# Программирование на Java

## Методическое руководство для преподавателей

### 27 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>

## Аннотация

Данное руководство предназначено для преподавателей информатики в ВУЗах России, принимающих участие по постановке курсов преподавания Java в рамках программы поддержки процесса обучения информационным технологиям в ВУЗах, представленной представительством компании Sun Microsystems в странах СНГ и Московским физико-техническим институтом (МФТИ) .

Методика преподавания опирается на четырехлетний опыт преподавания курсов по Java студентам МФТИ. По инициативе Sun Microsystems курс переработан таким образом, чтобы максимально облегчить процесс сертификации по Java для студентов, прослушавших курс. При подготовкие курсов использован опыт создания информационных систем на основе Java технологий, накопленный Центром Sun технологий МФТИ и Центром открытых систем и высоких технологий МФТИ.

Предлагается переподготовка преподавателей для чтения курсов по Java на основе совместного учебного центра Sun Microsystems и Центра дополнительного профессионального образования МФТИ.

# Оглавление

# Лекция 1. Что такое Java? История создания.

## Содержание лекции.



## 1. Что такое Java?

Что знают о Java обычные пользователи персональных компьютеров и Интернета? Что говорят о нем разработчики, которые не занимаются этой технологией профессионально?

Java широко известна как новейший объектно-ориентированный язык, легкий в изучении и позволяющий создавать программы, которые могут исполняться на любой платформе без каких-либо доработок (кросс-платформенность). Еще с Java почему-то всегда связана тема кофе (изображения логотипов, названия продуктов и т.д.). Программисты могут добавить к этому описанию, что язык похож на упрощенный C или C++ с добавлением garbage collector'a - автоматического сборщика "мусора" (механизм освобождения памяти, которая больше не используется программой). Также известно, что Java ориентирована на Интернет, и самое ее распространенное применение - небольшие программы, называющиеся апплеты, которые запускаются в браузере и являются частью HTML-страниц.

Критики в свою очередь утверждают, что язык вовсе не так прост в применении, многие замечательные свойства лишь обещаются, а на самом деле не очень-то работают, а

главное - программы на Java исполняются ужасно медленно. Следовательно, это просто некая модная технология, которая только на время привлечет к себе внимание, а затем исчезнет, как и многие другие.

Однако некоторые факты не позволяют согласиться с такой оценкой. Во-первых, со времени официального объявления Java прошло около семи с половиной лет - многовато для "просто модной технологии". Во-вторых, конференция разработчиков JavaOne, которая была впервые организована в 1996 году, уже через год собрала более 10.000 участников и стала крупнейшей конференцией по созданию программного обеспечения в мире (каждый следующий год число участников росло примерно на 5.000). Специальная программа Sun, объединяющая разработчиков Java по всему миру, Java Developer Connection также была запущена в 1996 году, через год она насчитывала более 100.000 разработчиков, а в 2000 году - более 1.5 миллионов. На данный момент число программистов на Java оценивается в 3 миллиона.

Было выпущено 5 основных версий языка, начиная с 1.0 в 1995 году и заканчивая 1.4 в феврале 2002 года. Следующая версия 1.5 планируется на 2003 год. Все версии и документация к ним всегда были доступны для бесплатного получения на официальном веб-сайте Java http://java.sun.com/. Один из первых продуктов для Java - JDK 1.1 (средство разработки на Java) в течение первых трех недель после объявления был загружен более 220.000 раз. Последняя версия 1.4 была загружена более 2 миллионов раз за первые 5 месяцев. Практически все ведущие производители программного обеспечения лицензировали технологию Java и регулярно объявляют о выходе новых продуктов, построенных на ней. Это и "голубой гигант" IBM, и создатель платформы Macintosh фирма Apple, и лидер в области реляционных БД Oracle, и даже первейший конкурент фирмы Sun - корпорация Microsoft - лицензировала Java еще в марте 1996 года.

В следующей главе описывается краткая история зарождения и развития идей, приведших к появлению Java, что поможет понять, чем на самом деле является эта технология, каковы ее истинные свойства и отличительные качества, для чего она предназначена, и откуда взялось такое количество различных слухов и мнений.

# 2. История создания Java

Если поискать в Интернете историю создания Java, то можно выяснить, что изначально язык назывался Oak (дуб), а работа по его созданию началась еще в 1990 году с довольно скандальной истории внутри корпорации Sun. Эти факты верны, однако на самом деле все было еще интереснее.

## 2.1. Сложности внутри Sun Microsystems

Действительно, события начинают разворачиваться в декабре 1990 года, когда бурного развития WWW (World Wide Web - "всемирная паутина") никто не мог еще даже предсказать. Тогда компьютерная индустрия была поглощена взлетом персональных компьютеров. Увы, фирма Sun Microsystems, занимающая заметную долю рынка серверов и высокопроизводительных станций, по мнению многих сотрудников и внешних экспертов не могла предложить ничего интересного для обычного пользователя "персоналок" - для них компьютеры от Sun представлялись "слишком сложными, очень некрасивыми и чересчур "тупыми" устройствами" [3].

Поэтому Скотт МакНили (Scott McNealy), член совета директоров, президент и CEO (исполнительный директор) корпорации Sun, не был удивлен, когда 25-летний хорошо зарекомендовавший себя программист Патрик Нотон (Patrick Naughton), проработав всего 3 года, объявил о своем желании перейти в компанию NeXT. Они были друзьями, и Патрик объяснил свое решение просто и коротко - "они все делают правильно". Скотт задумался на секунду, и произнес историческую фразу. Он попросил Патрика перед уходом описать, что, по его мнению, Sun делает не верно. Надо было не просто рассказать о проблеме, но предложить решение, не оглядываясь на существующие правила и традиции, как будто в его распоряжении имеются неограниченные ресурсы и возможности.

Патрик Нотон выполнил просьбу, вложив в свое письмо все свои мысли и сердце. Он безжалостно раскритиковал новую программную архитектуру NeWS, над которой фирма работала в то время, а также привел свои восторженные оценки только что объявленной операционной системы NeXTstep. Среди его предложений были: привлечь профессиональных художников-дизайнеров, чтобы сделать пользовательские интерфейсы Sun приятными и привлекательными; выбрать одно средство разработки и сконцентрировать усилия на одной оконной технологии, а не на нескольких сразу (Нотон был вынужден поддерживать сотни различных комбинаций технологий, платформ и интерфейсов, используемых в компании); наконец, уволить практически каждого из Window Systems Group (они будут просто не нужны, если выполнить предыдущие условия).

Конечно, Нотон был практически уверен, что его письмо просто проигнорируют, но все же задержал свой переход в NeXT в ожидании какой-нибудь ответной реакции. Однако она превзошла все ожидания.

МакНили разослал это письмо всему управляющему составу корпорации, а те переслали его своим ведущим специалистам. Откликнулись буквально все, и, по общему мнению, Нотон описал то, что все подозревали, но боялись признать. Решающей оказалась поддержка Билла Джоя (Bill Joy) и Джеймса Гослинга (James Gosling). Билл Джой - один из основателей и вице-президент Sun, а также участник проекта по созданию операционной системы UNIX в университете Беркли. Джеймс Гослинг пришел в Sun в 1984 году (до этого он работал в исследовательской лаборатории IBM) и был ведущим разработчиком, в частности автором первой реализации текстового редактора EMACS на C. Эти люди имели огромный авторитет в корпорации.

Чтобы развить этот успех, надо было предложить какой-то совершенно новый, новаторский проект. Нотон объединился с группой технических специалистов, и они просидели до 4.30 утра, обсуждая базовые концепции такого проекта. Их получилось всего три: главное - потребитель, и все строится исключительно в соответствии с его интересами; небольшая команда должна спроектировать небольшую же аппаратно-программную платформу; и воплотить эту платформу в устройстве, которое будет предназначено для персонального пользования, удобно и понятно в обращении - компьютер для обычных людей.

Этих идей оказалось достаточно, чтобы Джон Гейдж (John Gage), руководитель научных исследований Sun, смог организовать презентацию для высшего руководства корпорации. Нотон изложил свои условия, которые он считал необходимыми для успешного развития этого предприятия: команда должна расположиться вне офиса Sun, чтобы не испытывать никакого сопротивления революционным идеям; проект будет секретным для всех, кроме высшего руководства Sun; аппаратная и программная платформы могут быть не совместимыми с любыми продуктами Sun; на первый год группе необходим миллион долларов.

## 2.2. Проект Green

5 декабря 1990 года, в день, когда Нотон должен был перейти в компанию NeXT, Sun сделала ему встречное предложение. Руководство согласилось поддержать все его пожелания. Ожидаемый результат - "создать что-нибудь необычайное". 1 февраля 1991 года Патрик Нотон, Джеймс Гослинг и Майк Шеридан (Mike Sheridan) вплотную приступили к проекту, который получил название Green.

Цель они выбрали себе амбициозную - выяснить, что станет следующей волной развития компьютерной индустрии (первыми считаются появление полупроводников и персональных компьютеров), и что необходимо разработать для успешного участия в ней. С самого начала проект не рассматривался как чисто исследовательский, задача была создать реальный продукт, устройство.

Во время общения на ежегодном собрании Sun весной 1991 года, Гослинг заметил, что компьютерные чипы получили необычайное распространение, они применяются в видеомагнитофонах, тостерах, даже в дверных ручках гостиницы, где они жили! Тем не менее, до сих пор в каждом доме можно увидеть до трех пультов дистанционного управления - для телевизора, видеомагнитофона и музыкального центра. Так родилась идея разработать небольшое устройство с жидкокристаллическим сенсорным экраном, которое бы общалось с пользователем через анимацию, показывая, чем можно с его помощью управлять и как. Чтобы создать такой прибор Нотон начинает работать над специализированной графической системой, Гослинг берется за программное обеспечение, а Шеридан занимается бизнес-вопросами.

В апреле 1991 года команда выезжает из офиса Sun в новое помещение и отключается даже от внутренней сети корпорации. Они закупают разнообразные бытовые электронные устройства, такие как игровые приставки Nintendo, телевизионные приставки, пульты дистанционного управления, и играют в многочисленные игры целыми днями, чтобы лучше понять, как сделать пользовательский интерфейс легким в понимании и использовании. В качестве идеального примера Гослинг отмечал, что современные тостеры с микропроцессорами имеют точно такой же интерфейс, что и тостер его мамы, который служит уже 42 года. Он считал, что к этому должны стремиться все бытовые устройства, как, например, современный сигнал цветного телевидения можно принять с помощью черно-белого телевизора 50-х годов производства.

Очень быстро исследователи обнаружили, что практически все устройства построены на самых разных центральных процессорах. Это означает, что добавление новых функциональных возможностей крайне затруднено, так как необходимо учитывать ограничения и, как правило, довольно скудные возможности используемых чипов. Когда же Гослинг побывал на музыкальном концерте, где смог воочию наблюдать сложное переплетение проводов, огромное количество колонок и полуавтоматических прожекторов, которые, казалось, согласовано танцуют в такт музыке, он понял, что будущее за объединением сетей, компьютеров, и других электронных устройств в единую согласованную инфраструктуру.

Сначала Гослинг попытался модифицировать C++, чтобы создать язык для написания программ, минимально ориентированных под конкретные платформы. Однако очень скоро стало понятно, что это практически невозможно. Основное достоинство C++ - скорость программ, но отнюдь не их надежность. А надежность работы для обычных пользователей должна быть так же абсолютно гарантирована, как и совместимость обычных электрических вилки и розетки. Поэтому в июне 1991 года Гослинг, который написал свой первый язык

программирования в 14 лет, начинает разработку замены C++. Создавая новую директорию и раздумывая, как ее назвать, он выглянул в окно, и взгляд его остановился на растущем под ним дереве. Так язык получил свое первое название - Oak (дуб). Спустя несколько лет, на основе маркетинговых исследований имя сменили на Java.

Всего несколько месяцев потребовалось, чтобы довести разработку до стадии, когда стало возможным совместить новый язык с графической системой, над которой работал Нотон. Уже в августе команда смогла запустить первые программы, демонстрирующие работу будущего устройства.

Само устройство, по замыслу создателей, должно было быть размером с обычный пульт дистанционного управления, работать от батареек, иметь привлекательный и забавный графический интерфейс, и в конце концов стать любимой (и полезной!) домашней игрушкой. Чтобы построить этот не имеющий аналогов прибор, находчивые разработчики применили "технологию молотка". Они попросту находили какой-нибудь аппарат, в котором были подходящие детали или микросхемы, разбивали его молотком и таким образом добывали необходимые части. Так были получены основной жидкокристаллический экран, сенсорный экран и миниатюрные встроенные колонки. Центральный процессор и материнская плата были специально разработаны на основе высокопроизводительной рабочей станции Sun. Было придумано и оригинальное название - *7, или Star7 (с помощью этой комбинации кнопок можно было ответить с любого аппарата в офисе на звонок любой другого телефона, а поскольку редко кого из них можно было застать на рабочем месте, эти слова очень часто громко кричались на весь офис). Для придания привлекательности интерфейсу был создан забавный персонаж Дьюк (Duke), который всегда был готов помочь пользователю в выполнении его задач. В дальнейшем он стал неразлучным спутником Java, счастливым талисманом, он присутствует во многих документах, статьях, примерах кода.

Задача была совершенно новая, не на что было опереться, не было никакого опыта, никаких предварительных наработок. Команда трудилась, не прерываясь ни на один день. В августе 1991 года произошла первая демонстрация для Билла Джоя и Скотта МакНили. В ноябре группа снова подключились к сети Sun по модемной линии. Чем дальше развивался проект, тем больше новых членов присоединялось к команде разработчиков. Примерно в то время было придумано название для той идеологии, которую они создавали, - "1st Person" (условно можно перевести как "первое лицо").

Наконец, 4 сентября 1992 года Star7 был завершен и продемонстрирован МакНили. Это было небольшое устройство с 5" цветным (16 бит) сенсорным экраном без единой кнопки. Чтобы включить его, надо было просто дотронуться до экрана. Весь интерфейс был построен как мультик - никаких меню! Дьюк перемещался по комнатам забавно нарисованного, "мультяшного" дома, чтобы управлять им, надо было просто водить пальцем - никаких специальных органов управления. Можно было взять виртуальную телепрограмму с нарисованного дивана, выбрать передачу и "перетащить" ее на изображение видеомагнитофона, чтобы запрограммировать его на запись.

Результат превзошел все ожидания! Стоит напомнить, что устройства типа карманных компьютеров (PDA), начиная с Newton, появились заметно позже, не говоря уже о цветном экране. Это было время 286i и 386i процессоров Intel (486i уже появились, но были очень дороги) и MS DOS, даже мышь еще не была обязательным атрибутом персонального компьютера.

Руководство Sun было просто в восторге, появилось отличное оружие против таких могучих конкурентов, как HP, IBM и Microsoft. Созданная технология была способна отнюдь не

только демонстрировать мультики. Объектно-ориентированный язык Oak был готов стать достаточно мощным инструментом для написания программ, которые могут работать в сетевом окружении. Его объекты, свободно распространяющиеся по сети, работали бы на любом устройстве, начиная с персонального компьютера и заканчивая обычными бытовыми видеомагнитофонами и тостерами. На презентациях Нотон рисовал область применения Oak, изображая домашние компьютеры, машины, телефоны, телевизоры, банки и соединяя их единой сетью. Целое приложение, например, для работы с электронной почтой, могло быть построено в виде группы таких объектов, причем им было не обязательно располагаться на одном устройстве. Более того, как язык, ориентированный на распределенную архитектуру, Oak имел механизмы безопасности, шифрования, процедур аутентификации, причем все эти возможности были встроенные и, таким образом, незаметные и удобные для пользователя.

## 2.3. Компания FirstPerson

Крупные компании-производители, такие как Mitsubishi Electric, France Telecom, Dolby Labs, заинтересовались новой технологией, начались переговоры. Шеридан подготавливает бизнес-план с оригинальным названием "Beyond the Green Door" ("За зеленой дверью"), в котором предлагает Sun учредить дочернюю компанию для продвижения платформы Oak на рынок. 1 ноября 1992 года создается компания FirstPerson, которую возглавила Вэйн Роузинг (Wayne Rosing), перешедшая из Sun Labs. Арендуется роскошный офис, число сотрудников возрастает с 14 до 60 человек.

Однако в дальнейшем оказалось, что стоимость подобного решения (процессор, память, экран) составляет не менее $50. Производители же бытовой техники привыкли платить ничтожные суммы за дополнительную функциональность, облегчающую использование их продуктов.

В это время внимание компьютерной индустрии захватывает идея интерактивного телевидения, создается ощущение, что именно оно станет следующим революционным прорывом. Поэтому, когда в марте 1993 года Time Warner объявляет конкурс для производителей компьютерных приставок к телевизору для развертывания пробной сети интерактивного телевидения, FirstPerson полностью переключается на эту задачу. И снова неудача - победителем оказывается Джеймс Кларк (James Clark), основатель Silicon Graphics Inc., несмотря на то, что технологически его предложение уступает по возможности Oak. Впрочем, через год проект Time Warner и SGI проваливается, а Джеймс Кларк создает компанию Netscape, которая еще сыграет важную роль в успехе Java.

Другим потенциальным клиентом стал производитель игровых приставок 3DO. Понадобилось всего 10 дней, чтобы портировать Oak на эту платформу, однако после трехмесячных переговоров, директор 3DO потребовал полные права на новый продукт, и сделка не состоялась.

Наконец, в начале 1994 года стало понятно, что интерактивное телевидение оказалось ошибкой. Было много ожиданий, но им не суждено стать реальностью. Анализ состояния FirstPerson показал, что компания не имеет ни одного клиента или партнера, и ее дальнейшие перспективы довольно туманны. Руководство Sun требует немедленного составления нового бизнес-плана, позволяющего компании начать приносить прибыль.

## 2.4. World Wide Web

В погоне за призраком интерактивного телевидения многие участники компьютерного рынка совершенно пропустили поистине эпохальное событие. В апреле 1993 года Марк Андриссен (Marc Andreessen) и Эрик Бина (Eric Bina), работающие в Национальном Центре Суперкомпьютерных Приложений (National Center for Supercomputing Applications, NCSA) при университете Иллинойс, выпустили первую версию графического браузера ("обозревателя") Mosaic 1.0 для WWW. Хотя Internet существовал на тот момент уже около 20 лет, имеющимися протоколами связи (FTP, telnet и др.) пользоваться было очень неудобно, и Глобальная Сеть использовалась лишь в академической и государственной среде. Mosaic же основывался на новом языке разметки гипертекстовых документов (HyperText Markup Language, HTML), который с 1991 года разрабатывался в Европейском Институте Физики Частиц (CERN) специально для представления информации в Интернете. Этот формат позволял просматривать текст и изображения, а главное - поддерживал ссылки, с помощью которых можно было одним нажатием мыши перейти как на другую часть той же страницы, так и на страницу, которая могла располагаться совсем в другой части сети и географического мира. Именно такие перекрестные обращения, используя которые пользователь мог совершенно незаметно для себя посетить множество узлов Интернета, и позволили считать все HTML документы связанными частями единого целого - Всемирной Паутины (World Wide Web, WWW).

И самое важное - все эти новые достижения были совершенно бесплатно доступны для всех желающих. Впервые обычные пользователи персональных компьютеров безо всякой специальной подготовки могли пользоваться глобальной сетью не только для решения рабочих вопросов, а для поиска информации на самые разные темы. Количество документов в пространстве WWW стало расти экспоненциально, и очень скоро сеть Интернет стала поистине Всемирной. Правда, со временем обнаружилось, что такой способ организации и хранения информации очень напоминает свалку, в которой крайне трудно найти данные по какому-нибудь конкретному вопросу, однако, эта тема относится к совершенно другому этапу развития компьютерного мира.

Итак, совершенно необъяснимым способом Sun не замечает зарождения новой эпохи. Технический директор Sun впервые увидел Mosaic лишь три месяца спустя! И это притом, что около 50% серверов и рабочих станций в сети Интернет были произведены именно Sun.

Новый бизнес-план FirstPerson ставил цель, которая была неким промежуточным шагом от интерактивного телевидения к возможностям Интернета. Идея заключалась в создании платформы для кабельных компаний, конечными пользователями которой были бы обычные пользователи персональных компьютеров, объединенные сетями таких компаний. Используя технологию Oak, разработчики могли бы создавать приложения, по функциональности аналогичные CD-ROM программам, однако обладающие интерактивностью, позволяющей пользователям легко обмениваться любой информацией через сеть. Ожидалось, что такие сети в итоге и разовьются в полноценное интерактивное телевидение, и тогда Oak станет полноценным решением для этой индустрии. Об Интернете и Mosaic пока не говорилось ни слова.

По многим причинам этот план не устроил руководство Sun (он плохо соответствовал главному ожиданию - новая разработка должна была привести к увеличению спроса на продукты Sun). Из-за отсутствия перспектив половина сотрудников FirstPerson была переведена в только что созданную команду Sun Interactive, которая продолжила заниматься

мультимедиа-сервисами уже без Oak. Все предприятие оказалось под угрозой бесславной кончины, однако в этот момент Билл Джой снова оказал поддержку проекту, который вскоре дал миру платформу Java.

Когда создатели FirstPerson наконец обратили внимание на Интернет, они поняли, что функциональность тех сетевых приложений, для которых они создавали Oak, очень близки к WWW. Билл Джой вспомнил, как он двадцать лет назад принял участие в разработке UNIX в Беркли, и затем эта операционная система получила широчайшее распространение благодаря тому, что ее можно было загрузить по сети совершенно бесплатно. Такой принцип бесплатного распространения коммерческих продуктов создал саму WWW, тем же образом компания Netscape вскоре стала лидером рынка браузеров, так многие технологии получили возможность захватить долю рынка в кратчайшие сроки. Эти новые идеи при поддержке Джоя окончательно убедили руководство Sun, что Интернет может стать воскрешением платформы Oak (кстати, этот новый проект поначалу называли "Liveoak"). В итоге Джой садится писать очередной бизнес-план и отправляет Гослинга и Нотона начинать работу по адаптации Oak для Интернета. Гослинг пересматривает программный код платформы, а Нотон берется за написание "убойного" приложения, которое бы сразу продемонстрировало всю мощь Oak для Интернета.

В самом деле, эти технологии прекрасно подошли друг другу. Языки программирования всегда играли важную роль в развитии компьютерных технологий. Мейнфреймы не были особенно полезны, пока не появился Cobol. Благодаря языку Fortran от IBM, компьютеры стали широко применяться для научных вычислений и исследований. Basic - самый первый продукт от Microsoft - позволил всем программистам-любителям легко создавать программы для своих персональных компьютеров. Язык C++ стал основой для развития графических пользовательских интерфейсов, таких как Mac OS и Windows. Создатели Oak сделали все, чтобы эта технология сыграла такую же роль в программировании для Интернет.

Несмотря на то, что к середине 1994 года WWW достиг невиданных размеров (конечно, по меркам того времени), веб-страницы продолжали быть больше похожими на обычные бумажные издания, чем на интерактивные приложения. По большей части вся работа в сети заключалась в отправке запроса на веб-сервер и получении ответа, который содержал обычный статический HTML-файл, отображаемый браузером на стороне клиента. Уже тогда функциональность веб-серверов расширялась с помощью CGI (Common Gateway Interface). Эта технология позволяла по запросу клиента запускать обычную программу на сервере и ее результат отсылать обратно в качестве ответа. Поскольку в то время скорость каналов связи была невысокой (хотя, похоже, пользователи никогда не будут удовлетворены возможностями аппаратуры), то клиент мог ждать несколько минут, чтобы лишь увидеть сообщение, что он ошибся в одной букве запроса. Динамическое построение графиков при таком способе реализации означало бы генерацию GIF-файлов в реальном времени. А ведь зачастую клиентские машины являются полноценными персональными компьютерами, которые могли бы брать значительную часть работы взаимодействия с пользователем на себя, разгружая сервера.

Вообще, клиент-серверная архитектура, просто необходимая для большинства сложных корпоративных (enterprise) приложений, обладает рядом существенных технических сложностей. Основная идея - разместить общие данные на сервере, чтобы создать единое информационное пространство для работы многих пользователей, а программы, отображающие и позволяющие удобно редактировать эти данные, выполняются на клиентских машинах. Очень часто в корпорации используется несколько аппаратных платформ (это может быть как "историческое наследие", так и следствие того, что различные

подразделения, решая разные задачи, нуждаются в различных компьютерах). Следовательно, приложение необходимо развивать сразу в нескольких вариантах, что существенно удорожает поддержку. Кроме того, обновление клиентской части означает, что нужно перенастроить все компьютеры компании в кратчайший срок. А ведь часто обновлениями могут заниматься несколько групп разработчиков.

Попытка придать интернет-браузерам возможности полноценного клиентского приложения встречает еще большие трудности. Во-первых, обычные сложности предельно возрастают - в Интернете представлены поистине все существующие платформы, а количество и географическая распределенность пользователей делает быстрое обновление просто невозможным. Во-вторых, особенно остро встает вопрос безопасности. Через сеть удивительно быстро распространяется не только важная информация, но и вирусы. Текстовая информация и изображения не несут в себе никакой угрозы для клиентской машины, другое дело - исполняемый код. Наконец, приложения с красивым и удобным графическим интерфейсом, как правило, имели немаленький размер, недаром основным способом их распространения являлись CD-ROM'ы. Понятно, что для Интернета необходимо было серьезно поработать над компактностью кода.

Если оглянуться на историю развития Oak, то становится понятно, что эта платформа удивительным образом отвечает всем перечисленным требованиям интернет-программирования, хотя и создавалась во времена, когда про WWW никто даже и не думал. Видимо, это говорит о том, насколько верно предугадали развитие индустрии участники проекта Green.

## 2.5. Возрождение Oak

Для победного нашествия Oak не хватало последнего штриха - браузера, который бы поддерживал эту технологию. Именно он должен был стать тем самым "убойным" приложением Нотона, которое завершало почти пятилетнюю подготовительную работу перед официальным объявлением новой платформы.

Браузер назвали WebRunner. Нотону потребовался всего один выходной, чтобы написать основную часть программы. Это было в июле, а в сентябре 1994 года WebRunner уже демонстрировался руководству Sun. Небольшие программы, написанные на Oak для распространения через Интернет, назвали апплетами (applets), и на первом примере такого апплета Дьюк махал ручкой своим создателям.

Следующая демонстрация происходила на конференции, где встречались разработчики интернет-приложений и представители индустрии развлечений. Когда Гослинг начал презентацию WebRunner, аудитория не проявила большого интереса, решив, что это просто клон Mosaic. Тогда Гослинг провел мышкой над сложной трехмерной моделью химической молекулы.

Следуя за курсором, модель поворачивалась по всем направлениям! Сейчас это, возможно, не производит такого впечатления, однако надо представить, что в то время это было подобно переходу от картинки к кинематографу. Следующий пример демонстрировал анимированную сортировку. Вначале изображался набор отрезков разной длины. Затем синяя и красная линии начинали бегать по этому набору, сортируя их по размеру. Пример тоже нехитрый, однако наглядно демонстрирующий, что на стороне клиента появляется полноценная программная платформа. Оба эти апплета сейчас являются стандартными примерами и входят в состав Java Development Kit любой версии. Успех этой демонстрации,

которая закончилась бурными аплодисментами, показал, что Oak и WebRunner готовы устроить революцию в Интернете, так как все участники конференции начали переосмысление возможностей, которые предоставляет Всемирная Сеть.

Кстати, к тому времени в начале 1995 года, когда стало ясно, что официальное объявление уже близко, за дело взялись маркетологи. По результатам их исследований Oak был переименован в Java, а WebRunner стал называться HotJava. Многие тогда сильно удивлялись, что дало повод для такого решения. Основная легенда гласит, что Java - это сорт кофе (такой кофе действительно есть), который очень любили программисты. Видимо, примерно по той же логике появилось и название HotJava (горячая Java). Тема кофе навсегда останется в названиях (технология создания компонент названа Java Beans - зерна кофе, специальный формат для архивирования файлов с Java программами JAR - банка с кофе и т.д.) и логотипах, а сам язык критики стали называть "для кофеварок". Впрочем, сейчас все уже привыкли и не задумываются над названием, возможно, это и было целью (а тем, кто продолжает высказывать недовольство, приводят альтернативные варианты, которые рассматривались - Neon, Lyric, Pepper или Silk).

Согласно плану спецификация Java, реализация платформы и HotJava должны были свободно распространяться через Интернет. С одной стороны, это позволяло в кратчайшие сроки распространить технологию по всему миру и сделать ее стандартом де-факто для интернет-программирования. С другой стороны, при помощи всего сообщества разработчиков, которые бы высказывали свои замечания, можно было гораздо быстрее устранить все возможные ошибки и недоработки. Однако в конце 1994 года пока лишь считанные копии были распространены за пределы Sun. В феврале 1995 года выходит, возможно, первый пресс-релиз, сообщающий, что вскоре альфа-версии Oak и WebRunner будут доступны для всеобщего внимания.

Когда это случилось, команда стала считать каждый случай, когда кто-то загрузил продукты для просмотра. Вскоре пришлось считать уже сотнями. Затем решили, что если удастся достигнуть 10.000, то это будет означать "просто ошеломляющий успех". Дожидаться 10.000 пришлось совсем не так много времени, как они могли предполагать. Интерес нарастал лавинообразно, после просмотров приходило большое количество писем, и мощности интернет-канала стало все время не хватать. На письма всегда отвечали очень подробно, что по началу можно было делать, не отрываясь от работы. Затем по очереди стали назначать одного разработчика, чтобы он в течение недели только писал ответы. Наконец, потребовался специальный человек, так как письма приходили уже по 2-3 тысячи в день.

Вскоре руководство Sun осознало, что такой ошеломляющий успех Java не имеет никакого бюджета или плана для рекламы и других акций продвижения на рынок. Первым таким событием становится публикация 23 марта 1995 года в газете Sun Jose Mercury News статьи с описанием новой технологии, в которой приводился адрес официального сайта http://java.sun.com/, который по сей день является основным источником информации по Java.

## 2.6. Java выходит в свет

Наконец, вся подготовительная работа стала подходить к своему логическому завершению. Официальное объявление Java, уже получившей широкое признание и подающей самые радужные надежды, должно было произойти на конференции SunWorld. Ожидалось, что

это будет короткое информационное объявление, так как главная цель этого мероприятия - UNIX-системы. Однако все произошло не так, как планировалось.

В 4 часа утра в день конференции, после длинных и сложных переговоров, Sun подписывает важнейшее соглашение. Вторая сторона - компания Netscape, основанная в апреле 1994 года Джеймсом Кларком (он уже сыграл роль в судьбе Oak два года, когда перехватил предложение от Time Warner) и Марком Андриссеном (создателем NCSA Mosaic). Эта компания являлась лидером рынка браузеров после того, как в декабре 1994 года вышла первая версия Netscape Navigator, которая была открыта для бесплатного некоммерческого использования, что позволило занять на тот момент 75% рынка.

23 мая 1995 года технология Java и HotJava были официально объявлены Sun [14], и тут же было сообщено, что новая версия самого п¿опулярного браузера Netscape Navigator 2.0 будет поддерживать новую технологию [15]. По сути, это означало, что отныне Java становится такой же неотъемлемой составляющей WWW, как и HTML. Во второй раз презентация закончилась бурными аплодисментами всех присутствующих. Победное шествие Java началось.

# 3. История развития Java

Теперь, когда за Java стояли не только несколько создателей, но еще и целая армия разработчиков, корпорация Sun имела возможность построить впечатляющие планы развития технологии.

## 3.1. Браузеры

Конечно, основная линия развития оставалась связанной с браузерами. Хотя Интернет только начинал наполняться все новыми технологиями, уже возникали проблемы совместимости. Под разными платформами были немного разные браузеры, различались даже шрифты. В результате автор мог создать красивую аккуратную страницу, которая расползалась у клиента.

С помощью Java веб-страницу можно наполнить не только обычным текстом, но и динамическими элементами - простыми видео-вставками типа вращающегося земного шара или Дьюка, машущего рукой (хотя сейчас такие задачи хорошо решает анимированный GIF, а в более сложных случаях - Macromedia Flash); интерактивные элементы типа вращающейся модели химической молекулы; бегущие строки, содержащие, например, биржевые индексы или прогноз погоды.

Но на самом деле Java - это больше, чем симпатичное украшение HTML. Поскольку это полноценный язык программирования, то с его помощью можно создавать сложный пользовательский интерфейс. В самой первой версии Java Development Kit (средство разработки на Java) был пример апплета, представляющий простейшие электронные таблицы. Вскоре появился текстовый редактор, позволяющий менять стиль и цвет текста. Конечно, были игровые апплеты, обучающие, моделирующие физические и иные системы. Например, клиент, сделавший заказ в магазине или отправивший посылку почтой, получал возможность следить за доставкой через Интернет.

В отличие от обычных программ апплеты получили "в наследство" важное свойство HTML страниц. Прочитав сегодня содержание страницы новостей, клиент не сохраняет ее на

своем компьютере, а на следующий день читает обновленное содержание. Точно также, скачав апплет и поработав с ним, можно его удалить, а в следующий раз получить более новую версию. Таким образом, программы появляются и исчезают с машины клиента безо всякого усилия, не требуются ни специальные знания, ни действия, при этом автоматически поддерживаются самые последние версии.

С другой стороны, пользователь уже не привязан к своему основному рабочему месту, в любом интернет-кафе можно открыть нужную веб-страницу и начать работу с привычными программами. И все это без каких-либо опасений "подцепить" вирус. Для разработчиков было очень привлекательно, что их программы через день после выпуска могут увидеть самые разные пользователи по всему миру, независимо от того, какой компьютер, операционную систему и браузер они используют. Хотя браузер на стороне клиента должен поддерживать Java, как уже говорилось, пользователям предлагался HotJava, доступный на любой платформе. Самый популярный в то время Netscape Navigator, начиная с версии 2.0, также содержал Java. Однако, сегодня, как известно самый распространенный браузер - Microsoft Internet Explorer.

Компания Microsoft, добившись ошеломляющего успеха в области программного обеспечения для персональных компьютеров, стала (и в целом остается до сих пор) основным конкурентом в этой области для Sun, IBM, Netscape и других. Если в начале девяностых основные усилия Microsoft были направлены на операционную систему Windows и офисные приложения (MS Office), то в середине десятилетия стало очевидно, что пора всерьез заняться Internet. В начале 1995 года Билл Гейтс опубликовал планы объявления "войны" Netscape с целью занять такое же монопольное положение в WWW, как и в области операционных систем для персональных компьютеров. И когда вскоре Netscape подписывает лицензионное соглашение с Sun, Microsoft оказалась в трудной позиции.

Internet Explorer 2.0 был настолько непривлекательным, что никто не верил, что он может составить какую-нибудь заметную конкуренцию Netscape Navigator. А это значит, что новая версия IE 3.0 должна уметь все, что умеет только что вышедший NN 2.0. Поэтому 7 декабря 1995 года Microsoft объявляет о своем желании лицензировать Java, а в марте 1996 года соглашение о лицензировании подписано. Самая крупная компания по производству программного обеспечения была вынуждена поддерживать своего, возможно, самого опасного конкурента.

Сейчас мы имеем возможность оглянуться назад и оценить последствия прошлых событий. Теперь уже очевидно, что Microsoft полностью удалось осуществить свой план. Если Netscape Navigator 3.x еще соблюдал лидирующее положение, то Netscape 4.x уже начал уступать Internet Explorer 4.x. NN 5.x так и не вышел, а NN 6.x стал очередным разочарованием для бывших поклонников "Навигатора". Сейчас вышла версия 7.0, однако она не занимает серьезной доли рынка, в то время как Internet Explorer 5.0, 5.5 и 6.0 используют более 95% пользователей.

Забавно, что многие ожесточенно обвиняли Microsoft в том, что она боролась с Netscape нерыночными средствами. Однако сравним действия конкурентов. Среди многих шагов, предпринятых Microsoft для победы, была и поддержка независимой организации W3C, которая руководила разработкой нового стандарта HTML 3. Вначале Netscape считался локомотивом индустрии, постоянно развивая и модернизируя HTML, который изначально вообще-то не предназначался для графического оформления текста. Но когда за дело взялась Microsoft, она, вложив большое количество денег и людских ресурсов, смогла утвердить стандарты, которые отличались от уже реализованных в Netscape Navigator,

причем отличия порой были чисто формальными. В результате оказалось, что страницы, сделанные в соответствии с W3C спецификациями, отображались в Navigator искаженно. Немаловажно и то, что NN необходимо было скачивать (пусть и бесплатно) и устанавливать вручную, а IE быстро стал встроенным компонентом Windows, сразу готовым к использованию (и от которого, к слову, избавиться нельзя было принципиально).

А каким образом Netscape смог добиться лидирующего положения? В свое время подобными же методами компания пытался (успешно, в конце концов) выдавить с рынка NCSA Mosaic. Тогда HTML был особенно беден интересными возможностями, а потому интересные инновации, поддерживаемые Navigator'ом, сразу привлекали внимание разработчиков и пользователей. Однако такие страницы совершенно неправильно отображались в Mosaic, что склоняло его пользователей к переходу на решения компании Netscape.

В результате в связи с забвением Netscape и его Navigator многие вздохнули с облегчением. Хотя, безусловно, потеря конкуренции на рынке и воцарение такого опасного монополиста как Microsoft, никогда не идет на пользу конечным пользователям, однако, многие устали от "войны стандартов", когда и так небогатые возможности HTML приходилось изощренно подгонять таким образом, чтобы страницы выглядели одинаково в обоих браузерах.

Про HotJava, к сожалению, особенно сказать нечего. Некоторое время Sun поддерживала этот продукт и добавила возможность визуально создавать веб-страницы без знания HTML. Однако создать конкурентоспособный браузер не удалось, и вскоре развитие HotJava было остановлено. Сейчас еще можно скачать и посмотреть последнюю версию 3.0.

И последнее, на чем стоит остановиться, - это язык Java Script, который также весьма распространен и который до сих пор многие связывают с Java, видимо, по причине схожести имен. Впрочем, некоторые общие особенности у них действительно есть.

4 декабря 1995 года компании Netscape и Sun совместно объявляют новый "язык сценариев" (scripting language) Java Script. Как следует из пресс-релиза это открытый, кросс-платформенный объектный язык сценариев для корпоративных сетей и Интернета. Код Java Script описывается прямо в HTML тексте (хотя возможно и подгружать его из отдельных файлов с расширением .js). Этот язык предназначен для создания приложений, которые связывают объекты и ресурсы на клиентской машине или на сервере. Таким образом, Java Script с одной стороны расширяет и дополняет HTML, а с другой стороны - дополняет Java. С помощью Java пишутся объекты-апплеты, которыми можно управлять через язык сценариев.

Общие свойства Java Script и Java:

· легкость в освоении. По этому параметру Java Script сравнивают с Visual Basic - чтобы использоваться эти языки, серьезный опыт программирования не требуется.

· кросс-платформенность. Код Java Script выполняется браузером. Подразумевается, что браузеры на разных платформах должны обеспечивать одинаковую функциональность для страниц, использующих язык сценариев. Однако, это выполняется примерно в той же степени, что и поддержка самого HTML - различий все же очень много.

· открытость. Спецификация языка открыта для использования и обсуждения сообществом разработчиков.

· все перечисленные свойства позволяют утверждать, что Java Script хорошо приспособлен для интернет-программирования.

· синтаксисы языков Java Script и Java очень похожи. Впрочем, они также довольно сильно напоминают язык C.

· язык Java Script не объектно-ориентированный (хотя некоторые аспекты OO подхода поддерживаются), но позволяет использование различных объектов, предоставляемых браузером.

· похожая история появления и развития. Оба языка были объявлены компаниями Sun и Netscape с интервалом в несколько месяцев. Вышедший вскоре после это Netscape Navigator 2.0 поддерживал обе новые технологии. Есть предположение, что само название Java Script было дано для того, чтобы воспользоваться большой популярностью Java, либо для того, чтобы еще больше расширить понятие "платформа Java". Вполне вероятно, что основную работу по разработке языка провела именно Netscape.

Несмотря на большое количество схожих характеристик, Java и Java Script - совершенно различные языки, и в первую очередь - по назначению. Если изначально Java позиционировалась как язык для создания интернет-приложений (апплетов), то сейчас уже совершенно ясно, что Java - это полноценный язык программирования. Что касается Java Script, то он полностью оправдывает свое название языка сценариев, оставаясь расширением HTML. Впрочем, расширением довольно мощным, так как любители этой технологии ухитряются создавать вполне серьезные приложения, такие как 3D игры от первого лица (в сильно упрощенном режиме, естественно), хотя это скорее случай из области курьезов.

В заключение отметим, что код Java Script, исполняющийся на клиенте, оказывается доступен всем в открытом виде, что затрудняет охрану авторских прав. С другой стороны, из-за отсутствия полноценной поддержки объявления новых типов программы со сложной функциональностью зачастую оказываются слишком запутанными для того, чтобы ими могли воспользоваться другие.

## 3.2. Сетевые компьютеры

Когда стало понятно, что новая технология пользуется небывалым спросом, естественным желанием было укрепить и развить успех и распространенность Java. Для того чтобы Java не разделила судьбу NeWS (эта оконная система упоминалась в начале главы, она не получила развития, проиграв конкуренцию X Window), компания Sun старалась наладить сотрудничество с третьими фирмами для производства различных библиотек, средств разработчика, инструментариев. 9 января 1996 года было сформировано новое подразделение JavaSoft, которое и занялось разработкой новых Java-технологий и продвижением их на рынок. Главная цель - появление все большего количества самых разных приложений, написанных на этой платформе. Например, 1 июля 1997 года было объявлено, что ученые NASA (National Aeronautics and Space Administration, государственная организация США, занимающаяся исследованием космоса) с помощью Java-апплетов управляют роботом, изучающим поверхность Марса ("Java помогает делать историю!").

Пора остановиться подробнее на том, почему по отношению к Java используется этот термин - "платформа", чем Java отличается от обычного языка программирования?

Как правило, платформой называют сочетание, во-первых, аппаратной архитектуры ("железо"), которая определяется типом используемого процессора (Intel x86, Sun SPARC, PowerPC и др.), и, во-вторых, операционной системой (MS Windows, Sun Solaris, Linux, Mac OS и др.). При написании программ разработчик всегда пользуется средствами целевой

платформы для доступа к сети, поддержки потоков исполнения, работы с графическим пользовательским интерфейсом (GUI) и другим возможностям. Конечно, различные платформы в силу технических, исторических и других причин поддерживают различные интерфейсы (API, Application Programming Interface), а значит и программа может исполняться только под той платформой, под которую она была написана.

Однако часто заказчикам требуется одна и та же функциональность, а платформы они используют разные. Задача портирования приложений стоит перед разработчиками давно. Редко удается перенести сложную программу без существенной переделки, очень часто различные платформы слишком по-разному поддерживают многие возможности (например, операционная система Mac OS традиционно использует однокнопочную мышь, в то время как Windows изначально рассчитывалась на двухкнопочную).

А значит и языки программирования должны быть изначально ориентированы на какую-то конкретную платформу. Синтаксис и основные концепции легко распространить на любую систему (хотя это и не всегда эффективно), но библиотеки, компилятор и, естественно, бинарный, исполняемый код специфичен для каждой платформы. Так было с самого начала компьютерных вычислений, а потому лишь немногие, действительно удачные программы поддерживались сразу на нескольких системах, что приводило к некоторой изоляции миров программного обеспечения для различных операционных систем.

Было бы странно, если с развитием компьютерной индустрии разработчики не попытались создать универсальную платформу, под которой могли работать все программы. Особенно такому шагу способствовало бурное развитие Глобальной Сети Интернет, которая объединила пользователей независимо от типа используемых процессоров и операционных систем. Именно поэтому создатели Java задумали разработать не просто еще один язык программирования, а универсальную платформу для исполнения приложений, тем более что изначально Oak планировался для различных бытовых приборов, от которых ждать совместимости не приходится.

Каким же образом можно "сгладить" различия и многообразие операционных систем? Способ не новый, но эффективный - виртуальная машина. Приложения на языке Java исполняются в специальной, универсальной среде, которая называется Java Virtual Machine. JVM - это программа, которая пишется специально для каждой реальной платформы, чтобы с одной стороны скрыть все ее особенности, а с другой - предоставить единую среду исполнения для Java-приложений. Фирма Sun и ее партнеры создали JVM практически для всех современных операционных систем. Когда говорится о браузере с поддержкой Java, также подразумевается, что в нем имеется встроенная виртуальная машина.

Подробнее JVM рассматривается ниже, но необходимо сказать, что компания Sun прикладывала усилия, чтобы сделать эту машину вполне реальной, а не только виртуальной. 29 мая 1996 года объявляется операционная система Java OS (финальная версия выпущена в марте следующего года). Согласно пресс-релизу - "возможно, самая небольшая и быстрая операционная система, поддерживающая Java". Действительно, единственной целью ее создателей была возможность исполнять Java-приложения на широком спектре устройств - сетевые компьютеры, карманные компьютеры (PDA), принтеры, игровые приставки, мобильные телефоны и многие другие. Ожидалось, что Java OS будет реализована на всех аппаратных платформах. Это было необходимо для изначальной цели создателей Java - легкость добавления новой функциональности и совместимости в любые электрические приборы, которыми пользуется современный потребитель.

Это был первый шаг, распространяющий платформу Java на один уровень вниз - на уровень операционных систем. Предполагалась сделать и следующий шаг - создать аппаратную архитектуру, центральный процессор, который бы напрямую выполнял инструкции Java безо всякой виртуальной машины. Устройство с такой реализацией стало бы полноценным Java-устройством "на 100%".

Кроме бытовых приборов, компания Sun позиционировала такое решение и для компьютерной индустрии - сетевые компьютеры должны были заменить разнородные платформы персональных рабочих станций. Такой подход хорошо ложился в центральную концепцию Sun, выраженную в лозунге "Сеть - это компьютер". Возможности одного компьютера никогда не сравнятся с возможностями сети, объединяющей все ресурсы компании, а тем более - всего мира. Наверное, сегодня это уже очевидно, но во времена, когда WWW еще не опутала весь мир, идея была революционной.

Если же пытаться построить многофункциональную сеть, то к ее рабочим станциям предъявляются совсем другие требования - им не нужно быть особенно мощными - вычислительные задачи можно переложить на сервера. Более того, это особенно выгодно, так как позволит централизовать поддержку и обновление программного обеспечения, а также позволит сотрудникам не быть привязанным к своим рабочим местам. Достаточно войти с любого терминала в сеть, авторизоваться - и можно продолжать работу с того места, на котором она была оставлена. Это можно сделать в кабинете, зале для презентаций, кафе, в кресле самолета, дома - где угодно!

Кроме своих несомненных удобств, это начинание было с большим энтузиазмом поддержано индустрией и в силу того, что оно являлось сильнейшим оружием в борьбе с крупнейшей корпорацией-производителем программного обеспечения Microsoft. Тогда (да и сейчас) самой распространенной платформой являлась операционная система Windows на базе процессоров Intel (с чьей-то легкой руки теперь многими называемая Wintel). Этим компаниям удалось создать замкнутый круг, гарантирующий успех - все пользовались их платформой, так как под нее написано больше всего программ, что в свою очередь склоняло разработчиков создавать новые продукты именно для платформы Wintel. Поскольку корпорация Microsoft всегда очень агрессивно развивала свое преимущество в области персональных компьютеров (вспомним, как Netscape Navigator безнадежно проиграл конкуренцию MS Internet Explorer), это не могло не вызывать сильное беспокойство других представителей компьютерной индустрии. Понятно, что концепция сетевых компьютеров легко устраняла бы преимущества Wintel в случае широкого распространения. Разработчики и пользователи просто перестали бы задумываться, что находится внутри их рабочей станции, также как домашние потребители не имеют представления, на каких микросхемах собран их мобильный телефон или видеомагнитофон.

Выше рассказывалось, как и почему Microsoft лицензировала Java, хотя, казалось бы, этот шаг лишь способствовал опасному распространению новой технологии, ведь Internet Explorer завоевывал все большую популярность. Однако вскоре разразился судебный скандал. 30 сентября 1997 года вышел новый IE 4.0, а уже 7 октября Sun объявляет, что этот продукт не проходит тесты на соответствие со спецификацией виртуальной машины. 18 ноября Sun обращается в суд, чтобы запретить использование логотипа "Совместимый с Java" ("Java compatible") для MS IE 4.0. Оказалось, что Microsoft слегка "улучшила" язык Java, добавив несколько новых ключевых слов и библиотек. Не то чтобы это были сверхмощные расширения, однако достаточно привлекательные, чтобы заметная часть разработчиков начала ее использовать. К счастью, в Sun быстро осознали всю степень опасности такого шага. Java могла перестать быть универсальной платформой, для которой

верен знаменитый девиз "Write once, run everywhere" ("Написано однажды, работает везде"). В таком случае она потеряла бы основу своего успеха, превратившись всего лишь в "еще один язык программирования".

Компания Sun смогла отстоять свою технологию. 24 марта 1998 года суд согласился с требованиями компании (конечно, это было только предварительное решение, дело завершилось лишь 23 января 2001 года, Sun получил компенсацию в 20 миллионов долларов и добился выполнения лицензионного соглашения), а уже 12 мая Sun снова выступает с требованием обязать Microsoft включить полноценную версию Java в Windows 98 и другие программные продукты. Эта история продолжается до сих пор с переменным успехом сторон. Например, Microsoft исключила из виртуальной машины Internet Explorer'a библиотеку java.rmi, позволяющей легко создавать распределенные приложения, пытаясь привлечь разработчиков к DCOM - технологии, жестко привязанной к платформе Win32. В ответ многие компании стали распространять специальное дополнение (patch), устраняющее этот недостаток. В результате Microsoft остановила свою поддержку Java на версии 1.1, которая на данный момент является устаревшей и не имеет многих полезных возможностей. Это в свою очередь практически остановило широкое распространение апплетов, кроме случаев либо совсем несложной функциональности (типа бегущей строки или диалога с несколькими полями ввода и кнопками), либо приложений для внутренних сетей корпораций. Для последнего случая Sun выпустил специальный продукт Java Plug-in, который встраивается в MS IE и NN, позволяя им исполнять апплеты на основе Java самых последних версий, причем полное соответствие спецификациям гарантируется (первоначально продукт назывался Java Activator и впервые был объявлен 10 декабря 1997 года). На данный момент Microsoft то включает, то исключает Java из своей операционной системы Windows XP, видимо, пытаясь найти самый выгодный для себя вариант.

Что же касается сетевых компьютеров и Java OS, увы, они не смогли найти своих потребителей. Видимо, обычные персональные рабочие станции в совокупности с JVM требуют гораздо меньше технологических и маркетинговых усилий, и при этом вполне успешно справляются с прикладными задачами. А Java, в свою очередь, стала позиционироваться для создания сложных серверных приложений.

## 3.3. Платформа Java

Итак, Java обладает длинной и непростой историей развития, однако настало время рассмотреть, что же получилось у создателей, какими свойствами отличается эта технология.

Самое широко известное, и в тоже время вызывающее самые бурные споры, свойство много- или кросс-платформенности. Уже описывалось, что оно достигается за счет использования виртуальной машины JVM, которая является обычной программой, исполняемой операционной системой и предоставляющей все необходимые возможности Java-приложениям. Поскольку все параметры JVM специфицированы, то остается единственная задача - реализовать виртуальные машины на всех существующих и используемых платформах.

Наличие виртуальной машины определяет многие свойства Java, однако сейчас остановимся на следующем вопросе - является Java языком компилируемым или интерпретируемым? На самом деле, используются оба подхода.

Исходный код любой программы на языке Java представляется обычными текстовыми файлами, которые могут быть созданы в любом текстовом редакторе или специализированном средстве разработки и имеют расширение .java. Эти файлы подаются на вход Java-компилятора, который транслирует их в специальный Java байт-код. Именно этот компактный и эффективный набор инструкций поддерживается JVM и является неотъемлемой частью платформы Java.

Результат работы компилятора сохраняется в бинарных файлах с расширением .class. Java-приложение, состоящее из таких файлов, подается на вход виртуальной машине, которая начинает их исполнять, или интерпретировать, так как сама является программой.

Многие разработчики на раннем этапе жестко критиковали смелый лозунг Sun "Write once, run everywhere", обнаруживая все больше и больше несоответствий и нестыковок на различных платформах. Однако надо признать, что они были просто слишком нетерпеливыми. Java только появилась на свет, а первые версии спецификаций были недостаточно исчерпывающими. Кроме того, разработчики виртуальных машин также являются обычными людьми, и порой делают ошибки в своих продуктах.

Очень скоро Sun пришел к выводу, что просто свободно публиковать свои спецификации (что уже делалось задолго до Java) недостаточно. Необходимо еще и создавать специальные процедуры проверки новых продуктов на соответствие стандартам. Первый такой тест для JVM содержал всего около 600 проверок, через год их число выросло до десяти тысяч, и с тех пор все время увеличивается (именно его в свое время не смог пройти MS IE 4.0). Также, безусловно, авторы виртуальных машин все время совершенствовали их, устраняя ошибки и оптимизируя работу. Все-таки любая, даже очень хорошо задуманная, технология требует времени для создания высококачественной реализации. Аналогичный путь развития сейчас проходит Java 2 Micro Edition (J2ME), но об этом позже.

Следующим по важности является объектная ориентированность Java, что всегда упоминается во всех статьях и пресс-релизах. Важность самого ООП обсуждается в следующей главе, однако важно подчеркнуть, что в Java практически все реализовано в виде объектов - потоки выполнения (threads) и потоки данных (steams), работа с сетью, работа с изображениями, с пользовательским интерфейсом, обработка с ошибками и т.д. В конце концов, любое приложение на Java - это набор классов, описывающих новые типы объектов.

Подробное рассмотрение объектной модели Java проводится на протяжении всего курса, однако обозначим наиболее значимые особенности. Во-первых, создатели отказались от множественного наследования. Было решено, что оно слишком усложняет и запутывает программы. В языке используется альтернативный подход - специальный тип "интерфейс". Он подробно рассматривается в соответствующей главе.

Далее, в Java применяется строгая типизация. Это означает, что любая переменная и любое выражение имеет тип, известный уже на момент компиляции. Такой подход применен для упрощения выявления проблем, ведь компилятор сразу сообщает об ошибках и указывает их расположение в коде. Поиск же исключительных ситуаций (exceptions - так в Java называются некорректные ситуации) во время исполнения программы (runtime) потребует сложного тестирования, причем причина, породившая дефект, может обнаружиться в совсем другом классе. Таким образом, нужно прикладывать дополнительные усилия при написании кода, зато существенно повышается его надежность (а это одна из основополагающих целей, для которых и создавался новый язык).

В Java есть всего 8 типов данных, которые не являются объектами. Они были определены с самой первой версии и никогда не менялись. Это 5 целочисленных типов: byte, short, int, long, а также к ним относят символьный char. Затем 2 дробных типа float и double, и наконец булевский тип boolean. Такие типы называются простыми или примитивными (от английского primitive), и они подробно рассматриваются в главе, посвященной типам данных. Все остальные - объектные или ссылочные (англ. reference).

Синтаксис Java почему-то многих ввел в заблуждение. Он действительно создан на основе синтаксиса языков C/C++, так что если посмотреть на исходный код программ, написанных на этих языках и на Java, то не сразу удается понять, какая из них на каком языке написана. Это почему-то дало основание делать утверждения типа "Java - это упрощенный C++ с дополнительными возможностями, такими как garbage collector". Автоматический сборщик мусора (garbage collector) обсуждается чуть ниже, но считать что Java такой же язык, как и C++, - большое заблуждение.

Конечно, разрабатывая новую технологию, авторы Java опирались на широко распространенный язык программирования в силу целого ряда причин. Во-первых, они сами на тот момент считали C++ своим основным инструментом. Во-вторых, зачем придумывать что-то новое, когда есть хорошо подходящее старое? Наконец, очевидно, что незнакомый синтаксис отпугнет разработчиков и существенно осложнит внедрение нового языка, а ведь Java должна была максимально быстро получить широкое распространение. Поэтому синтаксис был лишь слегка упрощен, чтобы избежать слишком запутанных конструкций.

Но, как уже описывалось, C++ принципиально не годился для новых задач, которые поставили себе разработчики из компании Sun, поэтому идеология, модель Java была построена заново, причем в соответствии с совсем другими целями. Дальнейшие главы будут постепенно раскрывать конкретные различия.

Что же касается объектной модели, то она скорее была построена по образцу таких языков, как Smalltalk от IBM или разработанный еще в 60-е годы в Норвежском Вычислительном Центре язык Simula, на который ссылается сам создатель Java Джеймс Гослинг.

Другое немаловажное свойство Java - легкость в освоении и разработке - также получило неоднозначное трактование. Действительно, авторы позаботились избавить программистов от наиболее распространенных ошибок, которые порой допускают даже опытные разработчики на C/C++. И первое место здесь занимает работа с памятью.

В Java с самого начала был введен механизм автоматической сборки мусора (от английского garbage collector). Предположим, программа создает некоторый объект, работает с ним, а дальше наступает момент, когда он больше уже не нужен. Необходимо освободить занимаемую память, чтобы не мешать операционной системе нормально функционировать. В C/C++ это необходимо делать явным образом из программы. Понятно, что при таком подходе есть две опасности - либо удалить объект, который еще кому-то необходим (и если к нему действительно произойдет обращение, то возникнет ошибка), либо не удалить объект, ставший ненужным, а это означает утечку памяти, то есть программа начинает потреблять все большое количество оперативной памяти.

При разработке на Java программист вообще не думает об освобождении памяти. Виртуальная машина сама подсчитывает количество ссылок на каждый объект, и если оно становится равным нулю, то такой объект помечается для обработки garbage collector. Таким образом, программист должен следить лишь за тем, чтобы не оставалось ссылок на ненужные объекты. Сборщик мусора - это фоновый поток исполнения, который регулярно

просматривает существующие объекты, и удаляет уже не нужные. Из программы никак нельзя повлиять на работу garbage collector, можно только явно инициировать его очередной проход с помощью стандартной функции. Ясно, что это существенно упрощает разработку программ, особенно для начинающих программистов.

Однако опытные разработчики были недовольны тем, что они не могут полностью контролировать все, что происходит с их системой. Нет точной информации, когда именно будет удален объект, ставший ненужным, когда начнет работать (а значит и занимать системные ресурсы) поток сборщика мусора и т.д. Но, при всем уважении к опыту таких программистов, необходимо отметить, что подавляющее количество сбоев программ, написанных на C/C++, приходится именно на некорректную работу с памятью, причем порой это случается даже с широко распространенными продуктами весьма серьезных компаний.

Кроме того, особый упор делался на легкое освоение новой технологии. Как уже говорилось, ожидалось (и эти ожидания оправдались, что подтверждает правильность выбранного пути!), что Java должна получить максимально широкое применение, даже в таких компаниях, где никогда до этого не занимались программированием на таком уровне (бытовая техника типа тостеров и кофеварок, создание игр и других приложений для сотовых телефонов и т.д.). Был и целый ряд других соображений. Продукты для обычных пользователей, а не профессиональных программистов, должны быть особенно надежны. Интернет стал Всемирной Сетью за счет прихода непрофессиональных пользователей, а возможность создавать апплеты для них не менее привлекательна. Им требовался простой инструмент для создания надежных приложений.

Наконец, Интернет-бум 90-х годов набирал обороты и выдвигал новые, более жесткие требования на сроки разработки. Многолетние проекты, которые были обычным делом в прошлом, перестали отвечать потребностям заказчиков, новые системы надо было создавать максимум за год, а то и считанные месяцы.

Кроме введения garbage collector, были предприняты и другие шаги для облегчения разработки. Некоторые уже упоминались - отказ от множественного наследования, упрощение синтаксиса и др. Возможность создание многопоточных приложений было внесено с самой первой версии Java (исследования показали, что это очень удобно для пользователей, а существующие стандарты опираются на телетайпные системы, которые устарели много лет назад). Другие особенности будут рассмотрены в следующих главах. Однако то, что создание и поддержка систем действительно проще на Java, чем на C/C++, давно является общепризнанным фактом. Впрочем, все-таки эти языки созданы для разных целей, и каждый имеет свои неоспоримые преимущества.

Следующее важное свойство Java - безопасность. Изначальная нацеленность на распределенные приложение, и в особенности решение исполнять апплеты на клиентской машине, сделали вопрос защиты одним из самых приоритетных. При работе любой виртуальной машины Java действует целый комплекс мер. Далее приводится лишь краткое описание некоторых из них.

Во-первых, это правила работы с памятью. Уже говорилось, что очищение памяти производится автоматически. Резервирование ее также определяется JVM, а не компилятором или явным образом из программы, разработчик может лишь указать, что он хочет создать еще один новый объект. Указатели по физическим адресам отсутствуют принципиально.

Во-вторых, наличие виртуальной машины-интерпретатора значительно облегчает отсечение опасного кода на каждом этапе работы. Сначала байт-код загружается в систему, как правило, в виде class-файлов. JVM внимательно анализирует, что все они подчиняются общим правилам безопасности Java, а не были созданы злоумышленниками с помощью каких-то других средств (или не были искажены при передаче). Затем во время исполнения программы, интерпретатор легко может проверить каждое действие на допустимость. Возможности классов, которые были загружены с локального диска или по сети, существенно различаются (конечный пользователь легко может назначать или отменять конкретные права). Например, апплет по умолчанию никогда не получит доступ к локальной файловой системе. Такие встроенные ограничения есть и во всех стандартных библиотеках Java.

Наконец, существует механизм подписания апплетов и других приложений, загружаемых по сети. Специальный сертификат гарантирует, что пользователь получил код именно в том виде, в каком его выпустил производитель. Это, конечно, не дает дополнительных средств защит, но позволяет клиенту либо отказаться от работы с приложениями ненадежных производителей, либо сразу увидеть, что в программу внесены неавторизованные изменения. В худшем случае он знает, кто ответственен за причиненный ущерб.

Совокупность описанных свойств Java позволяет утверждать, что язык весьма приспособлен для разработки интернет- и интранет-(внутренние сети корпораций) приложений.

Наконец, важная отличительная особенность Java - это ее динамичность. Язык очень удачно задуман, в его развитии участвуют сотни тысяч разработчиков и многие крупные компании. Основные этапы этого развития кратко освещены в следующем разделе.

Итак, подведем итоги. Java-платформа обладает следующими преимуществами:

· переносимость, или кросс-платформенность;

· объектная ориентированность, создана эффективная объектная модель;

· привычный синтаксис C/C++;

· встроенная и прозрачная модель безопасности;

· ориентация на интернет-задачи, сетевые распределенные приложения;

· динамичность, легкость развития и добавления новых возможностей;

· легкость в освоении.

Но не следует считать, что более легкое освоение означает, что изучать язык не нужно вовсе. Чтобы писать действительно хорошие программы, создавать большие сложные системы, необходимо четкое понимание всех базовых концепций Java и используемых библиотек. Именно этому и посвящен данный курс.

Сразу оговоримся, что под продуктами здесь понимаются программные решения от компании Sun, являющиеся "образцами реализации" (reference implementation).

Итак, впервые Java была объявлена 23 мая 1995 года. Основными продуктами, доступными на тот момент в виде бета-версий, были:

· Java language specification, JLS, спецификация языка Java (описывающая лексику, типы данных, основные конструкции, и т.д.);

· спецификация JVM;

· Java Development Kit, JDK - средство разработчика, состоящее в основном из утилит, стандартных библиотек классов и демонстрационных примеров.

Спецификация языка была составлена настолько удачно, что практически без изменений используется по сей день. Конечно, было внесено большое количество уточнений, более подробных описаний, были добавлены и некоторые новые возможности (например, объявление внутренних классов), однако основные концепции остаются неизменными. Данный курс в большой степени опирается именно на спецификацию языка.

Спецификация JVM предназначена в первую очередь для создателей виртуальных машин, а потому практически не используется Java-программистами.

JDK долгое время было базовым средством разработки приложений. Оно не содержит никаких текстовых редакторов, а оперирует только с уже существующими java-файлами. Компилятор представлен утилитой javac (java compiler) . Виртуальная машина реализована программой java. Для тестовых запусков апплетов есть специальная утилита appletviewer. Наконец, для автоматической генерации документации на основе исходного кода прилагается средство javadoc.

Первая версия содержала всего 8 стандартных библиотек:

· java.lang - базовая классы, необходимые для работы любого приложения (название - сокращение от language);

· java.util - многие полезные вспомогательные классы;

· java.applet - классы для создания апплетов;

· java.awt, java.awt.peer - библиотека для создания графического интерфейса пользователя (GUI), называется Abstract Window Toolkit, AWT. Подробно описана в соответствующей главе.

· java.awt.image - дополнительные классы для работы с изображениями;

· java.io - работа с потоками данных (streams) и с файлами;

· java.net - работа с сетью.

Как видно, все библиотеки начинаются с java, именно они являются стандартными. Все остальные (начинающиеся с com, org и др.) могут меняться в любой версии без поддержки совместимости.

Финальная версия JDK 1.0 была выпущена в январе 1996 года.

Сразу поясним систему именования версий. Обозначение версии состоит из трех цифр. Первой пока всегда стоит 1. Это означает, что поддерживается полная совместимость между всеми версиями 1.x.x. То есть, программа, написанная на более старом JDK, всегда успешно выполнится на более новом. По возможности соблюдается и обратная совместимость - если программа откомпилирована более новым JDK, а никакие новые библиотеки не использовались, то в большинстве случаев старые виртуальные машины смогут выполнить такой код.

Вторая цифра изменилась от 0 до 4 (последняя на данный момент). В каждой версии происходило существенное расширение стандартных библиотек (212, 504, 1781, 2130 и

2738 - количество классов и интерфейсов с 1.0 по 1.4), а также добавлялись некоторые новые возможности в сам язык. Менялись и утилиты, входящие в JDK.

Наконец, третья цифра означает развитие одной версии. В языке или библиотеках ничего не меняется, лишь устраняются ошибки, производится оптимизация, могут меняться (добавляться) аргументы утилит. Так, последняя версия JDK 1.0 - 1.0.2.

Хотя с развитием версии 1.x ничего не удаляется, конечно, какие-то функции или классы устаревают. Они объявляются deprecated, и хотя они будут поддерживаться до объявления 2.0 (а про нее пока ничего не было слышно), пользоваться ими не рекомендуется.

Вместе с первым успехом JDK 1.0 пришла и критика. Основные недостатки, обнаруженные разработчиками, были следующими. Во-первых, конечно, производительность. Первая виртуальная машина работала очень медленно. Это связано с тем, что JVM по сути интерпретатор, который работает всегда медленнее, чем исполняется откомпилированный код. Однако, успешная оптимизация, устранившая этот недостаток, была еще впереди. Также отмечались довольно бедные возможности AWT, отсутствие работы с базами данных и другие.

В декабре 1996 года объявляется новая версия JDK 1.1, сразу выкладывается для свободного доступа бета-версия. В феврале 1997 года выходит финальная версия. Что было добавлено в новом выпуске Java?

Конечно, особое внимание было уделено производительности. Во-первых, многие части виртуальной машины были оптимизированы и переписаны с использованием Assembler, а не C, как до этого. Кроме этого, с октября 1996 года Sun развивает новый продукт - Just-In-Time компилятор, JIT. Его задача - транслировать Java байт-код программы в "родной" код операционной системы. Таким образом, время запуска программы увеличивается, но зато выполнение может ускоряться в некоторых случаях до 50 раз! С июля 1997 года появляется реализация под Windows, и JIT стандартно входит в JDK с возможностью отключения.

Были добавлены многие новые важные возможности. JavaBeans - технология, объявленная еще в 1996 году, позволяет создавать визуальные компоненты, которые легко интегрируются в визуальные средства разработки. JDBC (Java DataBase Connectivity) обеспечивает доступ к базам данных. RMI (Remote Method Invocation) позволяет легко создавать распределенные приложения. Были улучшены поддержка национальных языков и безопасность.

За первые 3 недели JDK 1.1 был скачан более 220.000 раз, менее чем через год - более 2-х миллионов раз. На данный момент версия 1.1 считается полностью устаревшей, и ее развитие остановилось на 1.1.8. Однако из-за того, что самый распространенный браузер MS IE до сих пор поддерживает только эту версию, она продолжает использоваться для написания небольших апплетов.

Кроме этого, с 11 марта 1997 года компания Sun начала предлагать Java Runtime Environment, JRE (среду выполнения Java). По сути дела это минимальная реализация виртуальной машины, необходимая для исполнения Java-приложений, без компилятора и других средств разработки. Если пользователь хочет только запускать программы, это именно то, что ему нужно.

Как видно, самым главным недостатком осталась слабая поддержка графического интерфейса пользователя (GUI).В декабре 1996 года компании Sun и Netscape объявляют новую библиотеку IFC (Internet Foundation Classes), разработанную Netscape полностью на Java и предназначенную как раз для создания сложного оконного интерфейса. В апреле

1997 года объявляется, что компании планируют объединить технологии AWT от Sun и IFC от Netscape для создания нового продукта Java Foundation Classes, JFC, в который должны войти:

· улучшенный оконный интерфейс, который получил особое название Swing;

· реализация Drag-and-Drop.

· поддержка 2D графики, улучшенная работа с изображениями;

· Accessibility API для пользователей с ограниченными возможностями; и другие возможности. Компания IBM также поддержала разработку новой технологии. В июле 1997 года стала доступна первая версия JFC. Первоначально библиотеки назывались, например, com.sun.java.swing для компонент Swing. В марте 1998 года вышла финальная версия этой технологии. За полгода продукт был скачан более 500.000 раз.

Выход следующей версии Java 1.2 много раз откладывался, но в итоге она настолько превзошла предыдущую 1.1, что с этого момента ее и все последующие версии начали называть платформой Java 2 (хотя номера, конечно, продолжались отсчитываться как 1.x.x, см выше описание правил нумерации). Первая бета-версия стала доступной в декабре 1997 года, а финальная версия была выпущена 8 декабря 1998 года, и за первые восемь месяцев ее скачали более миллиона раз.

Список появившихся возможностей очень широк, поэтому перечислим наиболее значимые из них:

· серьезно переработанная модель безопасности, введены понятия политики (policy) и разрешения (permission);

· JFC стал стандартной частью JDK, причем библиотеки стали называться, например, javax.swing для Swing (название javax указывает, что до этого библиотека считалась расширением Java);

· полностью переработанная библиотека коллекций (collection framework) - классов для хранения набора объектов;

· Java Plug-in был включен в JDK;

· улучшения в производительности, глобализации (независимости от особенностей разных платформ и стран), защита от "проблемы-2000".

С февраля 1999 года исходный код самой JVM был открыт для бесплатного доступа всем желающим.

Самое же существенное изменение произошло спустя полгода после выхода JDK 1.2. 15 июня 1999 года на конференции разработчиков JavaOne компания Sun объявила о разделении развития платформы Java 2 на три направления:

· Java 2 Platform, Standard Edition (J2SE);

· Java 2 Platform, Enterprise Edition (J2EE);

· Java 2 Platform, Micro Edition (J2ME).

На самом деле, подобная классификация уже давно назрела, так различные спецификации и библиотеки насчитывались в количестве нескольких десятков штук, а потому нуждались в четкой структуризации. Кроме того, такое разделение облегчало развитие и вывод на рынок технологии Java.

J2SE предназначается для использования на рабочих станциях и персональных компьютерах. На самом деле, Standard Edition - основа технологии Java и прямое развитие JDK (средство разработчика было переименовано в j2sdk).

J2EE содержит все необходимое для создание сложных, высоконадежных, распределенных серверных приложений. Условно можно сказать, что Enterprise Edition - это набор мощных библиотек (например, Enterprise Java Beans, EJB) и пример реализации платформы (сервера приложений, Application Server), которая их поддерживает. Работа такой платформы всегда опирается на j2sdk.

J2ME является усечением Standard Edition для того, чтобы удовлетворять жестким аппаратным требованием небольших устройств, таких как карманные компьютеры и сотовые телефоны.

Далее развитие этих технологий происходит разными темпами. Если J2SE уже была доступен более полугода, то финальная версия J2EE вышла лишь в декабре 1999 года. Последняя версия j2sdk 1.2 на данный момент - 1.2.2.

Тем временем борьба за производительность продолжалась, и Sun пытался еще больше оптимизировать виртуальную машину. В марте 1999 года объявляется новый продукт - высокоскоростная платформа (engine) Java HotSpot. Была оптимизирована работа с потоками исполнения, существенно переработаны алгоритмы автоматического сборщика мусора (garbage collector) и многое другое. Ускорение действительно было очень существенным, всегда заметное даже невооруженным взглядом за несколько минут работы с Java-приложением.

Новая платформа может работать в двух режимах - клиентском и серверном. Режимы различались настройками и другими оптимизирующими алгоритмами. По умолчанию работа идет в клиентском режиме.

Развитие HotSpot продолжалось более года, пока в начале мая 2000 года высокопроизводительная JVM не вошла в состав новой версии J2SE. В эту версию было внесено еще множество улучшений и исправлений, но именно большой прогресс в ускорении работы стал ключевым изменением нового j2sdk 1.3 (последняя подверсия 1.3.1).

Наконец, последняя на данный момент версия J2SE 1.4 вышла в феврале 2002 года. Она была разработана для более полной поддержки веб-сервисов (web services). Поэтому основные изменения коснулись работы с XML (Extensible Markup Language). Другое революционное добавление - выражение assert, позволяющее в отладочном режиме проверять верность условий, что должно серьезно упростить разработку сложных приложений. Наконец, были добавлены классы для работы с регулярными выражениями.

За первые пять месяцев j2sdk 1.4 было скачано более двух миллионов раз. В августе 2002 года была уже предложена версия 1.4.1, остающаяся самой современной на данный момент.

В заключение для демонстрации уровня развития Standard Edition приведем стандартные диаграммы, описывающие все их составляющие технологии, из документации к версиям 1.3: и 1.4:

# 4. Заключение

В этой лекции Вы узнали, какая сложная ситуация сложилась в корпорации Sun в эпоху развития персональных компьютеров в конце 1990 года. Патрик Нотон в своем письме сумел выявить истинные причины такого положения и обозначить истинные цели для создания успешного продукта. Благодаря этому при поддержке Джеймса Гослинга начался проект Green. Одним из продуктов, созданных в рамках этого проекта, стала совершенно новая платформа Oak. Для ее продвижения Sun учредила дочернюю компанию FirstPerson, но настоящий успех пришел, когда платформу, переименовав в Java, сориентировали на применение в Интернете. Глобальная сеть появилась в апреле 1993 года с выходом первого браузера Mosaic 1.0 и завоевывала пользовательскую аудиторию с поразительной скоростью. Первым примером Java-приложений стали апплеты, запускаемые при помощи специально созданного браузера HotJava. Наконец, после почти 4-х летней истории создания и развития, Java была официально объявлена миру. Благодаря подписанию лицензионного соглашения с Netscape, это событие стало поистине триумфальным.

Были рассмотрены различные варианты применения Java и насколько удачно удалось их развить и воплотить в жизнь. Отдельно был описан язык Java Script, который, несмотря на сходство в названии, имеет не так много общих черт с Java. Подробно рассмотрены отличительные особенности Java. Описаны базовые продукты от Sun: JDK и JRE. Кратко освещена история развития версий платформы Java, включая добавляемые технологии и продукты.

# 5. Контрольные вопросы

1-1. Перечислите основные свойства и преимущества платформы Java. Что такое JVM?

а.) Основные свойства языка:

- Кросс-платформенный
- Объектно-ориентированный
- - Строгая типизация
  - Наличие 8 примитивных типов
  - Вместо множественного наследования введены интерфейсы
- Легкий в освоении и разработке
- - Привычный синтаксис Java, являющийся развитием синтаксиса C/C++
  - Автоматическая сборка мусора (garbage collection)
  - Повышенная надежность Java-программ
  - Изначальная поддержка многопоточной архитектуры
- Высоко защищенный
- - Отсутствие указателей

  - Обязательная проверка виртуальной машиной кода загружаемых классов и действий, выполняемых программой
  - Встроенные возможности (работа с SSL и др.)
  - Подпись апплетов
- Приспособленный к разработке интернет-приложений
- Динамичный и активно развивающийся

JVM – это Java Virtual Machine, виртуальная машина Java, интерпретирующая байт-код, описываемый в class-файлах. Ее применение необходимо для обеспечения кросс-платформенности, а также для безопасности, однако создает определенные проблемы в вопросе производительности.

1-2. Является ли язык Java компилируемым или интерпретируемым?

a.) Используются оба подхода. Исходный код сначала компилируется в байт-код, который затем интерпретируется виртуальной машиной.

1-3. Что такое механизм автоматической сборки мусора (garbage collector)?

a.) Этот механизм автоматически подсчитывает количество ссылок на каждый объект Java. Когда на объект больше не указывает ни одна ссылка, он удаляется из памяти, освобождая ресурсы для программы.

1-4. В чем сходства и различия Java и C/C++?

a.) Основным сходством является во многом похожий синтаксис. Различий гораздо больше – Java обладает свойством кросс-платформенности, применяет интерпретатор, использует отличную объектную модель и др.

1-5. Почему Java является платформой, а не языком программирования?

a.) Применяя язык программирования, всегда необходимо учитывать используемую платформу при обращении к аппаратным ресурсам, таким как файловая система, работа с сетью, потоки исполнения и другие.Java сама является полноценной платформой, предоставляя приложениям единый интерфейс и скрывая различия используемой операционной системы и аппаратной платформы.

1-6. Из-за каких опасений корпорация Sun вела многолетнюю судебную тяжбу с Microsoft после выхода MS Internet Explorer 4.0?

a.) Одним из важнейших свойств Java является кросс-платформенность, для чего необходимо абсолютное соблюдение спецификации языка. Если разные реализации платформы будут иметь различные свойства, то совместимость будет утеряна.

Фирма Microsoft в своем продукте Internet Explorer 4.0 реализовала виртуальную машину с нарушениями лицензионного соглашения, а именно как раз внесла не стандартизованные возможности, что ставило под угрозу ключевое преимущество Java.

1-7. Из чего состоит и в каком виде записывается программа, написанная на Java?

а.) Программа на языке Java состоит из объявления классов, записанных в текстовых файлах с расширением java. Затем с помощью компилятора генерируются бинарные .class-файлы, содержащие байт-код, который затем интерпретируется виртуальной машиной.Часто класс-файлы упаковываются в архивы (.jar или .zip).

1-8. Что можно сказать относительно скорости выполнения Java-программ, и какие шаги предпринимала компания Sun в этой направлении?

а.) Быстродействие программ первых версий Java оставляло желать много лучшего. Виртуальная машина является интерпретатором, и ее первые реализации страдали от отсутствия должной оптимизации. Однако даже в то время для Java это не было принципиальным недостатком, поскольку основные конкурентные преимущества заключались в другом.

Компания Sun, непрерывно оптимизируя JVM, предложила продукт JIT (Just-in-time), который перед исполнением программы транслировал ее в «родной» код применяемой платформы, что существенно ускоряло работу приложений. С версии 1.3 применяется виртуальная машина HotSpot, применяющая улучшенные алгоритмы сборки мусора и существенно улучшающая быстродействие.

1-9. Что такое апплет?

а.) Апплет – это приложение, написанное на Java, распространяемое, как правило, через сеть Интернет и выполняемое виртуальной машиной браузера. Являются частью HTML-страницы.

Обычно имеет небольшой размер и существенно ограничивается в правах доступа к системе клиента, чтобы обеспечить безопасность выполнения кода, полученного из открытых источников.

1-10. Когда было официально объявлено о Java?

а.) 23 мая 1995 года на конференции SunWorld.

1-11. Где в Интернете можно найти самую полную и актуальную информацию о Java-платформе?

а.) Официальный сайт Java – http://java.sun.com

1-12. Какова система версий в Java? Что означает название Java2? Какая последняя выпущенная версия Java?

а.) Номер версии состоит их трех чисел.

Первое число всех существующих на данный момент версий – единица. Такие версии всегда совместимы между собой, то есть, программы, написанные на основе более старых версий, всегда будут корректно исполняться более новыми версиями. По возможности сохраняется и обратная совместимость – программы, созданные на основе более новых версий, будут работать и для более старых версий, если они обладают всеми используемыми свойствами.

Второе число изменилось от 0 до 4 (версии 1.0 по 1.4). Каждая новая версия обладает новыми возможностями по сравнению с предыдущей. Java версии 1.2 настолько превосходила платформу 1.1, что, начиная с

нее, все последующие версии называли платформой Java2 (при этом сама система номеров версий не изменилась).

Третья цифра означает номер версии поддержки платформы. Не добавляются никакие новые возможности, но исправляются ошибки, возможна дополнительная оптимизация. Последняя версия на данный момент Java 1.4.0.

1-13. Что означает сообщение deprecated?

a.) На данный момент новые версии Java продолжают поддерживать все возможности старых, однако некоторые классы и методы становятся не рекомендованными к использованию и, возможно, будут удалены их последующих версий. В этом случае они называются deprecated, и в процессе компиляции будут выданы предупреждения о том, что таких конструкций необходимо по возможности избегать.

1-14. На какие три направления было поделено развитие Java вскоре после выхода Java2?

a.) В середине 1999 года было объявлено о разделении Java на три направления:

- J2SE – Java2 Standard Edition, основа технологии Java, прямое развитие JDK
- J2EE – Java2 Enterprise Edition, платформа для создания сложных, распределенных, высоконадежных серверных приложений
- J2ME – Java2 Micro Edition, упрощенная J2SE для применения в небольших устройствах с ограниченными аппаратными ресурсами

1-15. Какая версия Java поддерживается в большинстве браузеров? Что такое Java Plug-in?

a.) Большинство браузеров поддерживает уже устаревшую версию Java 1.1, хотя практически каждый имеет некоторые отличительные особенности, отклонения от спецификации и т.п. Это делает довольно трудоемким создание универсальных апплетов, предназначенных для всех пользователей Интернет.

Поэтому компания Sun с конца 1997 года предлагает специальный продукт Java Plug-in, который можно установить на любой браузер и который позволяет запускать апплеты в точном соответствии со спецификацией Java. Plug-in доступен для любой версии платформы.

1-16. Что такое JDK и JRE? В чем сходство и разница между ними? Какие основные утилиты входят в их состав?

a.) JDK – это Java Development Kit, средство разработчика Java, включающее в себя набор утилит, стандартные библиотеки с их сходным кодом и набор демонстрационных примеров. Утилиты включают в себя:

- java – реализация JVM
- javac – компилятор Java
- appletviewer – средство для запуска апплетов

- jar – архиватор формата JAR
- javadoc – утилита для автоматической генерации документации

JRE – это Java Runtime Environment, среда выполнения Java, предназначена только для запуска готовых Java-приложений, а потому содержит лишь реализацию виртуальной машины и набор стандартных библиотек.

# Программирование на Java

## Лекция 2. Основы объектно-ориентированного программирования

### 27 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

В этой лекции излагаются основные концепции Объектно-Оринетированного Подхода (ООП) к проектированию программного обеспечения. Поскольку в Java почти все типы (за исключением 8 простейших) являются объектными, владение ООП становится необходимым условием для успешного применения языка. Лекция имеет вводный, обзорный характер. Для более детального изучения предлагается список дополнительной литературы и Интернет-ресурсов.

# Оглавление

# Лекция 2. Основы объектно-ориентированного программирования

## Содержание лекции.



## 1. Основы объектно-ориентированного программирования

### 1.1. Методология процедурно-ориентированного программирования

Появление первых электронных вычислительных машин или компьютеров ознаменовало новый этап в развитии техники вычислений. Казалось, достаточно разработать последовательность элементарных действий, каждое из которых можно преобразовать в понятные компьютеру инструкции, и таким образом любая вычислительная задача будет решена. Эта идея оказалась настолько жизнеспособной, что долгое время доминировала

над всем процессом разработки программ. Появились специализированные языки программирования, предназначенные для разработки программ, предназначенных для решения вычислительных задач. Примерами таких языков могут служить FOCAL (FOrmula CALculator) и FORTRAN (FORmula TRANslator).

Основой такой методологии разработки программ являлась процедурная или алгоритмическая организация структуры программного кода. Это было настолько естественно для решения вычислительных задач, что целесообразность такого подхода ни у кого не вызывала сомнений. Исходным понятием этой методологии было понятие алгоритма. Алгоритм - это способ решения вычислительных и др. задач, точно описывающий определенную последовательность действий, которые необходимо выполнить для достижения заданной цели или решения поставленной задачи

Примерами алгоритмов являются хорошо известные правила нахождения корней квадратного уравнения или линейной системы уравнений.

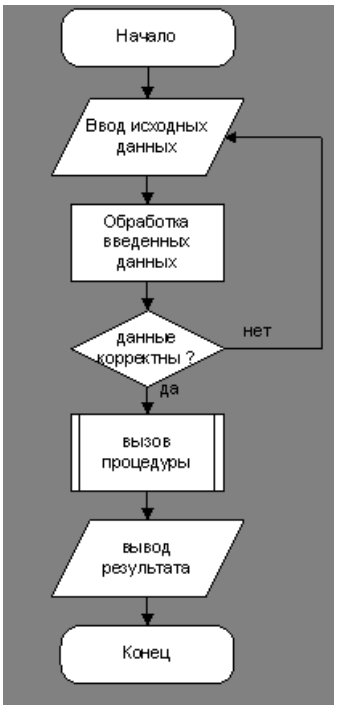
При увеличении объемов программ для упрощения их разработки появилась необходимость разбиения больших задач на подзадачи. В языках программирования возникло и закрепилось новое понятие процедуры. Использование процедур позволило разбивать большие задачи на подзадачи и таким образом упростило написание больших программ. Кроме того, использование процедурного подхода позволило уменьшить объем программного кода за счет написания часто используемых кусков кода в виде процедур и использования их в различных частях программы.

Так же, как и алгоритм, процедура представляет собой законченную последовательность действий или операций, направленных на решение отдельной задачи. В языках программирования появилась специальная синтаксическая конструкция, которая также получила название процедуры. Например, на языке Pascal описание процедуры выглядит следующим образом:

```
Procedure printGreeting(name: String)
Begin
   Print("Hello, ");
PrintLn(s);
End;
```

Назначение данной процедуры - вывести на экран приветствие "Hello, Name", где Name передается в процедуру в качестве входного параметра.

Со временем вычислительные задачи становились все сложнее, а значит, и решающие их программы увеличивались в размерах. Их разработка превратилась в серьезную проблему. Когда программа становится все больше, ее требуется разделять на все более мелкие фрагменты. Основой для такого разбиения как раз и стала процедурная декомпозиция, при которой отдельные части программы, или модули, представляли собой совокупность процедур для решения одной или нескольких задач. Одной из основных особенностей процедурного программирования заключается в том, что оно позволило создавать библиотеки подпрограмм (процедур), которые можно было бы использовать повторно в различных проектах или в рамках одного проекта. При процедурном подходе для визуального представления алгоритма выполнения программы используется так называемая Блок-схема. Соответствующая система графических обозначений была зафиксирована в ГОСТ 19.701-90. Пример блок-схемы изображен на рисунке (рис. 1.1).




Появление и интенсивное использование условных операторов и оператора безусловного перехода стало предметом острых дискуссий среди специалистов по программированию. Дело в том, что бесконтрольное применение в программе оператора безусловного перехода goto способно серьезно осложнить понимание кода. Такие запутанные программы сравнивали с порцией спагетти, называя их "bowl of spaghetti", имея в виду многочисленные переходы от одного фрагмента программы к другому, или, что еще хуже, возврат от конечных операторов программы к начальным. Ситуация казалась настолько драматичной, что в литературе зазвучали призывы исключить оператор goto из языков программирования. Именно с этого времени принято считать хорошим стилем отсутствие безусловных переходов.

Дальнейшее увеличение программных систем способствовало становлению новой точки зрения на процесс разработки программ и написания программных кодов, которая получила название методологии структурного программирования. Ее основой является процедурная декомпозиция предметной области решаемой задачи и организация отдельных модулей в виде совокупности процедур. В рамках этой методологии получило развитие нисходящее проектирование программ, или проектирование "сверху-вниз". Период наибольшей популярности идей структурного программирования приходится на конец 70-х - начало 80-х годов.

В этот период основным показателем сложности разработки программ считали ее размер. Вполне серьезно обсуждались такие оценки сложности программ, как количество строк программного кода. Правда, при этом делались некоторые предположения относительно синтаксиса самих строк, которые должны были удовлетворять определенным правилам. Например, каждая строка кода должна была содержать не более одного оператора. Общая трудоемкость разработки программ оценивалась специальной единицей измерения - "человеко-месяц" или "человеко-год". А профессионализм программиста напрямую связывался с количеством строк программного кода, который он мог написать и отладить в течение, скажем, месяца.

## 1.2. Методология объектно-ориентированного программирования

Увеличение размеров программ приводило к необходимости привлечения большего числа программистов, что, в свою очередь, потребовало дополнительных ресурсов для организации их согласованной работы. В процессе разработки приложений заказчик зачастую изменял функциональные требования, что еще более усложняло процесс создания программного обеспечения.

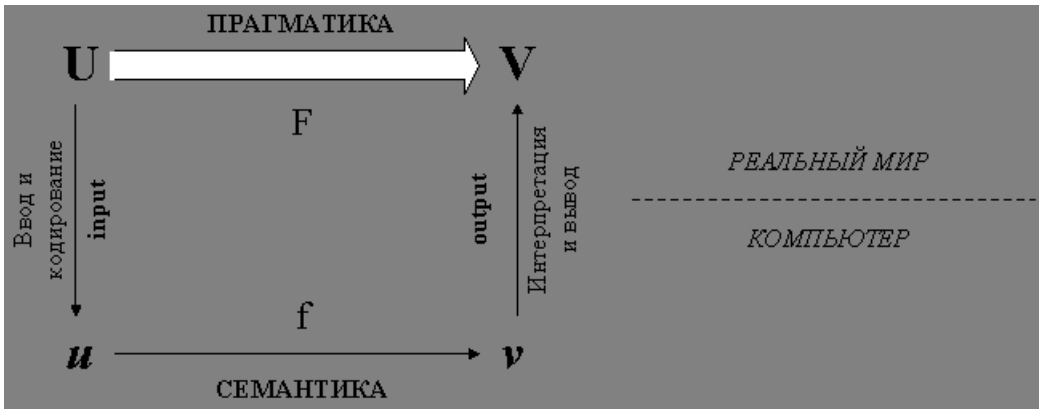
Но не менее важными оказались качественные изменения, связанные со смещением акцента использования компьютеров. В эпоху "больших машин" основными потребителями программного обеспечения были такие крупные заказчики, как большие производственные предприятия, финансовые компании, государственные учреждения. Стоимость таких вычислительных устройств была слишком высока для небольших предприятий и организаций.

Позже появились персональные компьютеры, которые имели гораздо меньшую стоимость и были значительно компактнее. Это позволило широко использовать их в малом и среднем бизнесе. Основными задачами в этой области являются обработка и манипулирование данными, поэтому вычислительные и расчетно-алгоритмические задачи с появлением персональных компьютеров отошли на второй план.

Как показала практика, традиционные методы процедурного программирования не способны справиться ни с нарастающей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Ею стало объектно-ориентированное программирование (ООП).

После составления технического задания, начинается этап проектирования, или дизайна, будущей системы. Объектно-ориентированный подход к проектированию основан на представлении предметной области задачи в виде множества моделей для языково-независимой разработки программной системы на основе ее прагматики.

Последний термин нуждается в пояснении. Прагматика определяется целью разработки программной системы, например обслуживание клиентов банка, управление работой аэропорта, обслуживание чемпионата мира по футболу и т.п. В формулировке цели участвуют предметы и понятия реального мира, имеющие отношение к создаваемой системе (см. рисунок 1.2). При объектно-ориентированном подходе эти предметы и понятия заменяются моделями, т.е. определенными формальными конструкциями, представляющими их в программной системе.




Модель содержит не все признаки и свойства представляемого ею предмета или понятия, а только те, которые существенны для разрабатываемой программной системы. Тем самым модель "беднее", а, следовательно, проще представляемого ею предмета или понятия.

Простота модели по отношению к реальному предмету позволяет сделать ее формальной. Благодаря такому характеру моделей при разработке можно четко выделить все зависимости и операции над ними в создаваемой программной системе. Это упрощает как разработку и изучение (анализ) моделей, так и их реализацию на компьютере.

Объектно-ориентированный подход помогает справиться с такими сложными проблемами, как

- уменьшение сложности программного обеспечения;
- повышение надежности программного обеспечения;
- обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов.
- обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

Более детально преимущества и недостатки объектно-ориентированного программирования будут рассмотрены в конце главы, так как для их понимания необходимо знание основных понятий и положений ООП.

Систематическое применение объектно-ориентированного подхода позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы. Этим объясняется интерес программистов к объектно-ориентированному подходу и объектно-ориентированным языкам программирования. ООП является одним из наиболее интенсивно развивающихся направлений теоретического и прикладного программирования.

## 1.3. Объекты

По определению будем называть объектом понятие, абстракцию или любой предмет с четко очерченными границами, имеющую смысл в контексте рассматриваемой прикладной проблемы. Введение объектов преследует две цели:

- понимание прикладной задачи (проблемы);
- введение основы для реализации на компьютере.

Примеры объектов: форточка, Банк "Империал", Петр Сидоров, дело № 7461, сберкнижка и т.д.

Каждый объект имеет определенное время жизни. В процессе выполнения программы или функционирования какой-либо реальной системы могут создаваться новые объекты и уничтожаться уже существующие.

Гради Буч дает следующее определение объекта:

Объект - это мыслимая или реальная сущность, обладающая характерным поведением, отличительными характеристиками и являющаяся важной в предметной области.[1]

Каждый объект имеет состояние, обладает некоторым хорошо определенным поведением и уникальной идентичностью.

### 1.3.1. Состояние.

Рассмотрим пример. Любой человек может находиться в некотором положении (состоянии) стоять, сидеть, лежать, и в то же время совершать какие либо действия.

Например, человек может прыгать, если он стоит, и не может - если он лежит, для этого ему потребуется сначала встать. Также в объектно-ориентированном программировании состояние объекта может определяться наличием или отсутствием связей между моделируемым объектом и другими объектами. Более подробно все возможные связи между объектами будут рассмотрены в разделе Типы отношений между классами.

Например, если у человека есть удочка (у него есть связь с объектом Удочка), то он может ловить рыбу, а если удочки нет, то такое действие невозможно. Из этих примеров видно, что набор действий, которые может совершать человек, зависит от параметров объекта, который его моделирует.

Для рассмотренных выше примеров такими характеристиками, или атрибутами, объекта Человек являются:

- текущее положение человека (стоит, сидит, лежит);
- наличие удочки (есть или нет).

В конкретной задаче могут появиться и другие свойства, например, физическое состояние, здоровье (больной человек обычно не прыгает).

Состояние (state) - совокупный результат поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно; в любой конкретный момент времени состояние объекта включает в себя перечень (обычно, статический) свойств объекта и текущие значения (обычно, динамические) этих свойств. [1]

### 1.3.2. Поведение

Каждый объект имеет определенный набор действий, которые с ним можно произвести. Например, возможные действия с некоторым файлом операционной системы ПК:

- создать
- открыть
- читать из файла

- писать в файл
- закрыть
- удалить

Результат выполнения действий зависит от состояния объекта на момент совершения действия, т.е. нельзя, например, удалить файл, если он открыт кем-либо (заблокирован). В то же время действия могут менять внутреннее состояние объекта - при открытии или закрытии файла свойство "открыт" принимает значения "да" или "нет" соответственно.

Программа, написанная с использованием ООП, обычно состоит из множества объектов, и все эти объекты взаимодействуют между собой. Обычно говорят, что взаимодействие между объектами в программе происходит посредством передачи сообщений между ними.

В терминологии объектно-ориентированного подхода понятия "действие", "сообщение" и "метод" являются синонимами. Т.е. выражения "выполнить действие над объектом", "вызвать метод объекта" и "послать сообщение объекту для выполнения какого-либо действия" эквивалентны. Последняя фраза появилась из следующей модели. Программу, построенную по технологии ООП, можно представить себе как виртуальное пространство, заполненное объектами, которые условно “живут” некоторой жизнью. Их активность проявляется в том, что они вызывают друг у друга методы, или посылают друг другу сообщения. Внешний интерфейс объекта, или набор его методов, это описание, какие сообщения он может получать.

Поведение (behavior) - действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта. [1]

### 1.3.3. Уникальность

Уникальность - это то, что отличает один объект от других. Например, у вас может быть несколько абсолютно одинаковых монет, даже если абсолютно все их свойства (атрибуты) одинаковы (год выпуска, номинал и.д.) и при этом вы можете использовать их независимо друг от друга - они по-прежнему остаются разными монетами.

В машинном представлении под параметром уникальности объекта наиболее часто понимается адрес размещения объекта в памяти.

Identity (уникальность) объекта состоит в том, что всегда возможно определить, указывают ли две ссылки на один и тот же объект, или на разные объекты. При этом два объекта могут во всем быть похожими, их образ в памяти может представляться одинаковыми последовательностями байтов, но, тем не менее, их Identity может быть различна.

Наиболее распространенной ошибкой является понимание уникальности как имя ссылки на объект. Это не верно, т.к. на один объект могут указывать несколько ссылок и ссылки могут менять свои значения (ссылаться на другие объекты) за время своего существования.

Итак, уникальность (identity) - природа объекта; то, что отличает его от других объектов.[1]

## 1.4. Классы

Все монеты из предыдущего примера принадлежат одному и тому же классу объектов (именно с этим связана их одинаковость). Номинальная стоимость монеты, металл, из которого она изготовлена, форма - это атрибуты класса: совокупность атрибутов и их

значений характеризует объект. Наряду с термином "атрибут" часто используют термины "свойство" и "поле", которые в объектно-ориентированном программировании являются синонимами.

Все объекты одного и того же класса описываются одинаковыми наборами атрибутов. Однако объединение объектов в классы определяется не наборами атрибутов, а семантикой. Так, например, объекты конюшня и лошадь могут иметь одинаковые атрибуты: цена и возраст. При этом они могут относиться к одному классу, если рассматриваются в задаче просто как товар, либо к разным классам, если в рамках поставленной задачи они будут использоваться различными способами, т.е. над ними будут совершаться различные действия.

Объединение объектов в классы позволяет рассмотреть задачу в более общей постановке. Класс имеет имя (например, лошадь), которое относится ко всем объектам этого класса. Кроме того, в классе вводятся имена атрибутов, которые определены для объектов. В этом смысле описание класса аналогично описанию типа структуры или записи (record), которые широко применяются в процедурном программировании; при этом каждый объект имеет тот же смысл, что и экземпляр структуры (переменная или константа соответствующего типа).

Формально класс - шаблон поведения объектов определенного типа с определенными параметрами, определяющими состояние. Все экземпляры одного класса (объекты, порожденные от одного класса)

- Имеют один и тот же набор свойств
- Общее поведение, одинаково реагируют на одинаковые сообщения

В соответствии с UML (Unified Modeling Language, унифицированный язык моделирования) класс имеет следующее графическое представление:

Класс изображается в виде прямоугольника, состоящего из трех частей. В верхней части помещается название класса, в средней - свойства объектов класса, в нижней - действия, которые можно выполнять с объектами данного класса (методы).

Каждый класс может также иметь специальные методы, которые автоматически вызываются при создании и уничтожении объектов этого класса:

- конструктор (constructor) - выполняется при создании объектов;
- деструктор (destructor) - выполняется при уничтожении объектов;

Обычно конструктор и деструктор имеют специальный синтаксис, который может отличаться от синтаксиса, используемого для написания обычных методов класса.

### 1.4.1. Инкапсуляция

Инкапсуляция (encapsulation) - это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса). При использовании объектно-ориентированного подхода не принято использовать прямой доступ к свойствам какого-либо класса из методов других классов. Для доступа к свойствам класса принято использовать специальные методы этого класса для получения и изменения его свойств.

Внутри объекта данные и методы могут обладать различной степенью открытости (или доступности). Степени доступности, принятые в языке Java, подробно будут рассмотрены в более поздних главах. Они позволяют более тонко управлять свойством инкапсуляции.

Открытые члены класса составляют внешний интерфейс объекта. Эта та функциональность, которая доступна другим классам. Закрытыми обычно объявляются все свойства класса, а так же вспомогательные методы, которые являются деталями реализации и от которых не должны зависеть другие части системы.

Благодаря сокрытию реализации за внешним интерфейсом класса можно менять внутреннюю логику отдельного класса, не меняя код остальных компонентов системы.

Обеспечение доступа к свойствам класса только через его методы также дает ряд преимуществ. Во-первых, так гораздо проще контролировать корректные значения полей, ведь прямое обращение к свойствам отслеживать невозможно, а значит им могут присвоить некорректные значения.

Во-вторых, не составит труда изменить способ хранения данных. Если информация станет храниться не в памяти, а в долговременном хранилище, таком как файловая система или база данных, то потребуется изменить лишь ряд методов одного класса, а не вводить эту функциональность во все части системы.

Наконец, программный код, написанный с использованием этого принципа легче отлаживать. Для того чтобы узнать, в какой момент времени и кто изменил свойство интересующего нас объекта, достаточно добавить вывод отладочной информации в тот метод объекта, посредством которого осуществляется доступ к свойству этого объекта. При использовании прямого доступа к свойствам объектов программисту бы пришлось добавлять вывод отладочной информации во все участки кода, где используется интересующий нас объект.

### 1.4.2. Полиморфизм

Полиморфизм является одним из фундаментальных понятий в объектно-ориентированном программировании наряду с наследованием и инкапсуляцией. Слово полиморфизм греческого происхождения и означает "имеющий много форм". Чтобы понять, что означает полиморфизм применительно к объектно-ориентированному программированию, рассмотрим пример.

Предположим мы хотим написать векторный графический редактор, в котором опишем в виде классов набор графических примитивов - Point, Line, Circle, Box, и т.д. У каждого из этих классов определим метод draw для отображения соответствующего примитива на экране.

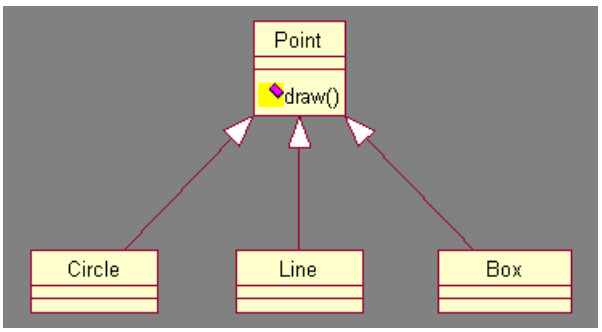
Очевидно, придется написать некоторый код, который при необходимости отобразить рисунок будет последовательно перебирать все примитивы, которые на момент отрисовки находятся на экране, и вызывать метод draw у каждого из них. Человек, незнакомый с полиморфизмом, вероятнее всего создаст несколько массивов: отдельный массив для каждого типа примитивов и напишет код, который последовательно переберет элементы из каждого массива и вызовет у каждого элемента метод draw. В результате получится примерно следующий код:

```
…
//создание пустого массива, который может содержать
//объекты Point с максимальным объемом 1000
Point[] p = new Point[1000];

Line[] l = new Line[1000];
```

```
Circle[] c = new Circle[1000];
Box[] b = new Box[1000];
…
// предположим, в этом месте происходит заполнение всех массивов
// соответствующими объектами
…
for(int i = 0; i < p.length;i++){ //цикл с перебором всех ячеек массива.
   //вызов метода draw() в случае,
   // если ячейка не пустая.
   if(p[i]!=null) p.draw();
}

for(int i = 0; i < l.length;i++){
   if(l[i]!=null) l.draw();
}

for(int i = 0; i < c.length;i++){
   if(c[i]!=null) c.draw();
}

for(int i = 0; i < b.length;i++){
   if(b[i]!=null) b.draw();
}
…
```

Недостатком написанного выше кода является дублирование практически идентичного кода для отображения каждого типа примитивов. Также неудобно то, что при дальнейшей модернизации нашего графического редактора и добавлении возможности рисовать новые типы графических примитивов, например Text, Star и т.д., при таком подходе придется менять уже существующий код и добавлять в него определения новых массивов, а также обработку элементов, содержащихся в них.

Используя полиморфизм, мы можем значительно упростить реализацию подобной функциональности. Прежде всего, создадим общий родительский класс для всех наших классов. Пусть таким классом будет класс Point. В результате получим иерархию классов, которая изображена на рисунке (1.3).




У каждого из дочерних классов метод draw переопределен таким образом, чтобы отображать экземпляры каждого класса соответствующим образом.

Для описанной выше иерархии классов, используя полиморфизм, можно написать следующий код:

```
…
Point p[] = new Point[1000];
p[0] = new Circle();
p[1] = new Point();
p[2] = new Box();
p[3] = new Line();
…
for(int i = 0; i < p.length;i++){
   if(p[i]!=null) p.draw();
}
…
```

В описанном выше примере массив p[] может содержать любые объекты, порожденные от наследников класса Point. При вызове какого-либо метода у любого из элементов этого массива будет выполнен метод того объекта, который содержится в ячейке массива. Например, если в ячейке p[0] находится объект Circle, то при вызове метода draw следующим образом

```
p[0].draw()
```

нарисуется круг, а не точка.

В заключение приведем формальное определение полиморфизма:

Полиморфизм (polymorphism) - положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов. Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций[1].

В процедурном программировании тоже существует понятие полиморфизма, которое отличается от рассмотренного механизма в ООП. Процедурный полиморфизм предполагает возможность создания нескольких процедур или функций с одинаковым именем, но разными количеством или типами передаваемых параметров. Такие одноименные функции называются перегруженными, а само явление - перегрузкой (overloading). Перегрузка функций существует и в ООП и называется перегрузкой методов.

Примером использования перегрузки методов в языке Java может служить класс PrintWriter, который используется в частности для вывода сообщений на консоль. Этот класс имеет множество методов println, которые различаются типами и/или количеством входных параметров. Вот лишь несколько из них:

```
void println() // переход на новую строку

// выводит значение булевской переменной (true или false)
void println(boolean x)
void println(String x) // выводит строку - значение текстового параметра
```

Определенные сложности возникают при вызове перегруженных методов. В Java существуют специальные правила, которые решают эту проблему. Они будут рассмотрены в соответствующей главе.

## 1.5. Типы отношений между классами

Как правило, любая программа, написанная на объектно-ориентированном языке, представляет собой некоторый набор классов, связанных между собой. Можно провести аналогию между строительством дома и написанием программы. Подобно тому, как здание строится из кирпичей, компьютерная программа с использованием ООП строится из классов. Причем эти классы должны знать друг о друге, для того чтобы взаимодействовать между собой и сообща выполнить поставленную задачу.

Возможны следующие связи между классами в рамках объектной модели (приводятся лишь наиболее простые и часто используемые виды связей, подробное их рассмотрение выходит за рамки этой ознакомительной главы):

- Агрегация (Aggregation)
- Ассоциация (Association)
- Наследование (Inheritance)
- Метаклассы (Metaclass)

### 1.5.1. Агрегация

Отношение между классами типа "содержит" или "состоит из" называется агрегацией или включением. Например, если аквариум наполнен водой и в нем плавают рыбки, то можно сказать, что аквариум агрегирует в себе воду и рыбок.

Такое отношение включения или агрегации (aggregation) изображается линией с ромбиком на стороне того класса, который выступает в качестве владельца или контейнера. Необязательное название отношения записывается посередине линии.

В нашем примере отношение "contain" является двунаправленным. Объект класса Aquarium содержит несколько объектов Fish. В то же время каждая рыбка "знает", в каком именно аквариуме она живет. Факт участия класса в отношении изображается посредством роли. В примере можно видеть роль "home" класса Aquarium (аквариум является домом для рыбок), а также роль "inhabitants" класса Fish (рыбки являются обитателями аквариума). Название роли обычно совпадает с названием соответствующего поля в классе. Изображение такого поля на диаграмме излишне, если уже изображено имя роли. Т.е. в данном случае класс Aquarium будет иметь свойство (поле) inhabitants, а класс Fish - свойство home.

Число объектов, участвующих в отношении, записывается рядом с именем роли. Запись "0..n" означает "от нуля до бесконечности". Приняты так же обозначения:

- "1..n" - от единицы до бесконечности;
- "0" - ноль;
- "1" - один;
- "n" - фиксированное количество;

- "0..1" - ноль или один.

Код, описывающий рассмотренную модель и явление агрегации, может выглядеть, например, следующим образом:

```
// определение класса Fish
public class Fish {

   // определения поля home (ссылка на объект Aquarium)
   private Aquarium home;

   public Fish() {
   }
}

// определение класса Aquarium
public class Aquarium {

   // определения поля inhabitants (массив ссылок на объекты Fish)
   private Fish inhabitants[];

   public Aquarium() {
   }
}
```

### 1.5.2. Ассоциация

Если объекты одного класса ссылаются на один или более объектов другого класса, но ни в ту, ни в другую сторону отношение между объектами не носит характера "владения" или контейнеризации, то такое отношение называют ассоциацией (association). Отношение ассоциации изображается так же,как и отношение агрегации, но линия, связывающая классы - простая, без ромбика.

В качестве примера можно рассмотреть программиста и его компьютер. Между этими двумя объектами нет агрегации, но существует четкая взаимосвязь. Так, всегда можно установить за какими компьютерами работает какой-либо программист, а также какие люди пользуются отдельно взятым компьютером. В рассмотренном примере имеет место ассоциация многие-ко-многим.

В данном случае между экземплярами классов Programmer и Computer в обе стороны используется отношение "0..n", т.к. программист теоретически может не пользоваться компьютером (если он теоретик или на пенсии). В свою очередь компьютер может никем не использоваться (если он новый и еще не установлен).

Код, соответствующий рассмотренному примеру, будет, например, следующим:

```
public class Programmer {
   private Computer computers[];

   public Programmer() {
   }
```

```
}

public class Computer {
   private Programmer programmers[];

   public Computer() {
   }
}
```

### 1.5.3. Наследование

Наследование (inheritance) - это отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов. Наследование вводит иерархию "общее/частное", в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение.

В качестве примера можно рассмотреть задачу, в которой необходимо реализовать классы "Легковой автомобиль" и "Грузовой автомобиль". Очевидно, эти два класса имеют много общей функциональности. Так, оба они имеют 4 колеса, двигатель, могут перемещаться и т.д. Всеми этими свойствами обладает любой автомобиль, не зависимо от того грузовой он или легковой, 5- или 12-местный. Разумно вынести эти общие свойства и функциональность в отдельный класс, например "Автомобиль", и наследовать от него классы "Легковой автомобиль" и "Грузовой автомобиль" чтобы избежать повторного написания одного и того же кода в разных классах.

Отношение обобщения обозначается сплошной линией с треугольной стрелкой на одном из концов. Стрелка указывает на более общий класс (класс-предок или суперкласс), а ее отсутствие - на более специальный класс (класс-потомок или подкласс)

Использование наследования способствует уменьшению количества кода, написанного для описания схожих сущностей, а так же способствует написанию более эффективного и гибкого кода.

В рассмотренном примере применено одиночное наследование. Некоторый класс так же может наследовать свойства и поведение сразу нескольких классов. Наиболее популярным примером применения множественного наследования является проектирование системы учета товаров в зоомагазине.

Все животные в зоомагазине являются наследниками класса "Животное", а также наследниками класса "Товар". Т.е. все они имеют возраст, нуждаются в пище и воде, и в то же время имеют цену и могут быть проданы.

Множественное наследование на диаграмме изображается точно так же как и одиночное наследование за исключением того, что линии наследования соединяют класс-потомок сразу с несколькими суперклассами.

Не все объектно-ориентированные языки программирования содержат в себе языковые конструкции для описания множественного наследования.

Многие объектно-ориентированные языки программирования не поддерживают множественное наследование и не имеют синтаксических конструкций для его реализации.

В языке Java множественное наследование имеет ограниченную поддержку через интерфейсы и будет рассмотрено в следующих главах.

### 1.5.4. Метаклассы

Итак, любой объект имеет структуру, состоящую из полей и методов. Объекты, имеющие одинаковую структуру и семантику, описываются одним классом, который и является, по сути, определением структуры объектов, порожденных от него.

В свою очередь каждый класс, или описание, всегда имеет строгий шаблон, задаваемый языком программирования или выбранной объектной моделью. Он определяет, например, допустимо ли множественное наследование, какие ограничения на имен½ование классов, как описываются п¿оля и методы, набор существующих типов данных и многое другое. Таким образом, класс можно рассматривать как объект, у которого есть свойства (имя, список полей и их типы, список методов, список аргументов для каждого метода и т.д.). Также класс может обладать поведением, то есть поддерживать методы. А раз для любого объекта существует шаблон, описывающий свойства и поведение этого объекта, то его можно определить и для класса. Такой шаблон, задающий различные классы, называется метакласс.

Чтобы легче представить себе, что такое метакласс, рассмотрим пример некой бюрократической организации. Будем считать, что все классы в такой системе - некоторые строгие инструкции, которые описывают, что нужно сделать, чтобы породить новый объект (например, нанять нового служащего или открыть новый отдел). Как и полагается классам, они описывают все свойства новых объектов (например, зарплату и профессиональный уровень для сотрудников, площадь и имущество для отделов) и их поведение (обязанности служащих и функции подразделений).

В свою очередь написание новой инструкции можно строго регламентировать. Скажем, необходимо использовать специальный бланк, придерживаться правил оформления и заполнить все обязательные поля (например, номер инструкции и фамилии ответственных работников) Такая "инструкция инструкций" и будет представлять собой метакласс в ООП.

Итак, объекты порождаются от классов, а классы - от метакласса. Он, как правило, в системе есть только один. Но существуют языки программирования, в которых можно создавать и использовать собственные метаклассы, например язык Python. Например, функциональность метакласса может быть следующая: при создании класса он будет просматривать список всех методов в классе и если имя метода имеет вид set_XXX или get_XXX, то автоматически создать поле с именем XXX, если такого не существует.

Поскольку метакласс сам является классом, то нет никакого смысла в заведении "мета-мета-классов".

В языке Java также есть такое понятие. Это класс, который так и называется - Class (описывает классы) и располагается в основной библиотеке java.lang. Виртуальная машина использует его по прямому назначению. Когда загружается очередной .class-файл, содержащий описание нового класса, JVM порождает объект класса Class, который будет хранить его структуру. Таким образом, Java использует концепцию метакласса в самых практических целях. С помощью Class реализована поддержка статических (static) полей и методов. Наконец, этот класс содержит ряд методов, полезных для разработчиков. Они будут рассмотрены в следующих главах.

## 1.6. Достоинства ООП

От любой методики разработки программного обеспечения мы ждем, что она поможет нам в решении наших задач. Но одной из самых значительных проблем проектирования является сложность. Чем больше и сложнее программная система, тем важнее становится разбить ее на небольшие, четко очерченные части. Чтобы справиться со сложностью, необходимо абстрагироваться от мелких деталей. Для этой цели классы представляют собой весьма удобный инструмент.

- Классы позволяют проводить конструирование из полезных компонент, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
- Данные и операции над ними вместе образуют определенную сущность, и они не разносятся по всей программе, как это нередко бывает в случае процедурного программирования, а описываются вместе. Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- Инкапсуляция позволяет привнести свойство модульности, что облегчает распараллеливание выполнения задачи между несколькими исполнителями иобновление версий отдельных компонент.

ООП дает возможность создавать расширяемые системы. Это одно из самых значительных достоинств ООП, и именно оно отличает данный подход от традиционных методов программирования. Расширяемость означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе исполнения программы.

Полиморфизм оказывается полезным преимущественно в следующих ситуациях.

- Обработка разнородных структур данных.

  Программы могут работать, не различая вида объектов, что существенно упрощает код. Новые виды могут быть добавлены в любой момент.

- Изменение поведения во время исполнения.

  На этапе исполнения один объект может быть заменен другим, что позволяет легко без изменения кода адаптировать алгоритм, в зависимости от того, какой используется объект.

- Реализация работы с наследниками.

  Алгоритмы можно обобщить настолько, что они уже смогут работать более чем с одним видом объектов.

- Создание “каркаса” (framework).

  Независимые от приложения части предметной области могут быть реализованы в виде набора универсальных классов, или каркаса (framework), и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Часто на практике многоразового использования программного обеспечения добиться не удается из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет нам извлечь максимум из многоразового использования компонент.

- Сокращается время на разработку, которое с выгодой может быть отдано другим задачам.
- Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
- Когда некая компонента используется сразу несколькими клиентами, то улучшения, вносимые в ее код, одновременно оказывают свое положительное влияние и на множество работающих с ней программ.
- Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.

## 1.7. Недостатки ООП

Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но также и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод. Для абстрактных методов, которые пусты, в документации должно даже говориться о том, для каких целей предполагается использовать переопределяемый метод.

В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу. Для получения такой информации нужны специальные инструменты вроде навигаторов классов. Если конкретный класс расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому классу. Реализация операции, таким образом, рассредоточивается по нескольким классам, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.

Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными. Зато количество методов намного выше. Короткие методы обладают тем преимуществом, что в них легче разбираться, неудобство же их связано с тем, что код для обработки сообщения иногда "размазан" по многим маленьким методам.

Абстракцией данных не следует злоупотреблять. Чем больше данных скрыто в недрах класса, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с классом этих данных знать не требуется.

Часто можно слышать, что ООП является неэффективным. Как же дело обстоит в действительности? Мы должны четко проводить грань между неэффективностью на этапе выполнения, неэффективностью в смысле распределения памяти и неэффективностью, связанной с излишней универсализацией.

1. Неэффективность на этапе выполнения. В языках типа Smalltalk сообщения интерпретируются во время выполнения программы путем осуществления поиска их в одной или нескольких таблицах и за счет выбора подходящего метода. Конечно, это медленный процесс. И даже при использовании наилучших методов оптимизации Smalltalk-программы в десять раз медленнее оптимизированных C-программ.

В гибридных языках типа Oberon-2, Object Pascal и C++ посылка сообщения приводит лишь к вызову через указатель процедурной переменной. На некоторых машинах сообщения выполняются лишь на 10% медленнее, чем обычные процедурные вызовы. И поскольку сообщения встречаются в программе гораздо реже других операций, их воздействие на время выполнения влияния практически не оказывает.

Однако существует другой фактор, который влияет на время выполнения: это инкапсуляция данных. Рекомендуется не предоставлять прямой доступ к полям класса, а выполнять каждую операцию над данными через методы. Такая схема приводит к необходимости выполнения процедурного вызова при каждом доступе к данным. Однако, когда инкапсуляция используется только там, где она необходима (т.е. в случаях, где это становится преимуществом), то замедление вполне приемлемое.

2. Неэффективность в смысле распределения памяти. Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информации о типе объекта. Такая информация хранится в дескрипторе типа, и он выделяется один на класс. Каждый объект имеет невидимый указатель на дескриптор типа для своего класса. Таким образом, в объектно-ориентированных программах требуемая дополнительная память выражается в одном указателе для объекта и в одном дескрипторе типа для класса.

3. Излишняя универсальность. Неэффективность может также означать, что программа имеет ненужные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, то они становятся мертвым грузом. Это не воздействует на время выполнения, но влияет на возрастание размера кода.

   Одно из возможных решений - строить базовый класс с минимальным числом методов, а затем уже реализовывать различные расширения этого класса, которые позволят нарастить функциональность.

   Другой подход - дать возможность компоновщику удалять лишние методы. Такие интеллектуальные компоновщики уже доступны для различных языков и операционных систем.

Но нельзя утверждать, что ООП неэффективно. Если классы используются лишь там, где это действительно необходимо, то потеря эффективности из-за повышенного расхода памяти и меньшей производительности незначительна. Кроме того, часто более важной является надежность программного обеспечения и небольшое время его написания, а не производительность.

## 1.8. Заключение

В этой главе Вы узнали об объектно-ориентированном подходе к разработке ПО, а также о том, что послужило предпосылками к его появлению и сделало его популярным. Были рассмотрены ключевые понятия ООП – объект и класс. Далее были описаны основные свойства объектной модели – инкапсуляции, наследование полиморфизм. Основными видами отношений между классами являются наследование, ассоциация, агрегация, метакласс. Также были описаны правила изображения классов и связей между ними на языке UML.

## 1.9. Контрольные вопросы

2-1. Почему объектно-ориентированное программирование пришло на смену процедурному программированию?

a.) Как показала практика, традиционные методы процедурного программирования не способны справиться ни с нарастающей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить эти проблемы. Такой методологией стало объектно-ориентированное программирование (ООП).

2-2. Что такое объект? Приведите примеры.

a.) объектом называют понятие, абстракцию или любой предмет с четко очерченными границами, имеющую смысл в контексте рассматриваемой прикладной проблемы. Введение объектов преследует две цели:

- понимание прикладной задачи (проблемы);
- введение основы для реализации на компьютере.

Примеры объектов: форточка, Банк "Империал", Петр Сидоров, дело № 7461, сберкнижка и т.д.

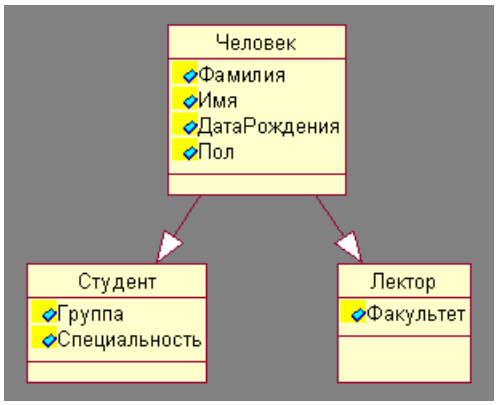
2-3. Что из перечисленного является классами, а что объектами:

1. яблоко;
2. Иван Сидорович Петров;
3. легковой автомобиль;
4. Страховое свидетельство №113-043-429-18.

a.) Яблоко и легковой автомобиль являются классами, а Иван Сидорович Петров и страховое свидетельство №113-043-429-18 - объектами.

Классом всегда является более общее понятие, а объектом – более конкретное. Например, для страхового свидетельства №113-043-429-18 классом будет являться просто страховое свидетельство, а примером объекта для класса «яблоко» может быть «большое красное яблоко».

2-4. Найдите ошибку на приведенной диаграмме:




a.) На диаграмме неверно изображены стрелки обозначающие наследование. Класс «Человек» является более общим по отношению к классам «Студент» и «Лектор» следовательно, он является предком этих классов. Направление стрелок обозначающих наследование должно быть от предка к наследнику.

2-5. Перечислите основные состояния для кофейного автомата.

a.) Примеры состояний кофейного автомата:

1. ожидание монеты;
2. ожидание выбора напитка покупателем;
3. наливает напиток в стакан;
4. ждет, пока покупатель заберет наполненный стакан.

2-6. Что используется для описания поведения объекта? Варианты ответов:

1. поля;
2. методы;
3. наследование.

a.) Ответ: для описания поведения объектов какого-либо класса используются методы.

2-7. Зачем нужен полиморфизм?

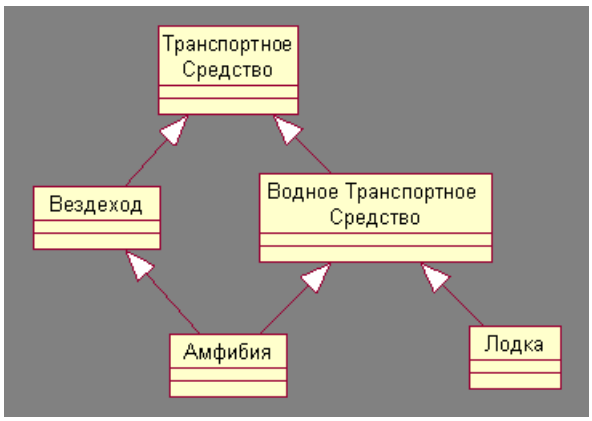
a.) Полиморфизм используется для написания общего кода, который будет одинаково работать с разными типами данных (объектами разных классов), при условии что классы, от которых созданы эти объекты, имеют общего предка.

2-8. Какое отношение существует между человеком и каким-либо внутренним органом этого человека: ассоциация или агрегация? Объясните почему.

а.) Ответ: агрегация. Человеческий организм состоит из органов. Обычно, если к классам применимо выражение «состоит из» или «содержит», то используется агрегация, а не ассоциация.

2-9. Даны объекты: моторная лодка, вездеход, амфибия. Какие классы Вы бы спроектировали для моделирования этих объектов? Постройте дерево наследования этих классов.

а.) Ответ:

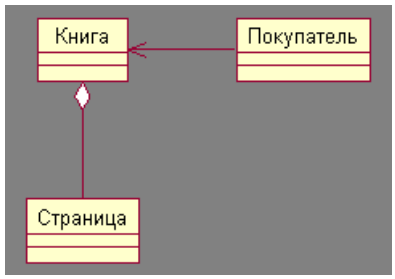



2-10. Даны объекты:

1. Книга «Java2: руководство разработчика»
2. Страница № 342 из книги «Java2: руководство разработчика»
3. Книга «Война и Мир»
4. Вася (покупатель)

Нарисуйте диаграмму классов для данных объектов.

а.)



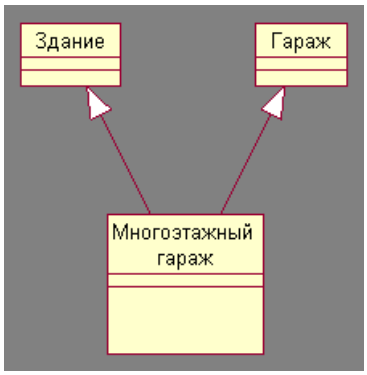

2-11. Даны классы «Здание» и «Гараж», наследования между ними нет. Класс «Здание» обладает такими свойствами, как адрес, количество этажей и т.п. Класс «Гараж» имеет свойства «вместимость» (максимальное количество автомобилей, которые могут быть размещены в нем), «размер» (максимально допустимая длина машины). Необходимо, используя существующие классы, спроектировать 2 новых класса:

1. Многоэтажный гараж (целое здание, предназначенное для парковки автомобилей)
2. Коттедж с гаражом (жилой дом с пристроенным гаражом, или гаражом в цокольном этаже)

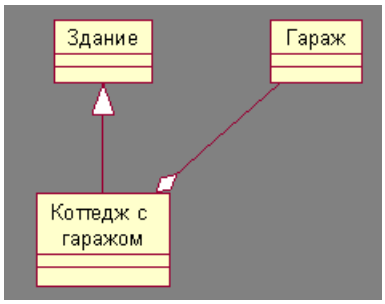
Нарисуйте диаграмму классов и обозначьте на ней отношения между классами.

а.) Ответы:

1.

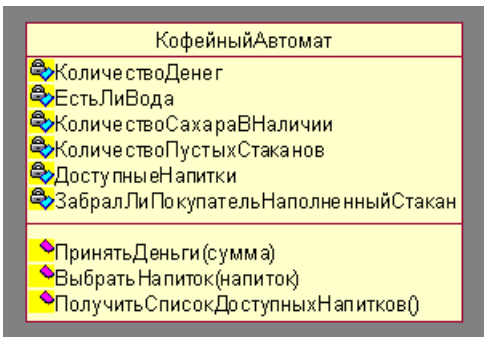



2.



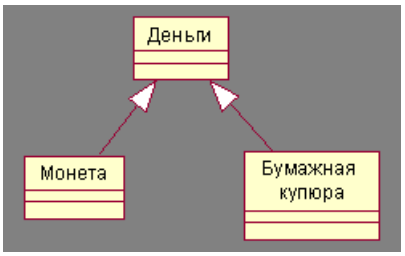

2-12. Спроектируйте класс «Кофейный автомат». Напишите, какие свойства и методы вы бы включили в этот класс.

а.) Один из вариантов класса для кофейного автомата:




2-13. Даны классы «Кофейный автомат», «Монета», «Бумажная купюра». Как можно улучшить модель, чтобы не писать различный код в классе «кофейный автомат» для работы с монетами и с бумажными деньгами, при условии, что монеты и бумажные купюры должны обрабатываться одинаковым образом.

а.) Ответ: создать новый класс «Деньги», наследниками которого сделать классы «Монета» и «Бумажная купюра». В классе «Кофейный автомат» написать код, который будет использовать класс «Деньги». Благодаря полиморфизму написанный код сможет также работать с экземплярами классов «Монета» и «Бумажная купюра».




2-14. Что из перечисленного является в классе «Лифт» деталями реализации, а что – внешним интерфейсом:

- кнопки управления;
- разводка проводов;
- реакция на нажатие какой-либо кнопки;
- количество людей в лифте;
- марка металла, из которого изготовлен трос.

a.) Ответ:

Внешний интерфейс:

- кнопки управления;
- количество людей в лифте;

Детали реализации:

- разводка проводов;
- реакция на нажатие какой-либо кнопки;
- марка металла, из которого изготовлен трос.

# Программирование на Java

## Лекция 3. Лексика языка

### 20 января2003

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Александр Хельвас (ЦОС и ВТ МФТИ) <hel@cos.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Лекция посвящена описанию лексики языка Java.

Лексика описывает, из чего состоит текст программы, каким образом он записывается, и на какие простейшие слова (лексемы) компилятор разбивает программу при анализе. Лексемы (или tokens в английском варианте) - это основные "кирпичики", из которых строится любая программа на языке Java.

Эта тема раскрывает многие детали внутреннего устройства языка, и невозможно написать ни одной строчки кода, не затронув ее. Именно поэтому курс начинается с основ лексического анализа.

# Оглавление

# Лекция 3. Лексика языка

## Содержание лекции.



## 1. Лексика языка

Лексика описывает, из чего состоит текст программы, каким образом он записывается, и на какие простейшие слова (лексемы) компилятор разбивает программу при анализе. Лексемы (или tokens в английском варианте) - это основные "кирпичики", из которых строится любая программа на языке Java.

Эта тема раскрывает многие детали внутреннего устройства языка, и невозможно написать ни одной строчки кода, не затронув ее. Именно поэтому курс начинается с основ лексического анализа.

## 1.1. Кодировка

Технология Java, как платформа, изначально спроектированная для Глобальной сети Интернет, должна быть многоязыковой, а значит обычный набор символов ASCII (American Standard Code for Information Interchange, Американский стандартный код обмена информацией), включающий в себя лишь латинский алфавит, цифры и простейшие специальные знаки (скобки, знаки препинания, арифметические операции и др.) не достаточен. Поэтому для записи текста программы применяется более универсальная кодировка Unicode.

Как известно, Unicode представляет символы кодом из 2 байт, описывая, таким образом, 65.535 символов. Это позволяет поддерживать практически все распространенные языки мира. Первые 128 символов совпадают с набором ASCII. Однако, понятно, что требуется некоторое специальное обозначение, чтобы иметь возможность задавать в программе любой символ Unicode, ведь никакая клавиатура не позволяет вводить более 65 тысяч различных знаков. Эта конструкция представляет символ Unicode, используя только символы ASCII. Например, если в программу нужно вставить знак с кодом 6917, необходимо его представить в шестнадцатеричном формате (1B05) и записать:

```
\u1B05
```

причем буква u должна быть прописной, а шестнадцатеричные цифры A, B, C, D, E, F можно использовать произвольно, как заглавные, так и строчные. Таким образом можно закодировать все символы Unicode от \u0000 до \uFFFF. Буквы русского алфавита начинаются с \u0410 (только буква Ё имеет код \u0401) по \u044F (код буквы ё \u0451). В последних версиях JDK в состав демонстрационных приложений и апплетов входит небольшая программа SymbolTest, позволяющая просматривать весь набор символов Unicode. Ее аналог несложно написать самостоятельно. Для перекодирования больших текстов служит утилита native2ascii, также входящая в JDK. Она может работать как в прямом режиме - переводить из разнообразных кодировок в Unicode, записанный ASCII-символами, так и в обратном (опция -reverse) - из Unicode в стандартную кодировку операционной системы.

В версиях языка Java до 1.1 применялся Unicode версии 1.1.5, в последнем выпуске 1.4 используется 3.0. Таким образом, Java следит за развитием стандарта и базируется на современных версиях. Для любой JDK точную версию Unicode, используемую в ней, можно узнать из документации к классу Character. Официальный веб-сайт стандарта, где можно получить дополнительную информацию - http://www.unicode.org/.

Итак, используя простейшую кодировку ASCII, можно ввести произвольную последовательность Unicode символов. Далее будет описано, что Unicode используется не для всех лексем, а только для тех, для которых важна поддержка многих языков, а именно: комментарии, идентификаторы, символьные и строковые литералы. Для записи остальных лексем вполне достаточно только ASCII символов.

## 1.2. Анализ программы

Компилятор, анализируя программу, сразу разделяет ее на

- пробелы (white spaces);
- комментарии (comments);

- основные лексемы (tokens).

### 1.2.1. Пробелы

Пробелами в данном случае называют все символы, разбивающие текст программы на лексемы. Это как сам символ пробела (space, \u0020, десятичный код 32), так и знаки табуляции и перевода строки. Они используются для разделения лексем, а также для оформления кода, чтобы его было легче читать. Например, следующую часть программы (вычисление корней квадратного уравнения):

```
 double a = 1, b = 1, c = 6;
double D = b * b - 4 * a * c;

if (D >= 0) {
double x1 = (-b + Math.sqrt (D)) / (2 * a);
double x2 = (-b - Math.sqrt (D)) / (2 * a);
}
```

можно записать и в таком виде:

```
 double a=1,b=1,c=6;double D=b*b-4*a*c;if(D>=0){double
x1=(-b+Math.sqrt(D))/(2*a);double x2=(-b-Math.sqrt(D))/(2*a);}
```

В обоих случаях компилятор сгенерирует абсолютно одинаковый код. Единственное соображение, которым должен руководствоваться разработчик - легкость чтения и дальнейшей поддержки такого кода.

Для разбиения текста на строки в ASCII используются два символа - "возврат каретки" (carriage return, CR, \u000d, десятичный код 13) и символ новой строки (linefeed, LF, \u000a, десятичный код 10). Чтобы не зависеть от особенностей используемой платформы, в Java применяется наиболее гибкий подход. Завершением строки считается

- ASCII-символ LF, символ новой строки;
- ASCII-символ CR, "возврат каретки";
- символ CR, за которым сразу же следует символ LF.

Разбиение на строки важно для корректного разбиения на лексемы (как уже говорилось, завершение строки также служит разделителем между лексемами), для правильной работы со строковыми комментариями (см. следующую тему "Комментарии"), а также для вывода отладочной информации (при выводе ошибок компиляции и времени исполнения указывается, на какой строке исходного кода они возникли).

Итак, пробелами в Java считаются:

- ASCII-символ SP, space, пробел, \u0020, десятичный код 32;
- ASCII-символ HT, horizontal tab, символ горизонтальной табуляции, \u0009, десятичный код 9;
- ASCII-символ FF, form feed, символ перевода страницы (был введен для работы с принтером), \u000c, десятичный код 12;
- завершение строки.

### 1.2.2. Комментарии

Комментарии не влияют на результирующий бинарный код и используются только для ввода пояснений к программе.

В Java комментарии бывают 2 видов:

- строчные
- блочные

Строчные комментарии начинаются с ASCII-символов // и длятся до конца текущей строки. Как правило, они используются для пояснения именно этой строки, например:

int y=1970; // год рождения
Блочные комментарии располагаются между ASCII-символами /* и */, могут занимать произвольное количество строк, например:

```
 /*
   Этот цикл не может начинаться с нуля
   из-за особенностей алгоритма
*/
for (int i=1; i<10; i++) {
 ...
}
```

Часто блочные комментарии оформляют следующим образом (каждая строка начинается с *):

```
 /*
 * Описание алгоритма работы
 * следующего цикла while
 */
while (x > 0) {
 ...
}
```

Блочный комментарий не обязательно должен располагаться на нескольких строках, он может даже находится в середине оператора:

```
 float s = 2*Math.PI/*getRadius()*/; // Закомментировано для отладки
```

В этом примере блочный комментарий разбивает арифметические операции. Выражение Math.PI предоставляет значение константы PI, определенное в классе Math. Вызов метода getRadius() теперь закомментирован и не будет произведен, переменная s всегда будет принимать значение 2 PI. Завершает строку строчный комментарий.

Комментарии не могут находиться внутри символьных и строковых литералах, идентификаторах (эти понятия подробно рассматриваются далее в этой лекции). Следующий пример содержит случаи неправильного применения комментариев:

```
 // В этом примере текст /*…*/ станет просто частью строки s
String s = "text/*just text*/";

/*
 * Следующая строка станет причиной ошибки при компиляции,
```

```
 * так как комментарий разбил имя метода getRadius()
 */
circle.get/*comment*/Radius();
```

А такой код допустим:

```
 // Комментарий может разделять вызовы функций:
cirle./*comment*/getRadius();

// Комментарий может заменять пробелы:
int/*comment*/x=1;
```

В последней строке между названием типа данных int и названием переменной x обязательно должен быть пробел или, как в данном примере, комментарий.

Комментарии не могут быть вложенными. Символы /*, */, // не имеют никакого особенного значения внутри уже открытых комментариев, как строчных, так и блочных. Таким образом, в следующем примере

```
 /* начало комментария /* // /** завершение тут: */
```

описан ровно один блочный комментарий. А в следующем примере (строки кода пронумерованы для удобства)

```
1. /*
2.   comment
3.  /*
4.    more comments
5.  */
6.   finish
7. */
```

компилятор выдаст ошибку. Блочный комментарий начался в строке 1 с комбинации символов /*. Вторая открывающая комбинация /* на строке 3 будет проигнорирована, так как находится уже внутри комментария. Символы */ в строке 5 завершат его, а строка 7 породит ошибку - попытка закрыть комментарий, который не был начат.

Любые комментарии полностью удаляются из программы по время компиляции, поэтому их можно использовать неограниченно, не опасаясь, что это повлияет на бинарный код. Основное их предназначение - сделать программу простой для понимания, в том числе и для других разработчиков, которым придется в ней разбираться по какой-либо причине. Также комментарии зачастую используются для временного исключения частей кода, например

```
 int x = 2;
int y = 0;
/*
if (x > 0)
 y = y + x*2;
else
 y = -y - x*4;
*/
```

```
y = y*y;// + 2*x;
```

В этом примере закомментировано выражение if-else и оператор сложения +2*x.

Как уже говорилось выше, комментарии можно писать символами Unicode, то есть на любом языке, удобном разработчику.

Кроме этого, существует особый вид блочного комментария - комментарий разработчика. Он применяется для автоматического создания документации кода. В стандартную поставку JDK, начиная с версии 1.0, входит специальная утилита javadoc. На вход ей подается исходный код классов, а на выходе получается удобная документация в HTML формате, которая описывает все классы, все их поля и методы. При этом активно используются гиперссылки, что существенно упрощает изучение программы по ее такому описанию (например, читая описание метода, можно одним нажатием мыши перейти на описание типов, используемых в качестве аргументов или возвращаемого значения). Однако понятно, что одного названия метода и перечисления его аргументов не достаточно для понимания его работы. Необходимы дополнительные пояснения от разработчика.

Комментарий разработчика записывается так же, как и блочный. Единственное различие в начальной комбинации символов - для документации комментарий необходимо начинать с /**. Например:

```
 /**
 * Вычисление модуля целого числа.
 * Этот метод возвращает
 * абсолютное значение аргумента x.
 */
int getAbs(int x) {
 if (x>=0)
  return x;
 else
  return -x;
}
```

Первое предложение должно содержать краткое резюме всего комментария. В дальнейшем оно будет использовано как пояснение этой функции в списке всех методов класса (ниже будут описаны все конструкции языка, для которых применяется комментарий разработчика).

Поскольку в результате создается HTML-документация, то и комментарий необходимо писать по правилам HTML. Допускается применение тегов, таких как <b> и <p> . Однако, теги заголовков с <h1> по <h6> и <hr> использовать нельзя, так как они активно применяются javadoc для создания структуры документации.

Символ * в начале каждой строки и предшествующие ему пробелы и знаки табуляции игнорируются. Их можно вообще не использовать, но они удобны, когда важно форматирование, например, в примерах кода.

```
 /**
 * Первое предложение - краткое описание метода.
 * <p>
 * Так оформляется пример кода:
 * <blockquote>
```

```
 * <pre>
 * if (condition==true) {
 *    x = getWidht();
 *    y = x.getHeight();
 * }
 * </pre></blockquote>
 * А так описывается HTML-список:
 * <ul>
 * <li>Можно использовать наклонный шрифт <i>курсив</i>,
 * <li>или жирный <b>жирный</b>.
 * </ul>
 */
public void calculate (int x, int y) {
 ...
}
```

Из этого комментария будет сгенерирован HTML-код, выглядящий примерно так:

```
 Первое предложение - краткое описание метода.

Так оформляется пример кода:

if (condition==true) {
   x = getWidht();
   y = x.getHeight();
}

А так описывается HTML-список:
· Можно использовать наклонный шрифт курсив,
· или жирный жирный.
```

Наконец, javadoc поддерживает специальные теги. Они начинаются с символа @. Подробное описание этих тегов можно найти в документации. Например, можно использовать тег @see, чтобы сослаться на другой класс, поле или метод, или даже на другой интернет-сайт.

```
 /**
 * Краткое описание.
 *
 * Развернутый комментарий.
 *
 * @see java.lang.String
 * @see java.lang.Math#PI
 * @see <a href="java.sun.com">Official Java site</a>
 */
```

Первая ссылка указывает на класс String (java.lang - название библиотеки, в которой находится этот класс), вторая - на поле PI класса Math (символ # разделяет название класса и его полей или методов), третья ссылается на официальный сайт Java.

Комментарии разработчика могут быть записаны перед объявлением классов, интерфейсов, полей, методов и конструкторов. Если записать комментарий /** … */ в другой части кода,

то ошибки не будет, но он не попадет в документацию, генерируемую javadoc. Кроме этого, можно описать и пакет (так называются библиотеки, или модули, в Java). Для этого необходимо создать специальный файл package.html, сохранить в нем комментарий, и поместить его в директории пакета. HTML-текст, содержащийся между тегами <body> и </body> , будет перемещен в документацию, а первое предложение будет использовано для краткой характеристики этого пакета.

Все классы стандартных библиотек Java поставляются в виде исходного текста, и можно увидеть, как хорошо они комментированы. Стандартная документация по этим классам сгенерирована утилитой javadoc. Для любой программы можно также легко подготовить подобное описание, необходимы лишь грамотные и аккуратные комментарии в исходном коде. Кроме того, Java предоставляет возможность генерировать с помощью javadoc документацию с нестандартным внешним видом.

### 1.2.3. Лексемы

Итак, были рассмотрены пробелы (в широком смысле этого слова, т.е. все символы, отвечающие за форматирование текста программы) и комментарии, применяемые для ввода пояснений к коду. С точки зрения программиста они применяются для того, чтобы сделать программу более читаемой и понятной для дальнейшего развития.

С точки зрения компилятора, а точнее его части, отвечающей за лексический разбор, основная роль пробелов и комментариев - служить разделителями между лексемами, причем сами разделители далее отбрасываются и не влияют на компилированный код. Например, все следующие примеры объявления переменной эквивалентны:

```
 // Используем пробел в качестве разделителя.
int x = 3 ;

// здесь разделителем является перевод строки
int
x
=
3
;

// здесь разделяем знаком табуляции
int x = 3 ;

/*
 * Единственный принципиально необходимый разделитель между
 * названием типа данных int и именем переменной x
 * здесь описан комментарием блочного типа.
 */
int/**/x=3;
```

Конечно, лексемы очень разнообразны, и именно они определяют многие свойства языка. Рассмотрим все их виды более подробно.

## 1.3. Виды лексем

Ниже перечислены все виды лексем в Java:

- идентификаторы (identifiers);
- ключевые слова (key words);
- литералы (literals);
- разделители (separators);
- операторы (operators).

Рассмотрим их по отдельности.

### 1.3.1. Идентификаторы

Идентификаторы - это имена, которые даются различным элементам языка для упрощения доступа к ним. Имена имеют пакеты, классы, интерфейсы, поля, методы, аргументы и локальные переменные (все эти понятия подробно рассматриваются в дальнейших главах). Идентификаторы можно записывать символами Unicode, то есть, на любом удобном языке. Длина имени не ограничена.

Идентификатор состоит из букв и цифр. Имя не может начинаться с цифры. Java-буквы, используемые в идентификаторах, включают в себя ASCII-символы A-Z (\u0041-\u005a), a-z (\u0061-\u007a), а также знаки подчеркивания _ (ASCII underscore, \u005f) и доллара $ (\u0024). Знак доллара используется только при автоматической генерации кода (чтобы исключить случайное совпадение имен), либо при использовании каких-либо старых библиотек, в которых допускались имена с этим символом. Java-цифры включают в себя обычные ASCII-цифры 0-9 (\u0030-\u0039).

Для идентификаторов не допускаются совпадения с зарезервированными словами (это ключевые слова, булевские литералы true и false и null-литерал null). Конечно, если 2 идентификатора включают в себя разные буквы, которые одинаково выглядят (например, латинская и русская буквы A), то они считаются различными.

В этой главе уже применялись следующие идентификаторы:

```
Character, a, b, c, D, x1, x2, Math, sqrt, x, y, i, s, PI, getRadius, circle,
getAbs, calculate, condition, getWidth, getHeight, java, lang, String


Компьютер,  COLOR_RED, _, aVeryLongNameOfTheMethod
```

### 1.3.2. Ключевые слова

```
 abstract     default     if              private       this
boolean      do          implements    protected     throw
break        double     import         public        throws
byte    else        instanceof     return        transient
case         extends   int             short         try
catch        final       interface      static        void
char    finally     long           strictfp      volatile
class        float       native         super         while
```

```
const       for         new           switch
continue    goto        package       synchronized
```

Ключевые слова goto и const зарезервированы, но не используются. Это сделано для того, чтобы компилятор мог правильно отреагировать на использование этих распространенных в других языках слов. Напротив, оба булевских литерала true, false и null-литерал null часто считают ключевыми словами (возможно потому, что многие средства разработки подсвечивают их таким же образом), однако это именно литералы.

Значение всех ключевых слов будет рассматриваться в последующих главах.

### 1.3.3. Литералы

Литералы позволяют задать в программе значения для числовых, символьных и строковых выражений, а также null-литералов. Всего в Java определены следующие виды литералов:

- целочисленный (integer);
- дробный (floating-point);
- булевский (boolean);
- символьный (character);
- строковый (string);
- null-литерал (null-literal).

Рассмотрим их по отдельности.

#### 1.3.3.1. Целочисленные литералы

Целочисленные литералы позволяют задавать целочисленные значения в десятеричном, восьмеричном и шестнадцатеричном виде. Десятеричный формат традиционен и ничем не отличается от правил, принятых в других языках. Значения в восьмеричном виде начинаются с нуля, и, конечно, использование цифр 8 и 9 запрещено. Запись шестнадцатеричных чисел начинаются с 0x или 0X (цифра 0 и латинская ASCII-буква X в произвольном регистре). Таким образом, ноль можно записать тремя различными способами:

```
 0
00
0x0
```

Как обычно, для записи цифр 10-15 в шестнадцатеричном формате используются буквы A, B, C, D, E, F, заглавные или прописные. Примеры таких литералов:

```
 0xaBcDeF, 0xCafe, 0xDEC
```

Типы данных рассматриваются ниже, однако здесь необходимо упомянуть два целочисленных типа int и long длиной 4 и 8 байт соответственно (или 32 и 64 бита соответственно). Оба эти типа знаковые, т.е. тип int хранит значения от -231 до 231-1, или от -2.147.483.648 до 2.147.483.647. По умолчанию целочисленный литерал имеет тип int, а значит, в программе допустимо использовать литералы только от 0 до 2147483648, иначе возникнет ошибка компиляции. При этом литерал 2147483648 можно использовать только как аргумент унарного оператора -:

```
 int x = -2147483648; \\верно
int y = 5-2147483648; \\ здесь возникнет ошибка компиляции
```

Соответственно, допустимые литералы в восьмеричной записи должны быть от 00 до 017777777777 (=231-1), с унарным оператором - допустимо также -020000000000 (= -231). Аналогично для шестнадцатеричного формата - от 0x0 до 0x7fffffff (=231-1), а также - 0x80000000 (= -231).

Тип long имеет длину 64 бита, а значит, позволяет хранить значения от -263 до 263-1. Чтобы ввести такой литерал, необходимо в конце поставить латинскую букву L или l, тогда все значение будет трактоваться как long. Аналогично можно выписать максимальные допустимые значения для них:

```
 9223372036854775807L
077777777777777777777L
0x7fffffffffffffL
// наибольшие отрицательные значения:
-9223372036854775808L
-01000000000000000000000L
-0x8000000000000000L
```

Другие примеры целочисленных литералов типа long:

```
 0L, 123l, 0xC0B0L
```

### 1.3.3.2. Дробные литералы

Дробные литералы представляют собой числа с плавающей десятичной точкой. Правила записи таких чисел такие же, как и в большинстве современных языков программирования.

Примеры:

```
 3.14
2.
.5
7e10
3.1E-20
```

Таким образом, дробный литерал состоит из следующих составных частей:

- целая часть;
- десятичная точка (используется ASCII-символ точка);
- дробная часть;
- показатель степени (состоит из латинской ASCII-буквы E в произвольном регистре и целого числа с опциональным знаком + или -);
- окончание-указатель типа.

Целая и дробная части записываются десятичными цифрами, а указатель типа (аналог указателя L или l для целочисленных литералов типа long) имеет 2 возможных значения

- латинская ASCII-буква D (для типа double) или F (для типа float) в произвольном регистре. Они вскоре будут подробно рассмотрены ниже.

Необходимыми частями являются:

- хотя бы одна цифра в целой или дробной части;
- десятичная точка или показатель степени или указатель типа.

Все остальные части необязательные. Таким образом, "минимальные" дробные литералы могут быть записаны, например, так:

```
 1.
.1
1e1
1f
```

В Java есть два дробных типа, упомянутые выше,- float и double. Их длина - 4 и 8 байт или 32 и 64 бита соответственно. Дробный литерал имеет тип float, если он заканчивается на латинскую букву F в произвольном регистре. В противном случае он рассматривается как значение типа double и может включать в себя окончание D или d, как признак типа double (это может служить только для наглядности).

```
 // float-литералы:
1f, 3.14F, 0f, 1e+5F

// double-литералы:
0., 3.14d, 1e-4, 31.34E45D
```

В Java дробные числа 32-битного типа float и 64-битного типа double хранятся в памяти в бинарном виде в формате, стандартизированном спецификацией IEEE 754 (полное название - IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE, New York)). В этой спецификации описаны не только конечные дробные величины, но кроме того еще несколько особых значений, а именно:

- положительная и отрицательная бесконечности (positive/negative infinity);
- значение "не число", Not-a-Number, или сокращенно NaN;
- положительный и отрицательный нули.

Для этих значений нет специальных обозначений. Чтобы получить такие величины необходимо либо произвести арифметическую операцию (например, результатом деления ноль на ноль 0.0/0.0 является NaN), либо обратиться к константам в классах Float и Double, а именно POSITIVE_INFINITY, NEGATIVE_INFINITY и NaN. Более подробно работа с этими особенными значениями рассматривается в следующей главе.

Типы данных накладывают ограничения на возможные значения литералов, как и для целочисленных типов. Максимальное положительное конечное значение дробного литерала:

- для float: 3.40282347e+38f
- для double: 1.79769313486231570e+308

Кроме этого, для дробных величин становится важным еще одно предельное значение - минимальное положительное ненулевое значение:

- для float:1.40239846e-45f
- для double: 4.94065645841246544e-324

Попытка указать литерал с слишком большим абсолютным значением (например, 1e40F) приведет к ошибке компиляции. Такая величина должна представляться бесконечностью. Аналогично, указание литерала со слишком малым ненулевым значением (например, 1e-350) также приводит к ошибке. Это значение должно быть округлено до нуля. Однако если округление приводит не к нулю, то компилятор произведет его сам:

```
 // ошибка, выражение должно быть округлено до 0
0.00000000000000000000000000000000000000000001f
// ошибки нет, компилятор сам округляет до 1
1.00000000000000000000000000000000000000000001f
```

Стандартных возможностей вводить дробные значения не в десятичной системе в Java нет, однако классы Float и Double предоставляют много вспомогательных методов, в том числе и для такой задачи.

#### 1.3.3.3. Логические литералы

truefalse

#### 1.3.3.4. Символьные литералы

```
'a' // латинская буква a
' ' // пробел
'К' // греческая буква каппа
```

Также позволяется специальная запись для описания символа через его код (см. тему "Кодировка"). Примеры:

```
'\u0041' // латинская буква A
'\u0410' // русская буква A
'\u0391' // греческая буква A
```

Символьный литерал должен содержать строго один символ, или специальную последовательность, начинающуюся с \. Для записи специальных символов (неотображаемых и служебных, таких как ", ', \) используются следующие обозначения:

```
\b \u0008 backspace BS - забой
\t \u0009 horizontal tab HT - табуляция
\n \u000a linefeed LF - конец строки
\f \u000c form feed FF - конец страницы
\r \u000d carriage return CR - возврат каретки
\" \u0022 double quote " - двойная кавычка
\' \u0027 single quote ' - одинарная кавычка
\\ \u005c backslash \ - обратная косая черта
```

```
\восьмеричный код \u0000 to \u00ff Код символа в восьмеричном формате.
```

Первая колонка описывает стандартные обозначения специальных символов, используемые в Java-программах. Вторая колонка представляет их в стандартном виде Unicode-символов. Третья колонка содержит английские и русские описания. Использование \ в комбинации с другими символами приведет к ошибке компиляции.

Поддержка ввода символов через восьмеричный код обеспечивается для совместимости с C. Например,

```
'\101' // Эквивалентно '\u0041'
```

Однако таким образом можно задать лишь символы от \u0000 до \u00ff (т.е. с кодом от 0 до 255), поэтому Unicode-последовательности предпочтительней.

Поскольку обработка Unicode-последовательностей (\uhhhh) производится раньше лексического анализа, то следующий пример является ошибкой:

```
'\u000a' // символ конца строки
```

Компилятор сначала преобразует \u000a в символ конца строки, и кавычки окажутся на разных строках кода, что является ошибкой. Необходимо использовать специальную последовательность:

```
'\n' // правильное обозначение конца строки
```

Аналогично и для символа \u000d (возврат каретки) необходимо использовать обозначение \r.

Специальные символы можно использовать в составе как символьных, так и строковых литералов.

### 1.3.3.5. Строковые литералы

Строковые литералы состоят из набора символов и записываются в двойных кавычках. Длина может быть нулевой или сколь угодно большой. Любой символ может быть представлен специальной последовательностью, начинающейся с \ (см. "Символьные литералы").

```
 "" // литерал нулевой длины
"\"" //литерал, состоящий из одного символа "
"Простой текст" //литерал длины 13
```

Строковый литерал нельзя разбивать на несколько строк в коде программы. Если требуется текстовое значение, состоящее из нескольких строк, то необходимо воспользоваться специальными символами \n и/или \r. Если же текст просто слишком длинный, чтобы уместиться на одной строке кода, то можно использовать оператор конкатенации строк +. Примеры строковых литералов:

```
 // выражение-константа, составленное из 2 литералов
"Длинный текст " +
   "с переносом"
```

```
 /*
 * Строковый литерал, содержащий текст из двух строк:
 * Hello, world!
 * Hello!
 */
"Hello, world!\r\nHello!"
```

На строковые литералы распространяются те же правила, что и на символьные, в отношении использования символов новой строки \u000a и \u000d.

Каждый строковый литерал является экземпляром класса String. Это определяет некоторые необычные свойства строковых литералов, которые будут рассмотрены в следующей главе.

#### 1.3.3.6. Null литерал

Null литерал может принимать всего одно значение: null. Это литерал ссылочного типа, причем эта ссылка никуда не ссылается, объект отсутствует. Разумеется, его можно применять к ссылкам любого объектного типа данных. Типы данных подробно рассматриваются в следующей главе.

#### 1.3.3.7. Разделители

```
( ) [ ] { } ; . ,
```

#### 1.3.3.8. Операторы

Операторы используются в различных операциях - арифметических, логических, битовых, операции сравнения, присваивания. Следующие 37 лексем (все состоят только из ASCII-символов) являются операторами языка Java:

```
 =   >   <   !   ~   ?   :
==   <=  >=  !=  &&  ||  ++  --
+   -   *   /   &   |   ^   %   <<  >>  >>>
+=  -=  *=  /=  &=  |=  ^=  %=  <<=  >>=  >>>=
```

Большинство из них вполне очевидны и хорошо известны из других языков программирования, однако некоторые нюансы в работе с операторами в Java все же присутствуют, поэтому в конце главы приводятся краткие комментарии по ним.

#### 1.3.3.9. Заключение

В этой главе были рассмотрены все типы лексем, из которых состоит любая Java-программа. Еще раз напомним, что использование Unicode возможно и необходимо в следующих конструкциях:

- комментарии;
- идентификаторы;
- символьные и строковые литералы.

Остальные же (пробелы, ключевые слова, числовые, булевские и null-литералы, разделители и операторы) легко записываются с применением лишь ASCII-символов. В то же время любой Unicode-символ можно задать в виде специальной последовательности

лишь ASCII-символов (условное обозначение - \uhhhh, где hhhh - код символа в шестнадцатеричном формате).

В заключение для примера приведем простейшую программу (традиционное Hello, world!), а затем классифицируем и подсчитаем используемые лексемы:

```
 public class Demo {

 /**
  * Основной метод, с которого начинается выполнение
  * любой Java программы.
  */
 public static void main (String args[]) {
  System.out.println("Hello, world!");
 }
}
```

Итак, в приведенной программе есть один комментарий разработчика, 7 идентификаторов, 5 ключевых слов, 1 строковый литерал, 13 разделителей и ни одного оператора. Этот текст можно сохранить в файле Demo.java, скомпилировать и запустить (работа с JDK и стандартными утилитами была рассмотрена в первой главе). Результатом работы будет, как очевидно:

```
 Hello, world!
```

## 1.4. Дополнение: Работа с операторами

Рассмотрим некоторые детали использования операторов в Java. Здесь будут описаны подробности, относящиеся к работе самих операторов. В следующей главе детально рассматриваются особенности, возникающие при использовании различных типов данных (например, значение операции 1/2 равно 0, а 1/2. - равно 0.5).

### 1.4.1. Операторы присваивания и сравнения

Во-первых, конечно же, различаются оператор присваивания = и оператор сравнения ==.

```
 x = 1; // присваиваем переменной x значение 1
x == 1 // сравниваем значение переменной x с единицей
```

Оператор присваивания не имеет никакого возвращаемого значения. Оператор сравнения всегда возвращает булевское значение true или false. Поэтому обычная опечатка в языке C, когда эти операторы спутывают:

```
 // пример вызовет ошибку компилятора
if (x=0) { // здесь должен применятся оператор сравнения ==
...
}
```

в Java легко устраняется. Поскольку выражение x=0 не имеет никакого значения (и тем более не воспринимается как всегда истинное), то компилятор сообщает об ошибке (необходимо писать x==0).

Условие "не равно" записывается как !=. Например:

```
 if (x!=0) {
float f = 1./x;
}
```

Сочетание какого-либо оператора с оператором присваивания = (см. нижнюю строку в полном перечне в разделе "Операторы") используется при изменении значения переменной, например, следующие две строки эквивалентны:

```
 x = x + 1;
x += 1;
```

### 1.4.2. Арифметические операции

Наряду с 4 обычными арифметическими операциями +, -, *, /, есть оператор получения остатка от деления %, который может быть применен как к целочисленным аргументам, так и к дробным.

Работа с целочисленными аргументами подчиняется простым правилам. Если делится значение a на значение b, то выражение (a/b)*b+(a%b) должно в точности равняться a. Здесь, конечно, оператор деления целых чисел / всегда возвращает целое число. Например:

```
 9/5 возвращает 1
9/(-5) возвращает -1
(-9)/5 возвращает -1
(-9)/(-5) возвращает 1
```

Остаток может быть положительным, только если делимое было положительным. Соответственно, остаток может быть отрицательным только в случае отрицательного делимого.

```
 9%5 возвращает 4
9%(-5) возвращает 4
(-9)%5 возвращает -4
(-9)%(-5) возвращает -4
```

Попытка получить остаток от деления на 0 приводит к ошибке.

Деление с остатком для дробных чисел может быть произведено по двум различным алгоритмам. Один из них повторяет правила для целых чисел, и именно он представлен оператором %. Если в рассмотренном примере деления 9 на 5 перейти к дробным числам, значение остатка во всех вариантах не изменится (оно будет также дробным, конечно).

```
 9.0%5.0 возвращает 4.0
9.0%(-5.0) возвращает 4.0
(-9.0)%5.0 возвращает -4.0
(-9.0)%(-5.0) возвращает -4.0
```

Однако стандарт IEEE 754 определяет другие правила. Такой способ представлен методом стандартного класса Math.IEEEremainder(double f1, double f2). Результат этого метода - значение, которое равно f1-f2*n, где n - целое число, ближайшее к значению f1/f2, а если 2 целых числа одинаково близки к этому отношению, то выбирается четное. По этому правилу значение остатка будет другим:

```
 Math.IEEEremainder(9.0, 5.0) возвращает -1.0
Math.IEEEremainder(9.0, -5.0) возвращает -1.0
Math.IEEEremainder(-9.0, 5.0) возвращает 1.0
Math.IEEEremainder(-9.0, -5.0) возвращает 1.0
```

Унарные операторы инкрементации ++ и декрементации --, как обычно, можно использовать как справа, так и слева.

```
 int x=1;
int y=++x;
```

В этом примере оператор ++ стоит перед переменной x, что означает, что сначала произойдет инкрементация, а затем значение x будет использовано для инициализации y. В результате после выполнения этих строк значения x и y будут равны 2.

```
 int x=1;
int y=x++;
```

А в этом примере сначала значение x будет использовано для инициализации y, а лишь затем произойдет инкрементация. В результате значение x будет равно 2, а y будет равно 1.

### 1.4.3. Логические операторы

Логические операторы "и" и "или" ( & и |) можно использовать в двух вариантах. Это связано с тем, что, как легко убедиться, для каждого оператора возможны случаи, когда значение первого операнда сразу определяет значение всего логического выражения. Если вторым операндом является значение некоторой функции, то появляется выбор - вызывать ее или нет, причем это решение может сказаться как на скорости, так и на функциональности программы.

Первый вариант операторов (&, |) всегда вычисляет оба операнда, второй же - ( &&, || ) не будет продолжать вычисления, если значение выражения уже очевидно. Например:

```
 int x=1;
(x>0) | calculate(x) // в таком выражении произойдет вызов calculate
(x>0) || calculate(x) // а в этом - нет
```

Логический оператор отрицания "не" записывается как !, и конечно имеет только один вариант использования. Этот оператор меняет булевское значение на противоположное.

```
 int x=1;
x>0 // выражение истинно
!(x>0) // выражение ложно
```

Оператор с условием ? : состоит из трех частей - условия и двух выражений. Сначала вычисляется условие (булевское выражение), и на основании результата значение всего оператора определяется первым выражением в случае получения истины, и вторым - если условие ложно. Например, так можно вычислить модуль числа x:

```
x>0 ? x : -x
```

### 1.4.4. Битовые операции

Прежде чем переходить к битовым операциям, необходимо уточнить каким именно образом целые числа представляются в двоичном виде. Конечно, для неотрицательных величин это практически очевидно:

```
 0 0
1 1
2 10
3 11
4 100
5 101
```

и так далее. Однако как представляются отрицательные числа? Во-первых, вводят понятие знакового бита. Первый бит начинает отвечать за знак, а именно 0 означает положительное число, 1 - отрицательное. Но не следует думать, что остальные биты остаются неизменными. Например, если рассмотреть 8-битовое представление:

```
 -1 10000001 // это НЕВЕРНО!
-2 10000010 // это НЕВЕРНО!
-3 10000011 // это НЕВЕРНО!
```

Такой подход неверен! В частности, мы получаем сразу два представления нуля - 00000000 и 100000000, что совсем нерационально. Правильный алгоритм можно представить себе так. Чтобы получить значение -1, надо из 0 вычесть 1:

```
  00000000
-00000001
---------
 11111111
```

Итак, -1 в двоичном виде представляется как 11111111. Продолжаем применять тот же алгоритм (вычитаем 1):

```
 0 00000000
-1 11111111
-2 11111110
-3 11111101
```

и так далее до значения 10000000, которое представляет собой наибольшее по модулю отрицательное число. Для 8-битового представления наибольшее положительное число 01111111 (=127), а наименьшее отрицательное 10000000 (=-128). Поскольку всего 8 бит

определяет 28=256 значений, причем одно из них отводится для нуля, то становится ясно, почему наибольшие по модулю положительные и отрицательные значения различаются на единицу, а не совпадают.

Как известно, битовые операции "и", "или", "исключающее или" принимают 2 аргумента, и выполняют логическое действие попарно над соответствующими битами аргументов. При этом используются те же обозначения, что и для логических операторов, но, конечно, только в первом (одиночном) варианте. Например, вычислим выражение 5&6 :

```
  00000101 // число 5 в двоичном виде
&00000110 // число 6 в двоичном виде
---------
 00000100 // проделали операцию "и" попарно над битами в каждой позиции
```

То есть, выражение 5&6 равно 4.

Исключение составляет лишь оператор "не" или "NOT", который для побитовых операций записывается как ~ (для логических было !). Этот оператор меняет каждый бит в числе на противоположный. Например, ~(-1)=0. Можно легко установить общее правило для получения битового представления отрицательных чисел:

Если n - целое положительное число, то -n в битом представлении равняется ~(n-1).

Наконец, осталось рассмотреть лишь операторы побитового сдвига. В Java есть один оператор сдвига влево и два варианта сдвига вправо. Такое различие связано с наличием знакового бита.

При сдвиге влево оператором << все биты числа смещаются на указанное количество позиций влево, причем освободившиеся справа позиции заполняются нулями. Эта операция аналогична умножению на 2n, и действует вполне предсказуемо как при положительных, так и при отрицательных аргументах.

Рассмотрим примеры применения операторов сдвига для значений типа int, т.е. 32-битных чисел. Пусть положительным аргументом будет число 20, а отрицательным -21.

```
 // Сдвиг влево для положительного числа 20
20 << 00 = 00000000000000000000000000010100 = 20
20 << 01 = 00000000000000000000000000101000 = 40
20 << 02 = 00000000000000000000000001010000 = 80
20 << 03 = 00000000000000000000000010100000 = 160
20 << 04 = 00000000000000000000000101000000 = 320
...
20 << 24 = 00010100000000000000000000000000 = 335544320
20 << 25 = 00101000000000000000000000000000 = 671088640
20 << 26 = 01010000000000000000000000000000 = 1342177280
20 << 27 = 10100000000000000000000000000000 = -1610612736
20 << 28 = 01000000000000000000000000000000 = 1073741824
20 << 29 = 10000000000000000000000000000000 = -2147483648
20 << 30 = 00000000000000000000000000000000 = 0
20 << 31 = 00000000000000000000000000000000 = 0
```

```
 // Сдвиг влево для отрицательного числа -21
-21 << 00 = 11111111111111111111111111101011 = -21
-21 << 01 = 11111111111111111111111111010110 = -42
-21 << 02 = 11111111111111111111111110101100 = -84
-21 << 03 = 11111111111111111111111101011000 = -168
-21 << 04 = 11111111111111111111111010110000 = -336
-21 << 05 = 11111111111111111111110101100000 = -672
...
-21 << 24 = 11101011000000000000000000000000 = -352321536
-21 << 25 = 11010110000000000000000000000000 = -704643072
-21 << 26 = 10101100000000000000000000000000 = -1409286144
-21 << 27 = 01011000000000000000000000000000 = 1476395008
-21 << 28 = 10110000000000000000000000000000 = -1342177280
-21 << 29 = 01100000000000000000000000000000 = 1610612736
-21 << 30 = 11000000000000000000000000000000 = -1073741824
-21 << 31 = 10000000000000000000000000000000 = -2147483648
```

Как видно из примера, неожиданности возникают тогда, когда значащие биты начинают занимать первую позицию и влиять на знак результата.

При сдвиге вправо все биты аргумента смещаются на указанное количество позиций, соответственно, вправо. Однако встает вопрос - каким значением заполнять освобождающиеся позиции слева, в том числе и отвечающую за знак. Есть два варианта. Оператор >> использует для заполнения этих позиций значение знакового бита, то есть результат всегда имеет тот же знак, что и начальное значение. Второй оператор >>> заполняет их нулями, то есть результат всегда положительный.

```
 // Сдвиг вправо для положительного числа 20

// Оператор >>
20 >> 00 = 00000000000000000000000000010100 = 20
20 >> 01 = 00000000000000000000000000001010 = 10
20 >> 02 = 00000000000000000000000000000101 = 5
20 >> 03 = 00000000000000000000000000000010 = 2
20 >> 04 = 00000000000000000000000000000001 = 1
20 >> 05 = 00000000000000000000000000000000 = 0

// Оператор >>>
20 >>> 00 = 00000000000000000000000000010100 = 20
20 >>> 01 = 00000000000000000000000000001010 = 10
20 >>> 02 = 00000000000000000000000000000101 = 5
20 >>> 03 = 00000000000000000000000000000010 = 2
20 >>> 04 = 00000000000000000000000000000001 = 1
20 >>> 05 = 00000000000000000000000000000000 = 0
```

Легко понять, что для положительного аргумента операторы >> и >>> работают совершенно одинаково. Дальнейший сдвиг на большее количество позиций будет также давать нулевой результат.

```
 // Сдвиг вправо для отрицательного числа -21
```

```
// Оператор >>
-21 >> 00 = 11111111111111111111111111101011 = -21
-21 >> 01 = 11111111111111111111111111110101 = -11
-21 >> 02 = 11111111111111111111111111111010 = -6
-21 >> 03 = 11111111111111111111111111111101 = -3
-21 >> 04 = 11111111111111111111111111111110 = -2
-21 >> 05 = 11111111111111111111111111111111 = -1

// Оператор >>>
-21 >>> 00 = 11111111111111111111111111101011 = -21
-21 >>> 01 = 01111111111111111111111111110101 = 2147483637
-21 >>> 02 = 00111111111111111111111111111010 = 1073741818
-21 >>> 03 = 00011111111111111111111111111101 = 536870909
-21 >>> 04 = 00001111111111111111111111111110 = 268435454
-21 >>> 05 = 00000111111111111111111111111111 = 134217727
...
-21 >>> 24 = 00000000000000000000000011111111 = 255
-21 >>> 25 = 00000000000000000000000001111111 = 127
-21 >>> 26 = 00000000000000000000000000111111 = 63
-21 >>> 27 = 00000000000000000000000000011111 = 31
-21 >>> 28 = 00000000000000000000000000001111 = 15
-21 >>> 29 = 00000000000000000000000000000111 = 7
-21 >>> 30 = 00000000000000000000000000000011 = 3
-21 >>> 31 = 00000000000000000000000000000001 = 1
```

Как видно из примеров, эти операции аналогичны делению на 2n. Причем, если для положительных аргументов с ростом n результат закономерно стремится к 0, то для отрицательных предельным значением служит -1.

## 1.5. Заключение

В этой главе были рассмотрены основы лексического анализа программ Java. Для их записи применяется универсальная кодировка Unicode, позволяющая использовать любой язык помимо традиционного английского. Специальная конструкция позволяет задавать любой символ Unicode с помощью лишь ASCII-символов.

Компилятор выделяет из текста программы «пробелы» (были рассмотрены все символы, которые рассматриваются как пробелы) и комментарии, которые полностью удаляются из кода (были рассмотрены все виды комментариев, в частности комментарий разработчика).

Пробелы и все виды комментариев служат для разбиения текста программы на лексемы. Были рассмотрены все виды лексем, в том числе все виды литералов.

В дополнении были рассмотрены особенности применения различных операторов.

## 1.6. Контрольные вопросы

3-1. Как записать в Java-программе символ с кодом 514?

a.) ‘514’

b.) \u0546

c.) \u222

✓d.) \u0222

Ответ a) некорректен, так как внутри одинарных кавычек может стоять ровно один символ, либо специальная последовательность, начинающаяся с обратного слеша. Причем если эта последовательность начинается с \u, то затем должны стоять 4 шестнадцатеричные цифры, поэтому ответ c) также некорректен. В примере b) задается символ с кодом 0x0546=1350.

3-2. Сколько пробелов в следующем примере кода:

```
int x = 3; int y=1;
int z = x+y;
```

a.) 7

b.) 8

✓c.) 9

d.) 11

Правильный ответ 9: 5 пробелов, 1 перенос строки, 3 табуляции.

3-3. Сколько комментариев в следующем примере кода:

```
int x = 0; /* text // text /*
int y=1; // text */ // text */
```

a.) 1 блочный, 0 строчных

✓b.) 1 блочный, 1 строчный

c.) 1 блочный, 2 строчных

d.) 2 блочных, 3 строчных

Правильный ответ b). Блочный комментарий начинается сразу после инициализации поля x, а закрывающая комбинация */ находится лишь во второй строке после первого слова text. Сразу после нее начинается строчный комментарий.

3-4. Что такое комментарий разработчика и для чего он служит?

a.) Комментарий разработчика записывается так же, как и блочный, но начинается с комбинации /**. С помощью утилиты javadoc можно автоматически сгенерировать документацию в формате HTML, описывающую поля, методы, конструкторы, классы и интерфейсы, пакеты. Поскольку текст комментария оказывается в HTML-странице, допускается применение тегов, кроме <hr> и <h1>..<h6>, которые используются самими javadoc. Кроме этого, поддерживается ряд специальных тегов, начинающихся с @, для вставки специальной информации.

3-5. Какие из перечисленных идентификаторов являются корректными?

✓a.) `abc`

b.) `1ab`

✓c.) `_bc`

✓d.) `_1c $ac`

✓e.) `$ac`

✓f.) `for_`

✓g.) `Int`

h.) `byte`

Идентификатор 1ab некорректен, так как начинается с цифры. Идентификатор byte некорректен, так как совпадает с зарезервированным словом byte. Остальные идентификаторы корректны. Использование знака доллара ($ac) рекомендовано только при автоматической генерации кода. Идентификатор Int также корректен, так как отличается от ключевого слова int регистром первой буквы.

3-6. Являются ли следующие слова ключевыми:

a.) `true`

✓b.) `goto`

c.) `null`

✓d.) `const`

e.) `false`

Слова goto и const являются ключевыми, но запрещены для использования. Это сделано в силу того, что эти слова распространены в других языках программирования, но не поддерживаются Java.

Слова true, false, null являются литералами, а не ключевыми словами.

3-7. Равны ли следующие числа:

✓a.) `5 и 05`

b.) `9 и 09`

c.) `10 и 010`

✓d.) `0x5A и 90L`

Числа 5 и 05 равны. Число 09 некорректно, компилятор выдаст ошибку. Число 010 равно 8 и не равно 10. Число 0x5A равно 90, а значит и 90L.

3-8. Какой будет результат следующих действий?

```
1/0 1./0 1/0. 1./0.
'\n'+'\r'
```

a.) Операция 1/0 приведет к арифметической ошибке. Следующие 3 операции будут иметь результат POSITIVE_INFINITY. Последнее сложение равняется 23.

3-9. Чему будет равно следующее выражение и значение переменной x после вычислений?

```
int x=0;
print(++x==x++);
```

a.) Выражение истинно. Левая часть его равна 1, так как переменная x увеличит свое значение на 1 до участия в сравнении. Правая часть также равна 1, так как переменная x еще раз увеличит свое значение уже после сравнения. После вычислений значение переменной равно 2.

3-10. Чему будет равно следующее выражение и значения переменных x и y после вычислений?

```
int x=0, y=0;
print((++x==1) || (y++==1));
```

a.) Выражение истинно, поскольку истинен его первый аргумент. Поскольку применен двойной оператор «или», то второй аргумент вычисляться не будет. Значит после вычислений значение переменной x равно 1, y – 0.

# Программирование на Java

## Лекция 4. Типы данных

### 20 января 2003

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Типы данных определяют основные возможности любого языка. Кроме того, Java является строго типизированным языком, а потому четкое понимание модели типов данных сильно помогает в написании качественных программ. Лекция начинается с введения понятия переменной, на примере которой иллюстрируются особенности применения типов в Java. Описывается разделение всех типов на простейшие и ссылочные, операции над значениями различных типов, а также особый класс Class, который выполняет роль метакласса в Java.

# Оглавление

# Лекция 4. Типы данных

## Содержание лекции.



## 1. Введение

Java является строго типизированным языком. Это означает, что любая переменная и любое выражение имеют известный тип еще на момент компиляции. Такое строгое правило позволяет выявлять многие ошибки уже во время компиляции. Компилятор, найдя ошибку, указывает точную строку и причину ее возникновения, а динамические "баги" (от английского bugs) необходимо сначала выявить тестированием (что может потребовать весьма значительных усилий), а затем найти место в коде, которое их породило. Поэтому четкое понимание модели типов данных в Java сильно помогает в написании качественных программ.

Все типы данных разделяются на две группы. Первую составляют 8 простых или примитивных (от английского primitive) типов данных. Они подразделяются на три подгруппы:

- целочисленные
  - byte
  - short
  - int
  - long
  - char (также является целочисленным типом)
- дробные
  - float

  - double
- булевский
  - boolean

Вторую группу составляют объектные или ссылочные (от английского reference) типы данных. Это все классы, интерфейсы и массивы. В стандартных библиотеках первых версий Java находилось несколько сотен классов и интерфейсов, сейчас их уже тысячи. Кроме стандартных, написаны многие и многие классы и интерфейсы, составляющие любую Java программу.

Иллюстрировать логику работы с типами данных проще всего на примере переменных.

# 2. Переменные

Переменные используются в программе для хранения данных. Любая переменная имеет три базовых характеристики:

- имя;
- тип;
- значение.

Имя уникально идентифицирует переменную и позволяет к ней обращаться в программе. Тип описывает, какие величины может хранить переменная. Значение - текущая величина, хранящаяся в переменной на данный момент.

Работа с переменной всегда начинается с ее объявления (declaration). Конечно, оно должно включать в себя имя объявляемой переменной. Как было сказано, в Java любая переменная имеет строгий тип, который также задается при объявлении и никогда не меняется. Значение может быть указано сразу (это называется инициализацией), а в большинстве случаев задание начальной величины можно и отложить.

Некоторые примеры объявления переменных примитивного типа int с инициализаторами и без таковых:

```
int a;
int b = 0, c = 3+2;
int d = b+c;
int e = a = 5;
```

Из примеров видно, что инициализатором может быть не только константа, но и арифметическое выражение. Иногда это выражение может быть вычислено во время компиляции (такое как 3+2), тогда компилятор сразу записывает результат. Иногда это действие откладывается на момент выполнения программы (например, b+c). В последнем случае нескольким переменным присваивается одно и то же значение, однако объявляется лишь первая из них (в данном примере e), остальные уже должны существовать.

Резюмируем: объявление переменных и возможная инициализация при объявлении описываются следующим образом. Сначала указывается тип переменной, затем ее имя и, если необходимо, инициализатор, который может быть константой или выражением, вычисляемым во время компиляции или исполнении программы. В частности, можно

пользоваться и уже объявленными переменными. Далее можно поставить запятую и объявить новую переменную точно такого же типа.

После объявления переменная может быть использована в различных выражениях, в которых будет браться ее текущее значение. Также в любой момент можно изменить значение, используя оператор присваивания, примерно так же, как это делалось в инициализаторах.

Кроме того, при объявлении переменной может быть использовано ключевое слово final. Его указывают перед типом переменной, и тогда ее необходимо сразу инициализировать и уже больше никогда не менять ее значение. Таким образом, final-переменные становятся чем-то вроде констант, но, на самом деле, некоторые инициализаторы могут вычисляться только во время исполнения программы, генерируя различные значения.

Простейший пример объявления final-переменной:

```
final double pi=3.1415;
```

# 3. Примитивные и ссылочные типы данных

Теперь на примере переменных можно проиллюстрировать различие между примитивными и ссылочными типами данных. Рассмотрим пример, когда объявляются 2 переменные одного типа, приравниваются друг другу, а затем значение одной их них изменяется. Что произойдет со второй переменной?

Возьмем простой тип int:

```
 int a=5; // объявляем первую переменную и инициализируем ее
int b=a; // объявляем вторую переменную, и приравниваем ее к первой
a=3; // меняем значение первой
print(b); // проверяем значение второй
```

Здесь и далее мы считаем, что функция print(...) позволяет нам некоторым (не важно, каким именно) способом узнать значение ее аргумента (как правило, для этого используют функцию из стандартной библиотеки System.out.println(...), которая выводит значение на системную консоль).

В результате мы увидим, что значение переменной b не изменилось, оно осталось равным 5. Это означает, что переменные простого типа хранят непосредственно свои значения, и при приравнивании двух переменных происходит копирование этого значения. Чтобы еще раз подчеркнуть эту особенность приведем еще один пример:

```
 byte b=3;
int a=b;
```

В данном примере происходит преобразование типов (преобразование подробно рассматривается в специальной главе). Для нас сейчас важно констатировать, что переменная b хранит значение 3 типа byte, а переменная a - значение 3 типа int. Это два разных значения, и во второй строке при присваивании произошло копирование.

Теперь рассмотрим ссылочный тип данных. Переменные таких типов всегда хранят ссылки на некоторые объекты. Рассмотрим для примера класс, описывающий точку на

координатной плоскости с целочисленными координатами. Описание класса - это отдельная тема, но в нашем простейшем случае оно тривиально:

```
 class Point {
 int x, y;
}
```

Теперь составим пример, аналогичный приведенному выше для int-переменных, считая, что выражение new Point(3,5) создает новый объект-точку с координатами (3,5).

```
 Point p1 = new Point(3,5);
Point p2=p1;
p1.x=7;
print(p2.x);
```

В третьей строке мы изменили горизонтальную координату точки, на которую ссылалась переменная p1, и теперь нас интересует, как это сказалось на точке, на которую ссылается переменная p2. Проведя такой эксперимент, можно убедиться, что в этот раз мы увидим обновленное значение. То есть объектные переменные после приравнивания остаются "связанными" друг с другом, изменения одной сказываются на другой.

Таким образом, примитивные переменные являются действительными хранилищами данных. Каждая переменная имеет значение, не зависящее от остальных. Ссылочные же переменные хранят лишь ссылки на объекты, причем различные переменные могут ссылаться на один и тот же объект, как это было в нашем примере. В этом случае их можно сравнить с наблюдателями, которые с разных позиций смотрят на один и тот же объект и одинаково видят все происходящие с ним изменения. Если же один наблюдатель сменит объект наблюдения, то он перестает видеть и изменения, происходящие с прежним объектом:

```
 Point p1 = new Point(3,5);
Point p2=p1;
p1 = new Point(7,9);
print(p2.x);
```

В этом примере мы получим 3, то есть после третьей строки переменные p1 и p2 ссылаются на различные объекты и поэтому имеют различные значения.

Теперь легко понять смысл литерала null. Такое значение может принять переменная любого ссылочного типа. Это означает, что ее ссылка никуда не указывает, объект отсутствует. Соответственно, любая попытка обратиться к объекту через такую переменную (например, вызвать метод или взять значение поля) приведет к ошибке.

Так же значение null можно передать в качестве любого объектного аргумента при вызове функций (хотя на практике многие методы считают такое значение некорректным).

Память в Java с точки зрения программиста представляется не нулями и единицами или набором байтов, а как некое виртуальное пространство, в котором существуют объекты. И доступ к памяти осуществляется не по физическому адресу или указателю, а лишь через ссылки на объекты. Ссылка возвращается при создании объекта и далее может быть сохранена в переменной, передана в качестве аргумента и т.д. Как уже говорилось,

допускается наличие нескольких ссылок на один объект. Возможна и противоположная ситуация - когда на какой-то объект не существует ни одной ссылки. Такой объект уже не доступен программе и является "мусором", то есть бесполезно занимает аппаратные ресурсы. Для их освобождения не требуется никаких усилий. В состав любой виртуальной машины обязательно входит автоматический сборщик мусора garbage collector - фоновый процесс, который как раз и занимается уничтожением ненужных объектов.

Очень важно помнить, что объектная переменная, в отличие от примитивной, может иметь значение другого типа, не совпадающего с типом переменной. Например, если тип переменной - некий класс, то переменная может ссылаться на объект, порожденный от наследника этого класса. Все случаи подобного несовпадения будут рассмотрены в следующих разделах курса.

Теперь рассмотрим примитивные и ссылочные типы данных более подробно.

## 3.1. Примитивные типы

Как уже говорилось, существует 8 простых типов данных, которые делятся на целочисленные (integer), дробные (floating-point) и булевские (boolean).

## 3.2. Целочисленные типы

Целочисленные типы - это byte, short, int, long, также к ним относят и char. Первые четыре типа имеют длину 1, 2, 4 и 8 байт соответственно, длина char - 2 байта, что непосредственно следует из того, что все символы Java описываются стандартом Unicode. Длины типов приведены только для оценки областей значения. Как уже говорилось, память в Java представляется виртуальной, и вычислить, сколько физических ресурсов займет та или иная переменная так прямолинейно не получится.

4 основных типа являются знаковыми. char добавлен к целочисленным типам данных, так как с точки зрения JVM символ и его код - взаимооднозначные понятия. Конечно, код символа всегда положительный, поэтому char - единственный беззнаковый тип. Инициализировать его можно как символьным, так и целочисленным литералом. Во всем остальном char - полноценный числовой тип данных, который может участвовать, например, в арифметических действиях, операциях сравнения и т.п. Ниже в таблице сведены данные по всем разобранным типам:

| Название типа | Длина (байты) | Область значений |
|---|---|---|
| byte | 1 | -128 .. 127 |
| short | 2 | -32.768 .. 32.767 |
| int | 4 | -2.147.483.648 .. 2.147.483.647 |
| long | 8 | -9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807 (примерно 1019) |
| char | 2 | '\u0000' .. '\uffff', или 0 .. 65.535 |

Обратите внимание, что int вмещает примерно 2 миллиарда, а потому подходит во многих случаях, когда не требуются сверхбольшие числа. Чтобы представить себе размеры типа long, укажем, что именно он используется в Java для отсчета времени. Как и во многих языках, время отсчитывается от 1 января 1970 года в миллисекундах. Так вот, вместимость

long позволяет отсчитывать время на протяжении миллионов веков(!), причем как в будущее, так и в прошлое.

Почему было обращено внимание именно на эти два типа, int и long? Дело в том, что целочисленные литералы имеют тип int по умолчанию или тип long, если стоит буква L или l. Именно поэтому корректным литералом считается только такое число, которое укладывается в 4 или 8 байт соответственно. Иначе компилятор сочтет это ошибкой. Таким образом, следующие литералы являются корректными:

```
 1
-2147483648
2147483648L
0L
11111111111111111L
```

Над целочисленными аргументами можно производить следующие операции:

- операции сравнения (возвращают булевское значение)
  - <, <=, >, >=
  - ==, !=
- числовые операции (возвращают числовое значение)
  - унарные операции + и -
  - арифметические операции +, -, *, /, %
  - операции инкремента и декремента (в префиксной и постфиксной форме): ++ и --
  - операции битового сдвига << , >>, >>>
  - битовые операции ~, & , |, ^
- оператор с условием ? :
- оператор приведения типов
- оператор конкатенации со строкой +

Операторы сравнения вполне очевидны и особого рассмотрения не требуют. Их результат всегда булевского типа (true или false).

Работа числовых операторов также понятна, либо пояснялась в предыдущей главе. Единственное уточнение можно сделать относительно операторов + и -, которые могут быть как бинарными (иметь два операнда), так и унарными (иметь один операнд). Бинарные операнды являются операторами сложения и вычитания соответственно. Унарный оператор + возвращает значение, равное аргументу (+x всегда равно x). Унарный оператор -, примененный к значению x, возвращает результат, равный 0-x. Неожиданный эффект происходит в случае, если аргумент равен наименьшему возможному значению примитивного типа.

```
int x=-2147483648; // Наименьшее возможное значение типа int
int y=-x;
```

Теперь значение переменной y на самом деле равно не 2147483648, поскольку такое число не укладывается в область значений типа int, а в точности равно значению x! Другими словами в этом примере выражение -x==x истинно!

Дело в том, что если при выполнении числовых операций над целыми числами возникает переполнение, и результат не может быть сохранен в данном примитивном типе, то Java не создает никаких ошибок. Вместо этого все старшие биты, которые превышают вместимость типа, просто отбрасываются. Это может привести не только к потере точной абсолютной величины результата, но и даже к искажению его знака, если на месте знакового бита окажется противоположное значение.

```
 int x= 300000;
print(x*x);
```

Результатом такого примера будет:

```
 -194313216
```

Возвращаясь к инвертированию числа -2147483648, становится ясно, что математический результат равен в точности +231 или, в двоичном формате, 10….0 (единица и 31 ноль). Но тип int рассматривает первую единицу как знаковый бит, и результат получается равным -2147483648.

Таким образом, явное выписывание в коде литералов, которые слишком велики для используемых типов, приводит к ошибке компиляции (см. главу "Лексика"). Если же переполнение возникает в результате выполнения операции, "лишние" биты просто отбрасываются.

Подчеркнем, что выражение типа -5 не является целочисленным литералом. На самом деле оно состоит из литерала 5 и оператора -. Напомним, что некоторые литералы (например, 2147483648) могут встречаться только в сочетании с унарным оператором -.

Кроме этого, числовые операции в Java обладают еще одной особенностью. Хотя целочисленные типы обладают длиной в 8, 16, 32 и 64 бита, вычисления проводятся только с 32-х и 64-х битной точностью. А это значит, что перед вычислениями может потребоваться преобразовать тип одного или нескольких операндов.

Если хотя бы один аргумент операции имеет тип long, то все аргументы приводятся к этому типу, и результат операции также будет типа long. Вычисление будет произведено с точностью в 64 бита, а более старшие биты, если таковые появляются в результате, отбрасываются.

Если же аргументов типа long нет, то вычисление производится с точностью в 32 бита, и все аргументы преобразуются в int (это относится к byte, short, char). Результат также имеет тип int. Все биты старше 32-го игнорируются.

Никаких способов узнать, произошло ли переполнение, нет. Расширим рассмотренный пример:

```
 int i=300000;
print(i*i); // умножение с точностью 32 бита
long m=i;
print(m*m); // умножение с точностью 64 бита
```

```
print(1/(m-i)); // попробуем получить разность значений int и long
```

Результатом такого примера будет:

```
-194313216
90000000000
```

затем мы получим ошибку деления на ноль, поскольку переменные i и m хоть и разных типов, но хранят одинаковое математическое значение, и их разность равна нулю. Первое умножение производилось с точностью в 32 бита, более старшие биты были отброшены. Второе - с точностью в 64 бита, ответ не исказился.

Вопрос приведения типов, и в том числе специальный оператор для такого действия, подробно рассматривается в следующих главах. Однако здесь хотелось бы отметить несколько примеров, которые не столь очевидны, и могут создать проблемы при написании программ. Во-первых, подчеркнем, что результатом операции с целочисленными аргументами всегда является целое число. А значит в следующем примере

```
double x = 1/2;
```

переменной x будет присвоено значение 0, а не 0.5, как можно было бы ожидать. Подробно операции с дробными аргументами рассматриваются ниже, но чтобы получить значение 0.5 достаточно написать 1./2 (теперь первый аргумент дробный, и результат не будет округлен).

Как уже упоминалось, время в Java измеряется в миллисекундах. Попробуем вычислить, сколько миллисекунд содержится в неделе и в месяце:

```
print(1000*60*60*24*7); // вычисление для недели
print(1000*60*60*24*30); // вычисление для месяца
```

Необходимо перемножить количество миллисекунд в одной секунде (1000), секунд - в минуте (60), минут - в часе (60), часов - в дне (24), и дней в неделе и месяце (7 и 30 соответственно). Получаем:

```
604800000
-1702967296
```

Очевидно, во втором вычислении произошло переполнение. Достаточно сделать последний аргумент величиной типа long:

```
print(1000*60*60*24*30L); // вычисление для месяца
```

Получаем правильный результат:

```
2592000000
```

Подобные вычисления разумно переводить на 64-битную точность не на последней операции, а заранее, чтобы избежать переполнения.

Понятно, что типы большей длины могут хранить больший спектр значений, а потому Java не позволяет присвоить переменной меньшего типа значение большего типа. Например, такие строки вызовут ошибку компиляции:

```
// пример вызовет ошибку компиляции
int x=1;
byte b=x;
```

Хотя программисту и очевидно, что переменная b должна получить значение 1, что легко укладывается в тип byte, однако компилятор не может вычислять значение переменной x при обработке второй строки, он знает лишь, что ее тип int.

А вот менее очевидный пример:

```
// пример вызовет ошибку компиляции
byte b=1;
byte c=b+1;
```

И здесь компилятор не сможет успешно завершить работу. При операции сложения значение переменной b будет преобразовано в тип int, и таким же будет результат сложения, а значит, его нельзя так просто присвоить переменной типа byte.

Аналогично:

```
// пример вызовет ошибку компиляции
int x=2;
long y=3;
int z=x+y;
```

Здесь результат сложения будет уже типа long. Точно также некорректна такая инициализация:

```
// пример вызовет ошибку компиляции
byte b=5;
byte c=-5;
```

Даже унарный оператор - проводит вычисления с точностью не меньше 32 бит.

Хотя во всех случаях инициализация переменных приводилась только для примера, а предметом рассмотрения были числовые операции, укажем корректный способ преобразовать тип числового значения:

```
byte b=1;
byte c=(byte)-b;
```

Итак, все числовые операторы возвращают результат типа int или long. Однако есть два исключения.

Во-первых, это операторы инкрементации и декрементации. Их действие заключается в прибавлении и или вычитании единицы из значения переменной, после чего результат сохраняется в этой переменной, и значение всей операции равно значению переменной

(до или после изменения в зависимости от того, является оператор префиксным или постфиксным). А значит, и тип значения совпадает с типом переменной. (На самом деле вычисления все равно производятся с точностью минимум 32 бита, однако при присвоении результата переменной его тип понижается.)

```
byte x=5;
byte y1=x++; // на момент начала исполнения x равен 5
byte y2=x--; // на момент начала исполнения x равен 6
byte y3=++x; // на момент начала исполнения x равен 5
byte y4=--x; // на момент начала исполнения x равен 6
print(y1);
print(y2);
print(y3);
print(y4);
```

В результате получаем:

```
5
6
6
5
```

Никаких проблем с присвоением результата операторов ++ и -- переменным типа byte. Завершая рассмотрение этих операторов, приведем еще один пример:

```
byte x=-128;
print(-x);

byte y=127;
print(++y);
```

Результатом будет:

```
128
-128
```

Этот пример иллюстрирует вопросы преобразования типов при вычислениях и случаи переполнения.

Вторым исключением является оператор с условием ? :. Если второй и третий операнды имеют одинаковый тип, то и результат операции будет такого же типа.

```
byte x=2;
byte y=3;
byte z=(x>y) ? x : y; // верно, x и y одинакового типа
byte abs=(x>0) ? x  : -x; // неверно!
```

Последняя строка неверна, так как третий аргумент содержит числовую операцию, стало быть, его тип int, а значит и тип всей операции будет int, и присвоение некорректно. Даже если второй аргумент имеет тип byte, а третий - short, значение будет типа int.

Наконец, рассмотрим оператор конкатенации со строкой. Оператор + может принимать в качестве аргумента строковые величины. Если одним из аргументов является строка, а вторым - целое число, то число будет преобразовано в текст, и строки объединятся.

```
int x=1;
print("x="+x);
```

Результатом будет:

```
x=1
```

Обратите внимание на следующий пример:

```
print(1+2+"text");
print("text"+1+2);
```

Его результатом будет:

```
3text
text12
```

Отдельно рассмотрим работу с типом char. Значения этого типа могут полноценно участвовать в числовых операциях:

```
char c1=10;
char c2='A'; // латинская буква A (\u0041, код 65)
int i=c1+c2-'B';
```

Переменная i получит значение 9.

Рассмотрим следующий пример:

```
char c='A';
print(c);
print(c+1);
print("c="+c);
print('c'+'='+c);
```

Его результатом будет:

```
A
66
c=A
225
```

В первом случае в метод print было передано значение типа char, поэтому отобразился символ. Во втором случае был передан результат сложения, то есть число, и именно число

появилось на экране. Далее при сложении со строкой тип char был преобразован в текст в виде символа. Наконец в последней строке произошло сложение трех чисел: 'c' (код 99), '=' (код 61) и переменной c (т.е. код 'A' - 65).

Для каждого примитивного типа существуют специальные вспомогательные классы-обертки (wrapper classes). Для типов byte, short, int, long, char это Byte, Short, Integer, Long, Character. Эти классы содержат многие полезные методы для работы с целочисленными значениями. Например, преобразование из текста в число. Кроме этого, есть класс Math, который хоть и предназначен в основном для работы с дробными числами, но также предоставляет некоторые возможности и для целых.

В заключение подчеркнем, что единственные операции с целыми числами, при которых Java генерирует ошибки - это деление на ноль (операторы / и %).

# 4. Дробные типы

Дробные типы - это float и double. Их длины - 4 и 8 байт соответственно. Оба типа знаковые. Ниже в таблице сведены их характеристики:

| Название типа | Длина (байты) | Область значений |
|---|---|---|
| float | 4 | 3.40282347e+38f ; 1.40239846e-45f |
| double | 8 | 1.79769313486231570e+308 ; 4.94065645841246544e-324 |

Для целочисленных типов область значений задавалась верхней и нижней границами, весьма близкими по модулю. Для дробных типов добавляется еще одно ограничение - насколько можно приблизиться к нулю, другими словами - каково наименьшее положительное ненулевое значение. Таким образом, нельзя задать литерал заведомо больший, чем позволяет соответствующий тип данных, это приведет к ошибке overflow. И нельзя задать литерал, значение которого по модулю слишком мало для данного типа, компилятор сгенерирует ошибку underflow.

```
// пример вызовет ошибку компиляции
float f = 1e40f; // значение слишком велико, overflow
double d = 1e-350; // значение слишком мало, underflow
```

Напомним, что если в конце литерала стоит буква F или f, то литерал рассматривается как значение типа float. По умолчанию дробный литерал имеет тип double, при желании это можно подчеркнуть буквой D или d.

Над дробными аргументами можно производить следующие операции:

- операции сравнения (возвращают булевское значение)
  - <, <=, >, >=
  - ==, !=
- числовые операции (возвращают числовое значение)
  - унарные операции + и -
  - арифметические операции +, -, *, /, %

  - операции инкремента и декремента (в префиксной и постфиксной форме): ++ и --
- оператор с условием ? :
- оператор приведения типов
- оператор конкатенации со строкой +

Практически все операторы действуют по тем же принципам, что и для целочисленных операторов (оператор деления с остатком % рассматривался в предыдущей главе, а операторы ++ и -- также увеличивают или уменьшают значение переменной на единицу). Уточним лишь, что операторы сравнения корректно работают и в случае сравнения целочисленных значений с дробными. Таким образом, в основном необходимо рассмотреть вопросы переполнения и преобразования типов при вычислениях.

Для дробных вычислений появляется уже два типа переполнения - overflow и underflow. Тем не менее, Java и здесь никак не сообщает о возникновении подобных ситуаций. Нет ни ошибок, ни других способов обнаружить их. Более того, даже деление на ноль не приводит к некорректной ситуации. А значит, дробные вычисления вообще не порождают никаких ошибок.

Такая свобода связана с наличием специальных значений дробного типа. Они определяются спецификацией IEEE 754 и уже перечислялись в разделе "Лексика":

- положительная и отрицательная бесконечности (positive/negative infinity);
- значение "не число", Not-a-Number, или сокращенно NaN;
- положительный и отрицательный нули.

Все эти значения представлены как для типа float, так и для double.

Положительную и отрицательную бесконечности можно получить следующим образом:

```
1f/0f // положительная бесконечность, тип float
-1d/0d // отрицательная бесконечность, тип double
```

Также в классах Float и Double определены константы POSITIVE_INFINITY и NEGATIVE_INFINITY. Как видно из примера, такие величины получаются при делении конечных величин на ноль.

Значение NaN можно получить, например, в результате следующих действий:

```
0.0/0.0 // деление ноль на ноль
(1.0/0.0)*0.0 // умножение бесконечности на ноль
```

Эта величина также представлена константами NaN в классах Float и Double.

Величины положительный и отрицательный ноль записываются очевидным образом:

```
0.0 // дробный литерал со значением положительного нуля
+0.0 // унарная операция +, ее значение - положительный ноль
-0.0 // унарная операция -, ее значение - отрицательный ноль
```

Все дробные значения строго упорядочены. Отрицательная бесконечность меньше любого другого дробного значения, положительная - больше. Значения +0.0 и -0.0 считаются равными, то есть выражение 0.0==-0.0 истинно, а 0.0>-0.0 - ложно. Однако другие операторы различают их, например, выражение 1.0/0.0 дает положительную бесконечность, а 1.0/-0.0 - отрицательную.

Единственное исключение - значение NaN. Если хотя бы один из аргументов операции сравнения равняется NaN, то результат заведомо будет false (для оператора != соответственно всегда true). Таким образом, единственное значение x, при котором выражение x!=x истинно, именно NaN.

Возвращаемся к вопросу возникновения переполнения в числовых операциях. Если получаемое значение слишком велико по модулю (overflow), то результатом будет бесконечность соответствующего знака.

```
print(1e20f*1e20f);
print(-1e200*1e200);
```

В результате получаем:

```
Infinity
-Infinity
```

Если результат, напротив, получается слишком мал (underflow), то он просто округляется до нуля. Также поступают и в случае, когда количество десятичных знаков превышает допустимое количество:

```
print(1e-40f/1e10f); // underflow для float
print(-1e-300/1e100); // underflow для double

float f=1e-6f;
print(f);
f+=0.002f;
print(f);
f+=3;
print(f);
f+=4000;
print(f);
```

Результатом будет:

```
0.0
-0.0

1.0E-6
0.002001
3.002001
4003.002
```

Как видно, в последней строке был утрачен 6-й разряд после десятичной точки.

Другой пример (из спецификации языка Java):

```
 double d = 1e-305 * Math.PI;
print(d);
for (int i = 0; i < 4; i++)
 print(d /= 100000);
```

Результатом будет:

```
3.141592653589793E-305
3.1415926535898E-310
3.141592653E-315
3.142E-320
0.0
```

Таким образом, как и для целочисленных значений, явное выписывание в коде литералов, которые слишком велики (overflow) или слишком малы (underflow) для используемых типов, приводит к ошибке компиляции (см. главу "Лексика"). Если же переполнение возникает в результате выполнения операции, то возвращается одно из специальных значений.

Теперь перейдем к преобразованию типов. Если хотя бы один аргумент имеет тип double, то значения всех аргументов приводятся к этому типу, и результат операции также будет иметь тип double. Вычисление будет произведено с точностью в 64 бита.

Если же аргументов типа double нет, а хотя бы один аргумент имеет тип float, то все аргументы приводятся к float, вычисление производится с точностью в 32 бита, и результат имеет тип float.

Эти утверждения верны и в случае, если один из аргументов целочисленный. Если хотя бы один из аргументов имеет значение NaN, то и результатом операции будет NaN.

Еще раз рассмотрим простой пример:

```
print(1/2);
print(1/2.);
```

Результатом будет:

```
0
0.5
```

Достаточно одного дробного аргумента, чтобы результат операции также имел дробный тип.

Более сложный пример:

```
int x=3;
int y=5;
print (x/y);
```

```
print((double)x/y);
print(1.0*x/y);
```

Результатом будет:

```
0
0.6
0.6
```

В первый раз оба аргумента были целыми, поэтому в результате получился ноль. Однако, поскольку оба операнда представлены переменными, в этом примере нельзя просто поставить десятичную точку, и таким образом перевести вычисления в дробный тип. Необходимо либо преобразовать один из аргументов (второй вывод на экран), либо вставить еще одну фиктивную операцию с дробным аргументом (последняя строка).

Приведения типов подробно рассматриваются в другой главе, однако обратим здесь внимание на несколько моментов.

Во-первых, при приведении дробных значений к целым типам, дробная часть просто отбрасывается. Например, число 3.84 будет преобразовано в целое 3, а -3.84 превратится в -3. Для математического округления необходимо воспользоваться методом класса Math.round(…).

Во-вторых, при приведении значений int к типу float и при приведении значений типа long к типу float и double возможны потери точности, не смотря на то, что эти дробные типы вмещают гораздо большие числа, чем соответствующие целые. Рассмотрим следующий пример:

```
long l=111111111111L;
float f = l;
l = (long) f;
print(l);
```

Результатом будет:

```
111111110656
```

Тип float не смог сохранить все значащие разряды, хотя преобразование от long к float произошло без специального оператора в отличие от обратного перехода.

Для каждого примитивного типа существуют специальные вспомогательные классы-обертки (wrapper classes). Для типов float и double это Float и Double. Эти классы содержат многие полезные методы для работы с дробными значениями. Например, преобразование из текста в число.

Кроме этого, класс Math предоставляет большое количество методов для операций над дробными значениями, например, извлечение квадратного корня, возведение в любую степень, тригонометрические и другие. Также в этом классе определены константы PI и основание натурального логарифма E.

# 5. Булевский тип

Булевский тип представлен всего одним типом boolean, который может хранить всего два возможных значения - true и false. Величины именно этого типа получаются в результате операций сравнения.

Над булевскими аргументами можно производить следующие операции:

- операции сравнения (возвращают булевское значение)
  - ==, !=
- логические операции (возвращают булевское значение)
  - !
  - &, |, ^
  - &&, ||
- оператор с условием ? :
- оператор конкатенации со строкой +

Операторы сравнения && и || обсуждались в предыдущей главе. В операторе с условием ? : первым аргументом может быть только значение типа boolean. Также допускается, чтобы второй и третий аргументы одновременно также имели булевский тип.

Операция конкатенации со строкой превращает булевскую величину в текст "true" или "false" в зависимости от значения.

Только булевские выражения допускаются для управления потоком вычислений, например, в качестве критерия условного перехода if.

Никакое число не может быть интерпретировано как булевское выражение. Если предполагается, что ненулевое значение эквивалентно истине (по правилам языка C), то необходимо записать x!=0. Ссылочные величины можно преобразовывать к boolean выражением ref!=null.

# 6. Ссылочные типы

Итак, выражение ссылочного типа имеет значение либо null, либо ссылку, указывающую на некоторый объект в виртуальной памяти JVM.

## 6.1. Объекты и правила работы с ними

Объект (object) - это экземпляр некоторого класса или экземпляр массива. Массивы будут подробно рассматриваться в соответствующей главе. Класс - это описание объектов одинаковой структуры, и если в программе такой класс используется, то описание присутствует в единственном экземпляре. Объектов этого класса может не быть вовсе, а может быть создано сколь угодно много.

Объекты всегда создаются с использованием ключевого слова new, причем одно слово new порождает строго один объект (или вовсе ни одного, если происходит ошибка). После ключевого слова указывается имя класса, от которого мы собираемся породить объект.

Создание объекта всегда происходит через вызов одного из конструкторов класса (их может быть несколько), поэтому в заключение ставятся скобки, в которых перечислены значения аргументов, передаваемых выбранному конструктору. В примерах выше, когда создавались объекты типа Point, выражение new Point(3,5) означало обращение к конструктору класса Point, у которого есть 2 аргумента типа int. Кстати, обязательное объявление такого конструктора отсутствовало в упрощенном объявлении класса. Объявление классов рассматривается в следующих главах, однако приведем правильное определение Point:

```
class Point {
 int x, y;

 /**
  * Конструктор принимает 2 аргумента,
  * которыми инициализирует поля объекта.
  */
 Point (int newx, int newy){
  x=newx;
  y=newy;
}
}
```

Если конструктор отработал успешно, то выражение new возвращает ссылку на созданный объект. Эту ссылку можно сохранить в переменной, передать в качестве аргумента в какой-либо метод или использовать другим способом. JVM всегда занимается подсчетом хранимых ссылок на каждый объект. Как только обнаруживается, что больше ссылок нет, то такой объект предназначается для уничтожения сборщиком мусора (garbage collector). Восстановить ссылку на такой "потерянный" объект невозможно.

```
Point p=new Point(1,2); // Создали объект, получили на него ссылку
Point p1=p; // теперь есть 2 ссылки на точку (1,2)
p=new Point(3,4); // осталась одна ссылка на точку (1,2)
p1=null;
```

Ссылок на объект-точку (1,2) больше нет, доступ к нему утерян и он вскоре будет уничтожен сборщиком мусора.

Любой объект порождается только с применением ключевого слова new. Единственное исключение - экземпляры класса String. Записывая любой строковый литерал, мы автоматически порождаем объект этого класса. Оператор конкатенации +, результатом которого является строка, также неявно порождает объекты без использования ключевого слова new.

Рассмотрим пример:

```
"abc"+"def"
```

При выполнении этого выражения будет создано 3 объекта класса String. Два объекта порождаются строковыми литералами, и третий будет представлять результат конкатенации.

Операция создания объекта - одна из самых ресурсоемких в Java. Поэтому следует избегать ненужных порождений. Так как при работе со строками их может создаваться довольно много, компилятор, как правило, пытается оптимизировать такие выражения. В рассмотренном примере, поскольку все операнды являются константами времени компиляции, компилятор сам осуществит конкатенацию и вставит в код уже результат, сократив таким образом количество создаваемых объектов до одного.

Кроме этого в версии Java 1.1 была введена технология reflection, которая позволяет обращаться к классам, методам и полям, используя лишь их имя в текстовом виде. С ее помощью также можно создать объект без ключевого слова new, однако эта технология довольно специфична, применяется в редких случаях, а, кроме того, довольно проста, и потому в данном курсе не рассматривается. Все же приведем пример, как выглядит ее использование:

```
Point p = null;
try {
 // в следующей строке, используя лишь текстовое
 // имя класса Point, порождается объект без
 // применения ключевого слова new
 p = (Point)Class.forName("Point").newInstance();

} catch (Exception e) { // обработка ошибок
 System.out.println(e);
}
```

Объект всегда "помнит", от какого класса он был порожден. С другой стороны, как уже указывалось, можно ссылаться на объект, используя ссылку другого типа. Приведем пример, который будем еще много раз использовать. Сначала опишем два класса, Parent и его наследник Child:

```
// Объявляем класс Parent
class Parent {
}


// Объявляем класс Child, и наследуем
// его от класса Parent
class Child extends Parent {
}
```

Пока нам не нужно определять какие-либо поля или методы. Далее объявим переменную одного типа, и проинициализируем ее значением другого типа:

```
Parent p = new Child();
```

Теперь переменная типа Parent указывает на объект, порожденный от класса Child.

Над ссылочными значениями можно производить следующие операции:

- обращение к полям и методам объекта
- оператор instanceof (возвращает булевское значение)
- операции сравнения == и != (возвращают булевское значение)
- оператор приведения типов
- оператор с условием ? :
- оператор конкатенации со строкой +

Обращение к полям и методам объекта можно назвать основной операцией над ссылочными величинами. Осуществляется она с помощью символа . (точка). Примеры применения будут еще многократно приведены в этом курсе.

Используя оператор instanceof, можно узнать, от какого класса произошел объект. Этот оператор имеет два аргумента. Слева указывается ссылка на объект, а справа - имя типа, на совместимость с которым проверяется объект. Например:

```
Parent p = new Child();

// проверяем переменную p типа Parent
// на совместимость с типом Child
print(p instanceof Child);
```

Результатом будет true. Таким образом, оператор instanceof опирается не на тип ссылки, а на свойства объекта, на который она ссылается. Но этот оператор возвращает истинное значение не только для точного того типа, от которого был порожден объект. Добавим к уже объявленным классам еще один:

```
// Объявляем новый класс и наследуем
// его от класса Child
class ChildOfChild extends Child {
}
```

Теперь заведем переменную нового типа:

```
Parent p = new ChildOfChild();
print(p instanceof Child);
```

В первой строке объявляется переменная типа Parent, которая инициализируется ссылкой на объект, порожденный от ChildOfChild. Во второй строке оператор instanceof анализирует совместимость ссылки типа Parent с классом Child, причем задействованный объект не порожден ни от первого, ни от второго класса. Тем не менее, оператор вернет true, поскольку класс, от которого этот объект порожден, наследуется от Child.

Добавим еще один класс:

```
class Child2 extends Parent {
}
```

И снова объявим переменную типа Parent:

```
Parent p=new Child();
print(p instanceof Child);
print(p instanceof Child2);
```

Переменная p имеет тип Parent, а значит, может ссылаться на объекты типа Child или Child2. Оператор instanceof помогает разобраться в ситуации:

```
true
false
```

Для ссылки равной null оператор instanceof всегда вернет значение false.

С изучением свойств объектной модели Java, мы будем возвращаться к алгоритму работы оператора instanceof.

Операторы сравнения = и != проверяют равенство (или неравенство) объектных величин именно по ссылке. Однако часто требуется альтернативное сравнение - по значению. Сравнение по значению имеет дело с понятием состояние объекта. Сам смысл этого выражения рассматривается в ООП, что же качается реализации в Java, то состояние объекта хранится в его полях. При сравнении по ссылке ни тип объекта, ни значения его полей не учитываются, true возвращается только в случае, если обе ссылки указывают на один и тот же объект.

```
Point p1=new Point(2,3);
Point p2=p1;
Point p3=new Point(2,3);
print(p1==p2);
print(p1==p3);
```

Результатом будет:

```
true
false
```

Первое сравнение оказалось истинным, так как переменная p2 ссылается в точности на тот же объект, что и p1. Второе же сравнение ложно, не смотря на то, что переменная p3 ссылается на объект-точку с точно такими де координатами. Однако это другой объект, который был порожден другим выражением new.

Если один из аргументов оператора = равен null, а другой - нет, то значение такого выражения будет false. Если же оба операнда null, то результат будет true.

Для корректного сравнения по значению существует специальный метод equals, который будет рассмотрен позже. Например, строки надо сравнивать следующим образом:

```
String s = "abc";
s=s+1;
print(s.equals("abc1"));
```

Операция с условием ? : работает обычным образом, и может принимать второй и третий аргументы, если они оба одновременно ссылочного типа. Результат такого оператора также будет иметь объектный тип.

Также как и простые типы, ссылочные величины можно складывать со строкой. Если ссылка равна null, то к строке добавляется текст "null". Если же ссылка указывает на объект, то у него вызывается специальный метод (он будет рассмотрен ниже, его имя toString()), и текст, который он вернет, будет добавлен к строке.

## 6.2. Класс Object

В Java отсутствует множественное наследование. Каждый класс может иметь только одного родителя. Таким образом, мы можем проследить цепочку наследования от любого класса, поднимаясь все выше. Существует класс, на котором такая цепочка всегда заканчивается, это класс Object. Именно от него наследуются все классы, в объявлении которых явно не указан другой родительский класс. А значит, любой класс напрямую или через своих родителей является наследником Object. Отсюда следует, что методы этого класса есть у любого объекта (поля в Object отсутствуют), а потому они представляют особенный интерес.

Рассмотрим основные из них:

- getClass()

Этот метод возвращает объект класса Class, который описывает класс, от которого был порожден этот объект. Класс Class будет рассмотрен ниже. У него есть метод getName()

, который возвращает имя класса:

```
String s = "abc";
Class cl=s.getClass();
print(cl.getName());
```

Результатом будет строка:

```
java.lang.String
```

В отличие от оператора instanceof, метод getClass() всегда возвращает точно тот класс, от которого был порожден объект.

- equals()

Этот метод имеет один аргумент типа Object и возвращает boolean. Как уже говорилось, equals() служит для сравнения объектов по значению, а не по ссылке. Сравнивается состояние объекта, у которого вызывается этот метод, с передаваемым аргументом.

```
Point p1=new Point(2,3);
```

```
Point p2=new Point(2,3);
print(p1.equals(p2));
```

Результатом будет true.

Поскольку сам Object не имеет полей, а значит, и состояния, в этом классе метод equals возвращает результат сравнения по ссылке. Однако при написании нового класса можно переопределить этот метод и описать правильный алгоритм сравнения по значению (что и сделано в большинстве стандартных классов). Соответственно в класс Point также необходимо добавить переопределенный метод сравнения:

```
public boolean equals(Object o) {
 // Сначала необходимо проверить, что переданный
 // объект совместим с типом Point
 if (o instanceof Point) {

  // Типы совместимы, можно провести преобразование
  Point p = (Point)o;

  // Возвращаем результат сравнения координат
  return p.x==x && p.y==y;
 }
// Если объект не совместим с Point, возвращаем false
 return false;
}
```

- hashCode()

Этот метод возвращает значение int. Цель hashCode() - представить любой объект целым числом. Особенно эффективно это используется в хэш-таблицах (в Java есть стандартная реализация такого хранения данных, она будет рассмотрена позже). Конечно, нельзя потребовать, чтобы различные объекты возвращали строго различные хэш-коды, но, по крайней мере, требуется, чтобы объекты равные по значению (метод equals() возвращает true) возвращали одинаковые хэш-коды.

В классе Object этот метод реализован на уровне JVM. Сама виртуальная машина генерирует число хеш-код, основываясь на расположении объекта в памяти.

- toString()

Этот метод позволяет получить текстовое описание любого объекта. Создавая новый класс, этот метод можно переопределить и возвращать более подробное описание. Для класса Object и его наследников, не переопределивших toString(), метод возвращает следующее выражение:

```
getClass().getName()+"@"+hashCode()
```

метод getName() класса Class уже приводился в пример, а хэш-код еще дополнительно обрабатывается специальной функцией для представления в шестнадцатеричном формате.

Например:

```
print(new Object());
```

Результатом будет:

```
java.lang.Object@92d342
```

В результате этот метод позволяет по текстовому описанию понять, от какого класса был порожден объект, и, благодаря хеш-коду, различать разные объекты, созданные от одного класса.

Именно этот метод вызывается при конвертации объекта в текст, когда он передается в качестве аргумента оператору конкатенации строк.

- finalize()

Этот метод вызывается при уничтожении объекта автоматическим сборщиком мусора (garbage collector). В классе Object этот метод ничего не делает, однако в классе-наследнике можно описать все действия, необходимые для корректного удаления объекта, такие как закрытие соединений с БД, сетевых соединений, снятие блокировок на файлы и т.д. В обычном режиме напрямую этот метод вызывать не нужно, он отработает автоматически. Если необходимо, можно обратиться к нему явным образом.

В методе finalize() нужно описывать только дополнительные действия, связанные с логикой работы программы. Все необходимое для непосредственно удаления объекта, JVM сделает сама.

## 6.3. Класс String

Как уже указывалось, класс String занимает в Java особое положение. Экземпляры только этого класса можно создавать без использования ключевого слова new. Каждый строковый литерал порождает экземпляр String, и это единственный литерал (кроме null), имеющий объектный тип.

Затем значение любого типа может быть приведено к строке, применяя оператор конкатенации строк, который был рассмотрен для каждого типа, как примитивного, так и объектного.

Еще одним важным свойством этого класса является неизменяемость. Это означает, что, породив объект, содержащий некое значение-строку, мы уже не можем изменить это значение, для этого необходимо создать новый объект.

```
String s="a";
s="b";
```

Во второй строке переменная сменила свое значение, но только создав новый объект класса String.

Поскольку каждый строковый литерал порождает новый объект, что есть очень ресурсоемкая операция в Java, то зачастую компилятор стремится оптимизировать эту работу.

Во-первых, если используются несколько литералов с одинаковым значением, то для всех них будет создан один и тот же объект.

```
String s1 = "abc";
String s2 = "abc";
String s3 = "a"+"bc";
print(s1==s2);
print(s1==s3);
```

Результатом будет:

```
true
true
```

То есть, в случае, когда строка конструируется из констант, известных уже на момент компиляции, оптимизатор также подставляет один и тот же объект.

Если же строка создается выражением, которое может быть вычислено только во время исполнения программы, то оно будет порождать новый объект:

```
String s1="abc";
String s2="ab";
print(s1==(s2+"c"));
```

Результатом будет false, так компилятор не может предсказать результат сложения значения переменной с константой.

В классе String определен метод intern(), который возвращает один и тот же объект-строку, для всех экземпляров, равных по значению. То есть, если для ссылок s1 и s2 верно выражение s1.equals(s2), то верно и s1.intern()==s2.intern().

Разумеется, в классе переопределены методы equals() и hashCode(). Метод toString() также переопределен, и возвращает он сам объект-строку, то есть для любой ссылки s типа String, не равной null, верно выражение s==s.toString().

## 6.4. Класс Class

Наконец, последний класс, который будет рассмотрен в этой главе.

Класс Class является метаклассом для всех классов Java. Когда JVM загружает файл .class, который описывает некоторый тип, в памяти создается объекта класса Class, который будет хранить это описание.

Например, если в программе есть строка:

```
Point p=new Point(1,2);
```

то это означает, что в системе созданы следующие объекты:

1. собственно, объект типа Point, описывающий точку (1,2)
2. объект класса Class, описывающий класс Point

3. объект класса Class, описывающий класс Object. Так как класс Point наследуется от Object, описание этого класса также необходимо.
4. объект класса Class, описывающий класс Class. Это обычный Java-класс, который должен быть загружен по общим правилам.

Одно из применений класса Class уже было рассмотрено - использование метода getClass() класса Object. Если продолжить последний пример с точкой:

```
Class cl=p.getClass(); // это объект №2 из списка
Class cl2=cl.getClass(); // это объект №4 из списка
Class cl3=cl2.getClass(); // опять объект №4
```

Выражение cl2==cl3 верно.

Другое применение класса Class также приводилось в примере применения технологии reflection.

Кроме прямого использования метакласса для хранения в памяти описания классов, Java использует эти объекты и для других целей, которые будут рассмотрены ниже (статические переменные, синхронизация статических методов и т.д.).

# 7. Заключение

Типы данных - одна из ключевых тем курса. Невозможно написать ни одной программы, не используя их. Вот список некоторых мест, где применяются типы:

- объявление типов;
- создание объектов;
- при объявлении полей - тип поля;
- при объявлении методов - входные параметры, возвращаемое значение;
- при объявлении конструкторов - входные параметры;
- оператор приведения типов;
- оператор instanceof;
- объявление локальных переменных;
- многие другие - обработка ошибок, import-выражения и т.д.

Принципиальные различия между примитивными и ссылочными типами данных будет и дальше рассматриваться по ходу курса. Изучение объектной модели Java даст основу для более детального изложения объектных типов - обычных и абстрактных классов, интерфейсов и массивов. После рассмотрения приведения типов будут описаны связи между типом переменной и типом ее значения.

В обсуждении будущей версии Java 1.5 упоминаются темплейты (templates), которые существенно расширят понятия типа данных, если действительно войдут в стандарт языка.

# 8. Заключение

В этой главе было рассказано, что Java является строго типизированным языком, то есть тип всех переменных и выражений определяется уже компилятором. Это позволяет существенно повысить надежность и качество кода, а также делает необходимым хорошее понимание объектной модели программистами.

Все типы в Java делятся на две группы – фиксированные простые, или примитивные, типы (8 типов) и многочисленная группа объектных типов (классов). Примитивные типы действительно являются хранилищами данных своего типа. Ссылочные переменные хранят ссылку на некоторый объект совместимого типа. Также они могут принимать значение null, не указывая ни на какой объект. JVM подсчитывает количество ссылок на каждый объект, и активизирует механизм автоматической сборки мусора для удаления неиспользуемых более объектов.

Были рассмотрены переменные. Они характеризуются тремя основными параметрами – имя, тип и значение. Любая переменная должна быть объявлена и может быть при этом инициализирована. Возможно использование модификатора final.

Примитивные типы состоят из 5 целочисленных, включая символьный тип, 2 дробных и 1 булевского. Целочисленные литералы имеют ограничения, связанные с типами данных. Были рассмотрены все операторы над примитивными типами, тип возвращаемого значения и тонкости их использования.

Затем изучались объекты, способы их создания, и операторы, выполняющие над ними различные действия, в частности принцип работы оператора instanceof. Далее были рассмотрены самые главные классы в Java – Object, Class, String.

В заключение перечислены все места применения типов в программе.

# 9. Контрольные вопросы

4-1. Каков будет результат следующего примера?

```
byte b=3;
int c=b;
c++;
print(++b==c);
```

a.) После выполнения второй строки обе переменные хранят значение 3. После третьей строки значение одной из них увеличится на единицу, а в последней строке значение второй переменной также увеличится, и они снова будут равны. Результат равен true.

4-2. Каков будет результат следующего примера?

```
Point p = new Point(1, 2);
int a=p.x;
p = new Point(3, 4);
print(a);
```

a.) Изменение значения ссылочной переменной никак не скажется на переменной простого типа. Результат равен 1.

4-3. Каков будет результат следующего примера?

```
Point p1 = new Point(3, 4);
Point p2 = p1;
p1.x=5;
p1 = new Point(4, 4);
print(p2.x-p1.x);
```

a.) В третьей строке обе переменные ссылаются на один и тот же объект, именно его координата x приравнивается 5. Далее переменная p1 начинает ссылаться на другой объект, чья координата x равна 4. Поэтому результат равен 5-4=1.

4-4. Какой будет результат следующих действий?

```
1/2 1./2 1/2. 1./2.
```

a.) Результат первого деления будет целым числом, поскольку оба аргумента целые числа, т.е. 0. Результатом следующих трех операций будет дробное число, поскольку как минимум один аргумент дробный, то есть 0.5.

4-5. Перечислите все операторы над числами, обозначение которых включает в себя знак +.

a.) Унарный плюс, числовая операция сложения, инкрементация (префиксная и постфиксная) и конкатенация со строкой. Также оператор присваивания имеет вариант со сложением (+=).

4-6. Приведите пример значения целочисленной переменной x, при которой следующие выражения не верны:

```
a) x*30/30==x
b) x/30*30==x
```

a.) Для выражения a) подходит любое значение большее, чем Integer.MAX_VALUE/30, т.е. 71582789. Для выражения b) подходит любое целое число, не делящееся нацело на 30, например, 15.

4-7. Какой будет результат следующей строки (код символа восклицательного знака 33)?

```
print("Hello"+'!');
```

a.) Результатом будет Hello!

4-8. Какой будет результат следующих действий?

```
double x=2./0;
double y=-1/0.;
print(x+y);
```

a.) Значение переменной x равно POSITIVE_INFINITY, а переменной y NEGATIVE_INFINITY. Их сумма равна NaN.

4-9. Перечислите возможные способы создания объектов в Java.

a.) Объекты всех классов создаются с помощью ключевого слова new, после которого идет имя класса, а затем в круглых скобках перечисляются аргументы.

Объекты класса String создаются для работы со строковыми литералами.

Кроме того, можно породить новый объект, вызвав метод newInstance() класса Class.

4-10. Какой будет результат следующих действий?

```
Point p1 = new Point(2, 3);
Point p2 = new Point(2, 3);
print(p1==p2);
p2=p1;
p1.x=3;
print(p1==p2);
```

a.) При выполнении третьей строки будет выведено значение false. Оператор сравнения проверяет ссылки, а переменные p1 и p2 ссылаются на различные объекты.

В четвертой строке они начинают указывать на один и тот же объект, поэтому последняя строка выведет значение true.

4-11. Эквиваленты ли две следующие операции над ссылочными переменными x1 и x2 (SomeClass2 – тип переменной x2)?

```
x1 instanceof SomeClass2
x1.getClass().getName().equals(x2.getClass().getName())
```

a.) Ответ «нет». Проверка имени класса (вторая операция) проверяет, от одного и того ли класса были созданы объекты, на которое ссылаются переменные x1 и x2. Оператор instanceof проверяет типы на совместимость. Если объекты созданы от одного и того же класса, то и их типы, конечно, совместимы. Однако, если значение переменной x1 – объект, порожденный от класса-наследника SomeClass2, а x2 – объект, порожденный от SomeClass2, то instanceof вернет true, а вторая операция – false.

4-12. При каком условии следующие выражения вернут значение false?

```
a) x.toString() instanceof String
b) (x+"") instanceof String
```

a.) Выражение a) может быть ложным только в случае, когда переменная x ссылается на объект, класс которого переопределил метод toString() так, что последний может вернуть null. Такое переопределение не рекомендуется. Выражение b) всегда вернет true.

4-13. При каком значении ссылочной переменной x следующее выражение всегда будет возвращать истину?

```
x.getClass()==x
```

a.) Это выражение будет истинным, только если x ссылается на объект класса Class, который описывает класс Class. Такую ссылку можно получить из любой объектной переменной y, не равной null:

```
x=y.getClass().getClass()
```

# Программирование на Java

## Лекция 5. Имена. Пакеты

### 20 января 2003

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>




Аннотация

В этой лекции рассматривается две темы – система именования элементов языка в Java и пакеты (packages), которые являются аналогами библиотек из других языков. Почти все конструкции в Java имеют имя для обращения к ним из других частей программы. По ходу изложения вводятся важные понятия, в частности – область видимости имени. При перекрытии таких областей возникает конфликт имен. Для того, чтобы минимизировать риск появления таких ситуаций, описываются соглашения по именованию, предложенные компанией Sun.

Пакеты осуществляют физическую и логическую группировку классов, и становятся необходимыми при создании больших систем. Вводится важное понятие модуля компиляции и описывается его структура.

# Оглавление

# Лекция 5. Имена. Пакеты

## Содержание лекции.



## 1. Введение

Имена (names) используются в программе для доступа к объявленным (declared) ранее "объектам", "элементам", "конструкциям" языка (все эти слова-синонимы были использованы здесь в их общем смысле, а не как термины ООП, например). Конкретнее, в Java имеют имена:

- пакеты;
- классы;
- интерфейсы;
- элементы (member) ссылочных типов:
  - поля;
  - методы;
  - внутренние классы и интерфейсы;
- аргументы:
  - методов;
  - конструкторов;
  - обработчиков ошибок;
- локальные переменные.

Соответственно, все они должны быть объявлены специальным образом, что будет постепенно рассматриваться по ходу курса. Кроме этого, также объявляются конструкторы, однако их имя совпадает с именем класса, поэтому они не попали в этот список.

Напомним, что пакеты (packages) в Java - это способ логически группировать классы, что необходимо, поскольку зачастую количество классов в системе составляет несколько тысяч или даже десятков тысяч. Кроме классов и интерфейсов в пакетах могут находиться вложенные пакеты. Синонимами этого слова в других языках являются библиотека или модуль.

# 2. Имена

## 2.1. Простые и составные имена. Элементы.

Имена бывают простыми (simple), состоящими из одного идентификатора (они определяются во время объявления), и составными (qualified), состоящими из последовательности идентификаторов, разделенных точкой. Для пояснения этих терминов необходимо рассмотреть еще одно понятие.

У пакетов и ссылочных типов (классов, интерфейсов, массивов) есть элементы (members). Доступ к элементам осуществляется с помощью выражения, состоящего из имен, например, пакета и класса, разделенных точкой.

Далее классы и интерфейсы будут называться объединяющим термином тип (type).

Элементами пакета являются классы и интерфейсы, содержащиеся в этом пакете, а также вложенные пакеты. Чтобы получить составное имя пакета, необходимо к полному имени пакета, в котором он располагается, добавить точку, а затем его собственное простое имя. Например, составное имя основного пакета языка Java - java.lang (то есть, простое имя этого пакета lang, и он находится в объемлющем пакете java). Внутри него есть вложенный пакет, предназначенный для типов технологии reflection, которая упоминалась в предыдущих главах. Простое название пакета reflect, а, значит, составное - java.lang.reflect.

Простое имя классов и интерфейсов дается при объявлении, например Object, String, Point. Чтобы получить составное имя таких типов надо к составному имени пакета, в котором находится тип, через точку добавить простое имя типа. Например, java.lang.Object, java.lang.reflect.Method или com.myfirm.MainClass. Смысл последнего выражения таков: сначала идет обращение к пакету com, затем к его элементу - вложенному пакету myfirm, а затем к элементу пакета myfirm - классу MainClass. Здесь com.myfirm - составное имя пакета, где лежит класс MainClass, а MainClass -простое имя этого класса. Составляем их и разделяем точкой - получается полное имя класса com.myfirm.MainClass.

Для ссылочных типов элементами являются поля и методы, а также внутренние типы (классы и интерфейсы). Элементы могут быть как непосредственно объявлены в классе, так и получены по наследству от родительских классов и интерфейсов, если таковые имеются. Простое имя элементов также дается при инициализации. Например, toString(), PI, InnerClass. Составное имя получается путем объединения простого или составного имени типа или переменной объектного типа с именем элемента. Например, ref.toString(), java.lang.Math.PI, OuterClass.InnerClass. Другие обращения к элементам ссылочных типов уже неоднократно применялись в примерах в предыдущих главах.

## 2.2. Имена и идентификаторы

Теперь, когда были рассмотрены простые и составные имена, уточним разницу между идентификатором (напомним, что это вид лексемы) и именем. Понятно, что простое имя

состоит из одного идентификатора, а составное - из нескольких. Однако не всякий идентификатор входит в состав имени.

Во-первых, в выражении объявления (declaration) идентификатор еще не является именем. Другими словами, идентификатор становится именем после первого появления в коде в месте объявления.

Во-вторых, есть возможность обращаться к полям и методам объектного типа не через имя типа или объектной переменной, а через ссылку на объект, полученную в результате выполнения выражения. Один пример такого случая уже приводился в предыдущих главах:

```
country.getCity().getStreet();
```

В данном примере getStreet является не именем, а идентификатором, так как соответствующий метод вызывается у объекта, полученного в результате вызова метода getCity(). Причем, country.getCity - как раз является составным именем метода.

Наконец, идентификаторы также используются для названий меток (label). Эта конструкция рассматривается позже, однако приведем пример, иллюстрирующий, что пространства имен и названий меток полностью разделены.

```
num:
   for (int num = 2; num <= 100; num++) {
      int n = (int)Math.sqrt(num)+1;
      while (--n != 0) {
         if (num%n==0) {
            continue num;
         }
      }
      System.out.print(num+" ");
   }
```

Результатом будут простые числа, меньшие 100:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Легко видеть, что применяются одноименные переменная и метка num, причем последняя используется для выхода из внутреннего цикла while на внешний for.

Очевидно, что удобнее использовать простое имя, а не составное, т.к. оно короче и его легче запомнить. Однако понятно, что если в системе есть очень много классов, у которых много переменных, то можно столкнуться с ситуацией, когда в разных классах есть одноименные переменные или методы. Для решения этой и других подобных проблем вводится новое понятие - область видимости.

## 2.3. Область видимости (введение)

Чтобы не заставлять программистов, совместно работающих над различными классами одной системы, координировать имена, которые они дают различным конструкциям языка, у каждого имени есть область видимости (scope). Если обращение к, например, полю идет

из части кода, попадающей в область видимости его имени, то можно пользоваться простым именем, если нет - то необходимо применять составное.

Например,

```
class Point {
   int x,y;

   int getX() {
      return x; // простое имя
   }
}

class Test {
   void main() {
      Point p = new Point();
      p.x=3; // составное имя
   }
}
```

Видно, что к полю x изнутри класса можно обращаться по простому имени. К нему же из другого класса можно обратиться только по составному имени. Оно составляется из имени переменной, ссылающейся на объект, и имени поля.

Теперь необходимо рассмотреть области видимости для всех элементов языка. Однако прежде познакомимся ближе с тем, что такое пакеты, как и для чего они используются.

# 3. Пакеты

Программа на Java представляет собой набор пакетов (packages). Каждый пакет может включать вложенные пакеты, то есть они образуют иерархическую систему.

Кроме этого, пакеты могут содержать классы и интерфейсы, и таким образом группируют типы, что необходимо сразу для нескольких целей. Во-первых, чисто физически невозможно работать с большим количеством классов, если они "свалены в кучу". Во-вторых, модульная декомпозиция облегчает проектирование системы. К тому же, как будет рассмотрено ниже, существует специальный уровень доступа, позволяющий типам из одного пакета более "тесно" взаимодействовать друг с другом, чем с классами из других пакетов. Таким образом, с помощью пакетов производится логическая группировка типов. Из ООП известно, что большая связность системы, то есть среднее количество классов, с которыми взаимодействует каждый класс, серьезно усложняет развитие и поддержку такой системы. С применением пакетов гораздо проще эффективно организовать взаимодействие подсистем друг с другом.

Наконец, каждый пакет имеет свое пространство имен, что позволяет создавать одноименные классы в различных пакетах. Таким образом, разработчикам не приходится тратить время на разрешение конфликта имен.

## 3.1. Элементы пакета

Еще раз повторим, что элементами пакета являются вложенные пакеты и типы (классы и интерфейсы). Одноименные элементы запрещены, то есть не может быть одноименных класса и интерфейса или вложенного пакета и типа. В противном случае возникнет ошибка компиляции.

Например, в JDK 1.0 пакет java содержал следующие пакеты: applet, awt, io, lang, net, util; и не содержал ни одного типа. Пакет java.awt содержал вложенный пакет image и 46 классов и интерфейсов.

Составное имя любого элемента пакета составляется из составного имени этого пакета и простого имени элемента. Например, для класса Object в пакете java.lang составным именем будет java.lang.Object, а для пакета image в пакете java.awt - java.awt.image.

Иерархическая структура пакетов была введена для удобства организации связанных пакетов, однако вложенные пакеты или соседние, то есть вложенные в один и тот же пакет, не имеют никаких дополнительных связей между собой, кроме ограничения на несовпадение имен. Например, пакеты space.sun, space.sun.ray, space.moon и factory.store совершенно "равны" между собой, и типы из одного из этих пакетов не имеют никакого особенного доступа к типам других пакетов.

## 3.2. Платформенная поддержка пакетов

Простейшим способом организации пакетов и типов является обычная файловая структура. Рассмотрим вырожденный пример, когда все пакеты, исходный и бинарный код располагаются в одной директории и ее поддиректориях.

В этой корневой директории должна быть папка java, соответствующая основному пакету языка, а в ней, в свою очередь, вложенные папки applet, awt, io, lang, net, util.

Предположим, разработчик работает над моделью солнечной системы, для чего создал классы Sun, Moon и Test, и расположил их в пакете space.sunsystem. В таком случае в корневой директории будет папка space, которая будет соответствовать одноименному пакету, а в ней - папка sunsystem, в которой хранятся классы этого разработчика.

Как известно, исходный код располагается в файлах с расширением .java, а бинарный - с расширением .class. Таким образом, содержимое папки sunsystem может выглядеть следующим образом:

```
Moon.java
Moon.class
Sun.java
Sun.class
Test.java
Test.class
```

Другими словами, исходный код классов

```
space.sunsystem.Moon
space.sunsystem.Sun
space.sunsystem.Test
```

хранится в файлах

```
space\sunsystem\Moon.java
space\sunsystem\Sun.java
space\sunsystem\Test.java
```

а бинарный код - в соответствующих .class-файлах. Обратите внимание, что преобразование имен пакетов в файловые пути потребовало замены разделителя . (точки) на символ-разделитель файлов (для Windows это обратный слеш \). Такое преобразование легко сможет сделать как компилятор для поиска исходных текстов и бинарного кода, так и виртуальная машина для загрузки классов и интерфейсов.

Обратите внимание, что было бы ошибкой запускать Java прямо из папки space\sunsystem и пытаться обращаться к классу Test, несмотря на то, что файл-описание лежит именно в ней. Необходимо подняться на два уровня директорий выше, чтобы Java, построив путь из имени пакета, смогла обнаружить нужный файл.

Кроме того, немаловажно то, что Java всегда различает регистр идентификаторов, а значит, и названия файлов и директорий должны точно отвечать запрограммированным именам. Хотя в некоторых случаях операционная система может обеспечить доступ, не взирая на регистр, но при изменении обстоятельств расхождения могут привести к сбоям.

Существует специальное выражение, объявляющее пакет (подробно рассматривается ниже). Оно предшествует объявлению типа и обозначает, какому пакету будет принадлежать этот тип. Таким образом, набор доступных пакетов определяется набором доступных файлов, содержащих объявления типов и пакетов. Например, если создать пустую директорию, или заполнить ее посторонними файлами, то это отнюдь не приведет к появлению пакета в Java.

Какие файлы доступны для утилит Java SDK (компилятора, интерпретатора и т.д.), устанавливается на уровне операционной системы, ведь утилиты - это обычные программы, которые выполняются под управлением ОС, и, конечно, следуют ее правилам. Например, если пакет содержит один тип, но файл-описание этого типа недоступен для чтения текущему пользователю ОС, то для Java этот тип и этот пакет не будут существовать.

Понятно, что далеко не всегда удобно хранить все файлы в одной директории. Зачастую разные классы находятся в разных местах, а некоторые могут даже распространятся в виде архивов, для ускорения загрузки через сеть. Требование копировать все такие файлы в одну папку было бы крайне затруднительным.

Поэтому Java использует специальную переменную окружения, которая называется classpath. Аналогично тому, как переменная path помогает системе находить и загружать динамические библиотеки, эта переменная помогает работать с Java-классами. Ее значение должно состоять из путей к директориям или архивам, разделенных точкой с запятой. С версии 1.1 поддерживаются архивы типов ZIP и JAR (Java ARchive) - специальный формат, разработанный на основе ZIP для Java.

Например, переменная classpath может иметь такое значение:

```
.;c:\java\classes;d:\lib\3Dengine.zip;d:\lib\fire.jar
```

В результате все указанные директории и содержимое всех архивов "добавляется" к исходной корневой директории. Java в поисках класса будет искать его по описанному

выше правилу во всех указанных папках и архивах по порядку. Обратите внимание, что первым в переменной указана текущая директория (представлена точкой). Это делается для того, чтобы поиск всегда начинался с исходной корневой директории. Конечно, такая запись не является обязательной и делается на усмотрение разработчика.

Несмотря на явные удобства такой конструкции, она таит в себе и опасности. Если при работе случилось так, что разрабатываемые классы лежат в некоторой директории, и она указана в classpath позже, чем некая другая директория, в которой обнаруживаются одноименные типы, то разобраться в такой ситуации будет непросто. В классы будут вноситься изменения, которые никак не проявляются при запуске из-за того, что Java на самом деле загружает одни и те же файлы из посторонней папки.

Поэтому к этой переменной среды окружения необходимо относится с вниманием. Полезно помнить, что необязательно устанавливать ее значение сразу для всей операционной системы. Его можно явно указывать при каждом запуске компилятора или виртуальной машины как опцию, что, во-первых, никогда не повлияет на другие Java-программы, а во-вторых, заметно упрощает поиск ошибок, связанных с некорректным значением classpath.

Наконец, возможно применять и альтернативные подходы к хранению пакетов и файлов с исходным и бинарным кодом. Например, в качестве такого хранилища может быть использована база данных. Более того, существует ограничение на размещение объявлений классов в .java-файлах, которое рассматривается ниже, а при использовании БД любые ограничения можно снять. Тем не менее, при таком подходе рекомендуется предоставлять утилиты импорта/экспорта с учетом ограничения для переходов из/в файлы.

## 3.3. Модуль компиляции

Модуль компиляции (complication unit) - хранится в текстовом .java-файле и является единичной порцией входных данных для компилятора. Он состоит из трех частей:

- объявление пакета;
- import-выражения;
- объявления верхнего уровня.

Объявление пакета одновременно указывает, какому пакету будут принадлежать все объявляемые ниже типы. Это выражение может отсутствовать, что означает, что эти классы располагаются в безымянном пакете (другое название - пакет по умолчанию).

import-выражения позволяют обращаться к типам из других пакетов по их простым именам, "импортировать" их. Эти выражения также необязательны.

Наконец, объявления верхнего уровня содержат объявления одного или нескольких типов. Название "верхнего уровня" противопоставляет эти классы и интерфейсы, располагающиеся в пакетах, внутренним типам, которые являются элементами и располагаются внутри других типов. Как ни странно, но эта часть также является необязательной, в том смысле, что компилятор не выдаст ошибки в случае ее отсутствия. Однако, никаких .class-файлов сгенерировано тоже не будет.

Доступность модулей компиляции определяется платформенной поддержкой, т.к. утилиты Java являются обычными программами, которые исполняются операционной системой по общим правилам.

Рассмотрим все 3 части более подробно.

### 3.3.1. Объявление пакета

Первое выражение в модуле компиляции - объявление пакета. Оно записывается с помощью ключевого слова package, после которого указывается полное имя пакета.

Например, первой строкой (после комментариев) в файле java/lang/Object.java идет:

```
package java.lang;
```

что служит одновременно объявлением пакета lang, вложенного в пакет java, и указанием, что объявляемый ниже класс Object, находится в этом пакете. Так складывается полное имя класса java.lang.Object.

Если это выражение отсутствует, то такой модуль компиляции принадлежит безымянному пакету. Этот пакет по умолчанию обязательно должен поддерживаться реализацией Java-платформы. Обратите внимание, что он не может иметь вложенных пакетов, так как составное имя пакета должно обязательно начинаться с имени пакета верхнего уровня.

Таким образом, самая простая программа может выглядеть следующим образом:

```
class Simple {
   public static void main(String s[]) {
      System.out.println("Hello!");
   }
}
```

Этот модуль компиляции будет принадлежать безымянному пакету.

Пакет по умолчанию был введен в Java для облегчения написания очень небольших или временных приложений, для экспериментов. Если же программа будет распространяться для пользователей, то рекомендуется расположить ее в пакете, который в свою очередь должен быть правильно назван. Соглашения по именованию рассматриваются ниже.

Доступность пакета определяется по доступности модулей компиляции, в которых он объявляется. Точнее, пакет доступен тогда и только тогда, когда выполняется любое из следующих двух условий:

- доступен модуль компиляции с объявлением этого пакета;
- доступен один из вложенных пакетов этого пакета.

Таким образом, для следующего кода:

```
package space.star;

class Sun {
}
```

если файл, который хранит этот модуль компиляции, доступен Java-платформе, то пакеты space и вложенный в него star (полное название space.star) также становятся доступны для Java.

Если пакет доступен, то область видимости его объявления - все доступные модули компиляции. Проще говоря, все существующие пакеты доступны для всех классов, никаких ограничений на доступ к пакетам в Java нет.

Требуется, чтобы пакеты java.lang и java.io, а значит и java, всегда были доступны для Java-платформы, поскольку они содержат классы, необходимые для работы любого приложения.

### 3.3.2. Импорт-выражения

Как будет рассмотрено ниже, область видимости объявления типа - пакет, в котором он располагается. Это означает, что внутри этого пакета допускается обращение к типу по его простому имени. Из всех других пакетов необходимо обращаться по составному имени, то есть полное имя пакета плюс простое имя типа, разделенные точкой. Поскольку пакеты могут иметь довольно длинные имена (например, дополнительный пакет в составе JDK1.2 называется com.sun.image.codec.jpeg), а тип может многократно использоваться в модуле компиляции, то такое ограничение может привести к усложнению исходного кода и сложностям в разработке.

Для решения этой проблемы вводятся import-выражения, позволяющие импортировать типы в модуль компиляции и далее обращаться к ним по простым именам. Существует два вида таких выражений:

- импорт одного типа;
- импорт пакета.

Важно подчеркнуть, что импортирующие выражения являются, по сути, подсказкой компилятора. Он пользуется ими, чтобы для каждого простого имени типа из другого пакета получить его полное имя, которое и попадает в компилированный код. Это означает, что импортирующих выражений может быть очень много, включая и такие, что импортируют неиспользуемые пакеты и типы,- все это никак не скажется ни на размере, ни на качестве бинарного кода. Также нет никакой разницы, обращаться ли к типу по его полному имени, или включить его в импортирующее выражение и обращаться по простому имени - результат будет идентичный.

Импортирующие выражения имеют эффект только внутри модуля компиляции, в котором они объявлены. Все объявления типов высшего уровня, находящиеся в этом же модуле, могут одинаково пользоваться импортированными типами. К импортированным типам допускается и обычный доступ по полному имени.

Выражение, импортирующее один тип, записываются с помощью ключевого слова import и полного имени типа. Например:

```
import java.net.URL;
```

Такое выражение означает, что в дальнейшем в этом модуле компиляции простое имя URL будет обозначать одноименный класс из пакета java.net. Попытка импортировать тип, недоступный на момент компиляции, вызовет ошибку. Если один и тот же тип импортируется несколько раз, то это не создает ошибки, а дублирующиеся выражения игнорируются. Если же импортируются типы с одинаковыми простыми именами из разных пакетов, то такая ситуация породит ошибку компиляции.

Выражение, импортирующее пакет, включает в себя полное имя пакета следующим образом.

```
import java.awt.*;
```

Это выражение делает доступными все типы, находящиеся в пакете java.awt, по их простому имени. Попытка импортировать пакет, недоступный на момент компиляции, вызовет ошибку. Импортирование одного пакета многократно не создает ошибки, дублирующиеся выражения игнорируются. Обратите внимание, что импортировать вложенный пакет нельзя.

Например:

```
// пример вызовет ошибку компиляции
import java.awt.image;
```

Можно предположить, что теперь возможно обращаться к типам пакета java.awt.image по упрощенному имени, например, image.ImageFilter. На самом деле пример вызовет ошибку компиляции, так как такое выражение расценивается как импорт типа, а в пакете java.awt отсутствует тип image.

Аналогично, выражение

```
import java.awt.*;
```

не делает доступнее классы пакета java.awt.image, их необходимо импортировать отдельно.

Поскольку пакет java.lang содержит типы, без которых невозможно создать ни одну программу, то он импортируется неявным образом в каждый модуль компиляции. Таким образом, все типы из этого пакета доступны по их простым именам без каких-либо дополнительных усилий. Попытка импортировать этот пакет еще раз будет проигнорирована.

Допускается одновременно импортировать пакет и какой-нибудь тип из него:

```
import java.awt.*;
import java.awt.Point;
```

Может появиться вопрос, как же лучше поступать - импортировать типы по отдельности, или сразу весь пакет? Есть ли какая-нибудь разница в этих подходах?

Разница заключается в алгоритме работы компилятора, который приводит каждое простое имя к полному. Он состоит из трех шагов:

- сначала просматриваются выражения, импортирующие типы;
- затем другие типы, объявленные в текущем пакете, в том числе, и в текущем модуле компиляции;
- наконец, просматриваются выражения, импортирующие пакеты.

Таким образом, если явно импортирован тип, то невозможно ни объявление нового типа с таким же именем, ни доступ по простому имени к одноименному типу в текущем пакете.

Например:

```
// пример вызовет ошибку компиляции
package my_geom;

import java.awt.Point;

class Point {
}
```

Этот модуль вызовет ошибку компиляции, так как имя Point в объявлении высшего типа будет рассматриваться как обращение к импортированному классу java.awt.Point, а его переопределять, конечно, нельзя.

Если в пакете объявлен тип:

```
package my_geom;

class Point {
}
```

то в другом модуле компиляции:

```
package my_geom;

import java.awt.Point;

class Line {
   void main() {
      System.out.println(new Point());
   }
}
```

складывается неопределенная ситуация - какой из классов, my_geom.Point или java.awt.Point, будет использован при создании объекта? Результатом будет:

```
java.awt.Point[x=0,y=0]
```

Как и регламентируется правилами, имя Point было трактовано на основе импорта типа. К классу текущего пакета все еще можно обращаться по полному имени: my_geom.Point. Если бы рассматривался безымянный пакет, то обратиться к такому "перекрытому" типу было бы уже невозможно, что является дополнительным аргументом к рекомендации располагать серьезные программы в именованных пакетах.

Теперь рассмотрим импорт пакета. Его еще называют "импорт по требованию", подразумевая, что не происходит никакой "загрузки" всех типов импортированного пакета сразу при указании импортирующего выражения, их полные имена подставляются по мере использования простых имен в коде. Возможно импортировать пакет и использовать только один тип (или даже ни одного) из него.

Изменим рассматриваемый выше пример:

```
package my_geom;

import java.awt.*;

class Line {
   void main() {
      System.out.println(new Point());
      System.out.println(new Rectangle());
   }
}
```

Теперь результатом будет:

```
my_geom.Point@92d342
java.awt.Rectangle[x=0,y=0,width=0,height=0]
```

Тип Point нашелся в текущем пакете, поэтому компилятору не пришлось делать поиск по пакету java.awt. Второй объект порождается от класса Rectangle, который не существует в текущем пакете, зато обнаруживается в java.awt.

Также корректен теперь пример:

```
package my_geom;

import java.awt.*;

class Point {
}
```

Таким образом, импорт пакета не препятствует объявлению новых или обращению к существующим типам текущего пакета по простым именам. Если все же нужно работать именно с внешними типами, то можно воспользоваться импортом типа или обращаться к ним по полным именам. Кроме этого, считается, что импорт конкретных типов помогает при прочтении кода сразу понять, какие внешние классы и интерфейсы используются в этом модуле компиляции. Однако полностью полагаться на такое соображение не стоит, так как возможны случаи, когда импортированные типы не используются и, напротив, в коде стоит обращение к другим типам по полному имени.

### 3.3.3. Объявление верхнего уровня

Далее модуль компиляции может содержать одно или несколько объявлений классов и интерфейсов. Подробно формат такого объявления рассматривается в следующих лекциях, однако приведем краткую информацию и здесь.

Объявление класса начинается с ключевого слова class, интерфейса - interface. Далее указывается имя типа, а затем в фигурных скобках описывается тело типа. Например:

```
package first;
```

```
class FirstClass {
}

interface MyInterface {
}
```

Область видимости типа - пакет, в котором он описан. Из других пакетов к типу можно обращаться либо по составному имени, либо с помощью импортирующих выражений.

Однако кроме области видимости в Java также есть средства разграничения доступа. По умолчанию тип объявляется доступным только для других типов своего пакета. Чтобы другие пакеты также могли использовать его, можно указать ключевое слово public:

```
package second;

public class OpenClass {
}

public interface PublicInterface {
}
```

Такие типы доступны для всех пакетов.

Объявления верхнего уровня описывают классы и интерфейсы, которые хранятся в пакетах. В версии Java 1.1 были введены внутренние (inner) типы, которые объявляются внутри других типов и являются их элементами наряду с полями и методами. Эта возможность является вспомогательной и довольно запутанной, поэтому не рассматривается подробно в курсе, хотя некоторые примеры и пояснения помогут в целом ее освоить.

Если пакеты, исходный и бинарный код хранятся в файловой системе, то Java может накладывать ограничение на объявления классов в модулях компиляции. Это ограничение создает ошибку компиляции в случае, если описание типа не обнаруживается в файле с названием, составленным из имени типа и расширения (например, ,java), и при условии:

- тип объявлен как public, и, значит, может быть использован из других пакетов;
- либо если тип используется из других модулей компиляции в своем пакете.

Эти условия означают, что в модуле компиляции может быть максимум один тип, отвечающий этим условиям.

Другими словами, в модуле компиляции может быть максимум один public тип, и его имя и имя файла должны совпадать. Если же в нем есть не-public типы, имена которых не совпадают с именем файла, то они должны использоваться только внутри этого модуля компиляции.

Если же для хранения пакетов используется БД, то такое ограничение не должно накладываться.

На практике же программисты зачастую помещают в один модуль компиляции ровно один тип, независимо от того, public он или нет. Это существенно упрощает работу с ними. Например, описание класса space.sun.Size хранится в файле space\sun\Size.java, а

бинарный код - в файле Size.class в той же директории. Именно так устроены все стандартные библиотеки Java.

Обращаем внимание, что при объявлении классов, вполне допускаются перекрестные обращения. Например, следующий пример совершенно корректен:

```
package test;

/*
 * Класс Human, описывающий человека
 */
class Human {
 String name;
 Car car; // принадлежащая человеку машина
}

/*
 * Класс Car, описывающий автомобиль
 */
class Car {
 String model;
 Human driver; // водитель, управляющий машиной
}
```

Кроме того, класс Car был использован ранее, чем был объявлен. Такое перекрестное использование типов также допускается в случае, если они находятся в разных пакетах. Компилятор должен поддерживать возможность транслировать их одновременно.

## 3.4. Уникальность имен пакетов

Поскольку Java создавалась как язык, предназначенный для распространения приложений через Интернет, а приложения состоят из структуры пакетов, то необходимо предпринять некоторые усилия, чтобы не произошел конфликт имен. Имена двух используемых пакетов могут совпасть по прошествии значительного времени после их создания. Исправить такое положение обычному программисту будет крайне затруднительно.

Поэтому создатели Java предлагают следующий способ уникального именования пакетов. Если программа создается разработчиком, у которого есть интернет-сайт, либо же он работает на организацию, у которой есть сайт, и доменное имя такого сайта, например, company.com, то имена пакетов должны начинаться с этих же слов, выписанных в обратном порядке: com.company. Дальнейшие вложенные пакеты могут носить названия подразделений компании, пакетов, фамилий, имена компьютеров и т.д.

Таким образом, пакет верхнего уровня всегда записывается ASCII-буквами в нижнем регистре и может иметь одно из следующих имен:

- трехбуквенные com, edu, gov, mil, net, org (этот список расширяется);
- двухбуквенные, обозначающие имена стран, такие как ru, su, de, uk и другие.

Если имя сайта противоречит требованиям к идентификаторам Java, то можно предпринять следующие шаги:

- если в имени стоит запрещенный символ, например тире, то его можно заменить знаком подчеркивания;
- если имя совпадает с зарезервированным словом, то можно в конце добавить знак подчеркивания;
- если имя начинается с цифры, можно в начале добавить знак подчеркивания.

Примеры имен пакетов, составленных по таким правилам:

```
com.sun.image.codec.jpeg
org.omg.CORBA.ORBPackage
oracle.jdbc.driver.OracleDriver
```

Однако, конечно, никто не требует, чтобы Java-пакеты были обязательно доступны на интернет-сайте, который дал им имя. Скорее была сделана попытка воспользоваться существующей системой имен вместо того, чтобы создавать новую для именования библиотек.

# 4. Область видимости имен

Областью видимости объявления некоторого элемента языка называется часть программы, откуда допускается обращение к этому элементу по простому имени.

При рассмотрении каждого элемента языка будет указываться его область видимости, однако полезно собрать в одном месте эту информацию.

Область видимости доступного пакета - вся программа, то есть любой класс может использовать доступный пакет. Однако необходимо помнить, что обращаться к пакету можно только по его полному составному имени. К п¿акету java.lang ни из какого места нельзя обратиться как к просто lang.

Областью видимости импортированного типа являются все объявления верхнего уровня в этом модуле компиляции.

Областью видимости типа (класса или интерфейса) верхнего уровня является пакет, в котором он объявлен. Из других пакетов доступ возможен либо по составному имени, либо с помощью импортирующего выражения, которое помогает компилятору воссоздать составное имя.

Область видимости элементов классов или интерфейсов - это все тело типа, в котором они объявлены. Если обращение к этим элементам происходит из другого типа, то тогда необходимо воспользоваться составным именем. Имя может быть составлено из простого или составного имени типа, имени объектной переменной или ключевых слов super или this, после чего через точку указывается простое имя элемента.

Аргументы метода, конструктора или обработчика ошибок видны только внутри этих конструкций и не могут быть доступы извне.

Область видимости локальных переменных начинается с момента их инициализации и до конца блока, в котором они объявлены. В отличие от полей типов, локальные переменные не имеют значений по умолчанию и должны инициализироваться явно.

```
int x;
for (int i=0; i<10; i++) {
   int t=5+i;
}
// здесь переменная t уже не доступна,
// так как блок, в котором она была объявлена
// уже завершен

// а переменная x еще не доступна,
// так как еще не была инициализирована
```

Определенные проблемы возникают, когда происходит перекрытие областей видимости и возникает конфликт имен различных конструкций языка.

## 4.1. "Затеняющее" объявление (Shadowing)

В классе Human (человек) объявлено поле age (возраст). Удобно определить также метод setAge(), который должен устанавливать новое значение возраста для человека. Вполне логично сделать у метода setAge() один входной аргумент, который также будет называть age (ведь в качестве этого аргумента будет передаваться новое значение возраста). Получается, что в реализации метода setAge() нужно написать age=age, в первом случае подразумевая поле класса, во втором - параметр метода. Понятно, что, хотя с точки зрения компилятора это корректная конструкция, попытка сослаться на две совершенно разные переменные через одно имя успехом завершиться не может. Надо заметить, что такие ошибки случаются порой даже у опытных разработчиков.

Во-первых, рассмотрим, из-за чего возникла конфликтная ситуация. Есть два элемента языка - аргумент метода и поле класса, области видимости которых пересеклись. Область видимости поля класса больше, она охватывает все тело класса. В то время как область видимости аргумента метода включает только сам метод. В таком случае внутри области пересечения по простому имени доступен именно аргумент метода, а поле класса "затеняется" (shadowing) объявлением параметра метода.

Остается вопрос, как все же обратиться к полю класса в такой ситуации. Если доступ по простому имени невозможен, надо воспользоваться составным. Здесь удобнее всего применить специальное ключевое слово this (оно будет подробно рассматриваться в дальнейших главах). Слово this имеет значение ссылки на объект, внутри которого оно применяется. Если вызвать метод setAge() у объекта класса Human и использовать слово this в этом методе, то его значение будет ссылкой на этот объект.

Исправленный вариант примера:

```
class Human {
   int age;// возраст

   void setAge(int age) {
      this.age=age; // верное присвоение!
   }
}
```

Конфликт имен, возникающий из-за затеняющего объявления, довольно легко исправить с помощью ключевого слова this или других конструкций языка в зависимости от обстоятельств. Наибольшей проблемой является то, что компилятор никак не сообщает о таких ситуациях, и самое сложное - выявить ее тестированием или контрольным просмотром кода.

## 4.2. "Заслоняющее" объявление (Obscuring)

Может возникнуть ситуация, когда простое имя может быть одновременно рассмотрено как имя переменной, типа или пакета.

Приведем пример, который частично иллюстрирует такой случай:

```
import java.awt.*;

public class Obscuring  {
   static Point Test = new Point(3,2);

   public static void main (String s[]) {
      print(Test.x);
   }
}

class Test {
   static int x = -5;
}
```

В методе main() простое имя Test одновременно обозначает имя поля класса Oscruing и имя другого типа, находящегося в том же пакете - Test. С помощью этого имени идет обращение к полю x, которое определено и в классе java.awt.Point и Test.

Результатом этого примера станет 3, то есть переменная имеет более высокий приоритет. В свою очередь, тип имеет более высокий приоритет, чем пакет. Таким образом, обращение к доступному в обычных условиях типу или пакету может оказаться невозможным, если есть более высокоприоритетное объявление одноименной переменной или типа. Такое объявление называется "заслоняющим" (obscuring).

Эта проблема скорее всего не возникнет, если следовать соглашениям по именованию элементов языка Java.

# 5. Соглашения по именованию

Для того чтобы код, написанный на Java, было легко читать и понимать не только его автору, но и другим разработчикам, а также для устранения некоторых конфликтов имен, предлагаются следующие соглашения по именованию элементов языка Java. Стандартные библиотеки и классы Java также следуют им, где это возможно.

Соглашения регулируют именование следующих конструкций:

- пакеты;

- типы (классы и интерфейсы);
- методы;
- поля;
- поля-константы;
- локальные переменные и параметры методов и др.;

Рассмотрим их последовательно.

Правила построения имен пакетов уже подробно рассматривались в этой главе. Имя каждого пакета начинается с маленькой буквы и представляет собой, как правило, одно недлинное слово. Если требуется составить название из нескольких слов, можно воспользоваться знаком подчеркивания или начинать следующее слово с большой буквы. Имя пакета верхнего уровня обычно соответствует доменному имени первого уровня. Названия java и javax (Java eXtension) зарезервированы компанией Sun для стандартных пакетов Java.

При возникновении ситуации "заслоняющего" объявления (obscuring) можно изменить имя локальной переменной, что не повлечет за собой глобальных изменений в коде. Случай же конфликта с именем типа не должен возникать согласно правилам именования типов.

Имена типов начинаются с большой буквы и могут состоять из нескольких слов, каждое следующее слово также начинается с большой буквы. Конечно, надо стремится к тому, чтобы имена были описательными, "говорящими".

Имена классов, как правило, являются существительными:

```
Human
HighGreenOak
ArrayIndexOutOfBoundsException
```

(Последний пример - ошибка, возникающая при использовании индекса массива, который выходит за допустимые границы).

Аналогично задаются имена интерфейсов, хотя они не обязательно должны быть существительными. Часто используется английский суффикс able:

```
Runnable
Serializable
Cloneable
```

Проблема "заслоняющего" объявления (obscuring) для типов редко встречается, так как имена пакетов и локальных переменных (параметров) начинаются с маленькой буквы, а типов - с большой.

Имена методов должны быть глаголами и обозначать действия, которое совершает данный метод. Имя должно начинаться с маленькой буквы, но может состоять из нескольких слов, каждое следующее слово начинается с заглавной буквы. Существует ряд принятых названий для методов:

- если методы предназначены для чтения и изменения значения переменной, то их имена начинаются с get и set соответственно. Например, для переменной size это будут getSize() и setSize().
- метод, возвращающий длину, называется length(), например, в классе String.
- метод, который проверяет булевское условие, имеет имя начинающееся с is, например, isVisible() у компоненты графического пользовательского интерфейса.
- метод, который преобразует величину в формат F, называется toF(), например, метод toSting(), который приводит любой объект к строке.

Вообще, рекомендуется везде, где возможно, называть методы похожим образом, как в стандартных классах Java, для того, чтобы они были легко понятны всем разработчикам.

Поля класса имеют имена, записываемые в том же стиле, что и для методов - начинаются с маленькой буквы, могут состоять из нескольких слов, каждое следующее слово начинается с заглавной буквы. Имена должны быть существительными, например, поле name в классе Human или size в классе Planet.

Как для полей решается проблема "заслоняющего" объявления (obscuring) уже обсуждалось.

Поля могут быть константами, если в их объявлении стоит ключевое слово final. Их имена состоят из последовательности слов, сокращений, аббревиатур. Записываются они только большими буквами, слова разделяются знаками подчеркивания:

```
PI
MIN_VALUE
MAX_VALUE
```

Иногда константы образуют группу, тогда рекомендуется использовать одно или несколько одинаковых слов в начале имен:

```
COLOR_RED
COLOR_GREEN
COLOR_BLUE
```

Наконец, рассмотрим имена локальных переменных и параметров методов, конструкторов и обработчиков ошибок. Они, как правило, довольно короткие, но тем не менее должны быть осмыслены. Например, можно использовать аббревиатуру (имя cp для ссылки на экземпляр класса ColorPoint) или сокращение (buf для buffer).

Распространенные однобуквенные сокращения:

```
byte b;
char c;
int i,j,k;
long l;
float f;
double d;
Object o;
String s;
Exception e; // объект, представляющий ошибку в Java
```

Двух- и трехбуквенные имена не должны совпадать с принятыми доменными именами первого уровня интернет-сайтов.

# 6. Заключение

В этой главе был рассмотрен механизм именования элементов языка. Для того чтобы различные части большой системы не зависели друг от друга, вводится понятия область видимости имени, вне которой необходимо использовать не простое, а составное имя. Затем было изучено важно понятия элементов (members), которые могут быть у пакетов и ссылочных типов. Также рассматривается связь терминов идентификатор (из темы Лексика) и имя.

Затем были рассмотрены пакеты, которые используются в Java для создания физической и логической структуры классов, а также и для более точного разграничения области видимости. Пакет содержит в себе вложенные пакеты и типы (классы и интерфейсы). Вопрос о платформенной поддержке пакетов привел к рассмотрению модулей компиляции как текстовых файлов, так как именно в виде файлов и директорий, как правило, хранятся и распространяются Java-приложения. Тогда же впервые был рассмотрен вопрос разграничения доступа, так как доступ к модулям компиляции определяется именно платформенной поддержкой, а точнее – операционной системой.

Модуль компиляции состоит из трех основных частей – объявление пакета, импорт-выражения и объявления верхнего уровня. Важную роль играет безымянный пакет, или пакет по умолчанию, хотя он и не рекомендуется для применения при создании больших систем. Были рассмотрены детали применения двух видов импорт-выражений – импорт класса и импорт пакета. Наконец, было начато рассмотрение объявлений верхнего уровня (эта тема будет продолжена в главе, описывающей объявление классов). Пакеты, как и другие элементы языка, имеют определенные соглашения по именованию, призванные облегчить понимание кода и уменьшить возможность появления ошибок и двусмысленных ситуаций в программе.

Описание области видимости для различных элементов языка приводит к вопросу о возможных перекрытиях таких областей и, как следствие, конфликтов имен. Рассматриваются «затеняющие» и «заслоняющие» объявления. Для устранения или уменьшения возможности возникновения таких ситуаций описываются соглашения по именованию для всех элементов языка.

# 7. Контрольные вопросы

5-1. Какие элементы языка Java имеют имена? Какие из них должны быть объявлены?

a.) Следующие элементы языка имеют имена:

- пакеты;
- классы;
- интерфейсы;
- элементы (member) ссылочных типов:
  - поля;

- методы;
- внутренние классы и интерфейсы;

• аргументы:
- методов;
- конструкторов;
- обработчиков ошибок;

• локальные переменные.

Все они должны быть объявлены, так как именно при объявлении указывается имя.

5-2. Что из перечисленных ниже слов является простым именем, составным именем, идентификатором?

```
MyClass
MyClass.name
MyClass.name.toString()
MyClass.name.toString().hashCode()
```

a.) MyClass – простое имя. MyClass.name, MyClass.name.toString – составные имена. Все они и hashCode – идентификаторы.

5-3. Могут ли пакет и вложенный пакеты содержать одноименные классы?

a.) Да, поскольку эти классы будут иметь непересекающуюся область видимости.

5-4. Из какой директории необходимо запускать компилятор, чтобы скомпилировать программу, состоящую из одного класса test.first.Start, описание которого сохранено в файле c:\Java\programs\test\first\Start.java?

a.) Компилятор можно запускать из любой директории, главное – правильно указать путь к файлу. Например,

```
c:\Java>javac programs\test\first\Start.java
```

или

```
c:\Java\programs\test>javac first\Start.java
```

5-5. Из какой директории необходимо запускать интерпретатор Java для выполнения программы, описанной в предыдущем вопросе?

a.) JVM нужно запускать из той директории, где лежит пакет по умолчанию, т.е.

```
c:\Java\programs>java test.first.Start
```

5-6. Можно ли исполнить программу, описанную в предыдущих 2 вопросах, если запускать виртуальную машину из директории c:\ ?

a.) Можно, для этого необходимо включить в classpath необходимый класс.

5-7. Ниже приведено несколько вариантов записи модуля компиляции. Какие из них корректны, если предполагается описать класс Point из пакета test.demo, причем класс активно использует классы java.awt.Point и несколько классов из пакета java.net?

```
a)
package test.demo;
import java.awt.Point;
import java.net.*;

b)
import java.awt.*;
import java.net.*;
package test.demo;

c)
package test.demo;
import java.net.*;
import java.awt.*;

d)
package test.demo.*
import java.net.*;
import java.awt.*;
```

a.) Ответ a) не годится, так как явное импортирование класса java.awt.Point не позволит объявить класс Point, как этого требует условие задачи. Ответ b) не корректен, так как объявление пакета должно идти до импорт-выражений. Ответ d) не верен, так объявление пакета должно содержать только имя пакета (безо всяких звездочек) и оканчиваться знаком точка с запятой. Ответ c) верен.

5-8. Корректен ли объявленный ниже класс? Если нет, то как его можно исправить?

```
class Box {
   private int weight=0;

   public int getWeight() {
      return weight;
   }

   void setWieght(int weight) {
      weight=weigth;
   }
```

```
}
```

а.) Такое объявление корректно с точки зрения, однако в методе setWeight() произошло «затеняющее» объявление. Чтобы устранить конфликт имен между аргументом метода и полем класса, необходимо переименовать одно из них. Как правило, меняют имя именно у аргумента, так как это затронет код лишь одного метода.

5-9. Корректно ли следующее объявление с точки зрения формального выполнения соглашений по именованию?

```
public class flat{
   private int floor_number;
   private int r; // количество комнат

   public int rooms() {
      retun r;
   }

   public int GetFloorNumber() {
      return floor_number;
   }
}
```

а.) Допущен целый ряд нарушений соглашений:

- Класс назван с прописной буквы, должно быть Flat.
- Имя поля floor_number содержит два слова, разделенных знаком подчеркивания, должно быть floorNumber.
- Имя поля r состоит из одной буквы, а оно должно быть более говорящим, например, rooms или roomsNumber
- Имя метода rooms является существительным, а должно быть глаголом, например, getRoomsNumber.
- Имя метода GetFloorNumber начинается с заглавной буквы, должно быть getFloorNumber.

Однако необходимо помнить, что соглашения призваны облегчать написание кода. Если есть какие-либо причины (например, они противоречат давно устоявшимся правилам написания программ), которые напротив усложняют процесс разработки, не нужны следовать соглашениям слишком формально.

# Программирование на Java

## Лекция 6. Объявление классов

### 20 января 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Центральная тема курса – объявление классов, поскольку любое Java-приложение является набором классов.

Первый рассматриваемый вопрос – система разграничения доступа в Java. Описывается, зачем вообще нужно управление доступом в ОО-языке программирования, и как оно осуществляется в Java. Затем подробно рассматривается структура объявления заголовка класса и его тела, которое состоит из элементов (полей и методов), конструкторов и инициализаторов. Дополнительно описывается сигнатура метода main, с которого начинается работа Java-приложения, правила передачи параметров различных типов в методы, перегруженные методы.

# Оглавление

# Лекция 6. Объявление классов

## Содержание лекции.



## 1. Введение

Объявление классов является центральной темой курса, поскольку любая программа на Java - это набор классов. Поскольку типы являются ключевой конструкцией языка, их структура довольно сложна, имеет много тонкостей. Поэтому эта тема разделена на две главы.

Эта глава начинается с продолжения темы прошлой главы - имена и доступ к именованным элементам языка. Необходимо рассмотреть механизм разграничения доступа в Java, как он устроен, для чего применяется. Затем будут описаны ключевые правила объявления классов.

Следующая глава подробно рассматривает особенности объектной модели Java. Вводится понятие интерфейса. Уточняются правила объявления классов, и описывается объявление интерфейса.

# 2. Модификаторы доступа

Во многих языках существуют права доступа, которые ограничивают возможность использования, например, переменной в классе. Например, легко представить два крайних вида прав доступа: это public, когда поле доступно из любой точки программы, и private, когда поле может быть использовано только внутри того класса, в котором оно объявлено.

Однако прежде чем переходить к подробному рассмотрению этих и других модификаторов доступа, необходимо внимательно разобраться, зачем они вообще нужны.

## 2.1. Предназначение модификаторов доступа

Существует весьма распространенное мнение, которое расценивает права доступа как некий элемент безопасности кода: мол, необходимо защищать классы от "неправильного" использования. Например, если в классе Human (человек) есть поле age (возраст человека), то какой-нибудь программист по злому умыслу или незнанию может установить этому полю отрицательное значение, после чего объект станет работать неправильным образом, могут появиться ошибки. Для защиты такого поля age необходимо объявить его private.

Такая точка зрения довольно распространена, однако нужно признать, что она далека от истины. Основной потребностью в разграничении прав доступа является обеспечение неотъемлемого свойства объектной модели - инкапсуляции, то есть, сокрытия реализации. Исправим пример таким образом, чтобы он корректно отражал предназначение модификаторов доступа. Итак, пусть в классе Human есть поле age целочисленного типа, и чтобы все желающие могли пользоваться этим полем, оно объявляется public.

```
public class Human {
   public int age;
}
```

Проходит время, и если в группу программистов, работающих над системой, входят десятки разработчиков, то логично предположить, что все или многие из них начнут использовать это поле.

Вдруг может возникнуть ситуация, что целочисленного типа данных уже недостаточно, и хотелось бы сменить тип поля на дробный. Однако если просто изменить int на double, то вскоре все разработчики, которые пользовались классом Human и его полем age, обнаружат, что в их коде появились ошибки, потому что поле вдруг стало дробным, и в строках, подобной этой:

```
Human h = getHuman();
int i=h.age; // Ошибка!!
```

будет возникать ошибка из-за попытки провести неявным образом сужение примитивного типа.

Получается, что подобное изменение (в общем, небольшое и локальное) потребует модификации многих и многих классов. Поэтому внесение его окажется недопустимым, неоправданным с точки зрения количества усилий, которые необходимо затратить. То есть, объявив один раз поле или метод как public, можно оказаться в ситуации, когда

малейшие изменения (имени, типа, характеристик, правил использования) в дальнейшем станут невозможны.

Напротив, если бы поле было объявлено как private, а для чтения и изменения его значения были бы введены дополнительные методы, то ситуация поменялась бы в корне:

```
public class Human {
   private int age;

   // метод, возвращающий значение age
   public int getAge() {
      return age;
 }

   // метод, устанавливающий значение age
   public void setAge(int a) {
      age=a;
   }
}
```

В этом случае с этим классом могло бы работать множество программистов, и могло бы быть создано большое количество классов, использующих тип Human, но модификатор private дает гарантию, что никто напрямую этим полем не пользуется, и изменение его типа было бы совершенно безболезненной операцией, связанной с изменением в ровно одном классе.

Получение величины возраста выглядело бы следующим образом:

```
Human h = getHuman();
int i=h.getAge(); // Обращение через метод
```

Рассмотрим, как выглядит процесс смены типа поля age:

```
public class Human {

   // поле получает новый тип double
   private /*int*/ double age;

   // старые методы работают с округлением значения
   public int getAge() {
      return (int)Math.round(age);
   }
   public void setAge(int a) {
      age=a;
   }

   // добавляются новые методы для работы с типом double
   public double getExactAge() {
      return age;
   }
```

```
   public void setExactAge(double a) {
      age=a;
   }

}
```

Видно, что старые методы, которые возможно уже применяются во множестве мест, остались без изменения. Точнее, остался без изменений их внешний формат, а внутренняя реализация усложнилась. Но такая перемена не потребует никаких модификаций остальных классов системы. Пример использования

```
Human h = getHuman();
int i=h.getAge(); // Корректно
```

остается верным, переменная i получает корректное целое значение. Однако изменения вводились для возможности работать с дробными величинами. Для этого были добавлены новые методы, и во всех местах, где требуется точное значение возраста, необходимо обращаться к ним:

```
Human h = getHuman();
double d=h.getExactAge(); // Точное значение возраста
```

Итак, в класс была добавлена новая возможность, не потребовавшая никаких изменений уже написанного кода.

За счет чего была достигнута такая гибкость? Необходимо выделить свойства объекта, которые необходимы будущим пользователям этого класса, и их сделать доступными (в данном случае, public). Те же элементы класса, что содержат детали внутренней реализации логики класса, желательно скрывать от внешнего мира, чтобы не образовались нежелательные зависимости, которые могут серьезно сдержать развитие системы.

Этот пример одновременно иллюстрирует и другое теоретическое правило написания объектов, а именно: в большинстве случаев доступ к полям лучше реализовывать через специальные методы (accessors) для чтения (getters) и записи (setters). То есть, само поле рассматривается как деталь внутренней реализации. Действительно, если рассматривать внешний интерфейс объекта как целиком состоящий из допустимых действий, то доступными элементами должны быть только методы, реализующие эти действия. Один из случаев, в котором такой подход приносит необходимую гибкость, уже рассмотрен.

Есть и другие соображения. Например, вернемся к вопросу о корректном использовании объекта и установки верных значений полям. Как следствие, правильное разграничение доступа позволяет ввести механизмы проверки входных значений:

```
public void setAge(int a) {
   (if a>=0) {
   age=a;
}
}
```

В этом примере поле age никогда не примет некорректное отрицательное значение. (Недостатком приведенного примера является то, что в случае неправильных входных данных, они просто игнорируется, нет никаких сообщений, позволяющих узнать, что изменения поля возраста на самом деле не произошло; для полноценной реализации метода необходимо освоить работу с ошибками в Java).

Бывают и более существенные изменения логики класса. Например, данные могут начать храниться не в полях класса, а в более надежном хранилище, например, файловой системе или базе данных. В этом случае методы-аксессоры опять изменят свою реализацию, и начнут обращаться к persistent storage (постоянное хранилище, например, БД) для чтения/записи значений. Если доступа к полям класса не было, а открытыми были только методы для работы с их значениями, то можно довольно легко изменить код этих методов, а наружные типы, которые использовали этот класс, совершенно не изменятся, логика их работы останется той же.

Подведем итоги. Функциональность класса необходимо разделять на открытый интерфейс, описывающий действия, которые будут использовать внешние типы, и на внутреннюю реализацию, которая используется только внутри самого класса. Внешний интерфейс в дальнейшем модифицировать невозможно или очень сложно для больших систем, поэтому его требуется продумывать особенно тщательно. Детали внутренней реализации могут быть изменены на любом этапе, если они не меняют логику работы всего класса. Благодаря такому подходу реализуется одна из базовых характеристик объектной модели - инкапсуляция, и обеспечивается важное преимущество технологии ООП - модульность.

Таким образом, модификаторы доступа вводятся не для защиты типа от внешнего пользователя, а, напротив, для защиты, или избавления, пользователя от излишних зависимостей от деталей внутренней реализации. Что же касается неправильного использования класса, то его создателям нужно стремится к тому, чтобы класс был легок и понятен в применении, тогда таких проблем не возникнет, ведь программист не станет сознательно писать код, который порождает ошибки в его программе.

Конечно, такое разбиение на внешний интерфейс и внутреннюю реализацию не всегда очевидно, часто условно. Для облегчения задачи технических дизайнеров классов в Java введено не 2 (public и private), а 4 уровня доступа. Рассмотрим их и весь механизм разграничения доступа в Java более подробно.

## 2.2. Разграничение доступа в Java

Уровень доступа элемента языка является статическим свойством, задается на уровне кода и всегда проверяется во время компиляции. Попытка обратиться к закрытому элементу вызовет ошибку.

В Java модификаторы доступа указываются для:

- типов (классов и интерфейсов) объявления верхнего уровня;
- элементов ссылочных типов (полей, методов, внутренних типов);
- конструкторов классов.

Как следствие, массив также может быть недоступен в том и только в том случае, если не доступен тип, на основе которого он объявлен.

Все 4 уровня доступа имеют только элементы типов и конструкторы. Это:

- public;
- private;
- protected;
- если не указан ни один из этих 3 типов, то уровень доступа определяется по умолчанию (default).

Первые два из них уже были рассмотрены. Последний уровень (доступ по умолчанию) упоминался в прошлой главе - он позволяет обращения из того же пакета, где объявлен и сам этот класс. По этой причине пакеты в Java являются не просто набором типов, а более структурированной единицей, так как типы внутри одного пакета могут больше взаимодействовать друг с другом, чем с типами из других пакетов.

Наконец, protected дает доступ наследникам класса. Понятно, что наследникам может потребоваться доступ к некоторым элементам родителя, с которыми не требуется иметь дело внешним классам.

Однако описанная структура не позволяет упорядочить модификаторы доступа так, чтобы каждый следующий строго расширял предыдущий. Модификатор protected может быть указан для наследника из другого пакета, а доступ по умолчанию позволяет обращения из классов-ненаследников, если они находятся в том же пакете. По этой причине возможности protected были расширены таким образом, что он включает в себя доступ внутри пакета. Итак, модификаторы доступа упорядочиваются следующим образом (от менее открытых - к более):

```
private
(none) default
protected
public
```

Эта последовательность будет использована далее при изучении деталей наследования классов.

Теперь рассмотрим, какие модификаторы доступа возможны для различных элементов языка.

- Пакеты всегда доступны, поэтому у них нет модификаторов доступа (можно сказать, что все они public, то есть, любой существующий в системе пакет может быть использован из любой точки программы).
- Типы (классы и интерфейсы) верхнего уровня объявления. При их объявлении есть всего две возможности: указать модификатор public или не указывать его. Если доступ к типу является public, то это означает, что он доступен из любой точки кода. Если же он не public, то уровень доступа назначается по умолчанию: тип доступен только внутри того пакета, где он объявлен.
- Массив имеет тот же уровень доступа, что и тип, на основе которого он объявлен (естественно, все примитивные типы являются полностью доступными).
- Элементы и конструкторы объектных типов. Обладают всеми 4 возможными значениями уровня доступа. Все элементы интерфейсов являются public.

Для типов объявления верхнего уровня нет необходимости во всех 4 уровнях доступа. private-типы образовывали бы закрытую мини-программу, никто не мог бы их использовать. Типы, доступные только для наследников, также не были признаны полезными.

Разграничения доступа сказываются не только на обращении к элементам объектных типов или пакетов (через составное имя или прямое обращение), но также при вызове конструкторов, наследовании, приведении типов. Запрещается импортировать недоступные типы.

Проверка уровня доступа проводится компилятором. Обратите внимание на следующие примеры:

```
public class Wheel {
   private double radius;

   public double getRadius() {
      return radius;
   }
}
```

Значение поля radius не доступно снаружи класса, однако открытый метод getRadius() корректно возвращает его.

Рассмотрим следующие два модуля компиляции:

```
package first;

// Некоторый класс Parent
public class Parent {
}
```

```
package first;

// Класс Child наследуется от класса Parent,
// но имеет ограничение доступа по умолчанию
class Child extends Parent {
}

public class Provider {
   public Parent getValue() {
      return new Child();
   }
}
```

К методу getValue() класса Provider можно обратиться и из другого пакета, не только из пакета first, поскольку метод объявлен как public. Как видно, этот метод возвращает экземпляр класса Child, который не доступен из других пакетов. Однако следующий вызов является корректным:

```
package second;
```

```
import first.*;

public class Test {
   public static void main(String s[]) {
      Provider pr = new Provider();
      Parent p = pr.getValue();
      System.out.println(p.getClass().getName());
      // (Child)p - приведет к ошибке компиляции!
   }
}
```

Результатом будет:

```
first.Child
```

То есть, на самом деле в классе Test работа идет с экземпляром недоступного класса Child, что возможно, поскольку обращение к нему делается через открытый класс Parent. Попытка же сделать явное приведение вызовет ошибку. Да, тип объекта "угадан" верно, но доступ к закрытому типу всегда запрещен.

Следующий пример:

```
public class Point {
   private int x, y;

   public boolean equals(Object o) {
      if (o instanceof Point) {
         Point p = (Point)o;
         return p.x==x && p.y==y;
      }
      return false;
   }
}
```

В этом примере объявляется класс Point с двумя полями, описывающими координаты точки. Обратите внимание, что поля полностью закрыты - private. Далее попытаемся переопределить стандартный метод equals() таким образом, чтобы для аргументов, являющихся экземплярами класса Point, или его наследников (логика работы оператора intsanceof) возвращалось истинное значение в случае равенства координат. Обратите внимание на строку, где делается сравнение координат - для этого приходится обращаться к private-полям другого объекта!

Тем не менее, такое действие совершенно корректно, поскольку private допускает обращения из любой точки класса, независимо от того, к какому именно объекту оно производится.

Другие примеры разграничения доступа в Java будут рассматриваться по ходу курса.

# 3. Объявление классов

Рассмотрим базовые возможности объявления классов.

Объявление класса состоит из заголовка и тела класса.

## 3.1. Заголовок класса

Вначале указываются модификаторы класса. Модификаторы доступа для класса уже обсуждались. Допустимым является public, либо его отсутствие - доступ по умолчанию.

Класс может быть объявлен как final. В этом случае не допускается создание наследников такого класса. На своей ветке наследования он является последним. Класс String и классы-обертки, например, являются final-классами.

После списка модификаторов указывается ключевое слово class, а затем имя класса - корректный Java-идентификатор. Таким образом, кратчайшим объявлением класса может являться такой модуль компиляции:

```
class A {}
```

Фигурные скобки обозначают тело класса, но о нем позже.

Указанный идентификатор становится простым именем класса. Полное составное имя класса строится из полного составного имени пакета, в котором он объявлен (если это не безымянный пакет), и простого имени класса, разделенных точкой. Область видимости класса, где он может быть доступен по своему простому имени - его пакет.

Далее заголовок может содержать ключевое слово extends, после которого должно быть указано имя (простое или составное) доступного не-final класса. В этом случае объявляемый класс наследуется от указанного класса. Если выражение extends не применяется, то класс наследуется напрямую от Object. Выражение extends Object допускается и игнорируется.

```
class Parent {} // = class Parent extends Object {}

final class LastChild extends Parent {}

// class WrongChild extends LastChild {} // ошибка!!
```

Попытка расширить final-класс приведет к ошибке компиляции.

Если в объявлении класса A указано выражение extends B, то класс A называют прямым наследником класса B.

Класс A считается наследником класса B если:

- A является прямым наследником B;
- либо существует класс C, который является наследником B, а A является наследником C (это правило применяется рекурсивно).

Таким образом можно проследовать цепочки наследования на несколько уровней вверх.

Если компилятор обнаруживает, что класс является своим наследником, то возникает ошибка компиляции:

```
// пример вызовет ошибку компиляции
class A extends B {}
class B extends C {}
class C extends A {} // ошибка! Класс A стал своим наследником
```

Далее в заголовке может быть указано ключевое слово implements, за которым должно следовать перечисление через запятую имен (простых или составных, повторения запрещены) доступных интерфейсов:

```
public final class String implements Serializable, Comparable {}
```

В этом случае говорят, что класс реализует перечисленные интерфейсы. Как видно из примера, класс может реализовывать любое количество интерфейсов. Если выражение implements отсутствует, то класс действительно не реализует никаких интерфейсов, здесь значений по умолчанию нет.

Далее следует пара фигурных скобок, которые могут быть пустыми или содержать описание тела класса.

## 3.2. Тело класса

Тело класса может содержать объявление элементов (members) класса:

- полей;
- методов;
- внутренних типов (классов и интерфейсов);

и остальных допустимых конструкций:

- конструкторов;
- инициализаторов;
- статических инициализаторов.

Элементы класса имеют имена и передаются по наследству, не-элементы - нет. Для элементов простые имена указываются при объявлении, составные формируются из имени класса или имени переменной объектного типа и простого имени элемента. Областью видимости элементов является все объявление тела класса. Допускается применение любого из всех 4 модификатора доступа. Напоминаем, что соглашения по именованию классов и их элементов обсуждались в прошлой главе.

Не-элементы не обладают именами, а потому не могут быть вызваны явно. Их вызывает сама виртуальная машина. Например, конструктор вызывается при создании объекта. По той же причине неэлементы не обладают модификаторами доступа.

Элементами класса являются элементы, описанные в объявлении тела класса и переданные по наследству от класса-родителя (кроме Object - единственного класса, не имеющего родителя) и всех реализуемых интерфейсов при условии достаточного уровня доступа. Таким образом, если класс содержит элементы с доступом по умолчанию, то его наследники из разных пакетов будут обладать разным набором элементов. Классы из того

же пакета могут пользоваться полным набором элементов, а из других пакетов - только protected и public. private-элементы не передаются по наследству.

Поля и методы могут иметь одинаковые имена, поскольку обращение к полям всегда записывается без скобок, а к методам - всегда со скобками.

Рассмотрим все эти конструкции более подробно.

## 3.3. Объявление полей

Объявление полей начинается с перечисления модификаторов. Возможно применение любого из 3 модификаторов доступа, либо никакого вовсе, что означает уровень доступа по умолчанию.

Поле может быть объявлено final, что означает, что оно инициализируется ровно один раз и больше не будет менять своего значения. Простейший способ работы с final-переменными - инициализация при объявлении:

```
final double PI=3.1415;
```

Также допускается инициализация final-полей в конце каждого конструктора класса.

Не обязательно использовать для инициализации константы компиляции, возможно обращение к различным функциям, например:

```
final long creationTime=System.currentTimeMillis();
```

Данное поле будет хранить время создания объекта. Существуют еще два специальных модификатора - transient и volatile. Они будут рассмотрены в соответствующих главах.

После списка модификаторов указывается тип поля. Затем идет перечисление одного или нескольких имен полей с возможными инициализаторами:

```
int a;
int b=3, c=b+5, d;
Point p, p1=null, p2=new Point();
```

Повторяющиеся имена полей запрещены. Указанный идентификатор при объявлении становится простым именем поля. Составное имя формируется из имени класса или имени переменной объектного типа и простого имени поля. Областью видимости поля является все объявление тела класса.

Запрещается использовать поле в инициализации других полей до его объявления.

```
int y=x;
int x=3;
```

Однако в остальном поля можно объявлять и ниже их использования:

```
class Point {
   int getX() {return x;}
```

```
   int y=getX();
   int x=3;

   public static void main (String s[]) {
      Point p=new Point();
      System.out.println(p.x+", "+p.y);
   }
}
```

Результатом будет:

```
3, 0
```

Данный пример корректен, но для понимания его результата необходимо вспомнить, что все поля класса имеют значение по умолчанию:

- для числовых полей примитивных типов это 0;
- для булевского типа это false;

Таким образом, при инициализации переменной y был использован результат метода getX(), который вернул значение по умолчанию переменной x, то есть 0. Затем переменная x получила значение 3.

## 3.4. Объявление методов

Объявление метода состоит из заголовка и тела метода. Заголовок состоит из:

- модификаторов (доступа в том числе);
- типа возвращаемого значения или ключевого слова void;
- имени метода;
- списка аргументов в круглых скобках (аргументов может не быть);
- специального throws-выражения.

Заголовок начинается с перечисления модификаторов. Для методов доступен любой из 3 возможных модификаторов доступа. Также допускается использование доступа по умолчанию.

Кроме этого, существует модификатор final, который говорит о том, что такой метод нельзя переопределять в наследниках. Можно считать, что все методы final-класса, а также все private-методы любого класса являются final.

Затем, поддерживается модификатор native. Метод, объявленный с таким модификатором, не имеет реализации на Java. Он должен быть написан на другом языке (C/C++, Fortran и т.д.) и добавлен в систему в виде загружаемой динамической библиотеки (например, DLL для Windows). Существует специальная спецификация JNI (Java Native Interface), описывающая правила создания и использования native-методов.

Такая возможность необходима для Java, поскольку многие компании имеют обширные программные библиотеки, написанные на более старых языках. Их было бы очень трудоемко и неэффективно переписывать на Java, поэтому необходима возможность подключать их в таком виде, в каком они есть. Безусловно, при этом Java-приложения теряют целый ряд

своих преимуществ, такие как переносимость, безопасность и другие. Поэтому применять JNI надо только в случае крайней необходимости.

Эта спецификация накладывает требования на имена процедур во внешних библиотеках (она составляет их из имени пакета, класса и самого native-метода), а поскольку библиотеки менять, как правило, очень неудобно, часто пишут специальные библиотеки-"обертки", к которым обращаются Java-классы через JNI, а они сами обращаются к целевым модулям, пришедшим из прошлого.

Наконец, существует еще один специальный модификатор synchronized, который будет рассмотрен в главе, описывающей потоки выполнения.

После перечисления модификаторов указывается имя (простое или составное) типа возвращаемого значения; это может быть как примитивный, так и объектный тип. Если метод не возвращает никакого значения, указывается ключевое слово void.

Затем определяется имя метода. Указанный идентификатор при объявлении становится простым именем метода. Составное имя формируется из имени класса или имени переменной объектного типа и простого имени метода. Областью видимости метода является все объявление тела класса.

Аргументы метода перечисляются через запятую. Для каждого указывается сначала тип, затем имя параметра. В отличие от объявления переменной здесь запрещается указывать два имени для одного типа:

```
// void calc (double x, y); - ошибка!
void calc (double x, double y);
```

Если аргументы отсутствуют, указываются пустые круглые скобки. Одноименные параметры запрещены. Создание локальных переменных в методе, с именами, совпадающими с именами параметров, запрещено. Для каждого аргумента можно указать ключевое слово final перед указанием его типа. В этом случае такой параметр не может менять своего значения в теле метода (формально говоря, участвовать в операции присвоения в качестве левого операнда).

```
public void process(int x, final double y) {
   x=x*x+Math.sqrt(x);
   // y=Math.sin(x); - так писать нельзя, т.к. y - final!
}
```

Как происходит изменение значений аргументов метода, рассматривается в конце этой главы.

Важным понятием является сигнатура (signature) метода. Сигнатура определяется именем метода и его аргументами (количеством, типом, порядком следования). Если для полей запрещается совпадение имен, то для методов в классе запрещено создание двух методов с одинаковыми сигнатурами.

Например,

```
class Point {
   void get() {}
```

```
   void get(int x) {}
   void get(int x, double y) {}
   void get(double x, int y) {}
}
```

Такой класс объявлен корректно. Следующие пары методов несовместимы друг с другом в одном классе:

```
void get() {}
int get() {}

void get(int x) {}
void get(int y) {}

public int get() {}
private int get() {}
```

В первом случае методы отличаются типом возвращаемого значения, которое, однако, не входит в определение сигнатуры. Стало быть, это два метода с одинаковыми сигнатурами, и они не могут одновременно появится в объявлении тела класса. Можно легко составить пример, который создал бы неразрешимую проблему для компилятора, если бы был допустим:

```
// пример вызовет ошибку компиляции
class Test {
   int get() {
      return 5;
   }
   Point get() {
      return new Point(3,5);
   }

   void print(int x) {
      System.out.println("it's int! "+x);
   }
   void print(Point p) {
      System.out.println("it's Point! "+p.x+", "+p.y);
   }

   public static void main (String s[]) {
      Test t = new Test();
      t.print(t.get()); // Двусмысленность!
   }
}
```

В классе определена запрещенная пара методов get() с одинаковыми сигнатурами и различными возвращаемыми значениями. Обратимся к выделенной строке в методе main, где возникает конфликтная ситуация, с которой компилятор не сможет справиться. Определены два метода print() (у них разные аргументы, а значит и сигнатуры, то есть это

допустимые методы), и чтобы разобраться, какой из них будет вызван, нужно знать точный тип возвращаемого значения метода get(), что невозможно.

На основе этого примера можно понять, как составлено понятие сигнатуры. Действительно, при вызове указывается имя метода и перечисляются его аргументы, причем компилятор всегда может определить их тип. Как раз эти понятия и составляют сигнатуру, и требование ее уникальности позволяет компилятору всегда однозначно определить, какой метод будет вызван.

Аналогично, в предыдущем примере вторая пара методов различается именем аргументов, которые также не входят в определение сигнатуры и делают невозможным определение, какой их двух методов должен быть вызван.

Аналогично, третья пара различается лишь модификаторами доступа, что также недопустимо.

Наконец, завершает заголовок метода throws-выражение. Оно применяется для корректной работы с ошибками в Java и будет подробно рассмотрено в соответствующей главе.

Пример объявления метода:

```
public final java.awt.Point createPositivePoint(int x, int y)
   throws IllegalArgumentException
{
   return (x>0 && y>0) ? new Point(x, y) : null;
}
```

Далее, после заголовка метода следует тело метода. Оно может быть пустым, и тогда записывается одним символом "точка с запятой". native-методы всегда имеют только пустое тело, поскольку настоящая реализация написана на другом языке.

Обычные же методы имеют непустое тело, которое описывается в фигурных скобках, что можно видеть в многочисленных примерах в этой и других главах. Если текущая реализация метода не выполняет никаких действий, тело все равно должно описываться парой пустых фигурных скобок:

```
public void empty() {}
```

Если в заголовке метода указан тип возвращаемого значения, а не void, то в теле метода обязательно должно встречаться return-выражение. При этом компилятор проводит анализ структуры метода, чтобы гарантировать, что при любых операторах ветвления возвращаемое значение будет сгенерировано. Например, следующий пример является некорректным:

```
// пример вызовет ошибку компиляции
public int get() {
   if (condition) {
      return 5;
   }
}
```

Видно, что хотя тело метода содержит return-выражение, однако не при любом развитии событий возвращаемое значение будет сгенерировано. А вот такой пример является правильным:

```
public int get() {
   if (condition) {
      return 5;
   } else {
      return 3;
   }
}
```

Конечно, значение, указанное после слова return, должно быть совместимое по типу с объявленным возвращаемым значением (это понятие подробно рассматривается в главе "Преобразование типов").

В методе без возвращаемого значения (указано void) также можно использовать выражение return без каких либо аргументов. Его можно указать в любом месте метода, в этой точке выполнение метода будет завершено:

```
public void calculate(int x, int y) {
   if (x<=0 || y<=0) {
      return; // некорректные входные значения, выход из метода
   }
   ... // основные вычисления
}
```

Выражений return (с параметром или без для методов с/без возвращаемого значения) в теле одного метода может быть сколько угодно. Однако, следует помнить, что множество точек выхода в одном методе может серьезно усложнить понимание логики его работы.

## 3.5. Объявление конструкторов

Формат объявления конструкторов похож на упрощенное объявление методов. Также выделяют заголовок и тело конструктора. Заголовок состоит, во-первых, из модификаторов доступа (никакие другие модификаторы не допустимы). Затем указывается имя класса, которое можно расценивать двояко. Можно считать, что имя конструктора совпадает с именем класса. А можно рассматривать конструктор как безымянный, а имя класса - как тип возвращаемого значения, ведь конструктор может породить только объект класса, в котором он объявлен. Это исключительно дело вкуса, так как на формате объявления никак не сказывается:

```
public class Human {
   private int age;

   protected Human(int a) {
      age=a;
   }
```

```
   public Human(String name, Human mother, Human father) {
      age=0;
   }
}
```

Как видно из примеров, далее следует перечисление входных аргументов по тем же правилам, что и для методов. Также завершает заголовок конструктора throws-выражение. Оно имеет особую важность для конструкторов, поскольку создать ошибку - это единственный способ для конструктора не создавать объект. Если конструктор выполнился без ошибок, то объект гарантированно создается.

Тело конструктора пустым быть не может и поэтому всегда описывается в фигурных скобках (для простейших реализаций скобки могут быть пустыми).

В отсутствие имени (или, что то же самое, из-за того, что у всех конструкторов одинаковое имя, совпадающее с именем класса) сигнатура конструктора определяется только набором входных параметров по тем же правилам, что и для методов. Аналогично, в одном классе допускается любое количество конструкторов, если у них различные сигнатуры.

Тело конструктора может содержать любое количество return-выражений без аргументов. Если процесс исполнения дойдет до такого выражения, то на этом месте выполнение конструктора будет завершено.

Однако логика работы конструкторов имеет и некоторые важные особенности. Поскольку при их вызове осуществляется создание и инициал»изация объекта, то становится понятно, что такой процесс не может происходить без обращения к конструкторам всех родительских классов. Поэтому вводится обязательное правило - первой строкой в конструкторе должно быть обращение к родительскому классу, которое записывается с помощью ключевого слова super.

```
public class Parent {
   private int x, y;

   public Parent() {
      x=y=0;
   }

   public Parent(int newx, int newy) {
      x=newx;
      y=newy;
   }
}

public class Child extends Parent {
   public Child() {
      super();
   }

   public Child(int newx, int newy) {
      super(newx, newy);
```

```
    }
}
```

Как видно, обращение к родительскому конструктору записывается с помощью super, за которым идет перечисление аргументов. Этот набор определяет, какой из родительских конструкторов будет использован. В приведенном примере в каждом классе есть по 2 конструктора, и каждый конструктор в наследнике обращается к аналогичному в родителе (это довольно распространенный, но, конечно, не обязательный способ).

Проследим мысленно весь алгоритм создания объекта. Он начинается при исполнении выражения с ключевым словом new, за которым следует имя класса, от которого будет порождаться объект, и набор аргументов для его конструктора. По этому набору определяется, какой именно конструктор будет использован, и происходит его вызов. Первая строка его тела содержит вызов родительского конструктора. В свою очередь, первая строка тела конструктора родителя будет содержать вызов далее к его родителю и так далее. Восхождение по дереву наследования заканчивается, очевидно, на классе Object, у которого есть единственный конструктор без параметров. Его тело пустое (в смысле, записывается парой пустых фигурных скобок), однако можно считать, что именно в этот момент JVM порождает объект, и далее начинается процесс его инициализации. Выполнение начинает обратный путь вниз по дереву наследования. У самого верхнего родителя, прямого наследника от Object, происходит продолжение исполнения конструктора со второй строки. Когда он будет полностью выполнен, необходимо перейти к следующему родителю на один уровень наследования вниз и завершить выполнение его конструктора и так далее. Наконец, можно будет вернуться к конструктору исходного класса, который был вызван с помощью new, и также продолжить его выполнение со второй строки. По его завершению объект считается полностью созданным, исполнение выражения new будет закончено, а в качестве результата будет возвращена ссылка на порожденный объект.

Проиллюстрируем этот алгоритм следующим примером:

```
public class GraphicElement {
   private int x, y; // положение на экране

   public GraphicElement(int nx, int ny) {
      super(); // обращение к конструктору родителя Object
      System.out.println("GraphicElement");
      x=nx;
      y=nx;
   }
}

public class Square extends GraphicElement {
   private int side;

   public Square(int x, int y, int nside) {
      super(x, y);
      System.out.println("Square");
       side=nside;
   }
}
```

```
public class SmallColorSquare extends Square {
   private Color color;

   public SmallColorSquare(int x, int y, Color c) {
      super(x, y, 5);
      System.out.println("SmallColorSquare");
      color=c;
   }
}
```

После выполнения выражения создания объекта на экране появится следующее:

```
GraphicElement
Square
SmallColorSquare
```

Выражение super может стоять только на первой строке конструктора. Часто можно увидеть конструкторы вообще без такого выражения. В этом случае компилятор первой строкой по умолчанию добавляет вызов родительского конструктора без параметров (super()). Если у родительского класса нет такого конструктора, выражение super обязательно должно быть записано явно (и именно на первой строке), поскольку необходима передача входных параметров.

Напомним, что, во-первых, конструкторы не имеют имени и их нельзя вызвать явно, только через выражение создания объекта. Кроме того, конструкторы не передаются по наследству. То есть, если в родительском классе объявлено пять разных полезных конструкторов, и требуется, чтобы класс-наследник имел аналогичный набор, необходимо их все заново описать.

Класс обязательно должен иметь конструктор, иначе невозможно порождать объекты ни от него, ни от его наследников. Поэтому если в классе не объявлен ни один конструктор, компилятор добавляет один по умолчанию. Это public-конструктор без параметров и с телом, описанным парой пустых фигурных скобок. Из этого следует, что такое возможно только для классов, у родителей которых объявлен конструктор без параметров, иначе возникнет ошибка компиляции. Обратите внимание, что если затем в такой класс добавляется конструктор (не важно, с параметрами или без), то конструктор по умолчанию больше не вставляется:

```
/*
 * Этот класс имеет один конструктор.
 */
public class One {
   // Будет создан конструктор по умолчанию
   // Родительский класс Object имеет
   // конструктор без параметров.
}

/*
```

```
 * Этот класс имеет один конструктор.
 */
public class Two {
   // Единственный конструктор класса Second.
   // Выражение new Second() ошибочно!
   public Second(int x) {
   }
}

/*
 * Этот класс имеет два конструктора.
 */
public class Three extends Two {
   public Three() {
      super(1); // выражение super требуется
   }

   public Three(int x) {
      super(x); // выражение super требуется
   }
}
```

В случае если класс имеет более одного конструктора, допускается в первой строке некоторых из них указывать не super, а this - выражение, вызывающее другой конструктор этого же класса.

Рассмотрим следующий пример:

```
public class Vector {
   private int vx, vy;
   protected double length;

   public Vector(int x, int y) {
      super();
      vx=x;
      vy=y;
      length=Math.sqrt(vx*vx+vy*vy);
   }

   public Vector(int x1, int y1, int x2, int y2) {
      super();
      vx=x2-x1;
      vy=y2-y1;
      length=Math.sqrt(vx*vx+vy*vy);
   }
}
```

Видно, что оба конструктора совершают практически идентичные действия, поэтому можно применить более компактный вид записи:

```
public class Vector {
   private int vx, vy;
   protected double length;

   public Vector(int x, int y) {
      super();
      vx=x;
      vy=y;
      length=Math.sqrt(vx*vx+vy*vy);
   }

   public Vector(int x1, int y1, int x2, int y2) {
      this(x2-x1, y2-y1);
   }
}
```

Большим достоинством такого метода записи является то, что удалось избежать дублирования идентичного кода. Например, если процесс инициализации объектов этого класса удлинится на один шаг (скажем, добавится проверка длины на ноль), то такое изменение надо будет внести только в одно место - в первый конструктор. Такой подход помогает избегать случайных ошибок, так как исчезает необходимость тиражировать изменения в нескольких местах.

Разумеется, такое обращение к конструкторам своего класса не должно приводить к зацикливаниям, иначе будет выдана ошибка компиляции. Цепочка this должна в итоге приводить к super, который должен присутствовать (явно или неявно) хотя бы в одном из конструкторов. После того, как отработают конструкторы всех родительских классов, будет продолжено выполнение каждого конструктора, вовлеченного в процесс создания объекта.

После выполнения выражения new Test(0) на консоли появится:

```
Test()
Test(int x)
```

В заключение рассмотрим применение модификаторов доступа для конструкторов. Может вызвать удивление возможность объявлять конструкторы как private. Ведь они нужны для порождения объектов, а к таким конструкторам ни у кого не будет доступа. Однако в ряде случаев модификатор private может быть полезен. Например:

- private-конструктор может содержать инициализирующие действия, а остальные конструкторы будут использовать его с помощью this, причем прямое обращение к этому конструктору по каким-то причинам нежелательно;
- запрет на создание объектов этого класса, например, невозможно создать экземпляр класса Math;
- реализация специального шаблона проектирования из ООП Singleton, для работы которого требуется контролировать создание объектов, что невозможно в случае наличия не-private конструкторов.

## 3.6. Инициализаторы

Наконец, последней допустимой конструкцией в теле класса является объявление инициализаторов. Записываются объектные инициализаторы очень просто - внутри фигурных скобок.

```
public class Test {
   private int x, y, z;

   // инициализатор объекта
   {
      x=3;
      if (x>0)
         y=4;
      z=Math.max(x, y);
   }
}
```

Инициализаторы не имеют имен, исполняются при создании объектов и не могут быть вызваны явно, не передаются по наследству (хотя, конечно, инициализаторы в родительском классе продолжают исполняться при создании объекта класса-наследника).

Было указано уже три вида инициализирующего кода в классах - конструкторы, инициализаторы переменных, а теперь добавились объектные инициализаторы. Необходимо разобраться в какой последовательности что выполняется, в том числе при наследовании. При создании экземпляра класса вызванный конструктор выполняется следующим образом:

- если первой строкой идет обращение к конструктору родительского класса (явное или добавленное компилятором по умолчанию), то этот конструктор исполняется;
- в случае успешного исполнения вызываются все инициализаторы полей и объекта в том порядке, в каком они объявлены в теле класса;
- если первой строкой идет обращение к другому конструктору этого же класса, то он вызывается. Повторное выполнение инициализаторов не производится.

Второй пункт имеет ряд важных следствий. Во-первых, из него следует, что в инициализаторах нельзя использовать переменные класса, если их объявление записано позже.

Во-вторых, теперь можно сформулировать наиболее гибкий подход к инициализации final-полей. Главное требование - чтобы такие поля были проинициализированы ровно один раз. Это можно обеспечить в следующих случаях:

- инициализировать поле при объявлении;
- инициализировать поле ровно один раз в инициализаторе объекта (он должен быть записан после объявления поля);
- инициализировать поле ровно один раз в каждом конструкторе, в первой строке которого стоит явное или неявное обращение к конструктору родителя. Конструктор, в первой строке которого стоит this, не может и не должен инициализировать final-поле, так как

цепочка this-вызовов приведет к конструктору с super, в котором эта инициализация обязательно присутствует.

Для иллюстрации порядка исполнения инициализирующих конструкций рассмотрим следующий пример:

```
public class Test {
   {
      System.out.println("initializer");
   }

   int x, y=getY();
   final int z;

   {
      System.out.println("initializer2");
   }

   private int getY() {
      System.out.println("getY() "+z);
      return z;
   }
   public Test() {
      System.out.println("Test()");
      z=3;
   }
   public Test(int x) {
      this();
      System.out.println("Test(int)");
      // z=4; - нельзя! final-поле уже было инициализировано
   }
}
```

После выполнения выражения new Test() на консоли появится:

```
initializer
getY() 0
initializer2
Test()
```

Обратите внимание, что для инициализации поля y вызывается метод getY(), который возвращает значение final-поля z, которое в свою очередь еще не было инициализировано. Поэтому в итоге поле y получит значение по умолчанию 0, а затем поле z получит постоянное значение 3, которое никогда уже не изменится.

После выполнения выражения new Test(3) на консоли появится:

```
initializer
getY() 0
initializer2
```

```
Test()
Test(int)
```

# 4. Дополнительные свойства классов

Рассмотрим в этом разделе некоторые особенности работы с классами в Java. Обсуждение этого вопроса продолжится в специальной главе, посвященной объектной модели в Java.

## 4.1. Метод main

Итак, виртуальная машина реализуется приложением операционной системы и запускается по обычным правилам. Программа, написанная на Java, является набором классов. Понятно, что требуется некая входная точка, с которой должно начинаться выполнение приложения.

Такой входной точкой, по аналогии с языками C/C++, является метод main(). Пример его объявления:

```
public static void main(String[] args) {
}
```

Модификатор static в этой главе не рассматривался и будет изучен позже. Он позволяет вызвать метод main(), не создавая объектов. Метод не возвращает никакого значения, хотя в C есть возможность указать код возврата из программы. В Java для этой цели есть метод System.exit(), который закрывает виртуальную машину и имеет аргумент типа int.

Аргументом метода main() является массив строк. Он заполняется дополнительными параметрами, которые были указаны при вызове метода.

```
package test.first;

public class Test {
   public static void main(String[] args) {
      for (int i=0; i<args.length; i++) {
         System.out.print(args[i]+" ");
      }
      System.out.println();
   }
}
```

Для вызова программы виртуальной машине передается в качестве параметра имя класса, у которого объявлен метод main(). Поскольку это имя класса, а не имя файла, то не должно быть указано никакого расширения (.class или .java), а расположение класса записывается через точку (разделитель имен пакетов), а не с помощью файлового разделителя. Компилятору же, напротив, передается именно имя и путь к файлу.

Если вышеприведенный модуль компиляции сохранен в файле Test.java, который лежит в директории test\first, то вызов компилятора записывается следующим образом:

```
javac test\first\Test.java
```

А вызов виртуальной машины:

```
java test.first.Test
```

Чтобы проиллюстрировать работу с параметрами, изменим строку запуска приложения:

```
java test.first.Test Hello, World!
```

Результатом работы программы будет:

```
Hello, World!
```

## 4.2. Параметры методов

Для лучшего понимания работы с параметрами методов в Java необходимо рассмотреть несколько вопросов.

Во-первых, как передаются аргументы в методы - по значению или по ссылке? С точки зрения программы вопрос формулируется, например, следующим образом. Пусть есть переменная, и она в качестве аргумента передается в некоторый метод. Могут ли произойти какие-либо изменения с этой переменной после завершения работы метода?

```
int x=3;
process(x);
print(x);
```

Предположим, используемый метод объявлен следующим образом:

```
public void process(int x) {
   x=5;
}
```

Какое значение появится на консоли после выполнения примера? Чтобы ответить на этот вопрос необходимо вспомнить, как переменные разных типов хранят свои значения в Java.

Напомним, что примитивные переменные являются истинными хранилищами своих значений, и изменения значения одной переменной никогда не скажется на значении другой. Параметр метода process(), хоть и имеет такое же имя x, на самом деле является полноценным хранилищем целочисленной величины. А потому присвоение ему значения 5 не скажется на внешних переменных. То есть, результатом примера будет 3, и аргументы примитивного типа передаются в методы по значению. Единственный способ изменить такую переменную в результате работы метода - возвращать нужные величины из метода и использовать их при присвоении:

```
public int double(int x) {
   return x+x;
}
```

```
public void test() {
   int x=3;
   x=double(x);
}
```

Перейдем к ссылочным типам.

```
public void process(Point p) {
   p.x=3;
}

public void test() {
   Point p = new Point(1,2);
   process(p);
   print(p.x);
}
```

Ссылочная переменная хранит ссылку на объект, находящийся в памяти виртуальной машины. Поэтому аргумент метода process() будет иметь своим значением ту же самую ссылку и, стало быть, ссылаться на тот же самый объект. Изменения состояния объекта, осуществленные с помощью одной ссылки, всегда видны при обращении к этому объекту с помощью другой. Поэтому результатом примера будет значение 3. Объектные значения передаются в Java по ссылке.

Однако если изменять не состояние объекта, а саму ссылку, то результат будет другим:

```
public void process(Point p) {
   p = new Point(4,5);
}

public void test() {
   Point p = new Point(1,2);
   process(p);
   print(p.x);
}
```

В этом примере аргумент метода process() после присвоения начинает ссылаться на другой объект, нежели исходная переменная p, а значит, результатом примера станет значение 1. Можно сказать, что ссылочные величины передаются по значению, но значением является именно ссылка на объект.

Теперь можно уточнить, что означает возможность объявлять параметры методов и конструкторов как final. Поскольку изменения значений параметров (но не объектов, на которые они ссылаются) никак не сказываются на переменных вне метода, то модификатор final говорит лишь о том, что значение этого параметра не будет меняться на протяжении работы метода. Разумеется, для аргумента final Point p выражение p.x=5 является допустимым (запрещается p=new Point(5, 5)).

## 4.3. Перегруженные методы

Перегруженными (overloading) методами называются методы одного класса с одинаковыми именами. Сигнатуры у них должны быть различными, и различие может быть только в наборе аргументов.

Если в классе параметры перегруженных методов заметно различаются, например, у одного метода один параметр, у другого - два, то для Java это совершенно независимые методы, и совпадение их имен может служить только для повышения наглядности работы класса. Каждый вызов в зависимости от количества параметров однозначно адресуется к тому или иному методу.

Однако, если количество параметров одинаковое, а типы их различаются незначительно, но при вызове может сложиться двойственная ситуация, когда несколько перегруженных методов одинаково хорошо подходят для использования. Например, если объявлены типы Parent и Child, где Child расширяет Parent, то для следующих двух методов:

```
void process(Parent p, Child c) {}
void process(Child c, Parent p) {}
```

можно сказать, что они допустимы, их сигнатуры различаются. Однако при вызове

```
process(new Child(), new Child());
```

обнаруживается, что оба метода одинаково годятся для использования. Другой пример, методы:

```
process(Object o) {}
process(String s) {}
```

и примеры вызовов:

```
process(new Object());
process(new Point(4,5));
process("abc");
```

Легко видеть, что для первых двух вызовов подходящим является только первый метод, и именно он будет вызван. Для последнего же вызова подходят оба перегруженных метода, однако класс String является более "специфичным", или узким, чем класс Object. Действительно, значения типа String можно передавать в качестве аргументов типа Object, обратное же неверно. Компилятор попытается отыскать наиболее специфичный метод, подходящий для указанных параметров, и вызовет именно его. Поэтому при третьем вызове будет использован второй метод.

Однако для предыдущего примера такой подход не дает однозначного ответа. Оба метода одинаково специфичны для указанного вызова, и поэтому возникнет ошибка компиляции. Необходимо, используя явное приведение, указать компилятору, какой метод необходимо использовать:

```
process((Parent)(new Child()), new Child());
```

```
// или
process(new Child(),(Parent)(new Child()));
```

Это верно и в случае использования значения null:

```
process((Parent)null, null);
// или
process(null,(Parent)null);
```

# 5. Заключение

В этой главе началось рассмотрение ключевой конструкции языка Java – объявление класса.

Первая тема посвящена средствам разграничения доступа. Главный вопрос – для чего этот механизм вводится в практически каждом современном языке высокого уровня. Необходимо понимать, что он предназначен не для обеспечения «безопасности» или «защиты» объекта от неких неправильных действий. Самая важная задача – разделить внешний интерфейс класса и детали его реализации с тем, чтобы в дальнейшем воспользоваться такими преимуществами ООП, как инкапсуляция и модульность.

Затем были рассмотрены все 4 модификатора доступа, а также возможность их применения для различных элементов языка. Проверка уровня доступа проверяется уже на момент компиляции и запрещает лишь явное использование типов. Например, с ними все же можно работать через их более открытых наследников.

Объявление класса состоит из заголовка и тела класса. Формат заголовка был подробно описан. Для изучения тела класса необходимо вспомнить понятие элементов (members) класса. Ими могут быть поля, методы и внутренние типы. Для методов важным понятием является сигнатура.

Кроме того, в теле класса объявляются конструкторы и инициализаторы. Поскольку они не являются элементами, к ним нельзя обратиться явно, они вызываются самой виртуальной машиной. Также они не передаются по наследству.

Дополнительно был рассмотрен метод main, который вызывается при старте виртуальной машины. Далее описываются тонкости, возникающие при передаче параметров, и связанный с этим вопрос о перегруженных методах.

Рассмотрение классов в Java будет продолжено в следующих главах.

# 6. Контрольные вопросы

6-1. Какие модификаторы позволяют обращаться к элементу из классов того же пакета?

a.) Модификаторы public и protected, а также доступ по умолчанию (без указания модификатора).

6-2. Если в классе заводится новый элемент, и пока нет никаких факторов, позволяющих выбрать тот или иной модификатор доступа. Какой модификатор использовать в таком случае?

а.) Лучше использовать private. Раз не требуется внести этот элемент в открытый интерфейс класса, значит, стоит его отнести к реализации. В дальнейшем при необходимости всегда можно расширить уровень доступа. Обратное действие – сужение – запрещено.

6-3. Пусть класс User описывает пользователя системы. В качестве имени используется его e-mail адрес, который всем известен, а пароль, конечно, не должен быть никому доступен, кроме самого пользователя. Корректна ли следующая реализация?

```
public class User {
   public String login; // открытый e-mail
   private String password; // закрытый пароль
}
```

а.) Хотя код корректен с точки зрения компилятора, он не верен с точки зрения ООП дизайна. В предложенном варианте любой класс может изменить значение login у пользователя, что вряд ли соответствует задуманному алгоритму работы системы. Рекомендуется закрыть доступ к полю login извне и добавить метод для чтения, который бы возвращал значение login. Например:

```
public class User {
   private String login; // открытый e-mail
   private String password; // закрытый пароль

   public String getLogin(){
      return login;
   }
}
```

6-4. Корректен ли следующий код?

```
public class Test {
   private int id;

   public Test(int i) {
      id=i;
   }

   public static boolean test(Test t, int id) {
      return t.id==id;
   }
}
```

а.) Да. Метод test является методом класса Test, а значит, имеет доступ к полю id любого объекта этого класса.

6-5. Из каких частей состоит заголовок объявления класса? Тело класса?

a.) Заголовок класса (именно в таком порядке):

1. опциональные модификаторы (public, abstract, final)
2. ключевое слово class и имя класса
3. опционально ключевое слово extends и имя суперкласса
4. опционально ключевое слово implements и список имен реализуемых интерфейсов
5. тело класса в фигурных скобках

Тело класса (в произвольном порядке):

- поля
- методы
- внутренние типы (классы и интерфейсы)
- конструкторы
- инициализаторы

6-6. Если метод использует переменную класса, должна ли она быть объявлена выше объявления метода?

a.) Нет. Областью видимости переменной класса является все объявление тела класса.

6-7. Из каких частей состоит заголовок объявления метода? Какие части обязательные?

a.) Заголовок метода (именно в таком порядке):

1. опциональные модификаторы (доступа public|private|protected и прочие static, final, native, synchronized)
2. тип возвращаемого значения или void, если его нет
3. имя метода
4. список типов и имен аргументов в круглых скобках
5. опционально throws выражение

Обязательно должны присутствовать имя метода, тип возвращаемого значения (или void), перечисление аргументов в круглых скобках (или пустые скобки, если их нет).

6-8. Пусть класс должен обладать методом со следующими вариациями. Метод должен принимать в качестве аргумента дробное значение типа double или float и возвращать результат округления. В случае float нужно возвращать либо «короткое» значение (byte), либо «полное» (int). Аналогично для double – int или long соответственно. Сколько методов должно быть объявлено в таком классе, и каковы их сигнатуры?

a.) Если формально следовать вопросу, то должно быть объявлено 4 метода:

```
byte round(float x) { ... }
int roundToInt(float x) { ... }
```

```
int round(double x) { ... }
long roundToLong(double x) { ... }
```

Как видно, методы имеют различные имена, поскольку два из них отличаются только возвращаемыми значениями, а объявлять несколько методов с одинаковыми сигнатурами нельзя.

Однако можно заметить, что сигнатуры 2 и 3 методов очень близки, поэтому их можно объединить. Но и здесь нужна осторожность. Если просто убрать второй метод, то при вызове, например, round(1f) будет вызываться 1 метод, а не третий, как ожидалось. Поэтому объединение возможно одним из двух способов. Нужно либо для округления от float к int записывать вызов как round((double)1f), либо назвать 3 метод уникальным именем, чтобы он не был перегружен с первым.

6-9. Может ли класс не иметь ни одного конструктора?

a.) Нет, такой класс объявить невозможно. Даже если не указать ни одного конструктора, компилятор добавит конструктор по умолчанию. Если возникнет противоречие с родительским классом (ведь конструктор по умолчанию требует наличие доступного конструктора без параметров у родительского класса), то возникнет ошибка, и класс не будет скомпилирован.

6-10. Что появится на консоли при вызове конструктора следующего класса?

```
class Test {
   private long id=getId();
   private String name=getName();
   private String login;

   public Test(int domain) {
      login=domain+" "+name;
      System.out.println(login);
   }

   private static long getId() {
      int id = 3;
      System.out.println(id);
      return id;
   }

   private String getName() {
      String name="name"+getId();
      System.out.println(name);
      return name;
   }
}
```

а.) Результатом будет:

```
3
3
name3
5 name3
```

Первой будет проинициализирована переменная id (первая 3 на консоли). Затем начнет инициализироваться поле name, которое в свою очередь опять вызовет метод getId() (вторая 3), после чего напечатает name3. Затем в самом конструкторе распечатается значение 5 name3.

6-11. Как записывается заголовок метода main?

а.) `public static void main(String[] args)`

6-12. Может ли изменится содержимое переменной типа String, если передать ее в качестве аргумента при вызове метода?

а.) Нет, так как класс String, а стало быть, и его объекты, не изменяемы.

# Программирование на Java

## Лекция 7. Преобразование типов

### 20 января 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Эта лекция посвящена вопросам преобразования типов. Поскольку Java – язык строго типизированный, компилятор и виртуальная машина всегда следят за работой с типами, гарантируя надежность выполнения программы. Однако во многих случаях то или иное преобразование необходимо осуществить для реализации логики прграммы. С другой стороны, некоторые безопасные переходы между типами Java позволяет осуществлять неявным для разработчика образом, что может привести к неверному пониманию работы программмы.

В лекции рассматриваются все виды преобразований, а затем все ситуации в программе, где они могут применяться. В заключение приводится начало классификации типов переменных и типов значений, которые они могут хранить. Этот вопрос будет детализироваться в будущих лекциях.

# Оглавление

# Лекция 7. Преобразование типов

## Содержание лекции.



## 1. Введение

Как уже говорилось, Java является строго типизированным языком, что означает, что каждое выражение и каждая переменная имеет строго определенный тип уже на момент компиляции. Тип устанавливается на основе структуры применяемых выражений и типов литералов, переменных и методов, используемых в этих выражениях.

Например:

```
long l=3;
l = 5+'A'+l;
print("l="+Math.round(l/2F));
```

Опишем, как в этом примере компилятор устанавливает тип каждого выражения, и какие преобразования (conversion) типов необходимо осуществить при каждом действии:

- В первой строке литерал 3 имеет тип по умолчанию, то есть int. При присвоении этого значения переменной типа long необходимо провести преобразование.
- Во второй строке сначала производится сложение значений типа int и char. Второй аргумент также будет преобразован, чтобы операция проводилось с точностью в 32 бита. Второй оператор сложения опять потребует преобразования, так как наличие переменной l увеличивает точность до 64 бит.
- В третьей строке сначала будет выполнена операция деления, для чего значение long надо будет привести к типу float, так как второй операнд - дробный литерал. Результат будет передан в метод Math.round, который произведет математическое округление, и вернет целочисленный результат типа int. Это значение необходимо преобразовано в текст, чтобы осуществить дальнейшую конкатенацию строк. Как рассматривается ниже, эта операция проводится в два этапа - сначала простой тип приводится к объектному классу-обертке (в данном случае int к Integer), а затем у полученного объекта вызывается метод toString(), что дает преобразование к строке.

Данный пример показывает, что даже простые строки могут содержать многочисленные преобразование, зачастую незаметные для разработчика. Часто бывают и такие случаи, когда программисту необходимо явно изменить тип некоторого выражения или переменной, например, чтобы воспользоваться подходящим методом или конструктором.

Вспомним уже рассмотренный пример:

```
byte b=1;
byte c=(byte)-b;
int i=c;
```

Здесь во второй строке необходимо провести явное преобразование, чтобы присвоить значение типа byte переменной типа int. В третьей же строке обратное приведение производится автоматически, неявным для разработчика образом.

Рассмотрим сначала, какие переходы между различными типами возможно осуществить.

# 2. Виды приведений

В Java возможны семь видов приведений:

- тождественное (identity);
- расширение примитивного типа (widening primitive);
- сужение примитивного типа (narrowing primitive);
- расширение объектного типа (widening reference);
- сужение объектного типа (narrowing reference);
- преобразование к строке (String);
- запрещенные преобразования (forbidden).

Рассмотрим их по отдельности.

## 2.1. Тождественное преобразование

Самым простым является тождественное преобразование. В Java преобразование выражения любого типа к точно такому же типу всегда допустимо и успешно выполняется.

Зачем нужно тождественное приведение? Есть две причины для того, чтобы выделить такое преобразование в особый вид.

Во-первых, с теоретической точки зрения теперь можно утверждать, что любой тип в Java может участвовать в преобразовании, хотя бы в тождественном. Например, примитивный тип boolean нельзя привести ни к какому другому типу, кроме него самого.

Вторая причина носит более практический смысл. Иногда в Java могут встретиться такие выражения, как длинный последовательный вызов методов:

```
print(getCity().getStreet().getHouse().getFlat().getRoom());
```

При исполнении такого выражения сначала вызывается первый метод getCity(). Можно предположить, что возвращаемым значением будет объект класса City. У этого объекта далее будет вызван следующий метод getStreet(). Чтобы узнать, значение какого типа он вернет, необходимо посмотреть описание класса City. У этого значения будет вызван следующий метод (getHouse()), и так далее. Чтобы узнать результирующий тип всего выражения, необходимо просмотреть описание каждого метода и класса.

Компилятор легко справится с такой задачей, однако разработчику может потребоваться большое количество усилий, чтобы проследить всю цепочку. В этом случае можно воспользоваться тождественным преобразованием, сделав приведение к точно такому же типу. Это ничего не изменит в структуре программы, но значительно облегчит чтение кода:

```
print((MyFlatImpl)(getCity().getStreet().getHouse().getFlat()));
```

## 2.2. Преобразование примитивных типов (расширение и сужение)

Легко видеть, что следующие четыре вида приведений легко представляются в виде таблицы.

| | |
|---|---|
| простой тип расширение | ссылочный тип расширение |
| простой тип сужение | ссылочный тип сужение |

Что это все означает? Начнем по порядку. Для простых типов расширение означает, что осуществляется переход от менее емкого типа к более емкому. Например, от типа byte (длина 1 байт) к типу int (длина 4 байта). Такие преобразования безопасны в том смысле, что новый тип всегда гарантированно вмещает в себя все данные, которые хранились в старом типе, и таким образом не происходит потери данных. Именно поэтому компилятор осуществляет его сам, незаметно для разработчика:

```
byte b=3;
int i=b;
```

В последней строке значение переменной b типа byte будет преобразовано к типу переменной i (то есть, int) автоматически, никаких специальных действий для этого предпринимать не надо.

Следующие 19 преобразований являются расширяющими:

- от byte к short, int, long, float, double
- от short к int, long, float, double
- от char к int, long, float, double
- от int к long, float, double
- от long к float, double
- от float к double

Обратите внимание, что нельзя провести преобразование к типу char от типов меньшей или равной длины (byte, short) или, наоборот, к short от char без потери данных. Это связано с тем, что char является беззнаковым в отличие от остальных целочисленных типов.

Тем не менее, следует помнить, что даже при расширении данные все-таки могут быть искажены в особых случаях. Они уже рассматривались в прошлой главе, это приведение значений int к типу float и приведение значений типа long к типу float или double. Хотя эти дробные типы вмещают гораздо большие числа, чем соответствующие целые, но у них меньше значащих знаков.

Повторим этот пример:

```
long l=111111111111L;
float f = l;
l = (long) f;
print(l);
```

Результатом будет:

```
111111110656
```

Обратное преобразование - сужение - означает, что переход осуществляется от более емкого типа к менее емкому. При таком преобразовании есть риск потерять данные. Например, если число типа int было больше 127, то при приведении его к byte значения битов старше восьмого будут потеряны. В Java такое преобразование должно совершаться явным образом, т.е. программист в коде должен явно указать, что он намеревается осуществить такое преобразование и готов идти на потерю данных.

Следующие 23 преобразования являются сужающими:

- от byte к char
- от short к byte, char
- от char к byte, short
- от int к byte, short, char
- от long к byte, short, char, int

- от float к byte, short, char, int, long
- от double к byte, short, char, int, long, float

При сужении целочисленного типа к более узкому целочисленному все старшие биты, не попадающие в новый тип, просто отбрасываются. Не производится никакого округления или других усилий для получения более корректного результата:

```
  print((byte)383);
 print((byte)384);
 print((byte)-384);
```

Результатом будет:

```
127
-128
-128
```

Видно, что знаковый бит при сужении не оказал никакого влияния, так как был просто отброшен - результат приведения обратных чисел (384 и -384) оказался одинаковым. Стало быть, может быть потеряно не только точное абсолютное значение, но и знак величины.

Это верно и для типа char:

```
char c=40000;
print((short)c);
```

Результатом будет:

```
-25536
```

Сужение дробного типа к целочисленному является более сложной процедурой. Она проводится в два этапа.

На первом шаге дробное значение преобразуется в long, если целевым типом является long, или в int в противном случае (целевой тип byte, short, char или int). Для этого исходное дробное число сначала математически округляется в сторону нуля, то есть дробная часть просто отбрасывается.

Например, число 3.84 будет округлено до 3, а -3.84 превратится в -3. При этом могут возникнуть особые случаи:

- если исходное дробное значение является NaN, то результатом первого шага будет 0 выбранного типа (т.е. int или long);
- если исходное дробное значение является положительной или отрицательной бесконечностью, то результатом первого шага будет соответственно максимально или минимально возможное значение для выбранного типа (т.е. для int или long);
- наконец, если дробное значение было конечной величины, но в результате округления получилось слишком большое по модулю число для выбранного типа (т.е. для int или long), то, также как и в предыдущем пункте, результатом первого шага будет

соответственно максимально или минимально возможное значение этого типа. Если же результат округления укладывается в диапазон значений выбранного типа, то он и будет результатом первого шага.

На втором шаге производится дальнейшее сужение от выбранного целочисленного типа к целевому, если таковое требуется, то есть может иметь место дополнительное преобразование от int к byte, short или char.

Проиллюстрируем описанный алгоритм преобразованием от бесконечности ко всем целочисленным типам:

```
float fmin = Float.NEGATIVE_INFINITY;
float fmax = Float.POSITIVE_INFINITY;
print("long: " + (long)fmin + ".." + (long)fmax);
print("int: " + (int)fmin + ".." + (int)fmax);
print("short: " + (short)fmin + ".." + (short)fmax);
print("char: " + (int)(char)fmin + ".." + (int)(char)fmax);
print("byte: " + (byte)fmin + ".." + (byte)fmax);
```

Результатом будет:

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

Значения long и int вполне очевидны - дробные бесконечности преобразовались в соответственно минимально и максимально возможные значения этих типов. Результат для следующих трех типов (short, char, byte) есть по сути дальнейшее сужение значений, полученных для int, согласно второму шагу процедуры преобразования. А делается, как было описано, просто отбрасыванием старших битов. Вспомним, что минимально возможное значение в битовом виде представляется как 1000..000 (всего 32 бита для int, то есть единица и 31 ноль). Максимально возможное - 1111..111 (32 единицы). Отбрасывая старшие биты получаем для отрицательной бесконечности результат 0, одинаковый для всех 3 типов. Для положительной же бесконечности получаем результат, все биты которого равняются 1. Для знаковых типов byte и short такая комбинация рассматривается как -1, а для беззнакового char - как максимально возможное значение, то есть 65.535.

Может сложиться впечатление, что для char приведение дает точное значение. Однако, это был частный случай - отбрасывание битов в большинстве случаев все же дает искажение. Например сужение дробного значения 2 миллиарда:

```
float f=2e9f;
print((int)(char)f);
print((int)(char)-f);
```

Результатом будет:

```
37888
27648
```

Обратите внимание на двойное приведение для значений типа char в двух последних примерах. Понятно, что преобразование от char к int не приводит к потере точности, но позволяет распечатывать не символ, а его числовой код, что более удобно для анализа.

В заключение еще раз обратим внимание на то, что примитивные значения типа boolean могут участвовать только в тождественных преобразованиях.

## 2.3. Преобразование ссылочных типов (расширение и сужение)

Переходим к ссылочным типам. Преобразование объектных типов лучше всего иллюстрируется с помощью дерева наследования. Рассмотрим небольшой пример наследования:

```
// Объявляем класс Parent
class Parent {
   int x;
}

// Объявляем класс Child, и наследуем
// его от класса Parent
class Child extends Parent {
   int y;
}

// Объявляем второго наследника
// класса Parent - класс Child2
class Child2 extends Parent {
   int z;
}
```

В каждом классе объявлено поле с уникальным именем. Будем рассматривать это поле как пример набора уникальных свойств, присущих некоторому объектному типу.

Объявленных 3 класса могут порождать 3 вида объектов. Объекты класса Parent обладают только одним полем x, а значит только ссылки типа Parent могут ссылаться на такие объекты. Объекты класса Child обладают полем y и полем x, полученным по наследству от класса Parent. Стало быть, на такие объекты могут указывать ссылки типа Child или Parent. Второй случай уже иллюстрировался следующим примером:

```
Parent p = new Child();
```

Обращаем внимание, что с помощью такой ссылки p можно обращаться лишь к полю x созданного объекта. Поле y не доступно, так как компилятор, проверяя корректность выражения p.y, не может предугадать, что ссылка p будет указывать на объект типа Child во время исполнения программы. Он анализирует лишь тип самой переменной, а она

объявлена как Parent, но в этом классе нет поля y, что и станет причиной возникновения ошибки компиляции.

Аналогично, объекты класса Child2 обладают полем z и полем x, полученным по наследству от класса Parent. Стало быть, на такие объекты могут указывать ссылки типа Child2 или Parent.

Таким образом, ссылки типа Parent могут указывать на объект любого из трех рассматриваемых типов, а ссылки типа Child и Child2 - только на объекты точно такого же типа. Теперь можно перейти к преобразованию ссылочных типов на основе такого дерева наследования.

Расширение означает переход от более конкретного типа к менее конкретному, т.е. переход от детей к родителям. В нашем примере преобразование от любого наследника (Child, Child2) к родителю (Parent) есть расширение, переход к более общему типу. Подобно случаю с примитивными типами, этот переход производится самой JVM при необходимости и незаметен для разработчика, то есть не требует никаких дополнительных усилий, так как он всегда успешен: всегда можно обращаться к объекту, порожденному от наследника, по типу его родителя.

```
Parent p1=new Child();
Parent p2=new Child2();
```

В обеих строках переменным типа Parent присваивается значение другого типа, а значит, происходит преобразование. Поскольку это расширение, оно производится автоматически и всегда успешно.

Обращаем внимание, что при подобном преобразовании с самим объектом ничего не происходит. Не смотря на то, что, например, поле y класса Child теперь больше не доступно, это не говорит о том, что оно исчезло. Такое существенное изменение структуры объекта невозможно. Он был порожден от класса Child, и навсегда сохранит все его свойства. Изменился лишь тип ссылки, через которую идет обращение к объекту. Эту ситуацию можно условно сравнить с рассматриванием некоего предмета через подзорную трубу. Если перейти от трубы с большим увеличением к более слабой, то видимых деталей станет меньше, но сам предмет, конечно, никак от этого не изменится.

Следующие преобразования являются расширяющими:

- от класса A к классу B, если A наследуется от B (важным частным случаем является преобразование от любого ссылочного типа к Object);
- от null-типа к любому объектному типу.

Второй случай иллюстрируется следующим примером:

```
Parent p=null;
```

Пустая ссылка null не обладает каким-либо конкретным ссылочным типом, поэтому иногда говорят о специальном null-типе. Однако на практике важно, что такое значение можно прозрачно преобразовать к любому объектному типу.

С изучением остальных ссылочных типов (интерфейсов и массивов) этот список будет расширяться.

Обратный переход, то есть движение по дереву наследования вниз, к наследникам, является сужением. Например для рассматриваемого случая, переход от ссылки типа Parent, которая может ссылаться на объекты трех классов, к ссылке типа Child, которая может ссылаться на объекты лишь одного из трех классов, очевидно, является сужением. Такой переход может оказаться невозможным. Если ссылка типа Parent ссылается на объект типа Parent или Child2, то переход к Child невозможен, ведь в обоих случаях объект не обладает полем y, которое объявлено в классе Child. Поэтому при сужении разработчику необходимо явным образом указывать на то, что необходимо попытаться провести такое преобразование. JVM во время исполнения проверит корректность перехода. Если он возможен, преобразование будет проведено. Если же нет - возникнет ошибка.

```
Parent p=new Child();
Child c=(Child)p; // преобразование будет успешным.
Parent p2=new Child2();
Child c2=(Child)p2; // во время исполнения возникнет ошибка!
```

Чтобы проверить, возможен ли желаемый переход, можно воспользоваться оператором instanceof:

```
Parent p=new Child();
if (p instanceof Child) {
   Child c = (Child)p;
}
Parent p2=new Child2();
if (p2 instanceof Child) {
   Child c = (Child)p2;
}
Parent p3=new Parent();
if (p3 instanceof Child) {
   Child c = (Child)p3;
}
```

В данном примере ошибок не возникнет. Первое преобразование возможно, и оно будет осуществлено. Во втором и третьем случаях условия операторов if не сработают, и попыток некорректного перехода не будет.

На данный момент можно назвать лишь одно сужающее преобразование:

- от класса A к классу B, если B наследуется от A (важным частным случаем является сужение типа Object до любого другого ссылочного типа);

С изучением остальных ссылочных типов (интерфейсов и массивов) этот список будет расширяться.

## 2.4. Преобразование к строке

Это преобразование уже не раз упоминалось. Любой тип может быть приведен к строке, т.е. к экземпляру класса String. Это преобразование является исключительным в силу того, что охватывает абсолютно все типы, в том числе и boolean, про который говорилось, что он не может участвовать ни в каком другом приведении, кроме тождественного.

Напомним, как преобразуются различные типы:

- Числовые типы записываются в текстовом виде без потери точности представления. Формально, такое преобразование происходит в два этапа. Сначала на основе примитивного значения порождается экземпляр соответствующего класса-обертки, а затем у него вызывается метод toString(). Но поскольку эти действия никак не заметны снаружи, то многие JVM оптимизируют их и преобразуют примитивные значения в текст напрямую.
- Булевская величина приводится к строке "true" или "false" в зависимости от значения.
- Для объектных величин вызывается метод toString(). Если метод возвращает null, то результатом будет строка "null".
- Для null-значения генерируется строка "null".

## 2.5. Запрещенные преобразования

Не все переходы между произвольными типами допустимы. Например, к запрещенным преобразованиям относятся: переходы от любого ссылочного типа к примитивному, от примитивного - к ссылочному (кроме преобразований к строке). Уже упоминавшийся пример - тип boolean нельзя привести ни к какому другому типу, отличному от boolean (как обычно - за исключением приведения к строке). Затем, невозможно привести друг к другу типы, находящиеся не на одной, а на соседних ветвях дерева наследования. В примере, который рассматривался для иллюстрации преобразований ссылочных типов, переход от Child к Child2 запрещен. В самом деле, ссылка типа Child может указывать на объекты, порожденные только от класса Child или его наследников. Что исключает вероятность того, что объект будет совместим с типом Child2.

На этом список запрещенных преобразований не исчерпывается. Однако, он довольно велик, и в то же время все варианты довольно очевидны, и поэтому подробно рассматриваться не будут. Желающие могут получить полную информацию из спецификации.

Разумеется, попытка осуществить запрещенное преобразование вызовет ошибку компиляции.

# 3. Применение приведений

Теперь, когда рассмотрены все виды преобразований, перейдем к ситуациям в коде, где могут встретиться или потребоваться приведения.

Такие ситуации могут быть сгруппированы следующим образом:

- присвоение значений переменным (assignment). Не все переходы допустимы при таком преобразовании - ограничения выбраны таким образом, чтобы не могла возникнуть ошибочная ситуация.
- вызов метода. Это преобразование применяется к аргументам вызываемого метода или конструктора. Допускаются почти те же переходы, что и для присвоения значений. Такое приведение никогда не порождает ошибок. Также приведение осуществляется при возвращении значения из метода.

- явное приведение. В этом случае явно указывается, к какому типу требуется привести исходное значение. Допускаются все виды преобразований кроме приведений к строке и запрещенных. Может возникать ошибка времени исполнения программы.
- оператор конкатенации производит преобразование к строке своих аргументов.
- числовое расширение (numeric promotion). Числовые операции могут потребовать изменения типа аргумента(ов). Это преобразование имеет особое название - расширение (promotion), так как выбор целевого типа может зависеть не только от исходного значения, но и от второго аргумента операции.

Рассмотрим все случаи более подробно.

## 3.1. Присвоение значений

Эти ситуации неоднократно применялись в этой главе для иллюстрации видов преобразования. Приведение может потребоваться, если переменной одного типа присваивается значение другого типа. Возможны следующие комбинации.

Если сочетание этих двух типов образует запрещенное приведение, возникнет ошибка. Например, примитивные значения нельзя присваивать объектным переменным, включая следующие примеры:

```
// пример вызовет ошибку компиляции

// примитивное значение нельзя
// присвоить объектной переменной
Parent p = 3;

// приведение к классу-обертке также запрещено
Long l=5L;

// универсальное приведение к строке
// возможно только для оператора +
String s=true;
```

Далее, если сочетание этих двух типов образует расширение (примитивных или ссылочных типов), то оно будет осуществлено автоматически, неявным для разработчика образом:

```
int i=10;
long l=i;
Child c = new Child();
Parent p=c;
```

Если же сочетание оказывается сужением, то возникает ошибка компиляции, такой переход не может быть проведен неявно:

```
// пример вызовет ошибку компиляции
int i=10;
short s=i; // ошибка! сужение!
```

```
Parent p = new Child();
Child c=p; // ошибка! сужение!
```

Как уже упоминалось, в подобных случаях необходимо преобразование делать явно:

```
int i=10;
short s=(int)i;
Parent p = new Child();
Child c=(Child)p;
```

Более подробно явное сужение рассматривается ниже.

Здесь может вызвать удивление следующая ситуация, которая не порождает ошибок компиляции:

```
byte b=1;
short s=2+3;
char c=(byte)5+'a';
```

В первой строке переменной типа byte присваивается значение целочисленного литерала типа int, что является сужением. Во второй строке переменной типа short присваивается результат сложения двух литералов типа int, а тип этой суммы также int. Наконец, в третьей строке переменной типа char присваивается результат сложения числа 5, приведенного к типу byte, и символьного литерала.

Однако, все эти примеры корректны. Для удобства разработчика компилятор проводит дополнительный анализ при присвоении значений переменным типа byte, short и char. Если таким переменным присваивается величина типа byte, short, char или int, причем ее значение может быть получено уже на момент компиляции, и оказывается, что это значение укладывается в диапазон типа переменной, то явного приведения не требуется. Если бы такой возможности не было, то пришлось бы писать так:

```
byte b=(byte)1; // преобразование необязательно
short s=(short)(2+3); // преобразование необязательно
char c=(char)((byte)5+'a'); // преобразование необязательно

// преобразование необходимо, так как
// число 200 не укладывается в тип byte
byte b2=(byte)200;
```

## 3.2. Вызов метода

Это приведение возникает в случае, когда вызывается метод с объявленными параметрами одних типов, а при вызове передаются аргументы других типов. Объявление методов рассматривается в следующих главах курса, однако такой простой пример вполне понятен:

```
// объявление метода с параметром типа long
void calculate(long l) {
   ...
}
```

```
void main() {
   calculate(5);
}
```

Как видно, при вызове метода передается значение типа int, а не long, как определено в объявлении этого метода.

Здесь компилятор предпринимает ровно те же шаги, что и при приведении при присвоении значений переменным. Если типы образуют запрещенное преобразование, то будет ошибка.

```
// пример вызовет ошибку компиляции

void calculate(long l) {
   ...
}

void main() {
   calculate(new Long(5)); // здесь будет ошибка
}
```

Если сужение, то компилятор не сможет осуществить приведение, и потребуются явные указания.

```
void calculate(int l) {
   ...
}

void main() {
   long l=5;
   // calculate(l); // сужение! так будет ошибка.
   calculate((int)l); // корректный вызов
}
```

Наконец, в случае расширения, компилятор осуществит приведение сам, как и было показано в примере в начале этого раздела.

Надо отметить, что в отличие от ситуации присвоения, при вызове методов компилятор не производит преобразований примитивных значений от byte, short, char или int к byte, short или char. Это привело бы к усложнению работы с перегруженными методами. Например:

```
// пример вызовет ошибку компиляции

// объявляем перегруженные методы
// с аргументами (byte, int) и (short, short)
int m(byte a, int b) { return a+b; }
int m(short a, short b) { return a-b; }

void main() {
```

```
   print(m(12, 2)); // ошибка компиляции!
}
```

В этом примере компилятор выдаст ошибку, так как при вызове аргументы имеют тип (int, int), а метода с такими параметрами нет. Если бы компилятор проводил преобразование для целых величин, подобно ситуации с присвоением значений, то пример стал бы корректным, но пришлось бы предпринимать дополнительные усилия, чтобы указать, какой из двух возможных перегруженных методов хотелось бы вызвать.

Затем, аналогичное преобразование требуется при возвращении значения из метода, если тип результата и заявленный тип возвращаемого значения не совпадают.

```
long get() {
   return 5;
}
```

Хотя в выражении return указан целочисленный литерал типа int, во всех местах, где будет вызван этот метод, будет получено значение типа long. Для такого преобразования действуют все правила, что и для присвоения значения.

В заключение рассмотрим пример, включающий в себя все рассмотренные случаи преобразования:

```
short get(Parent p) {
   return 5+'A'; // приведение при возвращении значения
}

void main() {
   long l = // приведение при присвоении значения
   get(new Child()); // приведение при вызове метода
}
```

## 3.3. Явное приведение

Явное приведение уже многократно использовалось в примерах. При таком преобразовании слева от выражения, тип значения которого необходимо преобразовать, в круглых скобках указывается целевой тип. Если преобразование пройдет успешно, то результат будет точно указанного типа. Примеры:

```
(byte)5
(Parent)new Child()
(Flat)getCity().getStreet().getHouse().getFlat()
```

Если комбинация типов образует запрещенное преобразование, возникает ошибка компиляции. Допускаются тождественные преобразования, расширения простых и объектных типов, сужения простых и объектных типов. Первые три выполнятся успешно всегда. Последние два могут стать причиной ошибки исполнения, если значения оказались не совместимыми. Как следствие, выражение null всегда может быть успешно преобразовано к любому ссылочному типу.

Легко найти способ все-таки закодировать запрещенное преобразование.

```
Child c=new Child();
// Child2 c2=(Child2)c; // запрещенное преобразование
Parent p=c; // расширение
Child2 c2=(Child2)p; // сужение
```

Такой код будет успешно скомпилирован, однако, разумеется, при исполнении он всегда будет генерировать ошибку в последней строке. "Обманывать" компилятор смысла нет.

## 3.4. Оператор конкатенации строк

Этот оператор уже был подробно рассмотрен. Если обоими его аргументами являются строки, то происходит обычная конкатенация. Если же тип String имеет лишь один из аргументов, то второй необходимо преобразовать в текст. Это единственная операция, при которой производится универсальное приведение любого значения к типу String.

Это одно из свойств, выделяющих класс String из всех остальных.

Правила преобразования уже были подробно описаны в этой главе, а оператор конкатенации рассматривался в главе "Типы данных".

Небольшой пример:

```
int i=1;
double d=i/2.;
String s="text";
print("i="+i+", d="+d+", s="+s);
```

Результатом будет:

```
i=1, d=0.5, s=text
```

## 3.5. Числовое расширение

Наконец, последний вид преобразований применяется при числовых операциях, когда требуется привести аргумент(ы) к типу длиной в 32 или 64 бита для проведения вычислений. Таким образом, при числовом расширении осуществляется только расширение примитивных типов.

Различают унарное и бинарное числовое расширение.

### 3.5.1. Унарное числовое расширение

Это преобразование расширяет примитивные типы byte, short или char до типов int по правилам расширения примитивных типов.

Унарное числовое расширение может происходить при следующих операциях:

- унарные операции + и -;
- битовое отрицание ~;
- операции битового сдвига <<, >>, >>>.

Операторы сдвига имеют два аргумента, но они расширяются независимо друг от друга, поэтому это преобразование является унарным. Таким образом, результат выражения 5<<3L имеет тип int. Вообще, результат операторов сдвига всегда имеет тип int или long.

Примеры работы всех этих операторов с учетом расширения подробно рассматривались в предыдущих главах.

### 3.5.2. Бинарное числовое расширение

Это преобразование расширяет все примитивные числовые типы кроме double до типов int, long, float, double по правилам расширения примитивных типов. Бинарное числовое расширение происходит при числовых операторах, имеющих два аргумента, по следующим правилам:

- если любой из аргументов имеет тип double, то и второй приводится к double;
- иначе, если любой из аргументов имеет тип float, то и второй приводится к float;
- иначе, если любой из аргументов имеет тип long, то и второй приводится к long;
- иначе оба аргумента приводятся к int.

Бинарное числовое расширение может происходить при следующих операциях:

- арифметические операции +, -, *, /, %;
- операции сравнения <, <=, >, >=, ==, !=;
- битовые операции &, |, ^;
- в некоторых случаях для операции с условием ? :.

Примеры работы всех этих операторов с учетом расширения подробно рассматривались в предыдущих главах.

# 4. Тип переменной и тип ее значения

Теперь, когда были подробно рассмотрены все примеры преобразований, нужно вернутся к вопросу переменной и ее значений.

Как уже говорилось, переменная определяется тремя базовыми характеристиками: имя, тип, значение. Имя дается произвольным образом и никак не сказывается на свойствах переменной. А вот значение всегда имеет некоторый тип, не обязательно совпадающий с типом самой переменной. Поэтому необходимо рассмотреть все возможные типы переменных и выяснить, значения каких типов они могут в себе хранить.

Начнем с переменных примитивных типов. Поскольку эти переменные действительно хранят само значение, то их тип всегда точно совпадает с типом значения.

Проиллюстрируем это правило на примере:

```
byte b=3;
char c='A'+3;
long m=b+c;
double d=m-3F;
```

Здесь переменная b будет хранить значение типа byte после сужения целочисленного литерала типа int. Переменная c будет хранить тип char после того, как компилятор осуществит сужающее преобразование результата суммирования, который будет иметь тип int. Для переменной l выполнится расширения результата суммирования типа от int к типу long. Наконец, переменная d будет хранить значение типа double, получившееся в результате расширения результата разности, который имеет тип float.

Переходим к ссылочным типам. Во-первых, значение любой переменной такого типа - ссылка, которая может указывать только на объекты, порожденные от тех или иных классов, и далее обсуждаются только свойства этих классов. (Также объекты могут порождаться от массивов, эта тема рассматривается в отдельной главе).

Кроме этого, ссылочная переменная любого типа может иметь значение null. Большинство действий над такой переменной, например, обращение к полям или методам, приведет к ошибке.

Итак, какова связь между типом ссылочной переменной и ее значения? Здесь главное ограничение - проверка компилятора, который следит, чтобы все действия, выполняющиеся над объектом, были корректны. Компилятор не может предугадать, на объект какого класса будет реально ссылаться та или иная переменная. Все, чем он располагает - тип самой переменной. Именно его и использует компилятор для проверок. А значит, все допустимые значения переменной должны гарантировано обладать свойствами, определенными в классе-типе этой переменной. Такую гарантию дает только наследование. Отсюда получаем правило: ссылочная переменная типа A может указывать на объекты, порожденные от самого типа A или его наследников.

```
Point p = new Point();
```

В этом примере переменная и ее значение одинакового типа, поэтому над объектом можно совершать все возможные для данного класса действия.

```
Parent p = new Child();
```

Такое присвоение корректно, так как класс Child порожден от Parent. Однако теперь допустимые действия над переменной p, а значит, над объектом, только что созданным на основе класса Child, ограничены только теми, которые были унаследованы от класса Parent. Например, если в классе Child определен некий новый метод newChildMethod(), то попытка его вызвать p.newChildMethod() будет порождать ошибку компиляции. Необходимо подчеркнуть, что не происходит никаких изменений с самим объектом, ограничение порождается используемым способом доступа к этому объекту - переменной типа Parent.

Чтобы показать, что объект не потерял никаких своих свойств, произведем следующее обращение:

```
((Child)p).newChildMethod();
```

Здесь в начале проводится явное сужение к типу Child. Во время исполнения программы JVM проверит, что тип объекта, на который ссылается переменная p совместим с типом Child. В нашем случае это именно так. В результате получается ссылка типа Child, поэтому становится допустимым вызов метода newChildMethod(), который вызывается у объекта, созданного в предыдущей строке.

Обратим внимание на важный частный случай - переменная типа Object может ссылаться на объекты любого типа.

В дальнейшем, с изучением новых типов (абстрактных классов, интерфейсов, массивов) этот список будет продолжаться, а пока приведем краткое обобщение того, что было рассмотрено в этом разделе.

| Тип переменной | Допустимые типы ее значения |
|---|---|
| Примитивный | В точности совпадает с типом переменной |
| Ссылочный | • null<br>• совпадающий с типом переменной<br>• классы-наследники от типа переменной |
| Object | • null<br>• любой ссылочный |

# 5. Заключение

В этой главе были рассмотрены правила работы с типами данных в строго типизированном языке Java. Поскольку компилятор строго отслеживает тип каждой переменной и каждого выражения, в случае изменения этого типа необходимо четко понимать, какие действия допустимы, а какие нет, с точки зрения компилятора и виртуальной машины.

Были рассмотрены все виды приведения типов в Java, то есть переход от одного типа к другому. Они разбиваются на 7 групп, начиная с тождественного и заканчивая запрещенными. Основные 4 вида определяются сужающими или расширяющими переходами между простыми или ссылочными типами. Важно помнить, что при явном сужении числовых типов, старшие биты просто отбрасываются, что порой приводит к неожиданному результату. Что касается преобразования ссылочных значений, то здесь действует правило – преобразование никогда не порождает новых и не изменяет существующих объектов. Меняется лишь способ работы с ними.

Особенными в Java является преобразование к строке.

Затем были рассмотрены все ситуации в программе, где могут происходить преобразования типов. Во-первых, это присвоение значений, когда зачастую преобразование происходит не заметно для программиста. Вызов метода во многом похож на инициализацию. Явное приведение позволяет осуществить желаемый переход в случае, когда компилятор не позволяет сделать это неявно. Преобразование при выполнении числовых операций оказывает существенное влияние на результат.

В заключение была рассмотрена связь между типом переменной и типом ее значения.

# 6. Контрольные вопросы

7-1. Корректен ли следующий пример кода, и если да, то сколько преобразований и каких видов будет произведено при его исполнении?

```
byte b=1;
long m=-b;
Object o="=";
```

```
print("m"+o+m);
```

a.) Пример корректен. Преобразования:

- первая строка: преобразование при присвоении, сужение от int к byte
- вторая: числовое расширение при вычислении оператора унарный минус от byte к int, расширение при присвоении значения от int к long
- третья: расширение при присвоении String к Object
- четвертая: приведения к строке: от Object к String и от long к String (через Long).

7-2. Произойдет ли потеря точности при следующем преобразовании?

```
float f = -16777217;
```

a.) Да. Число 16777217 записывается как 1000000000000000000000001 в двоичной системе, всего 25 бит. А тип float отводит 24 бита для хранения мантиссы (значащих цифр), причем один из них знаковый. После отбрасывания не уместившихся битов получаем 100000000000000000000000, или –16777216 в десятичной системе. Таким образом, теряется точность в последнем разряде. (Для получения правильного ответа достаточно запустить подобный пример и увидеть результат.)

7-3. Корректен ли следующий пример? Если нет, то в каких строках какие ошибки будут сгенерированы?

```
byte b=100-100;
byte b=100+100;
byte b=100*100;
```

a.) Результат вычитания в первой строке равен 0, что укладывается в тип byte, ошибки не будет. Результат вычислений в строках 2 и 3 выходит за рамки byte (максимально допустимое значение 127), поэтому неявное сужение не допустимо.

7-4. Для каких значений числовой переменной x будет верно следующее выражение?

```
x==-x
```

a.) Для целочисленных типов:

- 0
- самое наименьшее значение для типов int и long. Например, для типа int это –2147483648

Для дробных типов:

- значения 0.0 и –0.0

7-5. Верны ли следующие выражения для переменной d дробного типа?

```
(short)d==(short)(int)d
(int)d==(int)(long)d
```

a.) Первое выражение верно, так как при приведении к short, дробное значение всегда сначала приводится к int.

Второе выражение не верно, в чем можно легко убедится, вычислив его для любого d, большего, чем максимально допустимое значение типа int. Левая часть будет равна 231-1, в то время как правая будет вычислена путем отбрасывания битов, старше 32:

```
double d=3e9;
System.out.println((int)d);
System.out.println((int)(long)d);
```

Результатом будет:

```
2147483647
-1294967296
```

7-6. Корректны ли следующие преобразования?

```
Object o = (String)null;
String s = o;
```

a.) Нет. Хотя значение null можно привести к любому ссылочному типу, во 2 строке происходит сужение типа ссылки (Object) до класса String. Такое действие должно происходить явно:

```
String s = (String)o;
```

7-7. Пусть классы Wolf и Rabbit являются наследниками класса Animal. Корректен ли следующий пример?

```
Wolf w = new Wolf();
Animal a = (Animal)w;
Rabbit r = (Rabbit)a;
```

a.) Хотя этот пример будет корректно скомпилирован, при выполнении он всегда будет порождать ошибку. Не смотря на приведение, переменная «a» ссылается на объект типа Wolf, созданный в первой строке.

В третьей строке при приведении ссылки на объект класса Wolf к типу Rabbit будет возникать ошибка преобразования типов, поскольку эти классы не находятся на одной ветви дерева наследования.

7-8. Какие значения не могут участвовать в преобразовании к строке?

a.) Все могут.

7-9. Какие преобразования будут производиться при работе следующего метода?

```
public int add(byte a, byte b) {
   short x=(short)a;
   char y=(char)b;
   return x+y;
}
```

a.) Преобразования:

- первая строка: явное приведение от byte к short
- вторая: явное приведение от byte к char
- третья: числовое расширение при вычислении бинарного оператора сложения: от short и char к int

7-10. Какое значение появится на консоли после выполнения следующего кода?

```
char c=65;
print(c);
print(+c);
print("="+c);
```

a.) Ответ:

```
A
65
=A
```

В первом случае выводится значение типа char, поэтому отображается символ. Во втором случае выводится значение унарного оператора плюс, то есть число типа int. В третьем случае при сложении со строкой происходит преобразование от типа char к String, то есть запишется символ.

7-11. Значение какого типа будет хранить переменная после инициализации?

```
byte b=1+2;
```

a.) Примитивные переменные всегда хранят значения точного того же типа, что и они сами, то есть, в данном примере – byte. Это значение будет получено после неявного сужения при присвоении от типа int.

7-12. Значения какого типа будут хранить переменные после инициализации?

```
Object o = "123";
Child c = new Child();
Parent p=(Parent)new Child();
```

```
Child x=null;
```

a.) В порядке следования:

- String
- Child
- Child
- null

# Программирование на Java

## Лекция 8. Объектная модель в Java

### 20 января 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Эта лекция является некоторым отступлением от рассмотрения технических особенностей Java и посвящена в основном изучению ключевых свойств объектной модели Java, таких как статические элементы, абстрактные методы и классы, интерфейсы, заменяющие множественное наследование. Без этих мощных конструкций, язык Java был бы не способен решать серьезные задачи.

В заключение рассматриваются принципы работы полиморфизма для полей и методов, статических и динамических. Уточняется классификация типов переменных и типов значений, которые они могут хранить.

# Оглавление

# Лекция 8. Объектная модель в Java

## Содержание лекции.



## 1. Введение

В предыдущих главах были рассмотрены основы объявления классов, а затем взаимоотношения классов, связанных механизмом наследования. В этой главе будет продолжено изложение особенностей объектной модели Java, в том числе и альтернативный множественному наследованию подход - интерфейсы.

## 2. Статические элементы

До этого момента под полями объекта всегда понимали значения, которые имеют смысл только в контексте некоторого экземпляра класса. Например:

```
class Human {
```

```
   private String name;
}
```

Прежде чем обратиться к полю name необходимо получить ссылку на экземпляр класса Human, невозможно узнать имя вообще, оно всегда принадлежит какому-то конкретному человеку.

Но бывают данные и иного характера. Предположим, необходимо хранить количество всех людей (экземпляров класса Human, существующих в системе). Понятно, что общее число людей не является характеристикой какого-то одного человека, оно относится ко всему типу в целом. Отсюда появляется название "поле класса" в отличие от "поля объекта". Объявляются такие поля с помощью модификатора static:

```
class Human {
   public static int totalCount;
}
```

Чтобы обратиться к такому полю, ссылка на объект не требуется, вполне достаточно имени класса:

```
Humans.totalCount++; // рождение еще одного человека
```

Для удобства позволяется обращаться к статическим полям и через ссылки:

```
Human h = new Human();
h.totalCount=100;
```

Однако такое обращение конвертируется компилятором. Он использует тип ссылки, в данном случае переменная h объявлена как Human, поэтому последняя строка будет неявно преобразована в:

```
Human.totalCount=100;
```

В этом можно убедится, изучив следующий пример:

```
Human h = null;
h.totalCount+=10;
```

Значение ссылки равно null, но это не имеет значения в силу описанной конвертации. Данный код успешно скомпилируется и корректно исполнится. Таким образом, в следующем примере:

```
Human h1 = new Human(), h2 = new Human();
Human.totalCount=5;
h1.totalCount++;
System.out.println(h2.totalCount);
```

все обращения к переменной totalCount приводят к одному единственному полю, и результатом работы такой программы будет 6. Это поле будет существовать в единственном

экземпляре независимо от того, сколько объектов было порождено от этого класса, и был ли вообще создан хоть один объект.

Аналогично объявляются статические методы.

```
class Human {
   private static int totalCount;

   public static int getTotalCount() {
      return totalCount;
   }
}
```

Для вызова статического метода не требуется ссылки на объект.

```
Human.getTotalCount();
```

Хотя для удобства обращения через ссылку позволяются, но принимается во внимание только тип ссылки:

```
Human h=null;
h.getTotalCount(); // Два эквивалентных обращения к
Human.getTotalCount(); // одному и тому же методу
```

Хотя приведенный пример технически корректен, все же использование ссылки на объект для обращения к статическим полям и методам не рекомендуется, поскольку запутывает код.

Обращение к статическому полю является корректным независимо от того, были ли порождены объекты от этого класса и в каком количестве. Например, стартовый метод main() запускается до того, как программа создаст хотя бы один объект.

Кроме полей и методов статическими могут быть инициализаторы. Они также называются инициализаторами класса в отличие от инициализаторов объекта, рассматриваемых ранее. Их код выполняется один раз во время загрузки класса в память виртуальной машины. Их запись начинается с модификатора static:

```
class Human {
   static {
      System.out.println("Class loaded");
   }
}
```

Если объявление статического поля совмещается с его инициализацией, то поле инициализируется также однократно при загрузке класса. На объявление и применение статических полей накладываются аналогичные ограничения, что и для динамических - нельзя использовать поле в инициализаторах других полей или в инициализаторах класса до того, как это поле объявлено:

```
class Test {
```

```
   static int a;
   static {
      a=5;
      // b=7; // Нельзя использовать до объявления!
   }
   static int b=a;
}
```

Это правило распространяется только на обращения к полям по простому имени. Если использовать составное имя, то обращаться к полю можно будет раньше (выше в тексте программы), чем оно будет объявлено:

```
class Test {
   static int b=Test.a;
   static int a=3;
   static {
      System.out.println("a="+a+", b="+b);
   }
}
```

Если класс будет загружен в систему, на консоли появится текст:

```
a=3, b=0
```

Видно, что поле b при инициализации получило значение по умолчанию поля a, т.е. 0. Затем полю a было присвоено значение 3.

Статические поля также могут быть объявлены как final, что означает, что они должны быть проинициализированы строго один раз, и затем уже больше не менять своего значения. Аналогично, статические методы могут быть объявлены как final, что означает, что их нельзя перекрывать в классах-наследниках.

Для инициализации статических полей можно пользоваться статическими методами и нельзя обращаться к динамическим. Вводят специальные понятия - статический и динамический контексты. К статическому контексту относят статические методы, статические инициализаторы, инициализаторы статических полей. Все остальные части кода имеют динамический контекст.

При выполнении кода в динамическом контексте всегда есть объект, с которым идет работа в данный момент. Например, для динамического метода это объект, у которого он был вызван, и так далее.

Напротив, со статическим контекстом ассоциированных объектов нет. Например, как уже указывалось, стартовый метод main() вызывается в тот момент, когда ни один объект еще не создан. При обращении к статическому методу, например, MyClass.staticMethod(), также ни одного экземпляра MyClass может не быть. Обращаться к статическим методам класса Math можно, а создавать его экземпляры нельзя.

А раз нет ассоциированных объектов, то и пользоваться динамическими конструкциями нельзя. Можно только ссылаться на статические поля и вызывать статические методы.

Либо обращаться к объектам через ссылки на них, полученные в результате вызова конструктора или в качестве аргумента метода и т.п.

```
class Test {
   public void process() {
   }
   public static void main(String s[]) {
      // process(); - ошибка! у какого объекта его вызывать?

      Test test = new Test();
      test.process(); // так правильно
   }
}
```

# 3. Ключевые слова this и super

Эти ключевые слова уже упоминались, были рассмотрены некоторые ситуации их применения. Здесь они будут описаны более полно.

Если выполнение кода происходит в динамическом контексте, то должен быть объект, ассоциированный с ним. В этом случае ключевое слово this возвращает ссылку на этот объект:

```
class Test {
   public Object getThis() {
      return this; // Проверим, куда указывает эта ссылка
   }

   public static void main(String s[]) {
      Test t = new Test();
      System.out.println(t.getThis()==t); // Сравнение
   }
}
```

Результатом работы программы будет:

```
true
```

То есть, внутри методов слово this возвращает ссылку на объект, у которого этот метод вызван. Оно необходимо, если нужно передать аргумент, равный ссылке на этот объект, в какой-нибудь метод.

```
class Human {
   public static void register(Human h) {
      System.out.println(h.name+" is registered.");
   }

   private String name;
```

```
   public Human (String s) {
      name = s;
      register(this); // саморегистрация
   }

   public static void main(String s[]) {
      new Human("John");
   }
}
```

Результатом будет:

```
John is registered.
```

Другое применение this рассматривалось в случае "затемняющих" объявлений:

```
class Human {
   private String name;

   public void setName(String name) {
      this.name=name;
   }
}
```

Слово this можно использовать для обращения к полям, которые объявляются ниже:

```
class Test {
   // int b=a; нельзя обращаться к необъявленному полю!
   int b=this.a;
   int a=5;
   {
      System.out.println("a="+a+", b="+b);;
   }
   public static void main(String s[]) {
      new Test();
   }
}
```

Результатом работы программы будет:

```
a=5, b=0
```

Все происходит так же, как и для статических полей - b получает значение по умолчанию для a, т.е. ноль, а затем a инициализируется значением 5.

Наконец, слово this применяется в конструкторах для явного вызова в первой строке другого конструктора этого же класса. Там же может применяться и слово super, только уже для обращения к конструктору родительского класса.

Другие применения слова super также связаны с обращением к родительскому классу объекта. Например, оно может потребоваться в случае переопределения (overriding) родительского метода.

Переопределением называют объявление метода, сигнатура которого совпадает с одним из методов родительского класса.

```
class Parent {
   public int getValue() {
      return 5;
   }
}

class Child extends Parent {

   // Переопределение метода
   public int getValue() {
      return 3;
   }

   public static void main(String s[]) {
      Child c = new Child();

      // Пример вызова переопределенного метода
      System.our.println(c.getValue());
   }
}
```

Вызов переопределенного метода использует механизм полиморфизма, который подробно рассматривается в конце этой главы. Однако ясно, что результатом выполнения примера будет значение 3. Невозможно, используя ссылку типа Child, получить из метода getVaue() значение 5, родительский метод перекрыт и больше не доступен.

Часто бывают случаи, когда при переопределении было бы полезно воспользоваться результатом работы родительского метода. Предположим, он делал сложные вычисления, а переопределенный метод должен вернуть округленный результат этих вычислений. Понятно, что гораздо удобнее как-то обратиться к родительскому методу, чем заново описывать весь алгоритм. Здесь применяется слово super. Из класса наследника с его помощью можно обращаться к переопределенным методам родителя:

```
class Parent {
   public int getValue() {
      return 5;
   }
}

class Child extends Parent {

   // Переопределение метода
   public int getValue() {
```

```
      // Обращение к методу родителя
      return super.getValue()+1;
   }

   public static void main(String s[]) {
      Child c = new Child();
      System.our.println(c.getValue());
   }
}
```

Результатом работы программы будет значение 6.

Обращаться с помощью ключевого слова super к переопределенному методу родителя родителя, т.е. на два уровня наследования вверх, невозможно. Если родительский класс переопределил функциональность своего родителя, значит, она не будет доступна его наследникам.

Поскольку ключевые слова this и super требуют наличия ассоциированного объекта, т.е. динамического контекста, использование их в статическом контексте запрещено.

# 4. Ключевое слово abstract

Следующее важное понятие, которое необходимо рассмотреть - ключевое слово abstract.

Иногда бывает удобным описать только заголовок метода, без его тела, и таким образом объявить, что такой метод будет существовать в этом классе. Реализацию этого метода, то есть его тело, можно описать позже.

Рассмотрим пример. Предположим, необходимо создать набор графических элементов, неважно, каких именно. Например, они могут представлять собой геометрические фигуры - круг, квадрат, звезда и т.д.; или элементы пользовательского интерфейса - кнопки, поля ввода и т.д. Сейчас это не имеет решающего значения. Кроме этого, есть специальный контейнер, который занимается их отрисовкой. Понятно, что внешний вид каждой компоненты уникален, не похож на других, а значит, соответствующий метод (назовем его paint()) будет реализован в разных элементах совсем по-разному.

Но в то же время у компонент может быть много общего. Например, любая из них занимает некоторую прямоугольную область контейнера. Сложные контуры фигуры необходимо вписать в прямоугольник, чтобы можно было анализировать перекрытия, проверять, не вылезает ли компонент за размер контейнера и т.д. Каждая может иметь цвет, которым ее надо рисовать, может быть видимой или не видимой, и т.д. Очевидно, что полезно создать родительский класс для всех компонент, и один раз объявить в нем все общие свойства, чтобы каждая компонента лишь наследовала их.

Но как поступить с методом отрисовки? Ведь родительский класс не представляет собой какую-либо фигуру, у него нет визуального представления. Можно объявить метод paint() в каждой компоненте независимо. Но тогда контейнер должен будет обладать сложной функциональностью для анализа того, какая именно компонента сейчас обрабатывается, делать приведение типа и только после этого вызывать нужный метод.

Именно здесь удобно объявить абстрактный метод в родительском классе. У него нет внешнего вида, но известно, что он есть у каждого наследника. Поэтому заголовок метода

описывается в родительском классе, тело метода у каждого наследника свое, а контейнер может спокойно пользоваться только базовым типом, не делая никаких приведений.

Приведем упрощенный пример:

```
// Базовая арифметическая операция
abstract class Operation {
   public abstract int calculate(int a, int b);
}

// Сложение
class Addition {
   public int calculate(int a, int b) {
      return a+b;
   }
}

// Вычитание
class Subtraction {
   public int calculate(int a, int b) {
      return a-b;
   }
}

class Test {
   public static void main(String s[]) {
      Operation o1 = new Addition();
      Operation o2 = new Subtraction();

      o1.calculate(2, 3);
      o2.calculate(3, 5);
   }
}
```

Видно, что выполнения операций сложения и вычитания в методе main() записываются совершенно одинаковым образом.

Обратите внимание - поскольку абстрактный метод не имеет тела, после описания его заголовка ставится точка с запятой. А раз у него нет тела, то к нему нельзя обращаться, пока его наследники не опишут реализацию. Это означает, что нельзя создавать экземпляры класса, у которого есть абстрактные методы. Такой класс сам объявляется абстрактным.

Класс может быть абстрактным и в случае, если у него нет абстрактных методов, но должен быть абстрактным, если такие методы есть. Разработчик может указать ключевое слово abstract в списке модификаторов класса, если хочет запретить создание экземпляров этого класса. Классы-наследники должны реализовать (implements) все абстрактные методы (если они есть) своего абстрактного родителя, чтобы их можно было объявлять не абстрактными и порождать от них экземпляры.

Конечно, класс не может быть одновременно abstract и final. Это же верно и для методов. Кроме того, абстрактный метод не может быть private, native, static.

Сам класс может без ограничений пользоваться своими абстрактными методами.

```
abstract class Test {
   public abstract int getX();
   public abstract int getY();
   pulic double getLength() {
      return Math.sqrt(getX()*getX()+getY()*getY());
   }
}
```

Это корректно, поскольку метод getLength() может быть вызван только у объекта. Объект может быть порожден только от неабстрактного класса, который является наследником от Test, и должен был реализовать все абстрактные методы.

По этой же причине можно объявлять переменные типа абстрактный класс. Они могут иметь значение null или ссылаться на объект, порожденный от неабстрактного наследника этого класса.

# 5. Интерфейсы

Концепция абстрактных методов позволяет предложить альтернативу множественному наследованию. В Java класс может иметь только одного родителя, поскольку при множественном наследовании могут возникать конфликты, которые серьезно запутывают объектную модель. Например, если у класса есть два родителя, которые имеют одинаковый метод с различной реализацией, то какой из них унаследует новый класс? И как будет работать функциональность родительского класса, который лишился своего метода?

Все эти проблемы не возникают в случае, если наследуются только абстрактные методы от нескольких родителей. Даже если будет унаследовано несколько одинаковых методов, все равно у них нет реализации, и можно один раз описать тело метода, которое будет использовано при вызове любого из этих методов.

Именно так устроены интерфейсы в Java. От них нельзя порождать объекты, но другие классы могут реализовывать их.

## 5.1. Объявление интерфейсов

Объявление интерфейсов очень похоже на упрощенное объявление классов.

Объявление начинается с заголовка. Сначала указываются модификаторы. Интерфейс может быть объявлен как public, и тогда он будет доступен для всеобщего использования, либо модификатор доступа может не указываться, в этом случае интерфейс доступен только для типов своего пакета. Модификатор abstract для интерфейса не требуется, поскольку все интерфейсы являются абстрактными, его можно указать, но рекомендуется этого не делать, чтобы не загромождать код.

Далее записывается ключевое слово interface и имя интерфейса.

После этого может следовать ключевое слово extends и список интерфейсов, от которых будет наследоваться объявляемый интерфейс. Родительских типов может быть много,

главное, чтобы не было повторений, и чтобы отношение наследования не образовывало циклической зависимости.

Наследование интерфейсов действительно очень гибкое. Так, если есть два интерфейса A и B, причем B наследуется от A, то новый интерфейс C может наследоваться от них обоих. Впрочем, понятно, что указание наследования от A является избыточным, все элементы этого интерфейса и так будут получены по наследству через интерфейс B.

Затем в фигурных скобках записывается тело интерфейса.

```
public interface Drawble extends Colorable, Resizable {
}
```

Тело интерфейса состоит из объявления элементов, то есть полей-констант и абстрактных методов.

Все поля интерфейса должны быть public final static, поэтому эти модификаторы указывать необязательно и даже не желательно, чтобы не загромождать код. Поскольку поля объявляются финальными, необходимо их сразу инициализировать.

```
public interface Directions {
   int RIGTH=1;
   int LETFT=2;
   int UP=3;
   int DOWN=4;
}
```

Все методы интерфейса являются public abstract, и эти модификаторы также являются необязательными и нежелательными.

```
public interface Moveable {
   void moveRight();
   void moveLeft();
   void moveUp();
   void moveDown();
}
```

Как видно, описание интерфейса гораздо проще, чем объявление класса.

## 5.2. Реализация интерфейса

Любой класс может реализовывать любые доступные интерфейсы. При этом в классе должны быть реализованы все абстрактные методы, появившиеся при наследовании от интерфейсов или родительского класса, чтобы новый класс мог быть объявлен неабстрактным.

Если из разных источников наследуются методы с одинаковой сигнатурой, то достаточно один раз описать реализацию, и она будет применяться для всех них. Однако, если у этих методов различное возвращаемое значение, то возникает конфликт:

```
interface A {
```

```
   int getValue();
}

interface B {
   double getValue();
}
```

Если попытаться объявить класс, реализующий оба эти интерфейса, то возникнет ошибка компиляции. В классе оказывается два разных метода с одинаковой сигнатурой, что является неразрешимым конфликтом. Это единственное ограничение на набор интерфейсов, которые может реализовывать класс.

Подобный конфликт с полями-константами не столь критичен:

```
interface A {
   int value=3;
}

interface B {
   double value=5.4;
}

class C implements A, B {
   public static void main(String s[]) {
      C c = new C();
      // System.out.println(c.value); - ошибка!
      System.out.println(((A)c).value);
      System.out.println(((B)c).value);
   }
}
```

Как видно из примера, обращаться к такому полю через сам класс неверно, компилятор не сможет понять, какое из двух полей нужно использовать. Но можно с помощью явного приведения сослаться на одно из них.

Итак, если имя интерфейса указано после implements в объявлении класса, то класс реализует этот интерфейс. Наследники этого класса также реализуют интерфейс, поскольку им достаются по наследству его элементы.

Если интерфейс A наследуется от интерфейса B, а класс реализует A, то считается, что интерфейс B также реализуется этим классом по той же причине - все элементы передаются по наследству в два этапа - сначала интерфейсу A, а затем и классу.

Наконец, если класс C1 наследуется от класса C2, класс C2 реализуется интерфейс A1, а интерфейс A1 наследуется от интерфейса A2, то класс C1 также реализует интерфейс A2.

Вышесказанное позволяет утверждать, что переменные типа интерфейс также допустимы. Они могут иметь значение null или ссылаться на объекты, порожденные от классов, реализующих этот интерфейс. Поскольку объекты порождаются только от классов, а все они наследуются от Object, то это означает, что значения типа интерфейс обладают всеми элементами класса Object.

## 5.3. Применение интерфейсов

До сих пор интерфейсы рассматривались с технической точки зрения - как их объявлять, какие конфликты могут возникать, как их разрешать. Однако важно понимать, как применяются интерфейсы с концептуальной точки зрения.

Распространенное выражение, что интерфейс - это полностью абстрактный класс, в целом верно, но оно не отражает всех преимуществ, которые дают интерфейсы объектной модели. Как уже отмечалось, множественное наследование порождает ряд конфликтов, но отказ от него хоть и делает язык проще, но не устраняет ситуации, в которых требуются подобные подходы.

Рассмотрим пример. Возьмем в качестве примера дерева наследования классификацию живых организмов. Известно, что растения и животные принадлежат к разным царствам. Основным различием между ними является то, что растения поглощают неорганические элементы, а животные питаются органическими веществами. Среди животных есть две больших группы - птицы и млекопитающие. Предположим, что на основе этой классификации построено дерево наследования, в каждом классе определены элементы с учетом наследования от родительских классов.

Рассмотрим такое возможное свойство живого организма, как способность питаться насекомыми. Очевидно, что это свойство нельзя приписать всей группе птиц или млекопитающих, а тем более растений. Но существуют представители каждой такой группы, которые этим свойством обладают - для растений это росянка, для птиц, например, ласточки, а для млекопитающих - муравьеды. Причем, очевидно, "реализовано" это свойство у каждого вида совсем по-разному.

Можно было бы объявить соответствующий метод (скажем, consumeInsect(Insect)) у каждого представителя независимо. Но если задача состоит в моделировании, например зоопарка, то однотипную процедуру - кормление насекомыми - пришлось бы описывать для каждого вида независимо, что существенно осложнило бы код, причем без какой-либо пользы.

Java предлагает другое решение. Объявляется интерфейс InsectConsumer:

```
public interface InsectConsumer {
   void consumeInsect(Insect i);
}
```

Его реализуют все подходящие животные и растения:

```
// Росянка расширяет класс Растение
public class Sundew extends Plant implements InsectConsumer {
   public void consumeInsect(Insect i) {
      ...
   }
}

// Ласточка расширяет класс Птица
public class Swallow extends Bird implements InsectConsumer {
   public void consumeInsect(Insect i) {
      ...
```

```
    }
}

// Муравьед расширяет класс Млекопитающее
public class AntEater extends Mammal implements InsectConsumer {
    public void consumeInsect(Insect i) {
        ...
    }
}
```

В результате в классе, моделирующем служащего зоопарка, можно объявить соответствующий метод:

```
// Служащий, отвечающий за кормление, расширяет класс Служащий
class FeedWorker extends Worker {

    // С помощью этого метода можно накормить
    // и росянку, и ласточку, и муравьеда
    public void feedOnInsects(InsectConsumer consumer) {
        ...
        consumer.consumeInsect(insect);
        ...
    }
}
```

В результате удалось свести работу с одним свойством трех весьма разнородных классов в одно место, сделать код более универсальным. Обратите внимание, что при добавлении еще одного насекомоядного такая модель зоопарка не потребует никаких изменений, чтобы обслуживать новый вид, в отличие от первоначального громоздкого решения. Благодаря введению интерфейса удалось отделить классы, реализующие его (живые организмы) и использующие его (служащий зоопарка). После любых изменений этих классов при условии сохранения интерфейса их взаимодействие не нарушится.

Данный пример иллюстрирует, как интерфейсы предоставляют альтернативный, более строгий и гибкий подход вместо множественного наследования.

# 6. Полиморфизм

Ранее были рассмотрены правила объявления классов с учетом их наследования. В этой главе было введено понятие переопределенного метода. Однако полиморфизм требует более глубокого изучения. При объявлении одноименных полей или методов с совпадающими сигнатурами происходит перекрытие элементов из родительского и наследующегося класса. Рассмотрим, как именно функционируют классы и объекты в таких ситуациях.

## 6.1. Поля

Во-первых, нужно сказать, что такое объявление корректно. Наследники могут объявлять поля с любыми именами, даже совпадающими с родительскими. Затем, необходимо понять, как два одноименных поля будут сосуществовать. Действительно, объекты класса Child будут содержать сразу две переменных, а поскольку они могут отличаться не только значением, но и типом (ведь это два независимых поля), именно компилятор будет определять какое из значений использовать. Компилятор может опираться только на тип ссылки, с помощью которой делается обращение к полю:

```
Child c = new Child();
System.out.println(c.a);
Parent p = c;
System.out.println(p.a);
```

Обе ссылки указывают на один и тот же объект, порожденный от класса Child, но одна из них имеет такой же тип, а другая - Parent. Отсюда следуют и результаты:

```
3
2
```

Объявление поля в классе-наследнике "скрыло" родительское поле. Это объявление так и называется - "скрывающим" (hiding). Это особый случай перекрытия областей видимости, отличающийся от "затеняющего" (shadowing) и "заслоняющего" (obscuring) объявлений. Тем не менее, как хорошо видно, родительское поле продолжает существовать. К нему можно обратиться и явно:

```
class Child extends Parent {
   int a=3; // скрывающее объявление

   int b=((Parent)this).a; // более громоздкое объявление
   int c=super.a; // более простое
}
```

Переменные b и c получат значение, хранящееся в родительском поле a. Хотя выражение с super более простое, оно не позволит обратиться на два уровня вверх по дереву наследования. А ведь вполне возможно, что в родительском классе это поле также было скрывающим, и в родителе родителя хранится еще одно значение. К нему можно обратиться явным приведением, как это делается для b.

Рассмотрим следующий пример:

```
class Parent {
   int x=0;

   public void printX() {
      System.out.println(x);
   }
```

```
}

class Child extends Parent {
   int x=-1;
}
```

Каков будет результат следующих строк?

```
new Child().printX();
```

Значение какого поля будет распечатано? Метод вызывается с помощью ссылки типа Child, но это не сыграет никакой роли. Вызывается метод, определенный в классе Parent, и компилятор, конечно, расценил обращение к полю x в этом методе именно как к полю класса Parent. Поэтому результатом будет 0.

Перейдем к статическим полям. На самом деле, для них проблем и конфликтов, связанных с полиморфизмом, вообще не существует.

Рассмотрим пример:

```
class Parent {
   static int a=2;
}

class Child extends Parent {
   static int a=3;
}
```

Каков будет результат следующих строк?

```
Child c = new Child();
System.out.println(c.a);
Parent p = c;
System.out.println(p.a);
```

Нужно вспомнить, как компилятор обрабатывает обращения к статическим полям через ссылочные значения. Не имеет никакого значения, на какой объект указывает ссылка. Более того, она может быть даже равна null. Все определяется типом ссылки.

Поэтому рассматриваемый пример эквивалентен:

```
System.out.println(Child.a);
System.out.println(Parent.a);
```

А его результат сомнений уже не вызывает:

```
3
2
```

Можно привести следующее пояснение. Статическое поле принадлежит классу, а не объекту. В результате появление классов-наследников с скрывающими (hiding)

объявлениями никак не сказывается на работе с исходным полем. Компилятор всегда может определить, через ссылку какого типа происходит обращение к нему.

Обратите внимание на следующий пример:

```
class Parent {
   static int a;
}

class Child extends Parent {
}
```

Каков будет результат следующих строк?

```
Child.a=10;
Parent.a=5;
System.out.println(Child.a);
```

В этом примере поле a не было скрыто и передалось по наследству классу Child. Однако результат показывает, что это все же одно поле:

```
5
```

Несмотря на то, что к полю класса идут обращения через разные классы, переменная всего одна.

Итак, наследники могут объявлять поля с именами, совпадающими с родительскими полями. Такие объявления называют скрывающими. При этом объекты будут содержать оба значения, а компилятор будет определять в каждом случае, с которым из них надо работать.

## 6.2. Методы

Рассмотрим случай переопределения (overriding) методов:

```
class Parent {
   public int getValue() {
      return 0;
   }
}

class Child extends Parent {
   public int getValue() {
      return 1;
   }
}
```

И строки, демонстрирующие работу с этими методами:

```
Child c = new Child();
```

```
System.out.println(c.getValue());
Parent p = c;
System.out.println(p.getValue());
```

Результатом будет:

```
1
1
```

Можно видеть, что родительский метод полностью перекрыт, значение 0 никак нельзя получить через ссылку, указывающую на объект класса Child. В этом ключевая особенность полиморфизма - наследники могут изменять родительское поведение, даже если обращение к ним производится по ссылке родительского типа. Напомним, что, хотя старый метод уже не доступен снаружи, внутри класса-наследника к нему все же можно обратиться с помощью super.

Рассмотрим более сложный пример:

```
class Parent {
   public int getValue() {
      return 0;
   }
   public void print() {
      System.out.println(getValue());
   }
}

class Child extends Parent {
   public int getValue() {
      return 1;
   }
}
```

Что появится на консоли после выполнения следующих строк?

```
Parent p = new Child();
p.print();
```

С помощью ссылки типа Parent вызывается метод print(), объявленный в классе Parent. Из этого метода делается обращение к getValue(), которое в классе Parent возвращает 0. Но компилятор уже не может предсказать, к динамическому методу какого класса будет произведено обращение во время работы программы. Это определяет виртуальная машина на основе объекта, на который указывает ссылка. И раз этот объект порожден от Child, то существует лишь один метод getValue().

Результатом работы примера станет:

```
1
```

Данный пример демонстрирует, что переопределение методов должно производится с осторожностью. Если слишком сильно изменить логику их работы, нарушить принятые

соглашения (например, начать возвращать null в качестве значения ссылочного типа, если родительский метод такого не допускал), то это может привести к сбоям в работе родительского класса, а значит, объекта наследника. Более того, существуют и некоторые обязательные ограничения.

Вспомним, что заголовок метода состоит из модификаторов, возвращаемого значения, сигнатуры и throws-выражения. Сигнатура (имя и набор аргументов) остается неизменной, если говорить о переопределении. Возвращаемое значение также не может меняться, иначе это приведет к появлению двух разных методов с одинаковыми сигнатурами.

Рассмотрим модификаторы доступа.

```
class Parent {
   protected int getValue() {
      return 0;
   }
}

class Child extends Parent {
   /* ??? */ protected int getValue() {
      return 1;
   }
}
```

Пусть родительский метод был объявлен как protected. Понято, что метод наследника можно оставить с таким же уровнем доступа, но можно ли его расширить (public) или сузить (доступ по умолчанию)? Несколько строк для проверки:

```
Parent p = new Child();
p.getValue();
```

Обращение к методу идет с помощью ссылки типа Parent. Именно компилятор делает проверку уровня доступа, и он будет ориентироваться на родительский класс. Но ссылка-то указывает на объект, порожденный от Child, и по правилам полиморфизма исполняться будет метод именно этого класса. А значит, доступ к переопределенному методу не может быть более ограниченным, чем к исходному. Итак, методы с доступом по умолчанию можно переопределять с таким же доступом, либо protected или public. protected-методы переопределяются такими же или public, а для public менять модификатор доступа и вовсе нельзя.

Что касается private-методов, то они определены только внутри класса, снаружи не видны, а потому наследники могут без ограничений объявлять методы с такими же сигнатурами и произвольными возвращаемыми значениями, модификаторами доступа и т.д.

Аналогичные ограничения накладываются и на throws-выражение, которое будет рассмотрено в последующих главах.

Если абстрактный метод переопределяется неабстрактным, то говорят, что он его реализовал (implements). Как ни странно, но абстрактный метод может переопределить другой абстрактный или даже неабстрактный метод. В первом случае такое действие может иметь смысл только при изменении модификатора доступа (расширении), либо

throws-выражения. Во втором случае полностью утрачивается старая реализация метода, что может потребоваться в особенных случаях.

Перейдем к статическим методам. Рассмотрим пример:

```
class Parent {
   static public int getValue() {
      return 0;
   }
}

class Child extends Parent {
   static public int getValue() {
      return 1;
   }
}
```

И строки, демонстрирующие работу с этими методами:

```
Child c = new Child();
System.out.println(c.getValue());
Parent p = c;
System.out.println(p.getValue());
```

Аналогично случаю со статическими переменными, вспоминаем алгоритм обработки компилятором таких обращений к статическим элементам и получаем, что код эквивалентен следующим строкам:

```
System.out.println(Child.getValue());
System.out.println(Parent.getValue());
```

Результатом очевидно будет:

```
1
0
```

То есть, статические методы, подобно статическим полям, принадлежат классу, и появление наследников на них не сказывается.

Статические методы не могут перекрывать обычные, и наоборот.

## 6.3. Полиморфизм и объекты

В заключение рассмотрим несколько особенностей, вытекающих из свойств полиморфизма.

Во-первых, теперь можно точно сформулировать, что является элементами ссылочного типа. Ссылочный тип обладает следующими элементами:

- непосредственно объявленными в его теле;

- объявленными в его родительском классе и реализуемых интерфейсов, кроме:
  - private-элементов;
  - "скрытых" элементов (полей и статических методов, скрытых одноименными элементами);
  - переопределенных методов (динамических методов).

А во-вторых, продолжим рассмотрение взаимосвязи типа переменной и типов ее возможных значений. К случаям, рассмотренным в прошлой главе, добавляются два особых случая. Переменная типа абстрактный класс может ссылаться на объекты, порожденные неабстрактным наследником этого класса. Переменная типа интерфейс может ссылаться на объекты, порожденные от класса, реализующего этот интерфейс.

Сведем эти данные в таблицу:

| Тип переменной | Допустимые типы ее значения |
|---|---|
| Абстрактный класс | • null<br>• неабстрактный наследник |
| Интерфейс | • null<br>• классы, реализующие интерфейс, а именно:<br>- напрямую реализующие (заголовок содержит implements);<br>- наследующиеся от реализующих классов;<br>- реализующие наследников этого интерфейса;<br>- смешанный случай - наследование от класса, реализующего наследника интерфейса |

Таким образом, Java предоставляет гибкую и мощную модель объектов, позволяющую проектировать самые сложные системы. Необходимо хорошо разбираться в ее основных свойствах и механизмах - наследование, статические элементы, абстрактные элементы, интерфейсы, полиморфизм, разграничения доступа и другие. Все они позволяют избегать дублирующего кода, облегчают развитие системы, добавление новых возможностей и изменение старых, помогают обеспечивать минимальную связность между частями системы, то есть, повышают модульность. Также удачные технические решения можно многократно использовать в различных системах, сокращая и упрощая процесс их создания.

Для достижения таких важных целей требуется не только знание Java, но и владение объектно-ориентированным подходом, основными способами проектирования систем и проверки качества архитектурных решений. Платформа Java является основой и весьма удобным инструментом для применения всех этих технологий.

# 7. Заключение

В этой главе были рассмотрены особенности объектной модели Java. Это, во-первых, статические элементы, позволяющие использовать интерфейс класса без создания объектов. Нужно помнить, что, хотя для обращения к статическим элементам можно использовать ссылочную переменную, на самом деле ее значение не используется, компилятор основывается только на ее типе.

Для правильной работы со статическими элементами вводятся понятия статического и динамического контекста.

Далее рассматривалось использование ключевых слов this и super. Выражение this предоставляет ссылку, указывающую на объект, в контексте которого оно встречается. Оно помогает избегать конфликтов имен, а также применяется в конструкторах.

Слово super предоставляет возможность использовать свойства родительского класса, что необходимо для реализации переопределенных методов, а также в конструкторах.

Затем было введено понятие абстрактного метода и класса. Абстрактный метод не имеет тела, он лишь указывает, что метод с такой сигнатурой должен быть реализован в классе-наследнике. Поскольку он не имеет собственной реализации, классы с абстрактными методами также должны быть объявлены с модификатором abstract, который указывает, что от них нельзя порождать объекты. Основная цель абстрактных методов – описать в родительском классе как можно больше общих свойств наследников, пускай даже и в виде заголовков методов без реализации.

Следующее важное понятие – особый тип в Java, интерфейс. Его еще называют полностью абстрактным классом, так как все его методы обязательно абстрактные, а поля final static. Соответственно, на основе интерфейсов невозможно создавать объекты.

Интерфейсы являются альтернативой множественному наследованию. Классы не могут иметь более одного родителя, но они могут реализовывать сколько угодно интерфейсов. Таким образом, интерфейсы описывают общие свойства классов, не находящихся на одной ветви дерева наследования.

Наконец важным свойством объектной модели является полиморфизм. Были подробно изучены детали поведения полей и методов, как статических, так и динамических, при переопределении. После этого рассмотрения становится возможным развить вопрос соответствия типов переменной и ее значения.

# 8. Контрольные вопросы

8-1. Предположим, вы моделируете автомобиль, описывая его свойства в формате Java-класса. Какие из следующих полей нужно объявить динамическими, а какие – статическими?

- количество колес автомобиля;
- необходимое количество колес, полагающееся по проектной документации;
- максимально допустимая масса для этого класса автомобилей;
- максимально большое количество пассажиров, когда-либо одновременно перевозимых автомобилем;
- дата начала выпуска автомобилей;
- дата выпуска автомобиля.

a.) 1, 4, 6 – динамические, поскольку описывают свойства конкретного автомобиля

2, 3, 5 – статические, поскольку описывают свойства, присущие всем автомобилям этого класса.

8-2. Корректно ли следующее обращение к переменной x?

```
public class Test {
   static void perform() {
      ...
   }
   private Test x;

   public static void main(String s[]) {
      x.perform(); // корректно ли это выражение?
   }
}
```

a.) Нет, не корректно. Хотя при обращении к статическим элементам через имя переменной, используется лишь ее тип, а не значение, в данном примере производится попытка обратиться к динамической переменной из статического метода, чего делать нельзя, несмотря на то, что для вычисления выражения требуется лишь тип переменной.

8-3. Что окажется на консоли после выполнения следующей программы?

```
public class Parent {
   int x=2;

}

public class Child extends Parent {
   int x=3;

   void print(int x) {
      System.out.println(x);
      System.out.println(this.x);
      System.out.println(super.x);
   }

   public static void main(String s[]) {
      new Child().print(0);
   }
}
```

a.) Результатом будет:

```
0
3
2
```

В первом случае распечатывается значение аргумента метода, который передается из метода main, то есть 0. Во втором случае распечатывается значение переменной, объявленной в классе Child, то есть 3. Наконец, в

последнем случае распечатывается значение переменной, унаследованной от родительского класса Parent, то есть 2.

8-4. Для каких целей может быть использовано ключевое слово this?

a.) Для следующих целей:

- обращение из первой строки конструктора к другому конструктору этого же класса.
- получение ссылки на текущий объект, что может потребоваться в следующих случаях:
  - передача ссылки на сам объект в качестве аргумента вызываемого метода
  - разрешения конфликта имен в случае «затеняющих» объявлений
  - использование для инициализации одного поля другого поля, объявленного ниже
- также это слово применяется при работе с внутренними типами, что выходит за рамки этого курса

8-5. Можно ли при переопределении некоторого абстрактного метода perform() использовать выражение super.perform()?

a.) Нет, выражение super.perform() означает полноценный вызов родительского метода, что невозможно, если у него отсутствует тело, что верно для абстрактных методов.

8-6. Можно ли при наследовании не реализовывать абстрактный метод родительского класса?

a.) Можно, но тогда наследник должен оставаться абстрактным.

8-7. Если есть переменная типа абстрактный класс, можно ли с ее помощью обращаться к абстрактным методам этого класса?

a.) Да, поскольку ее значение, не равное null, будет ссылаться на объект, порожденный от неабстрактного класса-наследника. Следовательно, в нем реализованы все абстрактные методы.

8-8. Какие модификаторы элементов интерфейса подставляются по умолчанию, а потому не рекомендованы для явного указания?

a.) Для полей – public final static.

Для методов – public abstract.

8-9. Возможно ли не реализовывать все методы из интерфейса, указанного в выражении implements?

a.) Да, но такой класс должен быть объявлен абстрактным.

Для методов – public abstract.

8-10. Есть ли какие-либо ограничения на набор интерфейсов, которые может реализовывать класс?

a.) Да, они не могут иметь различных методов с одинаковыми сигнатурами, то есть различающихся типом возвращаемого значения.

8-11. Для каких элементов класса работает полиморфизм?

a.) Только для динамических методов.

8-12. Какое значение появится на консоли после выполнения следующей программы?

```
public class Parent {
   int x = 2;
   public void print() {
      System.out.println(x);
   }
}

public class Child extends Parent {
   int x = 3;

   public static void main(String s[]) {
      new Child().print();
   }
}
```

a.) Появится число 2, так как выводом занимается метод класса Parent, то и переменная будет использована та, что объявлена в этом классе.

8-13. Изменится ли результат программы из предыдущего вопроса, если добавить в объявление класса Child следующие строки?

```
public void print() {
   System.out.println(x);
}
```

a.) Хотя переопределенный метод выглядит точно так же, как и родительский, однако теперь он использует переменную класса Child, поэтому результатом будет 3.

8-14. Корректен ли следующий пример, и если да, то что появится после его выполнения?

```
public class Test {
   public static void test(Test t) {
      System.out.println("test "+t);
   }

   public static void main(String s[]) {
      Test t = null;
      t.test(t);
```

```
        }
    }
```

а.) Пример корректен, значение типа null подходит как для обращения к статическому элементу, так и для передачи в качестве аргумента. Результатом работы метода test станет текст test null.

8-15. Может ли переменная иметь тип абстрактный класс? Интерфейс? Если да, то какие значения она может хранить?

а.) Оба ответа – да, может. В обоих случаях переменные могут принимать значение null. Также переменная типа абстрактный класс может ссылаться на объекты, порожденные от неабстрактных классов-наследников. В случае переменных типа интерфейс, они могут ссылаться на объекты неабстрактных классов, реализующих этот интерфейс.

# Программирование на Java

## Лекция 9. Массивы

### 20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Лекция посвящена рассмотрению массивов в Java. Массивы издавна присутствуют в языках программирования, поскольку многие задачи требуют оперировать с целым рядом однотипных значений.

Массивы в Java – один из ссылочных типов, который однако имеет особенности при инициализации, создании и оперировании со своими значениями. Наибольшие различия проявляются при преобразовании таких типов. Также рассматривается, почему многомерные массивы в Java можно (и зачастую более правильно) рассматривать как одномерные. Завершается классификация типов переменных и типов значений, которые они могут хранить.

В заключение рассматривается механизм клонирования Java, позволяющий в любом классе легко описать возможность создавать точные копии объектов, порожденных от него.

# Оглавление

# Лекция 9. Массивы

## Содержание лекции.



## 1. Введение

Массивы являются важной составляющей языка программирования. В Java работа с массивами во многом похожа на то, как это делается в других языках, но есть также и важные особенности. Все они рассматриваются в этой главе. Завершается она изучением механизма клонирования в Java, в том числе, в применении к массивам.

## 2. Массивы, как тип данных в Java

В отличие от обычных переменных, которые хранят ровно одно значение, массивы (arrays) используются для хранения целого набора значений. Количество значений в массиве называется его длиной, сами значения - элементами массива. Значений может не быть вовсе, в этом случае массив считается пустым, а его длина равной нулю.

Элементы не имеют имен, доступ к ним осуществляется по номеру индекса. Если массив имеет длину n, отличную от нуля, то корректными значениями индекса являются числа от 0 до n-1. Все значения имеют одинаковый тип, и говорится, что массив основан на этом базовом типе. Массивы могут быть основаны как на примитивных типах (например, для хранения числовых значений 100 измерений), так и на ссылочных (например, если нужно хранить описание 100 автомобилей в гараже в виде экземпляров класса Car).

Сразу оговоримся, что в Java массив символов char[] и класс String являются различными типами. Их значения могут быть легко конвертированы друг в друга с помощью специальных методов, но они все же не относятся к идентичным типам.

Как уже говорилось, в Java массивы являются объектами (примитивных типов в Java всего восемь, и их количество не меняется), их тип напрямую наследуется от класса Object, поэтому все элементы этого класса доступны у объектов-массивов.

Базовый тип может также быть массивом. Таким образом конструируется массив массивов, или многомерный массив.

Работа с любым массивом включает в себя обычные операции, уже описанные для других типов - объявление, инициализация, рассмотрение элементов и т.д. Начнем последовательно изучать их в приложении к массивам.

## 2.1. Объявление массивов

В качестве примера рассмотрим объявление переменной типа массив, основанный на примитивном типе int

```
int a[];
```

Как видно, сначала указывается базовый тип. Затем идет имя переменной, а пара квадратных скобок указывает, что используемый тип является именно массивом. Также допустимой записью является:

```
int[] a;
```

Количество пар квадратных скобок указывает на размерность массива. Для многомерных массивов допускается смешанная запись:

```
int[] a[];
```

Переменная a имеет тип "двумерный массив, основанный на int". Аналогично объявляются массивы с базовым объектным типом:

```
Point p, p1[], p2[][];
```

Создание переменной типа массив еще не создает экземпляры этого массива. Такие переменные имеют объектный тип и хранят ссылки на объекты, однако изначально имеют значение null (если они являются полями класса; напомним, что локальные переменные необходимо явно инициализировать). Чтобы создать экземпляр массива, нужно воспользоваться ключевым словом new, после чего указывается тип массива и в квадратных скобках указывается длина массива.

```
int a[]=new int[5];
Point[] p = new Point[10];
```

Переменная инициализируется ссылкой, указывающей на только что созданный массив. После его создания можно обращаться к элементам, используя ссылку на массив, далее в квадратных скобках указывается индекс элемента. Индекс меняется от нуля, пробегая

всю длину массива, до максимально допустимого значения, на единицу меньшего длины массива.

```
int array[]=new int[5];
for (int i=0; i<5; i++) {
   array[i]=i*i;
}
for (int j=0; j<6; j++) {
   System.out.println(j+"*"+j+"="+array[j]);
}
```

Результатом выполнения программы будет:

```
0*0=0
1*1=1
2*2=4
3*3=9
4*4=16
```

И далее появится ошибка времени исполнения, так как индекс превысит максимально возможное значение для такого массива. Проверка, что индекс не выходит за допустимые пределы, происходит только во время исполнения программы, т.е. компилятор не пытается выявить такую ошибку, даже в таких явных случаях, как, например:

```
int i[]=new int[5];
i[-2]=0; // ошибка! Индекс не может быть отрицательным
```

Ошибка возникнет только на этапе выполнения программы.

Хотя при создании массива необходимо указывать его длину, это значение не входит в определение типа массива, важна лишь размерность. Таким образом, одна переменная может ссылаться на массивы разной длины:

```
int i[]=new int[5];
...
i=new int[7]; // переменная та же, длина массива другая
```

Однако, для объекта массива длина обязательно должна указываться при создании и уже никак не может быть изменена. В последнем примере для присвоения переменной ссылки на массив большей длины потребовалось породить новый экземпляр.

Поскольку для экземпляра массива длина является постоянной характеристикой, для всех массивов существует специальное поле length, позволяющее узнать ее значение. Например:

```
Point p[]=new Point[5];
for (int i=0; i<p.length; i++) {
   p[i]=new Point(i, i);
}
```

Значение индекса массива всегда имеет тип int. При обращении к элементу можно также использовать byte, short или char, поскольку эти типы автоматически расширяются до int. Попытка использовать long приведет к ошибке компиляции.

Соответственно, и поле length имеет тип int, а теоретическая максимально возможная длина массива равняется 2^31-1, то есть немногим больше 2 млрд.

Продолжая рассмотрение типа массива, подчеркнем, что в качестве базового типа может использоваться любой тип Java, в том числе:

- интерфейсы. В таком случае элементы массива могут иметь значение null или ссылаться на объекты любого класса, реализующего этот интерфейс.
- абстрактные классы. В этом случае элементы массива могут иметь значение null или ссылаться на объекты любого неабстрактного класса-наследника.

Поскольку массив является объектным типом данных, его значения могут быть приведены к типу Object или, что то же самое, быть присвоены переменной типа Object. Например,

```
Object o = new int[4];
```

Это дает интересную возможность для массивов, основанных на типе Object, хранить в качестве элемента ссылку на самого себя:

```
Object arr[] = new Object[3];
arr[0]=new Object();
arr[1]=null;
arr[2]=arr; // Элемент ссылается на весь массив!
```

## 2.2. Инициализация массивов

Теперь, когда понятно, как создавать экземпляры массива, рассмотрим, какие значения принимают его элементы.

Если создать массив на основе примитивного числового типа, то изначально после создания все элементы массива имеют значение по умолчанию, то есть 0. Если массив объявлен на основе примитивного типа boolean, то и в этом случае все элементы будут иметь значение по умолчанию false. Выше рассматривался пример инициализации элементов с помощью цикла for.

Рассмотрим создание массива на основе ссылочного типа. Предположим, это будет класс Point. При создании экземпляра массива с применением ключевого слова new не создается ни один объект класса Point, создается лишь один объект массива. Каждый элемент массива будет иметь пустое значение null. В этом можно убедится с помощью простого примера:

```
Point p[]=new Point[5];
for (int i=0; i<p.length; i++) {
   System.out.pritnln(p[i]);
}
```

Результатом будут лишь слова null.

Далее нужно инициализировать элементы массива по отдельности, например, в цикле. Вообще, создание массива длиной n можно рассматривать как заведение n переменных, и работать с элементами массива (в последнем примере p[i]) по правилам обычных переменных.

Кроме того, есть и другой способ создания массивов - инициализаторы. В этом случае ключевое слово new не используется, а ставятся фигурные скобки, и в них перечисляются через запятую значения всех элементов массива. Например, для числового массива явная инициализация записывается следующим образом:

```
int i[]={1, 3, 5};
int j[]={}; // эквивалентно new int[0]
```

Длина массива вычисляется автоматически, исходя из количества введенных значений. Далее создается массив такой длины, и каждому его элементу присваивается указанное значение.

Аналогично можно порождать массивы на основе объектных типов, например:

```
Point p=new Point(1,3);
Point arr[]={p, new Point(2,2), null, p};
// или
String sarr[]={"aaa", "bbb", "cde"+"xyz"};
```

Однако инициализатор нельзя использовать для анонимного создания экземпляров массива, то есть не для инициализации переменной, а, например, для передачи параметров метода или конструктора.

Например:

```
public class Parent {
   private String[] values;

   protected Parent(String[] s) {
      values=s;
   }
}

public class Child extends Parent {

   public Child(String firstName, String lastName) {
      super(???); // требуется анонимное создание массива
   }
}
```

В конструкторе класса Child необходимо сделать обращение к конструктору родителя и передать в качестве параметра ссылку на массив. Теоретически можно передать null, но это приведет в большинстве случаев к некорректной работе классов. Можно вставить выражение new String[2], но тогда вместо значений firstName и lastName будут переданы пустые строки. Попытка записать {firstName, lastName} приведет к ошибке компиляции, так можно только инициализировать переменные.

Правильное выражение выглядит так:

```
new String[]{firstName, lastName}
```

Что является некоторой смесью выражения, создающего массивы с помощью new и инициализатора. Длина массива определяется количеством указанных значений.

## 2.3. Многомерные массивы

Теперь перейдем к рассмотрению многомерных массивов. Например, в следующем примере:

```
int i[][]=new int[3][5];
```

переменная i ссылается на двумерный массив, который можно представить себе в виде таблицы 3x5. Суммарно в таком массиве содержится 15 элементов, к которым можно обращаться через комбинацию индексов от (0, 0) до (2, 4). Пример заполнения двумерного массива через цикл:

```
int pithagor_table[][]=new int[5][5];
for (int i=0; i<5; i++) {
   for (int j=0; j<5; j++) {
      pithagor_table[i][j]=i*j;
      System.out.print(pithagor_table[i][j]+ "\t");
   }
   System.out.println();
}
```

Результатом выполнения программы будет:

```
0   0   0   0   0
0   1   2   3   4
0   2   4   6   8
0   3   6   9   12
0   4   8   12   16
```

Однако, такой взгляд на двумерные и многомерные массивы является неполным. Более точный подход заключается в том, что в Java нет двумерных, и вообще многомерных, массивов, а есть массивы, базовыми типами которых являются также массивы. Например, тип int[] означает "массив чисел", а int[][] означает "массив массивов чисел". Поясним такую точку зрения следующими подробностями.

Если создать двумерный массив и определить переменную x, которая на него ссылается, то используя x и два числа в паре квадратных скобок каждое (например, x[0][0]), можно обратиться к любому элементу двумерного массива. Но в то же время используя x и одно число в паре квадратных скобок, можно обратиться к одномерному массиву, который является элементом двумерного массива. Его можно проинициализировать новым массивом с некоторой другой длиной, и таблица перестанет быть прямоугольной - она примет произвольную форму. В частности, можно одному из одномерных массивов присвоить даже значение null.

```
int x[][]=new int[3][5]; // прямоугольная таблица
x[0]=new int[7];
x[1]=new int[0];
x[2]=null;
```

После таких операций массив, на который ссылается переменная x назвать прямоугольным никак нельзя. Зато хорошо видно, что это просто набор одномерных массивов или значений null.

Полезно подсчитать, сколько объектов порождается выражением new int[3][5]. Правильный подсчет таков: создается один массив массивов (1 объект) и 3 массива чисел, каждый длиной 5 (3 объекта). Итого, 4 объекта.

В рассмотренном примере 3 из них (массивы чисел) были тут же переопределены новыми значениями. Для таких случаев полезно использовать упрощенную форму выражения создания массивов:

```
int x[][]=new int[3][];
```

Такая запись порождает один объект - массив массивов, и заполняет его значениями null. Теперь понятно, что и в этом, и в предыдущем варианте выражение x.length возвращает значение 3 - длину массива массивов. Далее можно с помощью выражений x[i].length узнать длину каждого вложенного массива чисел при условии, что i неотрицательно и меньше x.length, а также x[i] не равно null. Иначе будут возникать ошибки во время выполнения программы.

Вообще при создании многомерных массивов с помощью new необходимо указывать все пары квадратных скобок, соответственно количеству измерений. Но заполненной обязательно должна быть лишь самая левая пара, это значение задаст длину самого верхнего массива массивов. Если заполнить следующую пару, то этот массив заполнится не значениями по умолчанию null, а новыми созданными массивами с меньшей на единицу размерностью. Если заполнена вторая пара скобок, то можно заполнить третью и так далее.

Аналогично, для создания многомерных массивов можно использовать инициализаторы. В этом случае используется столько вложенных фигурных скобок, сколько требуется:

```
int i[][] = {{1,2}, null, {3}, {}};
```

В этом примере порождается 4 объекта. Это, во-первых, массив массивов длиной 4, а во-вторых, 3 массива чисел с длинами 2, 1, 0 соответственно.

Все рассмотренные примеры и утверждения одинаково верны для многомерных массивов, основанных как на примитивных, так и на ссылочных типах.

## 2.4. Класс массива

Поскольку массив является объектным типом данных, то можно попытаться представить себе, как бы выглядело объявление класса такого типа. На самом деле эти объявления не хранятся в файлах или еще каком-нибудь формате. Учитывая, что массив может быть

объявлен на основе любого типа и иметь произвольную размерность, это физически невыполнимо, да и не требуется. Вместо этого во время выполнения приложения виртуальная машина генерирует их динамически на основе базового типа и размерности, и затем они хранятся в памяти в виде таких же экземпляров класса Class, как и для любых других типов.

Рассмотрим гипотетическое объявление класса для массива, основанного на неком объектном типе Element.

Объявление класса начинается с перечисления модификаторов, среди которых особую роль занимают модификаторы доступа. Класс массива будет иметь такой же уровень доступа, как и базовый тип. Т.е., если Element объявлен как public-класс, то и массив будет иметь уровень доступа public. Для любого примитивного типа класс массива будет public. Можно также указать модификатор final, поскольку никакой класс не может наследоваться от класса массива.

После этого идет имя класса, на котором можно подробно не останавливаться, т.к. к типу массиву обращение идет не по его имени, а по имени базового типа и набору квадратных скобок.

Затем нужно указать родительский класса. Все массивы наследуются напрямую от класса Object. Далее перечисляются интерфейсы, которые реализует класс. Для массива это будут интерфейсы Cloneable и Serializable. Первый из них подробно рассматривается в конце этой главы, а второй будет рассмотрен в следующих главах.

Тело класса содержит объявление одного public final поля length типа int. Кроме того переопределен метод clone() для поддержки интерфейса Cloneable.

Сведем все вышесказанное в формальную запись класса:

```
[public] class A implements Cloneable, java.io.Serializable {
   public final int length; // инициализируется при создании

   public Object clone() {
      try {
         return super.clone();
      } catch (CloneNotSupportedException e) {
         throw new InternalError(e.getMessage());
      }
   }
}
```

Таким образом, тип массивы является полноценным объектом, который, в частности, наследует все методы, определенные в классе Object, например, toString(), hashCode() и остальные.

Например:

```
// результат работы метода toString()
System.out.println(new int[3]);
System.out.println(new int[3][5]);
System.out.println(new String[2]);
```

```
// результат работы метода hashCode()
System.out.println(new float[2].hashCode());
```

Результатом выполнения программы будет:

```
[I@26b249
[[I@82f0db
[Ljava.lang.String;@92d342
7051261
```

# 3. Преобразование типов для массивов

Теперь, когда массив введен как полноценный тип данных в Java, рассмотрим, какое влияние он окажет на вопрос преобразования типов.

Ранее подробно рассматривались переходы между примитивными и обычными (не являющимися массивами) ссылочными типами. Хотя массивы являются объектными типами, их также будет полезно разделить по базовому типу на две группы - основанные на примитивном или ссылочном типе.

Сразу скажем, что переходы между массивами и примитивными типами являются запрещенными. Преобразования между массивами и другими объектными типами возможны только для класса Object и интерфейсов Cloneable и Serializable. Массив всегда можно привести к этим 3 типам, обратный же переход является сужением, и должен производится явным образом по усмотрению разработчика. Таким образом, интерес представляют только переходы между разными типами массивов. Очевидно, что массив, основанный на примитивном типе, принципиально нельзя преобразовать к типу массива, основанному на ссылочном типе, и наоборот.

Пока не будем подробно на этом останавливаться, но заметим, что преобразования между типами массивов, основанных на различных примитивных типах, невозможно ни при каких условиях.

Для ссылочных же типов такого строгого правила нет. Например, если создать экземпляр массива, основанного на типе Child, то ссылку на него можно привести к типу массива, основанного на типе Parent.

```
Child c[] = new Child[3];
Parent p[] = c;
```

Вообще, существует универсальное правило: массив, основанный на типе A, можно привести к массиву, основанному на типе B, если сам тип A приводится к типу B.

```
// Если допустимо такое приведение:
B b = (B) new A();
// то допустимо и приведение массивов:
B b[]=(B[]) new A[3];
```

Применяя это правило рекурсивно можно преобразовывать многомерные массивы. Например, массив Child[][] можно привести к Parent[][], так как их базовые типы приводимы (Child[] к Parent[]) также на основе этого правила (поскольку базовые типы Child и Parent приводимы в силу правил наследования).

Как обычно, расширения можно проводить неявно (как записано в предыдущем примере), а сужения - только явным приведением.

Вернемся к массивам, основанным на примитивном типе. Невозможность их участия в преобразованиях типов связана, конечно, с различиями между простыми и ссылочными типами данных. Поскольку элементами объектных массивов являются ссылки, то они легко могут участвовать в приведении. Напротив, элементы простых типов действительно хранят числовые или булевские значения. Предположим, такое преобразование осуществимо:

```
// пример вызовет ошибку компиляции
byte b[]={1, 2, 3};
int i[]=b;
```

В таком случае, элементы b[0] и i[0] хранили бы значения разных типов. Стало быть преобразование потребовало бы копирования с одновременным преобразованием типа всех элементов исходного массива. В результате был бы создан новый массив, элементы которого равнялись бы по значению элементам исходного массива.

Но преобразование типа не может порождать новые объекты. Такие операции должны делаться только явным образом с применением ключевого слова new. По этой причине преобразования типов массивов, основанных на примитивных типах, запрещены.

Если же копирование элементов действительно требуется, то нужно сначала создать новый массив, а затем воспользоваться стандартной функцией System.arrayCopy(), которая эффективно производит копирование элементов одного массива в другой.

## 3.1. Ошибка ArrayStoreException

Преобразование между типами массивов, основанных на ссылочных типах, может стать причиной одной, довольно неочевидной ошибки.

Рассмотрим пример:

```
Child c[] = new Child[5];
Parent p[]=c;
p[0]=new Parent();
```

С точки зрения компилятора код совершенно корректен. Преобразование во второй строке допустимо. В третьей строке элементу массива типа Parent присваивается значение того же типа.

Однако при выполнении такой программы возникнет ошибка. Нельзя забывать, что преобразование не меняет объект, изменяется лишь способ доступа к нему. В свою очередь объект всегда "помнит", от какого типа он был порожден. С учетом этих замечаний становится ясно, что в третьей строке делается попытка добавить в массив Child значение типа Parent, что некорректно.

Действительно, ведь переменная c продолжает ссылаться на этот массив, а значит следующей строкой может быть следующее обращение:

```
c[0].onlyChildMethod();
```

где метод onlyChildMethod() определен только в классе Child. Такое обращение совершенно корректно, а значит недопустима ситуация, когда элемент c[0] ссылается на объект, несовместимый с Child.

Таким образом, несмотря на отсутствие ошибок компиляции, виртуальная машина при выполнении программы всегда делает дополнительную проверку перед присвоением значения элементу массива. Необходимо удостоверится, что реальный массив, существующий на момент исполнения, действительно может хранить присваиваемое значение. Если это условие нарушается, то возникает ошибка, которая называется ArrayStoreException.

Может сложиться впечатление, что разобранная ситуация является надуманной - зачем преобразовывать массив и тут же класть в него неверное значение? Однако преобразование при присвоении значений является лишь примером. Рассмотрим объявление метода:

```
public void process(Parent[] p) {
   if (p!=null && p.length>0) {
      p[0]=new Parent();
   }
}
```

Метод выглядит абсолютно корректным, все потенциально ошибочные ситуации проверяются if-выражением. Однако следующий вызов этого метода все равно приводит ошибке:

```
process(new Child[3]));
```

И это будет как раз ошибка ArrayStoreException.

## 3.2. Переменные типа массив, и их значения

Завершим рассмотрение, которое делалось на протяжении предыдущих глав, взаимосвязи типа переменной и типа значений, которая она может хранить.

Как обычно, массивы, основанные на простых и ссылочных типах, описываем раздельно.

Переменная типа массив примитивных величин может хранить значения только точно такого же типа, либо null.

Переменная типа массив ссылочных величин может хранить следующие значения:

- null;
- значения точно того же типа, что и тип переменной;
- все значения типа массив, основанный на типе, приводимом к базовому типу исходного массива.

Все эти утверждения непосредственно следуют из рассмотренных выше особенностей приведения типов массивов.

Еще раз напомним про исключительный класс Object. Переменные такого типа могут ссылаться на любые объекты, порожденные как от классов, так и от массивов.

Сведем все эти утверждения в таблицу:

| Тип переменной | Допустимые типы ее значения |
|---|---|
| Массив простых значений | • null<br>• в точности совпадающий с типом переменной |
| Массив ссылочных значений | • null<br>• совпадающий с типом переменной<br>• массивы ссылочных значений, удовлетворяющих следующему условию: Если тип переменной - массив на основе типа A, то значение типа массив на основе типа B допустимо тогда и только тогда, когда B приводимо к A. |
| Object | • null<br>• любой ссылочный, включая все массивы |

# 4. Клонирование

Механизм клонирования, как следует из названия, позволяет порождать новые объекты на основе существующего, которые бы обладали точно таким же состоянием, что и исходный. То есть, ожидается, что для исходного объекта, представленного ссылкой x, и результата клонирования, возвращаемого методом x.clone(), выражение

```
x != x.clone()
```

должно быть истинным, также как и выражение

```
x.clone().getClass() == x.getClass()
```

и, наконец, выражение

```
x.equals(x.clone())
```

также верно. Реализация такого метода clone() осложняется целым рядом потенциальных проблем, например:

- класс, от которого порожден объект, может иметь разнообразные конструкторы, которые к тому же могут быть недоступны (например, модификатор доступа private);
- цепочка наследования, которой принадлежит исходный класс, может быть довольно длинной, и каждый родительский класс может иметь свои поля, которые являются недоступными, но важными для воссоздания состояния исходного объекта;
- в зависимости от логики реализации возможна ситуация, когда не все поля должны копироваться для корректного клонирования. Какие-то могут оказаться лишними, какие потребуют дополнительных вычислений или преобразований;

- возможна ситуация, когда объект нельзя клонировать, дабы не нарушить целостность системы.

Поэтому было реализовано следующее решение.

Класс Object содержит метод clone(). Рассмотрим его объявление:

```
protected native Object clone() throws CloneNotSupportedException;
```

Именно он используется для клонирования. Далее возможно два варианта.

Во-первых, разработчик может в своем классе переопределить этот метод и реализовать его по своему усмотрению, решая перечисленные проблемы так, как этого требует логика разрабатываемой системы. Упомянутые условия, которые ожидаются быть истинными для клонированного объекта, не являются обязательными, и программист может им не следовать, если это требуется для его класса.

Второй вариант предполагает использование реализации метода clone() в самом классе Object. То, что он объявлен как native, говорит о том, что его реализация предоставляется виртуальной машиной. Понятно, что перечисленные трудности легко могут быть преодолены самой JVM, ведь она хранит в своей памяти все свойства объектов.

При выполнении метода clone() сначала делается проверка, можно ли клонировать исходный объект. Если разработчик хочет сделать объекты своего класса доступными для клонирования через Object.clone(), то он должен реализовать в своем классе интерфейс Cloneable. В этом интерфейсе нет ни одного элемента, он служит лишь признаком для виртуальной машины, что объекты допустимы для клонирования. Если проверка не выполняется успешно, метод порождает ошибку
CloneNotSupportedException

.

Если интерфейс Cloneable реализован, то порождается новый объект от точно того же класса, от которого был создан исходный объект. При этом копирование проводится на уровне виртуальной машины, никакие конструкторы не вызываются. Затем значения всех полей, объявленных, унаследованных, либо объявленных в родительских классах, копируются. Полученный объект возвращается в качестве клона.

Обратите внимание, что сам класс Object не реализует интерфейс Cloneable, а потому попытка вызова new Object().clone() будет приводить к ошибке времени исполнения. Метод clone() предназначен скорее для использования в наследниках, которые могут обращаться к нему с помощью выражения super.clone(). При этом могут быть сделаны следующие изменения:

- модификатор доступа расширен до public;
- убрано предупреждение об ошибке CloneNotSupportedException;
- результирующий объект может быть модифицирован любым образом на усмотрение разработчика.

Напомним, что все массивы реализуют интерфейс Cloneable и, таким образом, доступны для клонирования.

Важно помнить, что все поля клонированного объекта приравниваются, их значения никогда не клонируются. Рассмотрим пример:

```
public class Test implements Cloneable {
  Point p;
  int height;

  public Test(int x, int y, int z) {
   p=new Point(x, y);
   height=z;
  }

  public static void main(String s[]) {
   Test t1=new Test(1, 2, 3), t2;
   try {
   t2=(Test) t1.clone();
  } catch (CloneNotSupportedException e) {}
   t1.p.x=-1;
   t1.height=-1;
   System.out.println("t2.p.x=" + t2.p.x + ", t2.height=" + t2.height);
  }
}
```

Результатом работы программы будет:

```
t2.p.x=-1, t2.height=3
```

Из примера видно, что примитивное поле было скопировано и далее существует независимо в исходном и клонированном объектах. Изменение одного не сказывается на другом.

А вот ссылочное поле было скопировано по ссылке, оба объекта ссылаются на один и тот же экземпляр класса Point. Поэтому изменения, происходящие с исходным объектом, сказываются на клонированном.

Этого можно избежать, если переопределить метод clone() в классе Test.

```
 public Object clone() {
  Test clone=null;
  try {
  clone=(Test) super.clone();
 } catch (CloneNotSupportedException e) {
  throw new InternalError(e.getMessage());
}
clone.p=(Point)clone.p.clone();
return clone;
 }
```

Обратите внимание, что результат метода Object.clone() приходится явно приводить к типу Test, несмотря на то, что его реализация гарантирует, что клонированный объект будет порожден именно от этого класса. Однако тип возвращаемого значения в этом методе для универсальности объявлен как Object, поэтому явное сужение необходимо.

Теперь метод main можно упростить:

```
  public static void main(String s[]) {
  Test t1=new Test(1, 2, 3);
 Test t2=(Test) t1.clone();
  t1.p.x=-1;
  t1.height=-1;
  System.out.println("t2.p.x=" + t2.p.x + ", t2.height=" + t2.height);
 }
```

Результатом будет:

```
t2.p.x=1, t2.height=3
```

То есть, теперь все поля исходного и клонированного объектов стали независимыми.

Реализация такого "неглубокого" клонирования в методе Object.clone() необходима, так как в противном случае клонирование второстепенного объекта могло бы привести к огромным затратам ресурсов, ведь этот объект может содержать ссылки на более значимые объекты, а те при клонировании также начали бы копировать свои поля и так далее. Кроме этого, поля клонируемого объекта могут иметь своим типом класс, не реализующий Cloneable, что приводило бы к дополнительным проблемам. Как показано в примере, при необходимости дополнительное копирование можно легко добавить самостоятельно.

## 4.1. Клонирование массивов

Итак, любой массив может быть клонирован. В этом разделе хотелось бы рассмотреть особенности, возникающие из-за того, что Object.clone() копирует только один объект.

Рассмотрим пример:

```
int a[]={1, 2, 3};
int b[]=(int[])a.clone();
a[0]=0;
System.out.println(b[0]);
```

Результатом будет ноль, что вполне очевидно, так как весь массив представлен одним объектом, который не будет зависеть от своей копии. Усложняем пример:

```
int a[][]={{1, 2}, {3}};
int b[][]=(int[][]) a.clone();

if (...) {
// первый вариант:
a[0]=new int[]{0};
System.out.println(b[0][0]);
} else {
// второй вариант:
a[0][0]=0;
```

```
System.out.println(b[0][0]);
}
```

Разберем, что будет происходить в этих двух вариантах. Начнем с того, что в первой строке создается двухмерный массив, состоящий из 2 одномерных, итого 3 объекта. Затем, на следующей строке при клонировании будет создан новый двухмерный массив, содержащий ссылки на те же самые одномерные массивы.

Теперь несложно предсказать результат обоих вариантов. В первом случае в исходном массиве меняется ссылка, хранящаяся в первом элементе, что не принесет никаких изменений для клонированного объекта. На консоли появится 1.

Во втором случае модифицируется существующий массив, что скажется на обоих двумерных массивах. На консоли появится 0.

Обратите внимание, что если из примера убрать условие if-else, так, чтобы отрабатывал первый вариант, а затем последовательно второй, то результатом будет опять 1, поскольку в части второго варианта модифицироваться будет уже новый массив, порожденный в части первого варианта.

Таким образом, в Java предоставляется мощный, эффективный и гибкий механизм клонирования, который легко применять и модифицировать под конкретные нужды. Особенное внимание должно лишь уделяться копированию объектных полей, которые по умолчанию копируются только по ссылке.

# 5. Заключение

В этой главе были рассмотрено устройство массивов в Java. Подобно массивам в других языках, они представляют собой набор значений одного типа. Основным свойством массива является длина, которая в Java может равняться нулю. В противном случае, массив обладает элементами в количестве, равном длине, к которым можно обратиться, используя индекс, изменяющийся от 0 до величины длины без единицы. Длина задается при создании массива, и не может быть изменена у созданного массива. Однако, она не входит в определение типа, а потому одна переменная может ссылаться на массивы одного типа с различной длиной.

Создать массив можно как с помощью ключевого слова new, поскольку все массивы, включая определенные на основе примитивных значений, имеют объектный тип. Другой способ – воспользоваться инициализатором, и перечислить значения всех элементов. В первом случае элементы принимают значения по умолчанию (0, false, null).

Особым образом в Java устроены многомерные мас  сивы. Они, по сути, являются одномерными, основанными на массивах меньшей размерности. Такой подход позволяет единым образом работать с многомерными массивами. Также он позволяет создавать не только «прямоугольные» массивы, но и любой конфигурации.

Хотя массив и является ссылочным типом, работа с ним зачастую имеет некоторые особенности. Рассматриваются правила приведения типа массива. Как для любого объектного типа, приведение к Object является расширяющим. Приведение массивов, основанных на ссылочных типах, во многом подчиняется обычным правилам. А вот примитивные массивы преобразовывать нельзя. С преобразованиями связано и

возникновение ошибки ArrayStoreException, причина которой – невозможность точного отслеживания типов в преобразованном массиве для компилятора.

В заключение рассматриваются последние случаи взаимосвязи типа переменной и ее значения.

Наконец, изучается механизм клонирования, существующий с самых первых версий Java и позволяющий создавать точные копии объектов, если их классы позволяют это, реализую интерфейс Cloneable. Поскольку стандартное клонирование порождает строго один новый объект, это приводит к особым эффектам при работе с объектными полями классов и массивами.

# 6. Контрольные вопросы

9-1. Массивы каких типов и длин объявляются в следующем коде?

```
int x[], y[][];
byte[] a, b[][];
String s, s1[], s2={{}, {"a", "b"}, null};
```

a.) Ответ:

- переменная x имеет тип int[], y – int[][]. Поскольку массивы не созданы, длины у них нет.
- переменная a имеет тип byte[], b – byte[][][]. Поскольку массивы не созданы, длины у них нет.
- Переменная s не массив, s1 – String[], s2 – String[][]. Длина определена только у s2, она равна 3. Первым элементом этого двухмерного массива является одномерный массив длиной 0, вторым – массив длиной 2, третьим - null.

9-2. Корректен ли следующий код, и если нет, то в каких местах будут возникать ошибки, и какие?

```
int b[]=new int[5];
for (int i=1; i<=b.length(); i++) {
   b[i]=Math.sqrt(i);
}
```

a.) Код некорректен, в нем есть целый ряд ошибок. Во-первых, у массивов нет метода length(), есть только поле length. Во-вторых, в 3-ей строке делается попытка неявного приведения от типа double к int – результат работы метода Math.sqrt приравнивается целочисленной переменной.

Далее, во время исполнения программы будет возникать ошибка некорректного индекса массива. Если применить правильный способ получения длины массива, то на последней итерации цикла будет произведена попытка обратиться к элементу массива с индексом 5, в то

время как максимально допустимым является значение длины без единицы, то есть 4.

Наконец, перебор массива, начиная с индекса 1, пропускает элемент с индексом 0.

9-3. Может ли массив основываться на абстрактных классах? Интерфейсах? Если да, то какие значение могут принимать его элементы?

a.) Да. Элементы таких массивов будут ссылаться на объекты, порожденные от неабстрактных классов, которые являются наследниками данного абстрактного класса или реализуют данный интерфейс соответственно.

9-4. Как создать массив, эквивалентный объявляемому ниже, но без заведения переменной?

```
int x[][]=new int[2][3];
```

a.) Следующим образом:

```
new int[][]{{0, 0, 0}, {0, 0, 0}}
```

9-5. Корректен ли следующий код? Если нет, то какие исправления можно предложить?

```
byte b[]={1, 2, 3};
Object o=b;
o=new String[]{"", "a", "b"};
String s[]=o;
```

a.) Нет. В 4 строке делается попытка неявного сужения типов от Object к String[]. Такое действие нужно делать явно:

```
String s[]=(String[])o;
```

9-6. Сколько объектов порождается при инициализации массива new int[3][4]? new int[3][][]?

a.) В первом случае создается 3 одномерных массива длиной 4 и один двумерный массив, то есть всего 4 объекта.

Во втором случае создается только 1 трехмерный массив, все элементы которого null, то есть 1 объект.

9-7. От какого класса наследуются классы массивов? Какие интерфейсы реализуются? Какие элементы они объявляют или переопределяют по сравнению с родительским классом?

a.) Классы наследуются от java.lang.Object.

Реализуют 2 интерфейса – java.lang.Cloneable и java.io.Serializable.

Объявляется новое поле public final int length и переопределяется метод public Object clone().

9-8. Как определить, можно ли преобразовать один тип массива к другому?

a.) Во-первых, оба массива должны быть основаны на ссылочных типах данных.

Во-вторых, чтобы привести массив, основанный на типе A (то есть, A[]) к массиву B[], необходимо, чтобы тип A приводился к типу B.

С учетом того, что типы A и B также могут быть массивами, это правило работает и для многомерных массивов (в этом случае оно должно применяться рекурсивно).

9-9. Хотя примитивные массивы не могут участвовать в преобразованиях, однако массивы int[][] и byte[][] могут рассматриваться как одномерные объектные массивы, основанные на ссылочном типе «одномерный примитивный массив». Могут ли такие типы быть преобразованы из одного в другой?

a.) Нет. Если применять правило из предыдущего вопроса, необходимым условием является приводимость типов int[] и byte[], что неверно по тому же правилу.

9-10. Может ли возникнуть ошибка ArrayStoreException при работе следующих методов?

```
public void setCars(Car c[]) {
   c[0]=new Car();
}

public void setCars2(Car c[]) {
   if (c[0] instanceof Car) {
      c[0]=new Car();
   }
}

public void setNumbers(int x[]) {
   x[0]=0;
}
```

a.) Ошибка может возникнуть в методах setCars и setCars2, если в качестве аргумента передать массив, основанный на классе-наследнике Car. Причем, проверка во втором методе не спасает, так как оператор instanceof вернет true.

В третьем методе ошибки не будет, так как примитивные массивы хранят значения точно того типа, на котором они основаны.

9-11. Можно ли клонировать объекты следующего класса?

```
public class Point {
   private int x, y;
```

```
    public Point(int nx, int ny) {
       x=nx;
       y=ny;
    }

    public Object clone() {
       return new Point(x, y);
    }
}
```

a.) Да, определенный в этом классе метод clone работает безо всяких ошибок. То, что этот класс не реализует интерфейс Cloneable, не позволяет обращаться к методу Object.clone(), однако такая попытка и не производится.

9-12. Сколько объектов может быть создано в процессе выполнения клонирования одного объекта средствами JVM?

a.) Ровно один – сам клон.

9-13. Каков будет результат выполнения следующего кода?

```
Point p1[][]={null, {new Point(1, 1)}};
Point p2[][] = (Point[][])p1.clone();
p2[0]= new Point[]{new Point(2, 2)};
System.out.println(p1[0][0]);
```

a.) Не смотря на инициализацию первого элемента клонированного массива, на который ссылается переменная p2, первый элемент исходного массива (p1) остается равным null. Следовательно, попытка обратиться к элементу p1[0][0] приведет к ошибке (NullPointerException).

# Программирование на Java

## Лекция 10. Операторы и структура кода. Исключения

### 20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>




### Аннотация

После ознакомления с типами данных в Java, правилами объявления классов и интерфейсов, а также с массивами, из базовых свойств языка остается рассмотреть лишь управление ходом выполнения программы. Вводятся важные понятия, связанные с этой темой, описываются метки, операторы условного перехода, циклы, операторы break и continue и другие.

Следующая тема посвящена более концептуальным механизмам Java, а именно работе с ошибками, или исключительными ситуациями. Рассматриваются причины возникновения сбоев, способы их обработки, объявление собственных типов исключительных ситуаций. Описывается важное разделение всех ошибок на проверяемые и непроверяемые компилятором, а также ошибки времени исполнения.

# Оглавление

# Лекция 10. Операторы и структура кода

## Содержание лекции.

# 1. Управление ходом программы

Управление потоком вычислений является фундаментальной основой всего языка программирования. В данной главе будут рассмотрены основные языковые конструкции и способы их применения.

Синтаксис выражений весьма схож с синтаксисом языка C, что облегчает его понимание для программистов знакомых с этим языком, вместе с тем имеется ряд отличий, которые будут рассмотрены позднее и на которые следует обратить внимание.

Порядок выполнения программы определяется операторами. Операторы могут содержать в себе другие операторы или выражения.

# 2. Нормальное и прерванное выполнение операторов

Последовательность выполнения операторов может быть непрерывной, а может и прерываться (при возникновении определенных условий). Выполнение оператора может быть прервано, если в потоке вычислений будут обнаружены операторы

```
brake
continue
return
```

то управление будет передано в другое место (в соответствии с правилами обработки этих операторов, которые будут рассмотрены позже).

Нормальное выполнение оператора может быть прервано, так же, при возникновении исключительных ситуаций. Которые так же будут рассмотрены позже. Явное возбуждение исключительной ситуации с помощью оператора throw, так же прерывает нормальное выполнение оператора, и передает управление выполнением программы (далее просто управление) в другое место.

Прерывание нормального исполнения всегда вызывается определенной причиной. Приведем список таких причин

- break (без указания метки)
- break (с указанием метки)
- continue (без указания метки)
- continue (с указанием метки)
- return (с возвратом значения)
- return (без возврата значения)
- throw с указанием объекта Exception, а так же все исключения возбуждаемые виртуальной машиной Java.

Выражения так же могут завершаться нормально и преждевременно (аварийно). В данном случае термин аварийно вполне применим, т.к. причиной последовательности выполнения выражения отличной от нормальной может быть только возникновение исключительной ситуации.

Если в операторе содержится выражение, то в случае его аварийного завершения, выполнение оператора тоже будет завершено преждевременно. (т.е. нормальный ход выполнения оператора будет нарушен)

В случае если в операторе имеется вложенный оператор, и происходит ненормальное его завершение, то так же не нормально завершается оператора содержащего вложенный (в некоторых случаях это не так, но будет оговариваться особо)

# 3. Блоки и локальные переменные

Блок это последовательность операторов, объявлений локальных классов или локальных переменных заключенных в скобки. Область видимости локальных переменных и классов ограничена блоком, в котором они определены.

При обращении к локальным переменным не может быть использован квалификатор this или имя класса.

```
1. public class Test {
2.     static int x;
3.     public Test() {
4.     }
5.     public static void main(String[] args) {
6.       Test t = new Test();
7.       lbl: {
8.           int x = this.x;
9.           if ( x > 0) break lbl;
10.       }
11.     }
12. }
```

В строке 18 мы получим ошибку компиляции. Если строку заменить на int x = x;, то тоже будет получена ошибка компиляции. (т.к. локальная переменная не инициализирована переед первым использованием). А вот такое решение будет рабочим int x = (x=2)x;

Операторы в блоке выполняются слева направо, сверху вниз. Если все операторы (выражения) в блоке выполняются нормально, то весь блок выполняется нормально. Если какой - либо оператор (выражение) завершается ненормально, то весь блок завершается ненормально.

Нельзя объявлять несколько локальных переменных в пределах видимости блока. Приведенный ниже код вызовет ошибку времени компиляции.

```
public class Test {

    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int x;
        lbl: {
```

```
            int x = 0;
            System.out.println("X = " + x);
        }
    }
}
```

В то же время не следует забывать, что локальные переменные перекрывают видимость переменных-членов. Например, этот пример отработает нормально.

public class Test { static int x = 5; public Test() { } public static void main(String[] args) { Test t = new Test(); int x = 1; System.out.println("X = " + x); } }

И на консоль будет выведено X = 1. Следует напомнить, что перекрытие локальными переменными области видимости глобальных переменных является, частой, но трудно обнаруживаемой ошибкой.

То же самое правило применимо к фактическим параметров методов.

```
public class Test {
    static int x;
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.test(5);
        System.out.println("Member value x = " + x);
    }
    private void test(int x){
        this.x = x + 5;
        System.out.println("Local value x = " + x);
    }
}
```

В результате работы этого примера на консоль будет выведено следующее.

```
Local value x = 5
Member value x = 10
```

На следующем примере продемонстрируем, что область видимости локальной переменной ограничено областью видимости блока или оператора в пределах которого данная переменная объявлена.

```
public class Test {
    static int x = 5;
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        {
            int x = 1;
            System.out.println("First block x = " + x);
```

```
        }
        {
            int x = 2;
            System.out.println("Second block x =" + x);
        }
        System.out.print("For cycle x = ");
        for(int x =0;x<5;x++){
            System.out.print(" " + x);
        }

    }
}
```

Данный пример откомпилируется без ошибок и на консоль будет выведен следующий результат:

```
First block x = 1
Second block x =2
For cycle x = 0 1 2 3 4
```

Следует помнить, что определение локальной переменной есть исполняемый оператор. Если задана инициализация переменой, то выражение исполняется слева направо и его результат присваивается локальной переменной. Использование не инициализированных локальных переменных запрещено и вызывает ошибку компиляции.

Следующий пример кода

```
public class Test {
    static int x = 5;
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int x;
        int y = 5;
        if( y > 3) x = 1;
        System.out.println(x);
    }
}
```

вызовет ошибку времени компиляции, т.к. возможны условия, когда переменная x может быть не инициализирована до ее использования. (Несмотря на то, что в данном случае оператор if(y > 3)  и следующее за ним выражение  x = 1;  будут выполняться всегда)

# 4. Пустой оператор

; Является пустым оператором. Данная конструкция вполне применима там, где не предполагается выполнение никаких действий. Преждевременное завершение пустого оператора невозможно.

# 5. Метки

Любой оператор или блок может иметь метку. Метку можно указывать в качестве параметра для операторов break и continue. Область видимости метки ограничивается оператором или блоком, к которому она относится. Так в следующем примере мы получим ошибку компиляции.

```
public class Test {
    static int x = 5;
    static {

    }
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int x = 1;
        Lbl1:{
            if(x == 0) break Lbl1;
        }

        Lbl2:{
            if(x > 0) break Lbl1;
        }
    }
}
```

В случае если имеется несколько вложенных блоков и операторов, то метки внешних блоков будут видимы во внутренних.

Этот пример является вполне корректным.

```
public class Test {
    static int x = 5;
    static {

    }
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int L2 = 0;
```

```
        Test: for(int i = 0; i< 10;i++) {
            test: for(int j = 0; j< 10;j++) {
                if( i*j > 50) break Test;
            }
        }
    }
    private void test(){
        ;
    }
}
```

В этом же примере можно увидеть, что метки используют пространство имен отличное от пространства имен переменных, методов и классов.

Традиционно использование меток не рекомендуется, особенно в объектно-ориентированных языках, поскольку серьезно усложняет понимание порядка выполнения кода, а значит и его тестирование и отладку. Для Java этот запрет можно считать не столь строгим, поскольку самый опасный метод goto отсутствует. В некоторых ситуациях (как в рассмотренном примере со вложенными циклами) метки необходимы, но, конечно, их применение следует ограничивать лишь самыми необходимыми случаями.

# 6. Оператор if

Пожалуй, наиболее часто встречающейся конструкцией в Java, как и в любом другом структурном языке программирования, является оператор условного перехода.

В общем случае конструкция выглядит так:

```
if (логическое выражение) выражение или блок 1
else выражение или блок 2
```

Логическое выражение может быть любой языковой конструкцией, которая возвращает булевский результат. Заметим отличие от языка C, в котором в качестве логического выражения может быть использованы различные типы данных, где отличное от нуля выражение трактуется как истинное значение, а ноль как ложное. В Java возможно использование только логических выражений.

В случае если логическое выражение принимает значение истина, то выполняется выражение или блок 1, в противном случае выражение или блок 2. Вторая часть оператора (else) не является обязательной и может быть опущена. Т.е. конструкция if(x = 5) System.out.println("Five") является вполне допустимой.

Операторы if-else могут каскадироваться. . Иногда это называют многозвенный if else.

```
String test = "smb";
if( test.equals("value1"){
 ….
} else if (test.equals("value2"){
 …
} else if (test.equals("value3"){
```

```
 …
} else {
 …
}
```

Следует помнить, что оператор else относится к ближайшему к нему оператору if. в данном случае последнее условие else будет выполняться только в том случае, если не выполнено предыдущее. Заключительная конструкция else относится к самому последнему условию if b,и будет выполнено только в том случае, если ни одно из вышеперечисленных условий не будет истинным. В случае если одно из условий выполнено, то все последующие выполняться не будут.

Например:

```
…
int x = 5;
if( x < 4){
 System.out.println("Меньше 4");
} else if (x > 4){
 System.out.println("Больше 4");
} else if (x == 5){
 System.out.println("Равно 4");
} else{
 System.out.println("Другое значение");
}
```

Предложение "Равно 4" в данном случае напечатано не будет.

Следует обратить внимание на то, что в условии могут быть использованы только логические выражения.

Например

```
int x = 0;
if(x) ….
```

вызовет ошибку компиляции.

Выражение вот такого типа тоже будет ошибочным

```
int x = 0;
if( x = 5) ….
```

так как здесь происходит не операция сравнения а присвоение значения.

Следует заострить внимание на коротком пути (short circuit) вычисления логических выражений рассмотренных ранее.

В качестве полезного примера можно привести следующий

```
if( null == stringVal  || "" == stringVal )
 System.out.println("Значение stringVal не определено")
```

Данная конструкция выполнится успешно.

```
if( null == stringVal  | "" == stringVal )
 System.out.println("Значение stringVal не определено")
```

В этом случае мы получим ошибку времени выполнения, если stringVal,будет иметь пустое (null) значение.

Следует обратить внимание так же на то, что константы используетсяв левой части оператора сравнения. Если мы будем сравнивать переменную с булевской константой и допустим опечатку

```
if( x = true)
 …
```

то такая конструкция будет всегда истинна, что возможно отличается от того, что этой конструкцией хотел выразить программист. (Такого рода ошибки, могут обнаруживаться с помощью программ автоматического тестирования)

В этом же случае

```
if( true = x)
```

компилятор выдаст ошибку еще во время компиляции.

# 7. Оператор switch

Оператор switch() в случае необходимости множественного выбора. Выбор осуществляется на основе целочисленного значения.

Структура оператора:

```
switch(int value){
 case const1:
  выражение или блок
 case const2:
  выражение или блок
case constn:
   выражение или блок
 default:
  выражение или блок
}
```

Причем фраза default не является обязательной

В качестве параметра switch может быть использована переменная типа byte, short, int, char или выражение. Выражение должно в конечном итоге возвращать параметр одного из указанных ранее типов. В операторе case не могут применяться значения примитивного типа long и ссылочных типов Long, String, Integer, Byte и т.д.

При выполнении оператора switch производится последовательное сравнение значения x с константами указанными после case и в случае совпадения производится выполнение выражения следующего за этим условием. Если выражение выполнено нормально, и нет преждевременного его завершения, то производится выполнение сравнения для последующих case. Если же выражение, следующее за case, завершилось не нормально, то будет прекращено выполнение всего оператора switch.

Если не выполнен ни один оператор case, то выполнится оператор default, если он имеется в данном switch. Если оператора default нет, и ни одно из условий case не выполнено, то ни одно из выражений switch выполнено не будет.

Если какое либо условие case выполнено, то все выполнение switch не прекратится, а будут проверяться следующие за ним условия. Что бы избежать этого, после части кода, которая выполнена, и дальнейшее выполнение не является необходимым, применяется break.

После оператора case должен следовать литерал, который может быть интерпретирован как 32 битовое целое значение. Здесь не могут применяться выражения и переменные, если они не являются final static.

Наиболее часто встречается ошибка, когда программист забывает указать break после выполнения блока кода и производится дальнейшее выполнение условий сравнения в операторе switch(). Следует обратить на это внимание. В качестве хорошего стиля программирования можно порекомендовать использование оператора default.

Рассмотрим пример

```
int x = 2;
switch(x){
 case 1:
 case 2:
  System.out.println("Равно 1 или 2");
  break;
 case 2:
 case 3
  System.out.println("Равно 2 или 3");
  break;
 default:
  System.out.println("Значение не определено");
}
```

В данном случае на консоль будет выведен результат Равно 1 или 2.Если же убрать операторы break, то будут выведены все три строки.

Вот такая конструкция вызовет ошибку времени компиляции.

```
int x = 5;
switch(x){
 case y:
  …
  break;
}
```

Если в операторе switch() применяется выражение, следует обратить внимание на результирующий тип выражения. Например

```
float x = 1.0f;
int y = 4;
switch(x*y){
 case 10:
  …
  break;
 case 20:
  …
  break;
 default:
  …
}
```

вызовет ошибку времени компиляции.

Следует обратить внимание так же на следующий ньюанс. Если в операторе switch() указано значение типа byte или short, то соответственно в операторах case должны применяться константы, которые могут быть сохранены в переменной данного типа. Например:

```
byte x = 5;
switch(x){
 case 1:
  …
  break;
 case 132:
  …
  break;
 default:
  …
}
```

вызовет ошибку компиляции. Так как case 132 превышает максимальное значение, которое может быть сохранено в переменной типа byte; Оператор default не обязательно должен замыкать конструкцию switch case. Он может так же комбинироваться с любым оператором case

Например

```
public class Test {
    static int x = 5;
    static {

    }
    public Test() {
    }
    public static void main(String[] args) {
```

```
        Test t = new Test();
        int x = 5;
        switch(x){
            case 1:
                System.out.println("One");
                break;
            case 2:
                System.out.println("Two");
                break;
            default:
            case 3:
                System.out.println("Tree or other");
        }
    }
}
```

откомпилируется без ошибок и на консоль будет выведено

```
Tree or other
```

В операторе switch не может быть двух case с одинаковыми значениями.

Т.е. конструкция

```
switch(x){
     case 1:
         System.out.println("One");
         break;
     case 1:
         System.out.println("Two");
         break;
     case 3:
         System.out.println("Tree or other value");
}
```

не допустима.

Так же в конструкции switch может быть применен только один оператор default.

Можно порекомендовать использование оператора switch вместо многозвенного if else, т.к. switch выполняется быстрее.

# 8. Управление циклами

В языке Java имеется три основных конструкции управления циклами.

- цикл while
- цикл do
- цикл for

## 8.1. Цикл while

Основная форма цикла while может быть представлена так

```
while(логическое выражение)
 повторяющееся выражение или блок;
```

В данной языковой конструкции повторяющееся выражение или блок, будет исполняться до тех пор, пока логическое выражение будет иметь истинное значение.

Если выражение или блок представляющий тело цикла будет завершен не нормальным образом по причине

- встретился оператор continue, то часть тела цикла следующая за оператором continue будет пропущена и выполнение цикла продолжится с начала. Если continue используется с меткой и метка принадлежит к данному while, то выполнение его будет аналогичным. Если метка не относится к данному while, то его выполнение будет прекращено.
- встретился оператор break, то выполнение цикла будет прекращено
- если выполнение блока будет прекращено по другим причинам (возникла исключительная ситуация), то выполнение while будет прекращено по тем же причинам.

Рассмотрим несколько примеров

```
public class Test {
    static int x = 5;
    static {

    }
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int x = 0;
        while(x < 5){
            x++;
            if(x % 2 == 0) continue;
            System.out.print(" " + x);
        }
    }
}
```

На консоль будет выведено

```
1 3 5
```

т.е. вывод на печать всех четных чисел будет пропущен.

```
public class Test {
```

```
    static int x = 5;
    static {

    }
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        int x = 0;
        int y = 0;
        lbl: while(y  < 3){
            y++;
            while(x < 5){
                x++;
                if(x % 2 == 0) continue lbl;
                System.out.println("x=" + x + " y="+y);
            }
        }
    }
}
```

На консоль будет выведено

```
x=1 y=1
x=3 y=2
x=5 y=3
```

т.е. при выполнении условия if(x % 2 == 0) continue lbl; цикл по переменной x будет прерван, а цикл по переменной y продолжится с начала.

Советом по применению конструкции может служить использование фигурных скобок, даже если выражение следующее после while(), будет единственным. Т.о. если конструкция в дальнейшем будет расширяться, то можно избежать потенциальных ошибок, т.к. если фигурные скобки опустить, выражение следующее за первым будет трактоваться как независимое, а не связанное с условием while()

Типичный вариант использования выражения while()

```
int i = 0;
while( i++ < 5){
 System.out.println("Counter is " + i);
}
```

Следует помнить, что цикл while(),будет выполнен только в том случае, когда на момент начала его выполнения, логическое выражение будет истинным. Т.о. при выполнении программы, может иметь место ситуация, когда цикл while() не будет выполнен ни разу.

```
boolean b = false;
while(b){
```

```
 System.out.println("Executed");
}
```

в данном случае строка System.out.println("Executed"); выполнена не будет.

## 8.2. Цикл do

Основная форма цикла do имеет следующий вид

```
do
 повторяющееся выражение или блок;
while(логическое выражение)
```

В отличии от цикла while цикл do, будет выполняться до тех пор, пока логическое выражение будет ложным. Вторым важным отличием является то, что do будет выполнен как минимум один раз.

Следует еще раз обратить внимание на использование фигурных скобок. Так же следует подчеркунть, что условие выхода из уикла должно изменяться в самом цикле, в противном случае, однажды начавшись, цикл не будет закончен никогда. То же самое следует отметить для цикла while(). В качестве примера следует привести следующую конструкцию.

```
boolean  b = false;
int counter = 0;
do{
 counter++;
 System.out.println("Counter is " + counter);
}while(b);

Типичная конструкция цикла do

int counter = 0;
do{
 counter ++;
 System.out.println("Counter is " + counter);
}while(counter > 5);
```

В остальном выполнение цикла do совершенно аналогично выполнению цикла while.

## 8.3. Цикл for

Довольно часто, необходимо изменять значение какой-либо переменной в заданном диапазоне, и выполнять повторяющуюся последовательность операторов с использованием этой переменной. Для выполнения этой последовательности действий как нельзя лучше подходит конструкция цикла for.

Основная форма цикла for выглядит следующим образом:

```
for(выражение инициализации; условие; выражение обновления)
 повторяющееся выражение или блок;
```

Ключевыми элементами данной языковой конструкции являются предложения заключенные в круглых скобках и разделенные точкой с запятой.

- выражение инициализации - выполняется до начала выполнения тела цикла. Чаще всего используется как некое стартовое условие (инициализация, или объявление переменной)
- условие - должно быть логическим выражением и трактуется точно так же как логическое выражение в цикле loop(). Тело цикла будет выполняться до тех пор, пока логическое выражение будет истинным. Как и в случае с циклом while() тело цикла может не исполниться ни разу. Это условие срабатывает, если логическое выражение принимает значение ложь до начала выполнения цикла.
- выражение - выполняется сразу после исполнения тела цикла, и до того как проверено условие продолжения выполнения цикла. Обычно здесь используется выражение инкрементации, но может быть применено и любое другое выражение.

Пример использования цикла for()

```
...
for(counter=0;counter<10;counter++){
 System.out.println("Counter is " + counter);
}
```

В данном примере предполагается, что переменная counter была объявлена ранее. Цикл будет выполнен 10 раз, и будут напечатаны значения счетчика от 0 до 9.

Допускается определять переменную прямо в предложении:

```
for(int cnt = 0;cnt < 10; cnt++){
 System.out.println("Counter is " + cnt);
}
```

результат выполнения этой конструкции будет аналогичен предыдущему. Однако следует обратить внимание, что область видимости переменной cnt будет ограничена телом цикла. Следует помнить о диапазоне значений переменной счетчика.

В качестве примера можно привести следующую конструкцию:

```
for(int I = 0;int < 10;i+=){
 System.out.println("Value i = " + i);
}
System.out.println("After loop i value = " + i);
```

В данном случае воникнет ошибка времени компиляции.

```
for(byte x = 0; x < 256;x++){
```

```
    ...
}
```

А такая конструкция будет выполняться бесконечно … Максимальное значение x = 127. После превышения этого значения x ,будет присвоено значение -128 и т.о. значение 256 никогда достигнуто не будет.

Следует обратить внимание на то, что предложение будет выполнено в любом случае, будет выполнено тело цикла или нет. В качестве примера можно привести такую конструкцию

```
int counter = 10;
...
for(counter = 0;cnt > 0;counter++){
    ...
}
System.out.println("Counter is " + counter);
```

На выхоже будет получено Counter is 0

Напротив выражение выполняется только при выполнении тела цикла.

```
int counter = 0;
for(;cnt < 1;counter++){
    ...
}
System.out.println("Counter is " + counter);
```

На выходе будет получено Counter is 1

Любая часть конструкции for() может быть опущена. В вырожденном случае мы получим оператор for с пустыми значениями

```
for(;;){
    ...
}
```

В данном случае, цикл будет выполняться бесконечно. Эта конструкция аналогична конструкции while(true){}. Условия, в которых она может быть применена, будут рассмотрены позже.

Возможно так же расширенное использование синтаксиса оператора for(). Предложение и выражение могут состоять из нескольких частей разделенных запятыми.

```
for(i = 0, j = 0; i<5;i++, j+=2){
    ...
}
```

использование такой конструкции вполне правомерно.

Следует отметить, что при использовании в конструкции for() нескольких частей, невозможно определение нескольких переменных или смешение определения и инициализации

нескольких переменных. Так при попытке использования следующих выражений будет получена ошибка компиляции.

```
for(int i = 0, long j = 0,i<10; i++, j += 50) // неверно
```

Нельзя так же использовать выражения в предложении

```
…
int i = 0;
for(i++; int j = 0; i< 10; j++)    // неверно
   ...
```

однако такая конструкция будет верна

```
...
int j = 0;
for(int = 7, j = 0 ;i < 10; i++,j+=10)
   ...
```

# 9. Операторы break и continue

В некоторых случаях требуется изменить ход выполнения программы. В традиционных языках программирования для этих целей используется оператор goto, однако в Java его использование не предусмотрено. Для этих целей применяются операторы break и continue

## 9.1. Оператор continue

Оператор continue может применяться только в циклах while, do, for. Если в потоке вычислений встречается оператор continue, то выполнение текущей последовательности операторов (выражений) должно быть прекращено и управление будет передано на начало блока содержащего этот оператор.

```
...
int x = (int)(Math.random()*10);
int arr[10]  = {….}
for(int cnt=0;cnt<10;cnt++){
 if(arr[cnt] == x) continue;
...
}
```

В данном случае, если в массиве arr встретится значение равное x, то выполнится оператор continue, и все операторы до конца блока будут пропущены, а управление будет передано на начало цикла.

Следует обратить внимание, что инициализация переменной cnt не произойдет, а будет произведена следующая итерация. Если оператор continue будет применен не в контексте оператора цикла, то будет выдана ошибка времени компиляции.

Рассмотрим пример

```
1. public class Test {
2. public Test() {
3. }
4. public static void main(String[] args) {
5. Test t = new Test();
6.   for(int j=0; j < 10; j++){
7.     if(i* % 2 == 0) continue;
8.        System.out.print("i=" + i);
9.      }
10.   }
11. }
```

в результате работы на консоль будет выведено

```
1 3 5 7 9
```

При выполнении условия в строке 7 нормальная последовательность выполнения операторов будет прервана и управление будет передано на начало цикла. Т.о. на консоль будут выводиться только нечетные значения.

## 9.2. Оператор break

Этот оператор, так же как и оператор continue, изменяет последовательность выполнения, но не возвращает исполнение к началу цикла, а прерывает его.

```
1. public class Test {
2.     public Test() {
3.     }
4.     public static void main(String[] args) {
5.         Test t = new Test();
6.         int [] x = {1,2,4,0,8};
7.         int y  = 8;
8.         for(int cnt=0;cnt < x.length;cnt++){
9.             if(0 == x[cnt]) break;
10.             System.out.println("y/x = " + y/x[cnt]);
11.         }
12.     }
13. }
```

на консоль будет выведено

```
y/x = 8
y/x = 4
y/x = 2
```

при этом ошибки связанной с делением на ноль не произойдет, т.к. если значение элемента массива будет равно 0, то будет выполнено условие в строке 9 и выполнение цикла for будет прервано.

В качестве аргумента break может быть указана метка. Как и в случае с continue, нельзя указывать в качестве аргумента метки блоков, в которых данный оператор break не содержится.

# 10. Именованные блоки

В реальной практике, достаточно часто используются вложенные циклы. Соответственно может возникнуть ситуация когда из вложенного цикла нужно прервать внешний. Простое использование break или continue не разрешает этой задачи, однако в Java возможно именовать блок кода и явно указать этим операторам, к какому из них относится делаемое действие. Делается путем присвоения метки операторам do, while, for.

Метка - это любая допустимая в данном контексте лексема, оканчивающаяся двоеточием.

Рассмотрим следующий пример

```
…
int array[][] = {….};
for(int i=0;i<5;i++){
 for(j=0;j<4; j++){
  ...
  if(array[i][j] == caseValue) break;
  ...
 }
}
…
```

В данном случае, при выполнении условия будет прервано выполнение цикла по j, цикл по i продолжится со следующего значения. Для того, что бы прервать выполнение обоих циклов, используется ранее упомянутая языковая конструкция

```
…
int array[][] = {….};
outerLoop: for(int i=0;i<5;i++){
 for(j=0;j<4; j++){
  …
  if(array[i][j] == caseValue) break outerLoop;
  …
 }
}
…
```

Оператор continue так же может быть использован с именованными блоками.

Следует обратить внимание, что при использовании перехода continue на внешний блок, его действие схоже с действием break. Т.е. исполнение текущего блока будет прервано и управление передастся на внешний блок.

Пример.

```
…
int array[][] = {….};
outerLoop: for(int i=0;i<5;i++){
 for(j=0;j<4; j++){
  …
  if(array[i][j] == caseValue) continue outerLoop;
  …
 }
}
…
```

и

```
…
int array[][] = {….};
for(int i=0;i<5;i++){
 for(j=0;j<4; j++){
  …
  if(array[i][j] == caseValue) break;
  …
 }
}
…
```

Результат исполнения обоих вариантов кода будет идентичным

Между операторами break и continue есть еще одно существенное отличие. Оператор break может быть использован с любым именованным блоком, в этом случае его действие в чем-то похоже на действие goto. Оператор continue (как и отмечалось ранее) может быть использован только в теле цикла. Т.е. такая конструкция будет вполне приемлемой.

```
lbl:{
 …
  if( val > maxVal) break lbl;
…
}
```

в то время как оператор continue здесь применять нельзя. В данном случае при выполнении условия if, выполнение блока с меткой lbl будет прервано, т.е. управление будет передано на оператор (выражение), следующий непосредственно за закрывающей фигурной скобкой.

Следует отметить, что использование оператора goto JAVA запрещено, хотя это слово и является зарезервированным. Метки используют пространство имен, отличное от пространства имен классов, и методов.

Например, этот пример кода будет вполне работоспособным.

```
1.
2. public class Test {
```

```
3.      public Test() {
4.      }
5.      public static void main(String[] args) {
6.          Test t = new Test();
7.          t.test();
8.      }
9.      void test(){
10.          Test:{
11.              test: for(int i =0;true;i++){
12.                  if(i % 2 == 0) continue test;
13.                  if(i > 10) break Test;
14.                  System.out.print(i + " ");
15.              }
16.          }
17.      }
18. }
```

Однако следует акцентировать внимание, что хотя пространства имен и не совпадают, не рекомендуется без крайней на то необходимости использовать имена меток совпадающие с именами методов или классов.

Для составления меток применяются те же синтаксические правила, что и для переменных, за тем исключением, что метки всегда оканчиваются двоеточием. Метки всегда должны быть привязаны к какому-либо блоку кода. Допускается использование меток с одинаковыми именами, но нельзя использовать одинаковые имена в пределах видимости блока. Т.е. такая конструкция допустима

```
lbl: {
 …
 System.out.println("Block 1");
…
}
…
lbl: {
 …
 System.out.println("Block 2");
…
}
```

А такая нет

```
lbl:{
 …
 lbl:{
  …
 }
 …
}
```

# 11. Оператор return

Этот оператор предназначен для возврата управления из вызываемого метода в вызываемый. Если в последовательности операторов выполняется return то управление немедленно (если это не оговорено особо) передает управление в вызывающий метод.

Далее будут рассмотрены особенности выполнения retun в конструкции try catch finally

return может иметь, а может не иметь аргументов. Если аргументы отсутствуют, то этот оператор может быть использован для возврата управления только в методах с квалификатором void. Если при объявлении метода использован тип void, то return не может иметь аргументов, в противном случае, будет получена ошибка времени компиляции.

В качестве аргумента return может быть использовано выражение.

```
return (x*y +10) /11;
```

В этом случае сначала будет выполнено выражение, а затем результат его выполнения будет передан в вызывающий метод. В случае если выражение будет завершено не естественным образом, то и оператор return, будет завершен не естественным способом. Например, если во время выполнения выражения в операторе return возникнет исключение, то и return не будет выполнен так, как это ожидалось. Т.е. исключительная ситуация будет обработана в самом методе, или будет возбуждена в вызывающем методе.

В методе может быть более одного оператора return.

# 12. Оператор synchronized

Этот оператор применяется для исключения взаимного влияния нескольких потоков при выполнении кода, и будет подробно рассмотрен в главе 12, посвященной потокам исполнения.

# 13. Ошибки при работе программы. Исключения (Exceptions).

При выполнении программы зачастую могут возникать ошибки. В одних случаях это вызвано ошибками программиста, в других внешними причинами. Например, может возникнуть ошибка ввода/вывода при работе с файлом или сетевым соединением. В классических языках программирования, например в C, требовалось проверять некое условие которое указывало на наличие ошибки и, в зависимости от этого предпринимать определенные действия.

Например

```
…
int statusCode = someAction();
if (statusCode){
 … обработка ошибки
}else{
```

```
 statusCode = anotherAction();
 if(statusCode){
  … обработка ошибки …
}
}
…
```

В Java появилось более простое и элегантное решение - обработка исключительных ситуаций.

```
try{
 someAction();
 anotherAction()
}catch(Exception e){
 … обработка исключительной ситуации
}
```

Легко заметить, что такой подход является не только элегантным, но и более надежным и простым для понимания.

## 13.1. Причины возникновения ошибок

Существует три причины возникновения исключительных ситуаций.

- Попытка выполнить некорректное выражение.

  Например, деление на ноль, или обращение к объекту по ссылке, равной null, попытка использовать класс, описание которого (class-файл) отсутствует, и т.д.

  В таких случаях всегда можно точно указать, в каком месте произошла ошибка - именно в некорректном выражении.

- Выполнение оператора throw.

  Очевидно, что и здесь можно легко указать место возникновения исключительной ситуации.

- Асинхронные ошибки во время исполнения программы.

  Причиной таких ошибок могут быть сбои внутри самой виртуальной машины (ведь она также является программой), или вызов метода stop() у потока выполнения thread).

  В этом случае невозможно указать точное место программы, где происходит исключительная ситуация. Если мы пытаемся остановить поток выполнения (вызвав метод stop()), то мы не можем предсказать, при выполнении какого именно выражения этот поток остановится.

Таким образом, все ошибки в Java делятся на синхронные и асинхронные. Первые сравнительно проще, так как принципиально возможно найти точное место в коде, которое является причиной возникновения исключительно ситуации. Конечно, Java является строгим языком в том смысле, что все выражения до точки сбоя обязательно будут выполнены, в то же время ни одно последующее выражение никогда выполнено не будет. Важно помнить, что ошибки могут возникать как по причине недостаточной внимательности программиста (отсутствует нужный класс, или индекс массива вышел за допустимые границы), так и по

независящим от него причинам (произошел разрыв сетевого соединения, сбой аппаратного обеспечения, например, жесткого диска, и др.).

Асинхронные ошибки гораздо сложнее в обнаружении и исправлении. Обычному разработчику очень затруднительно выявить причины сбоев в виртуальной машине. Это могут быть ошибка создателей JVM, несовместимость с операционной системой, аппаратный сбой и многое другое. Все же современные виртуальные машины реализованы довольно хорошо, и подобные сбои происходят крайне редко (при условии использования качественных комплектующих).

Аналогичная ситуация наблюдается и в случае с принудительной остановкой потоков исполнения. Поскольку это действие выполняется операционной системой, никогда нельзя предсказать, в каком именно месте остановится поток. Это означает, что программа может многократно отработать корректно, а потом неожиданно дать сбой просто из-за того, что поток остановился в каком-то другом месте. По этой причине принудительная остановка крайне не рекомендуется. В соответствующей лекции приводятся примеры корректного управления жизненным циклом потока.

При возникновении исключительной ситуации управление передается от кода, вызвавшего исключительную ситуацию, на ближайший блок catch (или вверх по стеку), и создается объект, унаследованный от класса Throwable или его потомков (см. диаграмму иерархии классов исключений), который содержит информацию об исключительной ситуации и используется при ее обработке. Собственно в блоке catch указывается именно класс обрабатываемой ситуации. Подробно обработка ошибок рассматривается ниже.

Иерархия по которой передается информация об исключительной ситуации зависит от того, где эта исключительная ситуация возникла. Если это

- метод, то управление будет передаваться в то место, где этот метод был вызван;
- конструктор, то управление будет передаваться туда, где попытались создать объект (как правило, применяя оператор new);
- если это статический инициализатор, то управление будет передано туда, где произошло первое обращение классу, потребовавшее его инициализацию.

Допускается создание собственных классов исключительных ситуаций. Осуществляется это с помощью механизма наследования, т.е. класс пользовательской исключительной ситуации, должен быть унаследован от класс Throwable или его потомков.

## 13.2. Обработка исключительных ситуаций

### 13.2.1. Конструкция try-catch

В общем случае конструкция выглядит так.

```
try{
 …
}catch(SomeExceptionClass e){
 …
}catch(AnotherExceptionClass e){
 …
}
```

Работает она следующим образом. Сначала выполняется код заключенный в фигурные скобки оператора try. Если во время его выполнения не происходит никаких нештатных ситуаций, то далее управление передается, за закрывающую фигурную скобку, последнего оператор catch ассоциированного с данным оператором try.

В случае если в пределах try возникает исключительная ситуация, то далее выполнение кода производится по одному из ниже перечисленных сценариев.

- возникла исключительная ситуация, класс которой указан в качестве параметра одного из блоков catch. В этом случае производится выполнение блока кода ассоциированного с этим catch (заключенного в фигурные скобки). Далее если код в этом блоке завершается нормально, то и весть оператор try завершается нормально и управление передается на оператор (выражение) следующий за закрывающей фигурной скобкой последнего catch ассоциированного с данным try. Если код в catch завершается нештатно, то и весь try завершается нештатно по той же причине.
- если возникла исключительная ситуация, которая класс которой не указан в качестве аргумента, ни в одном catch, то выполнение всего try завершается нештатно.

Если в последовательности операторов могут возникнуть как ошибки ввода/вывода так и ошибки арифметических вычислений, вовсе нет нужды помещать различные фрагменты кода в разные операторы try{}catch(){}. Достаточно обеспечить несколько catch() для различных типов исключений.

### 13.2.2. Конструкция try-catch-finally

Оператор finally предназначен для того, что бы обеспечить гарантированное выполнение какого-либо фрагмента кода.

Последовательность выполнения такой конструкции будет следующей: Если оператор try выполнен нормально, то будет выполнен блок finally. В свою очередь, если оператор finally выполняется нормально, то весь оператор try выполняется нормально. Если происходит преждевременное окончание выполнения блока finally, то весть оператор try завершается предварительно по тем же причинам.

- существует оператор catch, который перехватывает данный тип исключения, происходит выполнение связанного с catch блока.
  - Если блок catch выполняется нормально, то выполняется блок finally
    - в свою очередь если блок finally завершается нормально, то весь try завершается нормально.
    - Если finally завершается предварительно, то и весь оператор try завершается предварительно по той же причине.
  - Если блок catch завершается ненормально, то выполняется блок finally
    - в свою очередь, если блок finally завершается нормально, то оператор try завершается не нормально, по той же причине, по которой не нормально завершился блок catch
    - если блок finally завершается ненормально, то весь блок try завершается ненормально, по той же причине, что и блок finally

- в списке операторов catch не находится такого, который обработал бы возникшее исключение. Все равно выполняется блок finally. В этом случае, если
  - finally завершится нормально, весь try завершится не нормально по той же причине по которой было нарушено исполнение try.
  - finally завершится ненормально, то try завершится ненормально по той же причине, по которой ненормально завершился finally

Если оператор try завершился нормально, то выполнится блок finally. И если

- блок finally завершится нормально, то весь try завершится нормально
- блок finally завершится не нормально, то весь try завершится не нормально, по той же причине.

Следует обратить внимание, что при использовании конструкции finally, блок кода ассоциированный с ним будет выполнятся всегда. Если во время обработки исключительной ситуации, возникнет новая исключительная ситуация, то исключительная ситуация, которая послужила первопричиной будет потеряна.

Рассмотрим пример применения конструкции try-catch-finally.

```
try{
 byte [] buffer = new byte[128];
 FileInputStream fis = new FileInputStream("file.txt");
 while(fis.read(buffer) > 0){
  … обработка данных
}
}catch(IOException es){
  … обработка исключения …
}finally{
 fis.flush();
 fis.close();
}
```

Следует обратить внимание, что использование flush() не является обязательным в данном контексте, т.к. буфер ввода/вывода будет очищен при вызове close() указания, и здесь использован для большей наглядности.

Если в данном примере поместить операторы очистки буфера и закрытия файла сразу после окончания обработки данных, то при возникновении ошибки ввода вывода, корректного закрытия файла не произойдет. Следует отметить, что блок finally, будет выполнен в любом случае, вне зависимости от того произошла обработка исключения или нет, возникло это исключение или нет.

В конструкции try-catch-finally обязательным является использование одной из частей оператора catch или finally. То есть, конструкция

```
try{
 ...
}finally{
```

```
 ...
}
```

является вполне допустимой. В этом случае блок finally при возникновении исключительной ситуации будет выполнен, хотя сама исключительная ситуация обработана не будет и будет передана для обработки на более высокий уровень иерархии.

Следует рассмотреть два специальных случая использования finally.

```
try{
 …
}catch(Exception e){
 …
 System.exit(0);
}finally{
 …
}
```

в этом случае блок finally выполнен НЕ БУДЕТ т.к. выполнение данного потока будет прекращено.

Пример применения оператора return:

```
try{
 …
 return 0;
}
catch(MyException ex){
 …
 System.out.println("Exception");
 return -1;
}
finally{
 …
 System.out.println("Finally");
}
```

В этом случае последовательность действий будет следующей.

Если исключения не произойдет будет исполнен лишь блок finally метод вернет значение 0

На консоль будет выведено Finally

Если произойдет исключение MyException, то будет выполнен блок finally и метод вернет значение -1.

На консоль будет выведено

```
Exception
Finally
```

То, есть сначала отработает блок finally и только после этого будет произведен возврат управления в вызвавший код.

Если,будет возбуждено исключение Exception, которое не обрабатывается в данной конструкции, то будет выполнен блок finally и исключительная ситуация будет передана на обработку вызываемому коду.

В случае, если обработка исключительной ситуации в коде не предусмотрена, то при ее возникновении выполнение метода будет прекращено, и исключительная ситуация будет передана для обработки коду более высокого уровня. Таким образом, если исключительная ситуация произойдет в вызываемом методе, то управление будет передано вызываемому методу и обработку исключительной ситуации должен произвести он. Если исключительная ситуация возникла в коде самого высокого уровня (методе main()), то управление будет передано исполняющей системе Java и выполнение программы будет прекращено.

## 13.3. Использование оператора throw

Помимо того, что предопределенная исключительная ситуация могла быть возбуждена исполняющей системой Java, программист сам может сгенерировать это условие. Делается это с помощью оператора throw.

Например

```
….
public int  calculate(int theValue){
if( theValue < 0){
throw new Exception("Параметр для вычисления не должен быть отрицательным");
}
}
…
```

В данном случае, предполагается, в качестве параметра методу может быть передано только положительное значение, если это условие не выполнено, то с помощью оператора throw возбуждается исключительная ситуация. В действительности данный код не будет откомпилирован, т.к. компилятор выдаст сообщение об ошибке. Если в методе возбуждается исключительная ситуация, то должно быть выполнено одно из двух правил

- исключительная ситуация должна быть обработана в теле метода (т.е. код должен возбуждающий исключительную ситуацию, должен быть помещен в блок try {} catch(UserException ue){})
- метод должен делегировать обработку исключительной ситуации вызвавшему его коду. Для этого в сигнатуре метода применятся ключевое слово throws, после которого должны быть перечислены через запятую все исключительные ситуации, которые может вызывать данный метод. Т.е. приведенный выше пример должен быть приведен к следующему виду

```
….
public int  calculate(int theValue) throws Exception{
if( theValue < 0){
throw new Exception("Some descriptive info");
```

```
}
}
…
```

Т.о. возбуждение исключительной ситуации в программе производится с помощью оператора throw, слева от которого указывается объект, который может быть приведен к типу Throwable. (Как правило этот объект создается в этом же месте с помощью оператора new, хотя это условие и не является обязательным)

В некоторых случаях после обработки исключительной ситуации, возможно, возникнет необходимость передать информацию о ней в вызывающий код.

В этом случае throw используется вторично.

Например

```
…
try{
 …
}catch(IOException ex){
 …
 // Обработка исключительной ситуации
 …
 // Повторное возбуждение исключительной ситуации
 throw ex;
}
```

Рассмотрим еще один случай.

Предположим, что оператор throw применяется внутри конструкции try catch.

```
try{
 …
 throw new IOException();
 …
}catch(Exception e){
 …
}
```

В этом случае, исключение возбужденное в блоке try не будет передано для обработки на более высокий уровень иерархии, а обработается в пределах блока try catch., т.к. тут содержится оператор, который может это исключение перехватить.Т.е. как бы произойдет неявная передача управления, на соответствующий блок cacth

Следует обратить внимание слушателей, что такой способ использования конструкции try catch не рекомендуется, т.к. затраты на обработку будут несравнимо выше, чем при использовании операторов if. Следует обратить так же внимание на то, что блок catch может быть пустым, т.е. не производить никакой обработки, тем не менее исключение будет считаться перехваченным и далее по иерархии передано не будет.

```
try{
```

```
 …
 throw new IOException();
 …
}catch(Exception e){
 ;
}
```

в данном случае оператор try завершится нормально.

## 13.4. Обрабатываемые и необрабатываемые исключения

Все исключительные ситуации можно разделить на две категории обрабатываемые (checked) и не обрабатываемые (unchecked).

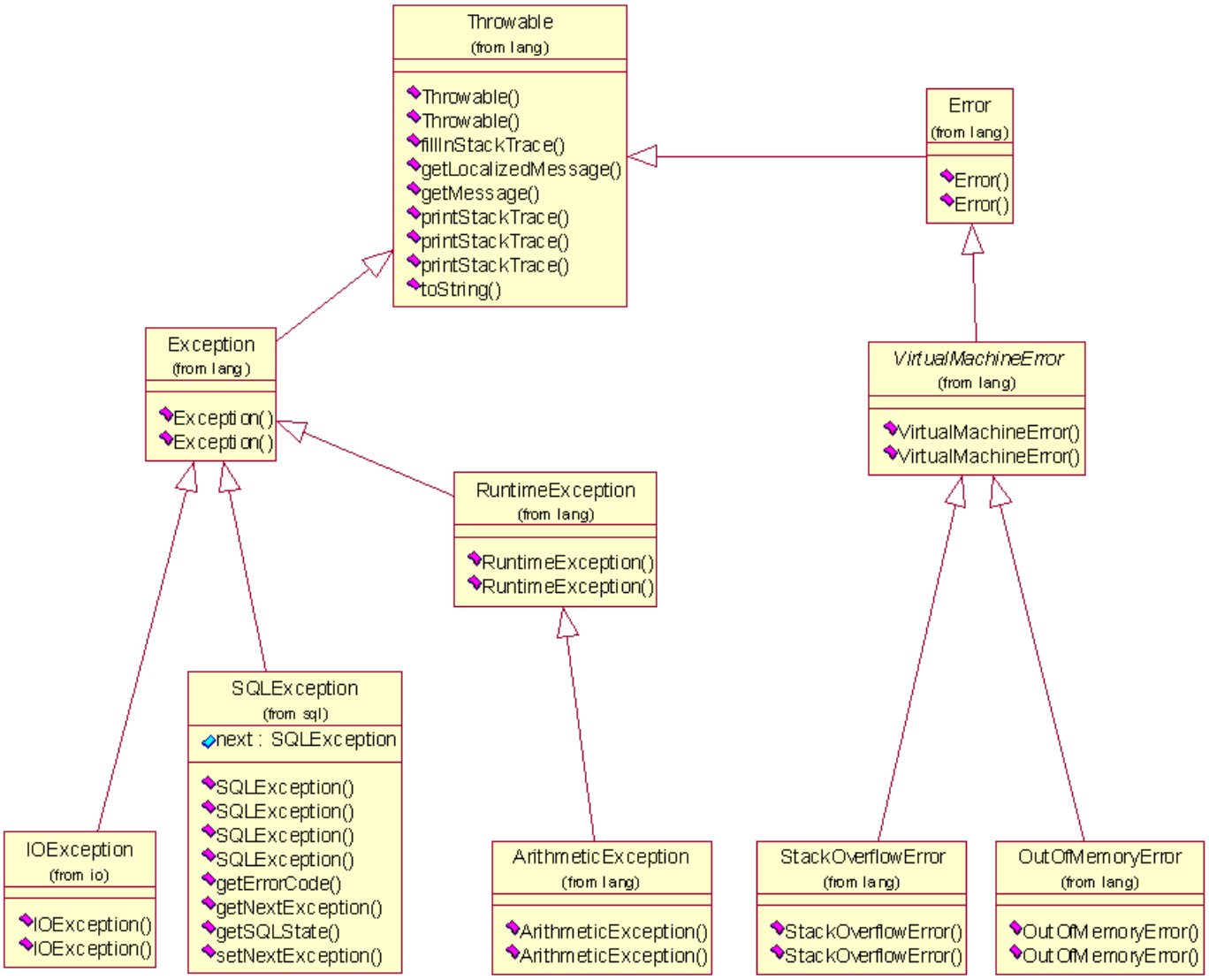
Все исключения, порожденные от java.lang.Exception являются обрабатываемыми. Т.е. во время компиляции проверяется - предусмотрена ли обработка возможных исключительных ситуаций.

Исключения, порожденные от java.lang.RunTimeException, являются необрабатываемыми, и компилятор не требует обязательной их обработки.

Как правило, обрабатываемые исключения предназначены для обработки ситуаций связанных с окружением программы (сетевым, файловым вводом-выводом и др.), которые могут возникнуть вне зависимости от того, корректно написан код или нет. Например, открытие сетевого соединения или файла может привести к возникновению ошибки, и компилятор требует от программиста предусмотреть некие действия для обработки возможных проблем. (Возможно никаких действий предпринято не будет, т.е. блок catch(){} можно оставить пустым, однако компилятор это трактует как обработку исключения и не выдаст сообщения об ошибке компиляции) Все пользовательские исключения, порожденные от java.lang.Exception (или его потомков) являются обрабатываемыми.

Необрабатываемые исключения, это ошибки программы, которые при правильном кодировании возникать не должны (например, java.lang.IndexOfBoundException, java.lang.ArifmeticException возникают соответственно при указании индекса выходящего за границы массива и при делении на ноль). Поэтому, чтобы не загромождать программу, компилятор разрешает не обрабатывать с помощью блоков try{} catch() исключения этого типа.

Исключения, порожденные от Error, так же не являются обрабатываемыми. Эти ошибки предназначены для того что бы уведомить программу о возникновении сбоев которые программным способом устранить сложно, или невозможно вообще. В качестве примера можно привести StackOverflowError, OutOfMemoryError.




Следует обратить внимание, что хотя необрабатываемые ошибки не обязательно помещать в блоки try{} catch() …, они обрабатываются точно так же как и обычные ошибки. Т.е. если вы не можете гарантировать, что Вов время выполнения не будет нарушена граница массива (например при динамическом определении размера массива), то код в котором производится обращение к элементам массива следует поместить в блок try{} catch(java.lang.IndexOfBoundException ex) { …}.

Далее, в случае если в подпрограмме могут возникать необрабатываемые ошибки, то указание ключевого слова throws не является обязательным.

Ошибки (Error), говорят о наличии фатальных ситуаций и нет необходимости обрабатывать их. Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые ошибки виртуальной машины Java. Например, исчерпание свободной памяти, переполнение стека и т.д.

В разработке и проектировании программного обеспечения используется понятие контракта. Более полно это понятие рассматривалось ранее. Одним из аспектов является четкое разграничение ответственности между вызывающим и вызываемым методами. Например при вызове метода производится вычисление квадратного корня (предполагается, что речь идет о натуральных числах), тогда на вход метода не может быть передано отрицательное число. Если это происходит, то метод должен вызвать исключительную ситуацию, так как контракт нарушен. Этот подход наиболее понятен и упрощает программирование, так как метод должен обрабатывать только правильно заданные параметры и не должен знать ничего об обработке неправильных. Кроме того он должен уведомить вызывающий метод о возникновении этой ситуации, сразу по ее возникновению. Для реализации этого подхода рекомендуется использовать обрабатываемее (checked) исключения.

Как было сказано ранее, возможно задание нескольких операторов catch для одного блока try. Если программист желает перехватывать некий набор исключительных ситуаций, он может указать в блоке catch родительский класс вместо перечисления нескольких подклассов. Например можно перехватывать все исключения порожденные от java.lang.Exception следующим образом

```
  try{
  …
 }
 catch(Exception e){
  …
 }
```

Так как родителем всех исключений является java.lang.Throwable, то если необходимо в каком - либо месте кода перехватывать все исключительные ситуации, то вполне применима вот такая конструкция

```
  try{
  …
 }
 catch(Throwable e){
  …
 }
```

однако следует избегать использования такой конструкции без явных на то причин. Т.к. в блоке catch будут обрабатываться самые разные ошибки, что нарушит структуру кода и усложнит его понимание.

В случае если в конструкции обработки исключений используется несколько операторов catch, то классы исключений нужно перечислять в них последовательно, от менее общих к более общим. Рассмотрим два примера

```
try{
 …
}
catch(Exception e){
 …
```

```
}
catch(IOException ioe){
 …
}
catch(UserExcetion ue){
 …
}
```

В данном примере при возникновении исключительной ситуации (класс которой порожден от Exception) будет выполняться всегда только первый блок catch. Остальные не будут выполнены ни при каких условиях. Эта ситуация отслеживается компилятором, который сообщает об UnreachableCodeException (ошибка - недостижимый код). Правильно данная конструкция будет выглядеть так

```
try{
 …
}
catch(UserExcetion ue){
 …
}
catch(IOException ioe){
 …
}
catch(Exception e){
 …
}
```

В этом случае будет произведена последовательная обработка исключений. И в случае если не предусмотрена обработка того типа исключения, которое возникло (например, AnotherUserException), будет выполнен блок catch(Exception e){…}

Если срабатывает один из блоков catch, то остальные блоки в данной конструкции try-catch выполняться не будут.

## 13.5. Создание пользовательских классов исключений

Как уже отмечалось ранее, допускается создание собственных классов исключений. Для этого достаточно создать свой класс, унаследовав его от любого класса являющегося дочерним по отношению к ajav.lang.Throwable. (Или от самого Throwable)

Пример.

```
  public class UserException extends Exception{
  public UserException(){
   super();
  }
  public UserException(String descry){
super(descr);
}
 }
```

соответственно возбуждаться данное исключение будет следующим образом:

```
  throw new UserException("Дополнительное описание");
```

Рассмотрим другой пример. В языке С существует конструкция assert, которая зачастую используется для целей отладки. Рассмотрим как в JAVA можно воспроизвести подобную конструкцию с помощью механизма исключений.

```
public class AssertionException extends RuntimeException{

public AssertionException(){
super("Asertion Exception");
}

  public AssertionException(String descr){
super(descr);
}
}

public class Assertion {
 public static Boolean ASSERTION_ON = true;

 private Assertion(){};

public static void assert(boolean flag)
throws AssertionException{
if(ASSERTION_ON && flag){
 throw new AssertionException()
}
}

public static void assert(boolean flag,Sring msg)
throws AssertionException{
if(ASSERTION_ON && flag){
 throw new AssertionException(msg)
}
}

}
```

Основная идея использования данного класса заключается в том, что бы в критичных участках программы встроить проверку некоторых граничных условий и, в случае их невыполнения возбуждать исключительную ситуацию.

Например

```
Assertion.assert(x <= xMinValue,"X too large");
```

В данном случае, если величина переменной x будет меньше некоего минимума будет возбуждена исключительная ситуация AssertionException. Т.к. это необрабатываемое

исключение, то использование блока try{} catch() не является обязательным. Однако, если мы все таки обработаем эту исключительную ситуацию, то можем выдать (например на консоль, или в лог-файл) сообщение об ошибке.

```
try{
 …
 // вызов кода использующего Assertion
 …
}cacth(AssertionException ae){
 System.err.println(ae);

}
```

## 13.6. Переопределение методов и исключения

При переопределении методов следует помнить что, если переопределяемый метод возбуждает исключение, то переопределяющий метод не может расширять класс этих исключений. Рассмотрим пример

```
public class BaseClass{
public void method () throws IOException{
 ...
 }
}

public class LegalOne extends BaseClass{
 public void method () throws IOException{
  ...
 }
}

public class LegalTwo extends BaseClass{
 public void method () {
  ...
 }
}

public class LegalTree extends BaseClass{
 public void method ()
  throws EOFException,MalformedURLException  {
  ...
 }
}

public class IllegalOne extends BaseClass{
 public void method ()
  throws IOException,IllegalAccessException  {
  ...
```

```
 }
}

public class IllegalTwo extends BaseClass{
 public void method () {
  ...
  throw new Exception();
 }
}
```

в данном случае

определение класса LegalOne будет корректным, т.к. переопределение метода method() будет верным.

определение класса LegalTwo будет корректным, т.к. переопределение метода method() будет верным. (Переопределяемый метод не возбуждает исключений и поэтому не создает конфликта с переопределяемым методом)

определение класса LegalTree будет корректным, т.к. переопределение метода method() будет верным. (Метод может возбуждать исключения, которые являются подклассами исключения возбуждаемого в переопределяемом методе)

определение класса IlegalOne будет некорректным, т.к. переопределение метода method() неверно. (IllegalAccessException не является подклассом IOException)

определение класса IlegalTwo будет некорректным, method() переопределен верно, но он возбуждает исключение не указанное в throws.

## 13.7. Особые случаи

Во время исполнения кода могут возникать ситуации, которые редко или вообще не описаны в литературе.

Рассмотрим такую ситуацию.

```
import java.io.*;
public class Test  {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        try {
            test.doFileInput("bogus.file");
        }
        catch (IOException ex) {
            System.out.println("Second exception hadle starck trace");
            ex.printStackTrace();
        }
    }
```

```
    private String doFileInput(String fileName)
throws FileNotFoundException,IOException{
        String retStr = "";
        java.io.FileInputStream fis = null;
        try {
             fis = new java.io.FileInputStream(fileName);
        }
        catch (FileNotFoundException ex) {
            System.out.println("First exception hadle starck trace");
            ex.printStackTrace();
            throw ex;
        }
        return retStr;
    }
}
```

Результат работы будет выглядеть следующим образом.

```
java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:33)
 at experiment.Test.main(Test.java:21)
First exception hadle starck trace
java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:33)
 at experiment.Test.main(Test.java:21)
Second exception hadle starck trace
```

Так как при вторичном возбуждении используется один и тот же один объект Exception, то стек в обоих случаях будет содержать одну и ту же последовательность вызовов. Т.е. при повторном возбуждении исключения, если мы используем тот же объект, изменения его параметров не происходит.

Рассмотрим другой пример.

```
import java.io.*;

public class Test  {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        try {
            test.doFileInput();
```

```
        }
        catch (IOException ex) {
            System.out.println("Exception hash code " + ex.hashCode());
            ex.printStackTrace();
        }
    }

    private String doFileInput() throws FileNotFoundException,IOException{
        String retStr = "";
        java.io.FileInputStream fis = null;
        try {
             fis = new java.io.FileInputStream("bogus.file");
        }
        catch (FileNotFoundException ex) {
            System.out.println("Exception hash code " + ex.hashCode());
            ex.printStackTrace();
            fis = new java.io.FileInputStream("anotherBogus.file");
    throw ex;
        }
        return retStr;
    }
}

java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:33)
 at experiment.Test.main(Test.java:21)
Exception hash code 3214658

java.io.FileNotFoundException:  (The system cannot find the path specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:38)
 at experiment.Test.main(Test.java:21)
Exception hash code 6129586
```

Несложно заметить, что, несмотря на то, что последовательность вызовов одна и та же, в вызываемом методе и вызывающем обрабатываются разные объекты исключений.

Здесь следует обратить внимание, что если при обработке исключения произойдет в свою очередь новое исключение и оно не будет обработано в данном методе, то собственно информация об исключении, которое послужило первоисточником нештатной ситуации будет утеряно и информация о нем в вызывающий метод передана не будет. В случае, если в коде обрабатывающем исключение тоже может возникнуть внештатная ситуация, следует использовать вложенные блоки try{} catch(). Если преобразовать код к следующему виду, то программа будет вести себя ожидаемым образом.

```
import java.io.*;

public class Test  {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        try {
            test.doFileInput();
        }
        catch (IOException ex) {
            System.out.println("Exception hash code " + ex.hashCode());
            ex.printStackTrace();
        }
    }

    private String doFileInput() throws FileNotFoundException,IOException{
        String retStr = "";
        java.io.FileInputStream fis = null;
        try {
             fis = new java.io.FileInputStream("bogus.file");
        }
        catch (FileNotFoundException ex) {
            try {
                System.out.println("Exception hash code " + ex.hashCode());
                ex.printStackTrace();
                fis = new java.io.FileInputStream("");
            }
            catch (FileNotFoundException ex2) {
            }
            throw ex;
        }
        return retStr;
    }
}
```

```
java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:24)
 at experiment.Test.main(Test.java:12)

Exception hash code 3214658

java.io.FileNotFoundException: bogus.file (The system cannot find the file
specified)
```

```
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:24)
 at experiment.Test.main(Test.java:12)

Exception hash code 3214658
```

# 14. Заключение

В данной главе рассмотрены основные языковые конструкции.

Циклы

- Три основных конструкции для выполнения циклов while(), do(), for().
- Выражение управляется выражением которое должно иметь булевский тип.
- В циклах for() while() условие выполнения цикла проверяется в начел цикла и до его выполнения. Таким образом тело цикла может быть не выполнено ни разу.
- В do циклах условие выполнения проверяется в конце цикла, таким образом цикл будет выполнен как минимум один раз.
- Для цикла for в скобках указывается три выражения. Первое выполняется до первой итерации цикла. Как правило здесь определяются и инициализируются переменные использующиеся в теле цикла. Второй параметр должен иметь булевское значение и определяет условие продолжения цикла. Третье выражение вычисляется сразу после выполнения тела цикла и до того как будет протестировано условие продолжения выполнения.
- Область видимости переменных объявленных в конструкции for ограничена телом цикла.
- Все три выражения цикла for не являются обязательными. Если условие продолжения цикла опущено, то оно трактуется как истинное.
- Предложение continue отменяет выполнение оставшейся части цикла для циклов while и do. В случае цикла for производится выполнение третьего выражения в конструкции и далее вычисляется условие продолжения цикла.
- Оператор break прекращает выполнение цикла. Для цикла for не производится тестирование условия продолжения цикла, выполнение третьего выражения не производится.
- Break и continue могут иметь в идее параметра метку, что позволяет прерывать выполнение вложенных циклов. Метка должна быть расположена перед объявлением цикла и заканчиваться двоеточием.

Операторы ветвления

- Оператор if() принимает на вход логическое (булевское) выражение
- else является необязательной частью оператора if()
- оператор switch принимает на входе целочисленное значение. Это может быть один из типов byte,short,char,int.
- Аргумент оператора case должен быть константой или выражением которое может быть вычислено во время компиляции
- В операторе case может быть только одна метка, в случае необходимости использовать несколько меток, следует применять несколько операторов case и не использовать brake для прекращения выполнения конкретного блока

- Если не выполнено ни одно из условий case, будет выполнен блок default (если он имеется)

Последовательность выполнения исключений

- В случае возникновения исключения управление передается за закрывающую скобку блока try, даже если исключение было вызвано из другого метода, находящегося в переделах блока try. В последнем случае выполнение вызываемого метода будет прекращено.
- Если встречается блок catch связанный с блоком try, и в качестве аргумента catch указан класс возникшего исключения или его родительский класс, будет выполнен первый встретившийся такой блок. Если это произошло, исключение считается обработанным, если этого не произошло, исключение считается не обработанным и передается для обработки вызвавшему коду.
- Блок finally выполняется вне зависимости от того возникло ли исключение, было - ли оно обработано или нет.
- Если исключения не возникло или исключение было обработано выполнение продолжается после блока finally
- Если исключение не обработано в текущем блоке, оно передается на обработку вышестоящему блоку try если оно не обрабатывается и там, то передается вверх по иерархии до тех пор, пока не достигнет самого верхнего уровня (главный поток приложения). В этом случае поток будет завершен и в System.err будет выведен дамп стека.

Возбуждение исключений

- Для возбуждения исключительной ситуации используется конструкция throw new XXXException()
- Любой подкласс java.lang.Exception будет обрабатываемым исключением, за исключением классов порожденных от java.lang.RuntimeException
- В методах, код которых может вызывать исключения, должен быть обрамлен блоками try{} catch(){} или в объявлении метода должно быть указано
- Метод не может возбуждать исключения если они не указаны после ключевого слова throws, за исключением RuntimeException и Error
- Если метод может возбуждать исключение (указано ключевое слово throws в объявлении метода), то вовсе не обязательно, что бы код возбуждающий это исключение был обязательно включен в этот метод
- Переопределенный (overridden) не может возбуждать исключение, если метод который он переопределяет не может возбуждать этого исключения или родительского класса исключения.

# 15. Контрольные вопросы

10-1. Приведенная ниже программа должна вывести на консоль Hello World! Выберите строки, которые нужно модифицировать в вашей программе, что бы получить правильный результат.

```
1. public class Test  {
2.     public Test() {
3.     }
4.     public static void main(String[] args) {
```

```
5.      Test test = new Test();
6.      String [] arr = {"H","e","l","l","o","
","w","o","r","l","d","!"};
7.      String result = "";
8.      int i= 0;
9.      for(;;;){
10.         result += arr[i++];
11.      }
12.      System.out.println(result);
13.    }
14. }
```

- a Заменить строку 9 на for(i = 0; i < arr.length;){
- b Заменить строку 9 на for(int int i = 0; i < arr.length;){
- c Заменить строку 9 на for(i = 0; i < arr.length;i++){
- d Заменить строку 9 на for(i = 1; i <= arr.length;i++){

a.) Правильный ответ a

Ответ b не верен так как переменная i уже определена в методе main Здесь следует еще раз напомнить, что область видимости переменной (если она объявлена в цикле for) будет ограничиваться лишь телом цикла. Однако в данном случае перемена с таким именем уже объявлена в теле метода и соответственно находится в той же области видимости. Если переменная была бы объявлена, как переменная класса, то этот код откомпилировался бы вполне успешно.

Ответ c не является верным так как, увеличение значения i в теле цикла будет произведено дважды, т.о. на печать будут выведены лишь четные элементы массива.

Ответ d не является верным по двум причинам. Первая – элементы массива нумеруются с 0, соответственно первым будет выбран второй элемент массива, вторая – когда выполнится условие окончания цикла, будет нарушена граница массива и будет вызвано исключение IndexOf-BoundException

10-2. Каков будет результат выполнения программы

```
1. public class Test  {
2.    public Test() {
3.    }
4.
5.    public static void main(String[] args) {
6.      Test test = new Test();
7.      int i = 5;
8.      while(i = 5){
9.         System.out.println(i++);
10.      }
11.    }
12. }
```

- a Компилятор выдаст сообщение об ошибке в строке 8
- b На консоль будут последовательно выведены значения 01234
- c На консоль будут последовательно выведены значения 43210
- d Программа откомпилируется,но на консоль ничего выведено не будет

a.) Правильный ответ a. В операторе while может быть использовано только булево значение. В данном случае используется оператор присваивания,а не сравнения, т.о. компилятор выдаст ошибку.

Так как здесь предложено выбрать только один ответ остальные ответы неверные

10-3. В данном случае выберите все правильные ответы.

```
private void say(int digit){
    switch(x){
        case 1: System.out.print("ONE");
              break;
        case 2: System.out.print("TWO");
        case 3: System.out.print("TREE");
        default:System.out.pritn("Unknown value")
    }
}
```

- a digit = 1 ONE
- b digit = 0 TWO TREE
- c digit = 2 TWO Unknown value
- d digit = 3 TREE Unknown value

a.) В данном случае правильными будут ответы a,d.

Рассмотрим последовательно вывод для всех возможных значений.

```
digit = 0 Unknown value.
```

В данном случае ни одно из значений case выполнено не будет и будет исполнен default.

```
digit = 1 ONE
```

В данном случае будет выполнено условие case 1: на консоль будет выведена соответствующая надпись. Далее следует оператор break и управление будет передано за фигурную скобку закрывающую блок switch

```
digit = 2 TWO TREE Unknown value
```

В данном случае будет выполнено условие case 2: на консоль будет выведено TWO. Т.к. далее не встречается оператор break, то выполнение switch будет продолжено, и на экран будут выведены TREE Unknown value

```
digit = 3  TREE Unknown value
```

В данном случае будет выполнено условие будет выполнено условие case 3: на консоль будет выведено TREE, как и в предыдущем случае оператор break здесь не используется, соответственно так же будет выполнен оператор default.

```
digit = 4 Unknown value
```

В данном случае ни одно из условий case выполнено не будет, поэтому отработает только оператор default.

10-4. Какая строка будет выдана на консоль после выполнения фрагмента кода приведенного ниже.

```
1. public class Test {
2.    public Test() {
3.    }
4.    public static void main(String[] args) {
5.       int i,j;
6.       lab: for(i = 0; i < 6; i++){
7.            for (j = 3; j > 1; j--){
8.                 if(i == j){
9.                   System.out.println(" " + j);
10.                   break lab;
11.                 }
12.            }
13.       }
14.    }
15. }
```

1. 2345
2. 234
3. 3
4. 2

a.) Правильный ответ d

Условие if в данном примере будет выполнено, когда переменные i и j будут равны 2. После чего на консоль будет выведено 2 и выполнится оператор break. Т.к. break содержит ссылку на метку, то будет прерван не текущий цикл (внутренний, по переменной j), а цикл по переменной i (внешний), т.о. образом выполнение программы будет прекращено.

10-5. Выберите все правильные варианты ответов в этом примере

```
1. public class Test {
2.   float fVal = 0.0f;
3.   public Test() {
4.   }
5.   public static void main(String[] args) {
6.      Test t = new Test();
7.      String testVal = "0.123";
8.      System.out.println("Was returned " + t.testParse(testVal) + "
with value " + t.fVal);
9.    }
10.   private boolean testParse(String val){
11.      try {
12.        fVal = Float.parseFloat(val);
13.        return true;
14.      }
15.      catch (NumberFormatException ex) {
16.        System.out.println("Test.testParse() Bad number -> " + val);
17.        fVal = Float.NaN;
18.      } finally{
19.        System.out.println("Finally part executed");
20.      }
21.      return false;
22.   }
23. }
```

1. testVal=“0.123”; Finally part executed Was returned true with value 0.123
2. testVal = “0,123”; Finally part executed Was returned false with value 0.123
3. testVal = null; Finally part executed Далее будет вызвано исключение NullPointer
4. testVal = “0.123”; Finally part executed Was returned false with value null

a.) Правильные ответы a,c

Вариант b не верен потому что, будет вызван оператор return в строке 13. Ответ d будет неверным т.к. при выполнении строки 12 будет возбуждено исключение, которое не обрабатывается ни в процедуре, ни в вызывающей программе, поэтому сообщение об ошибке будет выведено на консоль и выполнение программы прекратится.

10-6. Рассмотрим следующий пример.

Эти исключения имеют следующую иерархию наследования StringIndexOutOfBoundsException и ArrayIndexOutOfBoundsException

```
java.lang.Object
  |
  +--java.lang.Throwable
```

```
|
+--java.lang.Exception
      |
      +--java.lang.RuntimeException
            |
            +--java.lang.IndexOutOfBoundsException
                  |
                  +--java.lang.StringIndexOutOfBoundsException
                  |
                  +--java.lang.ArrayIndexOutOfBoundsException
```

Предположим, что в методе testSomeValue могут быть возбуждены оба вида этих исключений, при этом они не обрабатываются в блоке try – catch. Какое из ниже перечисленных суждений будет верным ?

- a Определение метода testSomeValue должно включать throws StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException
- b Если метод вызывающий testSomeValue перехватывает IndexOutOfBoundsException, то исключения StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException тоже будут перехыватываться.
- c Так как в определении метода указано throws StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException, то любой вызывающий его метод должен перехватывать эти типы исключений, вне зависимости возбуждается во время работы исключение или нет.
- d При объявлении метода testSomeValue не обязательно указывать возбуждаемые исключения

a.) Правильные ответы b,d

Ответ b будет правильным потому, что StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException являются потомками IndexOutOfBoundsException. И если перехватывается родительский класс исключений, то будут перехвачены и все его потомки.

Ответ d верен так, как StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException унаследованы от RuntimeException, то они являются необрабатываемыми исключениями и их указание в разделе throws определения метода не является обязательным

Ответ a неверен т.к. если метод возбуждает исключения унаследованные от RuntimeException, то указывать их в разделе throws нет необходимости, хотя это и не будет ошибкой.

Ответ c неверен по тем же причинам, что и вопрос 1. Даже если мы укажем в разделе throws StringIndexOutOfBoundsException, ArrayIndexOutOfBoundsException все равно нет необходимости указывать их в разделе catch.

10-7. Предположим необходимо создать собственную иерархию исключений.

Рассмотрим следующий пример.

```
Exception
  |
  +--LengthException
     |
     +--TooLongException
     |
     +--TooShortException

class BaseMeasurer{
    public BaseMeasurer(){
    }

    int measureLength(Dimension d) throws LengthException{
    }
}

class DerivedMeasurer extends BaseMeasurer{
    public BaseMeasurer(){
   }

  XXX {
  }
}
```

Какое из ниже перечисленных выражений можно использовать в строке 13 с тем, что бы код успешно откомпилировался

- a int measureLength(Dimension d) throws LengthException
- b int measureLength(Dimension d) throws Exception
- c int measureLength(Dimension d) throws TooLongException
- d int measureLength(Dimension d)

a.) Правильные ответы a,c,d

Ответ a верен, т.к. переопределенный метод может возбуждать тот же тип исключений, что и переопределяемый.

Ответ c верен, т.к. переопределенный метод возбуждает исключение которое является подклассом, исключения возбуждаемого в переопределяемом методе

Ответ d верен, т.к. переопределяемый метод не возбуждает исключений.

Ответ b неверен т.к. переопределенный метод расширяет список исключений возбуждаемых в переопределяемом методе.

10-8. Как и в предыдущем примере создадим собственную иерархию классов исключений.

```
Exception
  |
```

```
  +--LengthException
     |
     +--TooLongException
     |
     +--TooShortException
```

```
1. class TooShortException extends Exception{
2.     public TooShortException(String description){
3.         super(description);
4.     }
5. }
6.
7. class Measurer{
8.     public Measurer(){
9.         super();
10.     }
11.
12.     int measureLength(Dimension d) throws LengthException{
13.         XXX
14.     }
15. }
```

В строке 13 необходимо вызвать исключение. Какой из предложенных вариантов будет правильным ?

- a new TooShortException(“Shhhhort”);
- b throws new TooShortException(“Shhhhort”);
- c throw new TooShortException(“Shhhhort”);
- d throw TooShortException(“Shhhhort”);

a.) Правильным будет вариант c. В этом случае создается и возбуждается исключение.

Ответ a неверен т.к. создается экземпляр класса TooShortException, но возбуждения исключения не производится (т.е. не используется оператор throw)

Ответ b неверен т.к. вместо оператора throw используется throws

Ответ d неверен т.к. возбуждается исключение, но при этом экземпляр класса исключения не создается

10-9. Каков будет результат работы следующего кода

```
1. public class Test {
2.
3.    public Test() {
4.    }
5.    public static void main(String[] args) {
6.       Test t = new Test();
```

```
7.       XXX
8.    }
9.    private int check(String x,int n){
10.       if( n ==0 )return n;
11.       else if(n == 1){
12.          if ( x != null) return 5;
13.       }
14.       else if ( n == 2 && x != null){
15.          if(x.equals("YES")) return 3;
16.          else if ( x.equals("NO")) return 4;
17.       }
18.       return -1;
19.    }
20. }
```

Если в строке 7 поместить код вызова метода check, то какое из предложений будет верным ?

- a t.check(“ANY”,1) в этом случае обязательно будет выполнена строка 14.
- b t.check(“NO”,2) в этом случае функция вернет значение 4.
- c t.check(“YES”,1) в этом случае функция вернет значение 3
- d else в строке 14 относится к if в строке 11.

a.) Верными ответами будут b,d

Предложение a неверно, т.к. если n=1 и x = null то функция вернет значение 5 в строке 12.

Предложение c неверно потому, что если n = 1, то строки 15, 16 не выполняются, а значение 3 возвращается именно из этих строк.

10-10. Рассмотрим пример связанный с конструкцией switch…case

```
1. public class Test {
2.   public Test() {
3.   }
4.   public static void main(String[] args) {
5.     Test t = new Test();
6.     XXX
7.   }
8.
9.   private String check(int n){
10.     String retStr = "x";
11.     if (n < 3) n--;
12.     switch(n){
13.       case 1:
14.          return "one";
15.       case 2:
16.         n = 3;
17.       case 3:
```

```
18.         break;
19.       case 4:
20.       default:
21.         return retStr;
22.     }
23.     return "Result " + n;
24.   }
25. }
```

Если в строке 6 поместить вызов метода check, то выражения из ниже перечисленных можно считать верными ?

- a t.check(1) “one”
- b t.check(2) “Result 3”
- c t.check(3) “Result 3”
- d t.check(4) “X”
- e t.check(5) “X”

a.) Правильные ответы c,d,e. Рассмотрим логику работы метода при различных входных параметрах.

n=1 В строке 11 производится декремент n т.о. он примет значение 0 и из оператора switch case будет выбрано default

n=2 В строке 11 производится декремент n т.о. он примет значение 1 и будет выполнено условие case 1 и метод вернет значение “one”

n=3 Условие if в строке 11 выполнено не будет т.о. значение n останется без изменений и будет выполнено условие case 3, и нем прерывается выполнение switch и метод вернет Result 3

n=4 Условие if в строке 11 выполнено не будет т.о. значение n останется без изменений и будет выполнено условие case 4 и будет возвращено значение “X”

n=5 Условие if в строке 11 выполнено не будет т.о. значение n останется без изменений и будет выполнено условие default и будет возвращено значение “X”

# Программирование на Java

## Лекция 11. Пакет java.awt

### 20апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Лекция посвящена рассмотрению простейшей графической библиотеки java.awt ....

# Оглавление

# Лекция 11. Пакет java.awt

## Содержание лекции.

# 1. Введение

Трудность при создании независимой от платформы библиотеки заключается в том, что ее разработчикам либо приходится требовать, чтобы все приложения на всех платформах вели себя и выглядели одинаково, либо для поддержки, скажем, трех различных разновидностей интерфейса приходится писать в три раза больше кода. Существуют два взгляда на эту проблему. Один подход заключается в том, что упор делается на графику низкого уровня - рисование пикселей, при этом разработчики библиотеки сами заботятся о внешнем виде каждого компонента. При другом подходе создаются абстракции, подходящие для библиотек каждой из операционных систем, и именно "родные" пакеты данной операционной системы служат подъемной силой для архитектурно-нейтральной библиотеки на каждой из платформ. В Java при создании библиотеки Abstraction Window Toolkit (AWT) выбран второй подход.

Начнем изучение AWT с апплетов.

# 2. Апплеты

Апплеты (applets) - это маленькие приложения, которые размещаются на серверах Internet, транспортируются клиенту по сети, автоматически устанавливаются и запускаются на месте, как часть документа HTML. Когда апплет прибывает к клиенту, его доступ к ресурсам ограничен.

Ниже приведен исходный код канонической программы HelloWorld, оформленной в виде апплета:

```
import java.awt.*;
import java.applet.*;

public class HelloWorldApplet extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello World!", 20, 20);
   }
}
```

Этот апплет начинается двумя строками, которые импортируют все пакеты иерархий java.applet и java.awt. Дальше в нашем примере присутствует метод paint, замещающий одноименный метод класса Applet. При вызове этого метода ему передается аргумент, содержащий ссылку на объект класса Graphics. Последний используется для прорисовки нашего апплета. С помощью метода drawString, вызываемого с этим объектом типа Graphics, в позиции экрана (20,20) выводится строка "Hello World".

Для того, чтобы с помощью браузера запустить этот апплет, нам придется написать несколько строк html-текста.

```
<applet code="HelloWorldApplet" width=200 height=40>
</applet>
```

Вы можете поместить эти строки в отдельный html-файл, либо вставить их в текст этой программы в виде комментария и запустить программу appletviewer с его исходным текстом в качестве аргумента. На экране появится строка приветствия.

## 2.1. Тег HTML <Applet>

Тег <applet> используется для запуска апплета как из HTML-документа, так и из программы appletviewer. Программа appletviewer выполняет каждый найденный ей тег <applet> в отдельном окне, в то время как браузеры позволяют разместить на одной странице несколько апплетов. Синтаксис тэга <APPLET> в настоящее время таков:

```
<APPLET
   CODE = appletFile
   OBJECT = appletSerialFile
   WIDTH = pixels
   HEIGHT = pixels
   [ARCHIVE = jarFiles]
   [CODEBASE = codebaseURL]
   [ALT = alternateText]
   [NAME = appletInstanceName]
   [ALIGN = alignment]
   [VSPACE = pixels]
   [HSPACE = pixels]
>
[< PARAM NAME = AttributeName1 VALUE = AttributeValue1 >]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue2 >]
[HTML-текст, отображаемый при отсутствии поддержки Java]
</APPLET>
```

- CODE = appletClassFile

  CODE - обязательный атрибут, задающий имя файла, в котором содержится оттранслированный код апплета. Имя файла задается относительно codebase, то есть либо от текущего каталога, либо от каталога, указанного в атрибуте CODEBASE. В Java 1.1 вместо этого атрибута может использоваться атрибут OBJECT.

- OBJECT = appletClassSerialFile

  Указывает имя файла, содержащего сериализованный апплет, из которого последний будет восстановлен. При запуске определяемого таким образом апплета должен вызываться не метод init(), а метод start(). Для апплета необходимо задать либо атрибут CODE, либо атрибут OBJECT, но задавать эти атрибуты одновременно нельзя.

- WIDTH = pixels
- HEIGHT = pixels

  WIDTH и HEIGHT - обязательные атрибуты, задающие начальный размер видимой области апплета.

- ARCHIVE = jarFiles

  Этот необязательный атрибут задает список jar-файлов (разделяется запятыми), которые предварительно загружаются в Web-броузер. В этих архивных файлах могут содержаться

файлы классов, изображения, звуки и любые другие ресурсы, необходимые апплету. Для создания архивов используется утилита JAR, синтаксис вызова которой напоминает вызов команды TAR Unix:

```
c:\> jar cf soundmap.jar *.class image.gif sound.wav
```

Очевидно, что передача сжатых jar-файлов повышает эффективность работы. Поэтому многие средства разработки (Lotus JavaBeans, Borland JBuilder) имеют средства для публикации апплетов в виде jar-файлов.

- CODEBASE = codebaseURL

  CODEBASE - необязательный атрибут, задающий базовый URL кода апплета, являющийся каталогом, в котором будет выполняться поиск исполняемого файла апплета (задаваемого в признаке CODE). Если этот атрибут не задан, по умолчанию используется каталог данного HTML-документа. CODEBASE не обязательно должен указывать на тот же узел, с которого был загружен HTML-документ.

- ALT = alternateAppletText

  Признак ALT - необязательный атрибут, задающий короткое текстовое сообщение, которое должно быть выведено (как правило, в виде всплывающей подсказки при нахождении курсора мыши над областью апплета) в том случае, если используемый браузер распознает синтаксис тега <applet> , но выполнять апплеты не умеет. Это не то же самое, что HTML-текст, который можно вставлять между <applet> и </applet> для браузеров, вообще не поддерживающих апплетов.

- NAME = appletInstanceName

  NAME - необязательный атрибут, используемый для задания имени для данного экземпляра апплета. Присвоение апплетам имен необходимо для того, чтобы другие апплеты на этой же странице могли находить их и общаться с ними. Для того, чтобы получить доступ к подклассу MyApplet класса Applet с именем "Duke", нужно написать: MyApplet a = getAppletContext().getApplet("Duke"); После того, как вы получили таким образом дескриптор именованного экземпляра апплета, вы можете вызывать его методы точно так же, как это делается с любым другим объектом.

- ALIGN = alignment

  ALIGN - необязательный атрибут, задающий стиль выравнивания апплета. Этот атрибут трактуется так же, как в теге IMG, возможные его значения - LEFT, RIGHT, TOP, TEXT-TOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM, ABSBOTTOM.

- VSPACE = pixels
- HSPACE = pixels

  Эти необязательные атрибуты задают ширину свободного пространства в пикселях сверху и снизу апплета (VSPACE), и слева и справа от него (HSPACE). Они трактуются точно так же, как одноименные атрибуты тега IMG.

- PARAM NAME = appletAttribute1 VALUE = value1

  Этот тег дает возможность передавать из HTML-страницы апплету необходимые ему аргументы. Апплеты получают эти атрибуты, вызывая метод getParameter(), описываемый ниже.

## 2.2. Передача параметров

- getParameter(String)

Метод getParameter возвращает значение типа String, соответствующее указанному имени параметра. Если вам в качестве параметра требуется значение какого-либо другого типа, вы должны преобразовать строку-параметр самостоятельно. Вы сейчас увидите некоторые примеры использования метода getParameter для извлечения параметров из приведенного ниже примера:

```
<applet code=Testing width=40 height=40>
<param name=fontName value=Univers>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
```

Ниже показано, как извлекается каждый из этих параметров:

```
String FontName = getParameter("fontName");
String FontSize = Integer.parseInt(getParameter("fontSize"));
String Leading = Float.valueOf(getParameter("leading"));
String PaidUp = Boolean.valueOf(getParameter("accountEnabled"));
```

## 2.3. Контекст апплета

- getDocumentBase и getCodeBase

  Возможно, Вы будете писать апплеты, которым понадобится явно загружать данные и текст. Java позволяет апплету загружать данные из каталога, в котором располагается HTML-документ, запустивший апплет (база документа - getDocumentBase), и из каталога, из которого был загружен class-файл с кодом апплета (база кода - getCodeBase).

- AppletContext и showDocument

  AppletContext представляет собой средства, позволяющие получать информацию об окружении работающего апплета. Метод showDocument приводит к тому, что заданный его параметром документ отображается в главном окне браузера или фрейме.

## 2.4. Отладочная печать

Отладочную печать можно выводить в два места: на консоль и в статусную строку программы просмотра апплетов. Для того, чтобы вывести сообщение на консоль, надо написать:

```
  System.out.println("Hello there, welcome to Java");
```

Сообщения на консоли очень удобны, поскольку консоль обычно не видна пользователям апплета, и в ней достаточно места для нескольких сообщений. В браузере Netscape консоль Java доступна из меню Options, пункт "Show Java Console".

Метод showStatus выводит текст в статусной области программы appletviewer или браузера с поддержкой Java. В статусной области можно вывести только одну строку сообщения.

## 2.5. Порядок инициализации апплета

Ниже приведен порядок, в котором вызываются методы класса Applet, с пояснениями, нужно или нет переопределять данный метод.

- init

  Метод init вызывается первым. В нем вы должны инициализировать свои переменные.

- start

  Метод start вызывается сразу же после метода init. Он также используется в качестве стартовой точки для возобновления работы после того, как апплет был остановлен. В то время, как метод init вызывается только однажды - при загрузке апплета, start вызывается каждый раз при выводе HTML-документа, содержащего апплет, на экран. Так, например, если пользователь перейдет к новой WWW-странице, а затем вернется назад к странице с апплетом, апплет продолжит работу с метода start.

- paint

  Метод paint вызывается каждый раз при повреждении апплета. AWT следит за состоянием окон в системе и замечает такие случаи, как, например, перекрытие окна апплета другим окном. В таких случаях, после того, как апплет снова оказывается видимым, для восстановления его изображения вызывается метод paint.

- update

  Используемый по умолчанию метод update класса Applet сначала закрашивает апплет цветом фона по умолчанию, после чего вызывает метод paint. Если вы в методе paint заполняете фон другим цветом, пользователь будет видеть вспышку цвета по умолчанию при каждом вызове метода update - то есть, всякий раз, когда вы перерисовываете апплет. Чтобы избежать этого, нужно заместить метод update. В общем случае нужно выполнять операции рисования в методе update, а в методе paint, к которому будет обращаться AWT, просто вызвать update.

- stop

  Метод stop вызывается в тот момент, когда браузер покидает HTML-документ, содержащий апплет. При вызове метода stop апплет еще работает. Вы должны использовать этот метод для приостановки тех подпроцессов, работа которых необязательна при невидимом апплете. После того, как пользователь снова обратится к этой странице, вы должны будете возобновить их работу в методе start.

- destroy

  Метод destroy вызывается тогда, когда среда (например, браузер Netscape) решает, что апплет нужно полностью удалить из памяти. В этом методе нужно освободить все ресурсы, которые использовал апплет.

## 2.6. Перерисовка

Возвратимся к апплету HelloWorldApplet. В нем мы заместили метод paint, что позволило апплету выполнить отрисовку. В классе Applet предусмотрены дополнительные методы рисования, позволяющие эффективно закрашивать части экрана. При разработке первых апплетов порой непросто понять, почему метод update никогда не вызывается. Для инициации update предусмотрены три варианта метода repaint.

repaint

Метод repaint используется для принудительного перерисовывания апплета. Этот метод, в свою очередь, вызывает метод update. Однако, если ваша система медленная или сильно загружена, метод update может и не вызваться. Близкие по времени запросы на перерисовку могут объединяться AWT, так что метод update может вызываться спорадически. Если вы хотите добиться ритмичной смены кадров изображения, воспользуйтесь методом repaint(time) - это позволит уменьшить количество кадров, нарисованных не вовремя.

repaint(time)

Вы можете вызывать метод repaint, устанавливая крайний срок для перерисовки (этот период задается в миллисекундах относительно времени вызова repaint).

repaint(x, y, w, h)

Эта версия ограничивает обновление экрана заданным прямоугольником, изменены будут только те части экрана, которые в нем находятся.

repaint(time, x, y, w, h)

Этот метод - комбинация двух предыдущих.

## 2.7. Задание размеров графических изображений

Графические изображения вычерчиваются в стандартной для компьютерной графики системе координат, в которой координаты могут принимать только целые значения, а оси направлены слева направо и сверху вниз. У апплетов и изображений есть метод size, который возвращает объект Dimension. Получив объект Dimension, вы можете получить и значения его переменных width и height:

```
Dimension d = size();
System.out.println(d. width + "," + d.height);
```

## 2.8. Простые методы класса Graphics

У объектов класса Graphics есть несколько простых функций рисования. Каждую из фигур можно нарисовать заполненной, либо прорисовать только ее границы. Каждый из методов drawRect, drawOval, fillRect и fillOval вызывается с четырьмя параметрами: int x, int y, int width и int height. Координаты x и y задают положение верхнего левого угла фигуры, параметры width и height определяют ее границы.

- drawLine

```
drawline(int x1, int y1, int x2, int y2)
```

Этот метод вычерчивает отрезок прямой между точками с координатами (x1,y1) и (x2,y2). Эти линии представляют собой простые прямые толщиной в 1 пиксель. Поддержка разных перьев и разных толщин линий не предусмотрена.

- drawArc и fillArc

  Сигнатура методов drawArc и fillArc следующая:

  ```
  drawArc(int x, int y, int width, int height, int startAngle, int sweepAngle)
  ```

  Эти методы вычерчивают (fillArc заполняет) дугу, ограниченную прямоугольником (x,y,width, height), начинающуюся с угла startAngle и имеющую угловой размер sweepAngle. Ноль градусов соответствует положению часовой стрелки на 3 часа, угол отсчитывается против часовой стрелки (например, 90 градусов соответствуют 12 часам, 180 - 9 часам, и так далее).

- drawPolygon и fillPolygon

  Прототипы для этих методов:

  ```
  drawPolygon(int[], int[], int)
  fillPolygon(int[], int[], int)
  ```

  Метод drawPolygon рисует контур многоугольника (ломаную линию), задаваемого двумя массивами, содержащими x и y координаты вершин, третий параметр метода - число пар координат. Метод drawPolygon не замыкает автоматически вычерчиваемый контур. Для того, чтобы прямоугольник получился замкнутым, координаты первой и последней точек должны совпадать.

Рассмотрим поддержку цвета в Java и вернемся к рассмотрению методов Graphics.

## 2.9. Цвет

Цветовая система AWT разрабатывалась так, чтобы была возможность работы со всеми цветами. После того, как цвет задан, Java отыскивает в диапазоне цветов дисплея тот, который ему больше всего соответствует. Вы можете запрашивать цвета в той семантике, к которой привыкли - как смесь красного, зеленого и голубого, либо как комбинацию оттенка, насыщенности и яркости. Вы можете использовать статические переменные класса Color.black для задания какого-либо из общеупотребительных цветов - black, white, red, green, blue, cyan, yellow, magenta, orange, pink, gray, darkGray и lightGray.

Для создания нового цвета используется один из трех описанных ниже конструкторов.

- Color(int, int, int)

  Параметрами для этого конструктора являются три целых числа в диапазоне от 0 до 255 для красного, зеленого и голубого компонентов цвета.

- Color(int)

  У этого конструктора - один целочисленный аргумент, в котором в упакованном виде заданы красный, зеленый и голубой компоненты цвета. Красный занимает биты 16-23, зеленый - 8-15, голубой - 0-7.

- Color(float, float, float)

  Последний из конструкторов цвета, Color(float, float, float), принимает в качестве параметров три значения типа float (в диапазоне от 0.0 до 1.0) для красного, зеленого и голубого базовых цветов.

### 2.9.1. Методы класса Color

- HSBtoRGB(float, float, float)
- RGBtoHSB(int, int, int, float[])

  HSBtoRGB преобразует цвет, заданный оттенком, насыщенностью и яркостью (HSB), в целое число в формате RGB, готовое для использования в качестве параметра конструктора Color(int). RGBtoHSB преобразует цвет, заданный тремя базовыми компонентами, в массив типа float со значениями HSB, соответствующими данному цвету.

  Цветовая модель HSB (Hue-Saturation-Brightness, оттенок-насыщенность-яркость) является альтернативой модели Red-Green-Blue для задания цветов. В этой модели оттенки можно представить как круг с различными цветами (оттенок может принимать значения от 0.0 до 1.0, цвета на этом круге идут в том же порядке, что и в радуге - красный, оранжевый, желтый, зеленый, голубой, синий, фиолетовый). Насыщенность (значение в диапазоне от 0.0 до 1.0) - это шкала глубины цвета, от легкой пастели до сочных цветов. Яркость - это также число в диапазоне от 0.0 до 1.0, причем меньшие значения соответствуют более темным цветам, а большие - более ярким.

- getRed(), getGreen(), getBlue()

  Каждый из этих методов возвращает в младших восьми битах результата значение соответствующего базового компонента цвета.

- getRGB()

  Этот метод возвращает целое число, в котором упакованы значения базовых компонентов цвета, причем

```
red = Oxff & (getRGB() >> 16);
green = Oxff & (getRGB() >> 8);
blue = Oxff & getRGB();
```

Продолжаем описание методов класса Graphics:

- setPaintMode() и setXORMode(Color)

  Режим отрисовки paint - используемый по умолчанию метод заполнения графических изображений, при котором цвет пикселей изменяется на заданный. XOR устанавливает режим рисования, когда результирующий цвет получается выполнением операции XOR (исключающее или) для текущего и указанного цветов (особенно полезно для анимации).

## 2.10. Шрифты

Библиотека AWT обеспечивает большую гибкость при работе со шрифтами благодаря предоставлению соответствующих абстракций и возможности динамического выбора

шрифтов. Вот очень короткая программа, которая печатает на консоли Java имена всех имеющихся в системе шрифтов.

```
/*
 * <applet code="WhatFontsAreHere" width=100 height=40>
 * </applet>
 *
 */
import java.applet.*;
import java.awt.*;

public class WhatFontsAreHere extends Applet {
   public void init() {
      String fontList[];

      // Устаревший способ получения набора шрифтов:
      // Toolkit.getDefaultToolkit().getFontList()
      fontList = GraphicsEnvironment.getLocalGraphicsEnvironment().
         getAvailableFontFamilyNames();
      for (int i=0; i < fontList.length; i++) {
         System.out.println(i + ": " + fontList[i]);
      }
   }
}
```

- drawString

  В предыдущих примерах использовался метод drawString(String, x, y). Этот метод выводит строку с использованием текущих шрифта и цвета. Точка с координатами (x,y) соответствует левой границе базовой линии символов, а не левому верхнему углу, как это принято в других методах рисования. Для того, чтобы понять, как при этом располагается описывающий строку прямоугольник, ниже будет рассмотрен класс FontMetrics.

### 2.10.1. Использование шрифтов

Конструктор класса Font создает новый шрифт с указанным именем, стилем и размером в пунктах:

```
Font StrongFont = new Font("Helvetica", Font.BOLD|Font.ITALIC, 24);
```

В настоящее время доступны следующие имена шрифтов: Dialog, Helvetica, TimesRoman, Courier и Symbol. Для указания стиля шрифта внутри данного семейства предусмотрены три статические переменные. - Font.PLAIN, Font.BOLD и Font.ITALIC, что соответствует обычному стилю, курсиву и полужирному.

Теперь давайте посмотрим на несколько дополнительных методов.

- getFamily и getName

Метод getFamily возвращает строку с именем семейства шрифтов. С помощью метода getName можно получить логическое имя шрифта.

- getSize

  Этот метод возвращает целое число, представляющее собой размер шрифта в пунктах.

- getStyle

  Этот метод возвращает целое число, соответствующее стилю шрифта. Полученный результат можно побитово сравнить со статическими переменными класса Font: - PLAIN, BOLD и ITALIC.

- isBold, isItalic, isPlain

  Эти методы возвращают true в том случае, если стиль шрифта - полужирный (bold), курсив (italic) или обычный (plain), соответственно.

### 2.10.2. Позиционирование и шрифты: FontMetrics

В Java используются различные шрифты, а класс FontMetrics позволяет программисту точно задавать положение выводимого в апплете текста. Прежде всего нам нужно понять кое-что из обычной терминологии, употребляемой при работе со шрифтами:

- Высота (height) - размер от верхней до нижней точки самого высокого символа в шрифте.
- Базовая линия (baseline) - линия, по которой выравниваются нижние границы символов (не считая снижения (descent)).
- Подъем (ascent) - расстояние от базовой линии до верхней точки символа.
- Снижение (descent) - расстояние от базовой линии до нижней точки символа.

### 2.10.3. Использование FontMetrics

Ниже приведены некоторые методы класса FontMetrics:

stringWidth

Этот метод возвращает длину заданной строки для данного шрифта.

bytesWidth, charsWidth

Эти методы возвращают ширину указанного массива байтов для текущего шрифта.

getAscent, getDescent, getHeight

Эти методы возвращают подъем, снижение и ширину шрифта. Сумма подъема и снижения дают полную высоту шрифта. Высота шрифта - это не просто расстояние от самой нижней точки букв g и y до самой верхней точки заглавной буквы T и символов вроде скобок. Высота включает подчеркивания и т.п.

getMaxAscent и getMaxDescent

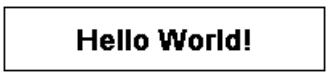
Эти методы служат для получения максимальных подъема и снижения всех символов в шрифте.

### 2.10.4. Центрирование текста

Давайте теперь воспользуемся методами объекта FontMetrics для получения подъема, снижения и длины строки, которую требуется нарисовать, и с помощью полученных значений отцентрируем ее в нашем апплете.

```
/*
 * <applet code="HelloWorld" width=200 height=100>
 * </applet>
 *
 */
import java.applet.*;
import java.awt.*;

public class HelloWorld extends Applet {
   final Font f = new Font("Helvetica", Font.BOLD, 18);
   public void paint(Graphics g) {
      Dimension d = this.size();
      g.setColor(Color.white);
      g.fillRect(0,0,d.width,d.height);
      g.setColor(Color.black);
      g.setFont(f);
      drawCenteredString("Hello World!", d.width, d.height, g);
      g.drawRect(0,0,d.width-1,d.height-1);
   }
   public void drawCenteredString(String s, int w, int h, Graphics g) {
      FontMetrics fm = g.getFontMetrics();
      int x = (w - fm.stringWidth(s)) / 2;
      int y = (fm.getAscent() + (h - (fm.getAscent() + fm.getDescent()))/2);
      g.drawString(s, x, y);
   }
}
```

Вот как выглядит апплет в действии:




# 3. Базовые классы

Теперь, когда рассмотрены классы Graphics и Fonts изучим базовую архитектуру AWT, касающуюся интерфейсных объектов.

- Компоненты

  Component - это абстрактный класс, который инкапсулирует все атрибуты визуального интерфейса - обработка ввода с клавиатуры, управление фокусом, взаимодействие с мышью, уведомление о входе/выходе из окна, изменения размеров и положения окон, прорисовка своего собственного графического представления, сохранение текущего

текстового шрифта, цветов фона и переднего плана (более 100 методов). Перейдем к некоторым конкретным подклассам класса Component.

- Container

  Container - это абстрактный подкласс класса Component, определяющий дополнительные методы, которые дают возможность помещать в него другие компоненты, что дает возможность построения иерархической системы визуальных объектов. Container отвечает за расположение содержащихся в нем компонентов с помощью интерфейса LayoutManager, описание которого будет позднее в этой главе.

- Panel

  Класс Panel - это очень простая специализация класса Container. В отличие от последнего, он не является абстрактным классом. Поэтому о Panel можно думать, как о допускающем рекурсивную вложенность экранном компоненте. С помощью метода add в объекты Panel можно добавлять другие компоненты. После того, как в него добавлены какие-либо компоненты, можно вручную задавать их положение и изменять размер с помощью методов setLocation, setSize и setBounds класса Component.

  В предыдущей главе мы уже использовали один из подклассов Panel - Applet. Каждый раз, когда мы создавали Applet, методы paint и update рисовали его изображение на поверхности объекта Panel. Прежде, чем мы углубимся в методы Panel, давайте познакомимся с основными компонентами AWT, которые можно вставлять в пустую Panel при создании графических приложений.

# 4. Основные компоненты

- Canvas

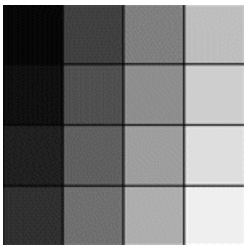
Основная идея использования объектов Canvas в том, что они являются семантически свободными компонентами. Вы можете придать объекту Canvas любое поведение и любой желаемый внешний вид. Его имя подразумевает, что этот класс является пустым холстом, на котором вы можете "нарисовать" любой компонент - такой, каким вы его себе представляете.

Произведем от Canvas подкласс GrayCanvas, который будет просто закрашивать себя серым цветом определенной насыщенности. Наш апплет будет создавать несколько таких объектов, каждый со своей интенсивностью серого цвета.

```
/* <applet code = "PanelDemo"
width=300
height=300>
   </applet>
 */
import java.awt.*;
import java.applet.*;

class GrayCanvas extends Canvas {
   Color gray;
   public GrayCanvas(float g) {
```

```
      gray = new Color(g, g, g);
   }

   public void paint(Graphics g) {
      Dimension size = size();
      g.setColor(gray);
      g.fillRect(0, 0, size.width, size.height);
      g.setColor(Color.black);
      g.drawRect(0, 0, size.width-1, size.height-1);
   }
}

public class PanelDemo extends Applet {
   static final int n = 4;
   public void init() {
      setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
            float g = (i * n + j) / (float) (n * n);
            Canvas c = new GrayCanvas(g);
            add(c);
            c.setSize(width / n, height / n);
            c.setLocation(i * width / n, j * height / n);
         }
      }
   }
}
```

Вот как этот апплет выглядит на экране:



Мы устанавливаем размер каждого из объектов Canvas на основе значения, полученного с помощью метода size, который возвращает объект класса Dimension. Обратите внимание на то, что для размещения объектов Canvas в нужные места используются методы resize и move. Такой способ станет очень утомительным, когда мы перейдем к более сложным компонентам и более интересным вариантам расположения. А пока в нашем апплете для выключения упомянутого механизма использован вызов метода setLayout(null).

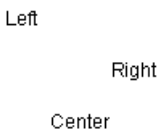
- Label

Функциональность класса Label сводится к тому, что он знает, как нарисовать объект String - текстовую строку, выровняв ее нужным образом. Шрифт и цвет, которыми отрисовывается строка метки, являются частью базового определения класса Component. Для работы с

этими атрибутами предусмотрены пары методов getFont/setFont и getForeground/setForeground. Задать или изменить текст строки после создания объекта с помощью метода setText. Для задания режимов выравнивания в классе Label определены три константы - LEFT, RIGHT и CENTER. Ниже приведен пример, в котором создаются три метки, каждая - со своим режимом выравнивания.

```
/* <applet code = "LabelDemo" width=100 height=100>
   </applet>
 */
import java.awt.*;
import java.applet.*;

public class LabelDemo extends Applet {
   public void init() {
      setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      Label left = new Label("Left", Label.LEFT);
      Label right = new Label("Right", Label.RIGHT);
      Label center = new Label("Center", Label.CENTER);
      add(left);
      add(right);
      add(center);
      left.setBounds(0, 0, width, height / 3);
      right.setBounds(0, height / 3, width, height / 3);
      center.setBounds(0, 2 * height / 3, width, height / 3);
   }
}
```

На этот раз, чтобы одновременно переместить и изменить размер объектов Label, мы использовали метод reshape. Ширина каждой из меток равна полной ширине апплета, высота - 1/3 высоты апплета. Вот как этот апплет должен выглядеть, если его запустить:



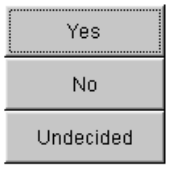

- Button

Объекты-кнопки помечаются строками, причем эти строки нельзя выравнивать подобно строкам объектов Label (они всегда центрируются внутри кнопки). Позднее в данной главе речь пойдет о том, как нужно обрабатывать события, возникающие при нажатии и отпускании пользователем кнопки. Ниже приведен пример, в котором создаются три расположенные по вертикали кнопки.

```
/* <applet code = "ButtonDemo" width=100 height=100>
   </applet>
 */
import java.awt.*;
```

```
import java.applet.*;

public class ButtonDemo extends Applet {
   public void init() {
      setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      Button yes = new Button("Yes");
      Button no = new Button("No");
      Button maybe = new Button("Undecided");
      add(yes);
      add(no);
      add(maybe);
      yes.setBounds(0, 0, width, height / 3);
      no.setBounds(0, height / 3, width, height / 3);
      maybe.setBounds(0, 2 * height / 3, width, height / 3);
   }
}
```

Вот как выглядит работающий апплет:



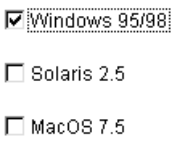

- Checkbox

Класс Checkbox часто используется для выбора одной из двух возможностей. При создании объекта Checkbox ему передается текст метки и логическое значение, чтобы задать исходное состояние окошка с отметкой. Программно можно получать и устанавливать состояние окошка с отметкой с помощью методов getState и setState. Ниже приведен пример с тремя объектами Checkbox, задаваемое в этом примере исходное состояние соответствует отметке в первом объекте.

```
/* <applet code = "CheckBoxDemo" width=120 height=100>
   </applet>
 */
import java.awt.*;
import java.applet.*;

public class CheckboxDemo extends Applet {
   public void init() {
      setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      Checkbox win95 = new Checkbox("Windows 95/98", null, true);
      Checkbox Solaris = new Checkbox("Solaris 2.5");
      Checkbox mac = new Checkbox("MacOS 7.5");
```

```
      add(win95);
      add(solaris);
      add(mac);
      win95.setBounds(0, 0, width, height / 3);
      Solaris.setBounds(0, height / 3, width, height / 3);
      mac.setBounds(0, 2 * height / 3, width, height / 3);
   }
}
```

Ниже приведен внешний вид работающего апплета:



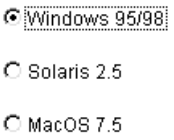

- CheckboxGroup

Второй параметр конструктора Checkbox (в предыдущем примере мы ставили там null) используется для группирования нескольких объектов Checkbox. Для этого сначала создается объект CheckboxGroup, затем он передается в качестве параметра любому количеству конструкторов Checkbox, при этом предоставляемые этой группой варианты выбора становятся взаимоисключающими (только один может быть задействован). Предусмотрены и методы, которые позволяют получить и установить группу, к которой принадлежит конкретный объект Checkbox - getCheckboxGroup и setCheckboxGroup. Вы можете пользоваться методами getCurrent и setCurrent для получения и установки состояния выбранного в данный момент объекта Checkbox. Ниже приведен пример, отличающийся от предыдущего тем, что теперь различные варианты выбора в нем взаимно исключают друг друга.

```
/* <applet code = "CheckboxGroupDemo" width=120 height=100>
   </applet>
 */
import java.awt.*;
import java.applet.*;

public class CheckboxGroupDemo extends Applet {
   public void init() {
      setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      CheckboxGroup g = new CheckboxGroup();
      Checkbox win95 = new Checkbox("Windows 95/98", g, true);
      Checkbox solaris = new Checkbox("Solaris 2.5", g, false);
      Checkbox mac = new Checkbox("MacOS 7.5", g, false);
      add(win95);
      add(solaris);
      add(mac);
      win95.setBounds(0, 0, width, height / 3);
      solaris.setBounds(0, height / 3, width, height / 3);
```

```
      mac.setBounds(0, 2 * height / 3, width, height / 3);
   }
}
```

Обратите внимание - окошки изменили свою форму, теперь они не квадратные, а круглые:



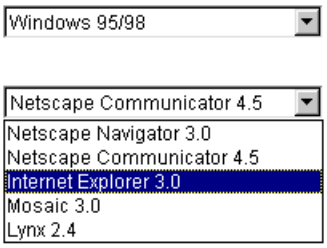

- Choice

Класс Choice (выбор) используется при создании раскрывающихся списочных меню (выпадающих списков типа ComboBox в Windows). Компонент Choice занимает ровно столько места, сколько требуется для отображения выбранного в данный момент элемента, когда пользователь щелкает мышью на нем, раскрывается меню со всеми элементами, в котором можно сделать выбор. Каждый элемент меню - это строка, которая выводится, выровненная по левой границе. Элементы меню выводятся в том порядке, в котором они были добавлены в объект Choice. Метод countItems возвращает количество пунктов в меню выбора. Вы можете задать пункт, который выбран в данный момент, с помощью метода select, передав ему либо целый индекс (пункты меню перечисляются с нуля), либо строку, которая совпадает с меткой нужного пункта меню. Аналогично, с помощью методов getSelectedItem и getSelectedIndex можно получить, соответственно, строку-метку и индекс выбранного в данный момент пункта меню. Вот очередной простой пример, в котором создается два объекта Choice.

```
/* <applet code = "ChoiceDemo" width=200 height=100>
   </applet>
 */
import java.awt.*;
import java.applet.*;

public class ChoiceDemo extends Applet {
   public void init() {
      setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      Choice os = new Choice();
      Choice browser = new Choice();
      os.addItem("Windows 95/98");
      os.addItem("Solaris 2.5");
      os.addItem("MacOS 7.5");
      browser.addItem("Netscape Navigator 3.0");
      browser.addItem("Netscape Communicator 4.5");
      browser.addItem("Internet Explorer 3.0");
      browser.addItem("Mosaic 3.0");
      browser.addItem("Lynx 2.4");
      browser.select("Netscape Communicator 4.5");
      add(os);
```

```
      add(browser);
      os.setBounds(0, 0, width, height / 2);
      browser.setBounds(0, height / 2, width, height / 2);
   }
}
```

А вот как выглядят эти выпадающие списки:

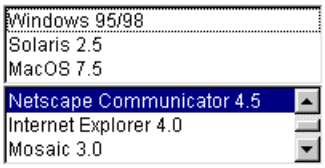



- List

Класс List представляет собой компактный список с возможностью выбора нескольких вариантов и с прокруткой (аналог ListBox в Windows). Ниже приведен пример с двумя списками выбора, один из которых допускает выбор нескольких элементов, а второй - выбор единственного элемента.

```
/* <applet code = "ListDemo" width=200 height=100>
   </applet>
 */
import java.awt.*;
import java.applet.*;

public class ListDemo extends Applet {
   public void init() { setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      List os = new List(0, true);
      List browser = new List(0, false);
      os.addItem("Windows 95/98");
      os.addItem("Solaris 2.5");
      os.addItem("MacOS 7.5");
      browser.addItem("Netscape Navigator 3.0");
      browser.addItem("Netscape Communicator 4.5");
      browser.addItem("lnternet Explorer 4.0");
      browser.addItem("Mosaic 3.0");
      browser.addItem("Lynx 2.4");
      browser.select(1);
      add(os);
      add(browser);
      os.setBounds(0, 0, width, height / 2);
      browser.setBounds(0, height / 2, width, height / 2);
   }
}
```

Заметьте, что у нижнего списка имеется линейка прокрутки, поскольку все его элементы не уместились в заданный нами размер:




- Scrollbar

Объекты Scrollbar (линейки прокрутки) используются для выбора подмножества значений между заданными минимумом и максимумом. Визуально у линейки прокрутки есть несколько органов управления, ориентированных либо вертикально, либо горизонтально. Стрелки на каждом из ее концов показывают, что, нажав на них, вы можете продвинуться на один шаг в соответствующем направлении. Текущее положение отображается с помощью движка линейки прокрутки, которым пользователь также может управлять, устанавливая требуемое положение линейки.

Конструктор класса Scrollbar позволяет задавать ориентацию линейки прокрутки - для этого предусмотрены константы VERTICAL и HORIZONTAL. Кроме того, с помощью конструктора можно задать начальное положение и размер движка, а так же минимальное и максимальное значения, в пределах которых линейка прокрутки может изменять параметр. Для получения и установки текущего состояния линейки прокрутки используются методы getValue и setValue. Кроме того, воспользовавшись методами getMinimum и getMaximum, вы можете получить рабочий диапазон объекта. Ниже приведен пример, в котором создается и вертикальная, и горизонтальная линейки прокрутки.

```
/* <applet code = "ScrollbarDemo" width=200 height=100>
   </applet>
 */
import java.awt.*;
import java.applet.*;

public class ScrollbarDemo extends Applet {
   public void init() {
      setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer. parseInt(getParameter("height"));
      Scrollbar hs = new Scrollbar(Scrollbar.HORIZONTAL, 50, width / 10, 0,
100);
      Scrollbar vs = new Scrollbar(Scrollbar.VERTICAL, 50, height / 2, 0,
100);
      add(hs);
      add(vs);
      int thickness = 16;
      hs.setBounds(0, height - thickness, width - thickness, thickness);
      vs.setBounds(width - thickness, 0, thickness, height - thickness);
   }
}
```

В этом примере скроллируется, конечно, пустая область:

- TextField

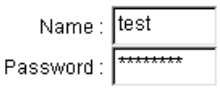
Класс TextField представляет собой реализацию однострочной области для ввода текста. Такие области часто используются в формах для пользовательского ввода. Вы можете "заморозить" содержимое объекта TextField с помощью метода setEditable, а метод isEditable сообщит вам, можно ли редактировать текст в данном объекте. Текущее значение объекта можно получить методом getText и установить методом setText. С помощью метода select можно выбрать фрагмент строки, задавая его начало и конец, отсчитываемые с нуля. Для выбора всей строки используется метод selectAll.

Метод setEchoChar задает символ, который будет выводиться вместо любых вводимых символов. Вы можете проверить, находится ли объект TextField в этом режиме, с помощью метода echoCharIsSet, и узнать, какой именно символ задан для эхо-печати, с помощью метода getEchoChar. Вот пример, в котором создаются классические поля для имени пользователя и пароля.

```
/* <applet code = "TextFieldDemo" width=200 height=100>
</applet>
*/
import java.awt.*;
import java.applet.*;

public class TextFieldDemo extends Applet {
   public void init() {
      setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      Label namep = new Label("Name : ", Label.RIGHT);
      Label passp = new Label("Password : ", Label.RIGHT);
      TextField name = new TextField(8);
      TextField pass = new TextField(8);
      pass.setEchoChar('*');
      add(namep);
      add(name);
      add(passp);
      add(pass);
      int space = 25;
      int w1 = width / 3;
      namep.setBounds(0, (height - space) / 2, w1, space);
      name.setBounds(w1, (height - space) / 2, w1, space);
      passp.setBounds(0, (height + space) / 2, w1, space);
      pass.setBounds(w1, (height + space) / 2, w1, space);
```

```
    }
}
```

Ниже приведен внешний вид работающего апплета:

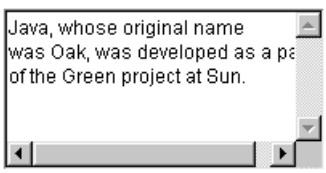



- TextArea

Порой одной строки текста оказывается недостаточно для конкретной задачи. AWT включает в себя очень простой многострочный редактор обычного текста, называемый TextArea. Конструктор класса TextArea воспринимает значение типа String в качестве начального текста объекта. Кроме того, в конструкторе указывается число колонок и строк текста, которые нужно выводить. Есть три метода, которые позволяют программе модифицировать содержимое объекта TextArea: appendText добавляет параметр типа String в конец буфера; insertText вставляет строку в заданное отсчитываемым от нуля индексом место в буфере; replaceText копирует строку-параметр в буфер, замещая ею текст, хранящийся в буфере между первым и вторым параметрами-смещениями. Ниже приведена программа, создающая объект TextArea и вставляющая в него строку.

```
/* <applet code = "TextAreaDemo" width=200 height=100>
   </applet>
 */
import java.awt.*;
import java.applet.*;

public class TextAreaDemo extends Applet {
   public void init() {
      setLayout(null);
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      String val = "Java, whose original name\n"+
         "was Oak, was developed as a part\n"+
         "of the Green project at Sun.\n";
      System.out.println(val);
      TextArea text = new TextArea(val, 80, 40);
      add(text);
      text.setBounds(0, 0, width, height);
   }
}
```

Ниже приведен внешний вид работающего апплета:



# 5. Менеджеры компоновки

- Layout

Все компоненты, с которыми мы работали до сих пор в этой главе, размещались "вручную". И в каждом примере мы вызывали загадочный метод setLayout(null). Этот вызов запрещал использование предусмотренного по умолчанию механизма управления размещением компонентов. Для решения подобных задач в AWT предусмотрены диспетчеры размещения (layout managers).

- LayoutManager

Каждый класс, реализующий интерфейс LayoutManager, следит за списком компонентов, которые хранятся с именами типа String. Всякий раз, когда вы добавляете компонент в Panel, диспетчер размещения уведомляется об этом. Если требуется изменить размер объекта Panel, то идет обращение к диспетчеру посредством методов minimumLayoutSize и preferredLayoutSize. В каждом компоненте, который приходится обрабатывать диспетчеру, должны присутствовать реализации методов preferredSize и minimumSize. Эти методы должны возвращать предпочтительный и минимальный размеры для прорисовки компонента, соответственно. Диспетчер размещения по возможности будет пытаться удовлетворить эти запросы, в то же время заботясь о целостности всей картины взаимного расположения компонентов.

В Java есть несколько предопределенных классов - диспетчеров размещения, описываемых ниже.

- FlowLayout

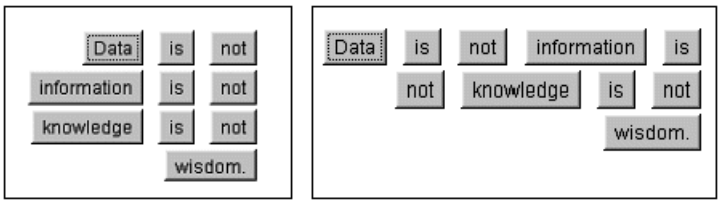
Класс FlowLayout реализует простой стиль размещения, при котором компоненты располагаются, начиная с левого верхнего угла, слева направо и сверху вниз. Если в данную строку не помещается очередной компонент, он располагается в левой позиции новой строки. Справа, слева, сверху и снизу компоненты отделяются друг от друга небольшими промежутками. Ширину этого промежутка можно задать в конструкторе FlowLayout. Каждая строка с компонентами выравнивается по левому или правому краю, либо центрируется в зависимости от того, какая из констант LEFT, RIGHT или CENTER была передана конструктору. Режим выравнивания по умолчанию - CENTER, используемая по умолчанию ширина промежутка - 5 пикселей.

Ниже приведен пример, в котором в Panel включается несколько компонентов Label. Объект Panel использует FlowLayout с выравниванием RIGHT.

```
/* <applet code = "FlowLayoutDemo" width=200 height=100>
   </applet>
   <applet code = "FlowLayoutDemo" width=250 height=100>
   </applet>
 */

import java.awt.*;
import java.applet.*;
import java.util.*;

public class FlowLayoutDemo extends Applet {
   public void init() {
```

```
      setLayout(new FlowLayout(FlowLayout.RIGHT, 10, 3));
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      String val = "Data is not information " +
      "is not knowledge is not wisdom.";
      StringTokenizer st = new StringTokenizer(val);
      while (st.hasMoreTokens()) {
         add(new Button(st.nextToken()));
      }
   }
}
```

Необходимо вызвать пример для двух различных страниц (ширина апплета 200 и 250 пикселей) для того, чтобы проиллюстрировать, как объекты Label перетекают из строки в строку, и при этом строки выравниваются по правому краю:



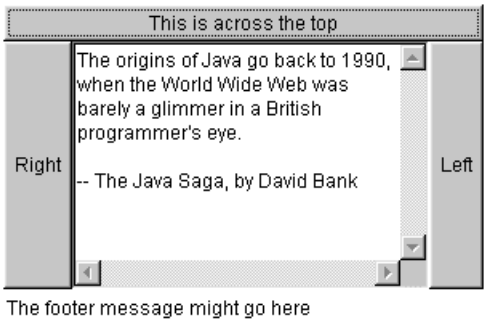

- BorderLayout

Класс BorderLayout реализует обычный стиль размещения для окон верхнего уровня, в котором предусмотрено четыре узких компонента фиксированной ширины по краям, и одна большая область в центре, которая может расширяться и сужаться в двух направлениях, занимая все свободное пространство окна. У каждой из этих областей есть строки-имена: String.North, String.South, String.East и String.West соответствуют четырем краям, а Center - центральной области. Ниже приведен пример BorderLayout с компонентом в каждой из названных областей.

```
/* <applet code = "BorderLayoutDemo" width=300 height=200>
   </applet>
 */
import java.awt.*;
import java.applet.*;
import java.util.*;

public class BorderLayoutDemo extends Applet {
   public void init() {
      setLayout(new BorderLayout());
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      add("North", new Button("This is across the top"));
      add("South", new Label("The footer message might go here"));
      add("East", new Button("Left"));
      add("West", new Button("Right"));
      String msg = "The origins of Java go back to 1990,\n"+
         "when the World Wide Web was\n"+
```

```
         "barely a glimmer in a British\n"+
         "programmer's eye.\n\n"+
         "-- The Java Saga, by David Bank";
      add("Center", new TextArea(msg));
   }
}
```

Ниже приведен внешний вид работающего апплета:

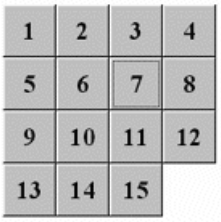



- GridLayout

Класс GridLayout размещает компоненты в простой равномерной сетке. Конструктор этого класса позволяет задавать количество строк и столбцов. Ниже приведен пример, в котором GridLayout используется для создания сетки 4x4, 15 квадратов из 16 заполняются кнопками, помеченными соответствующими индексами. Как вы уже, наверное, поняли, это - панель для игры в "пятнашки".

```
/* <applet code = "GridLayoutDemo" width=200 height=200>
   </applet>
 */
import java.awt.*;
import java.applet.*;

public class GridLayoutDemo extends Applet {
   static final int n = 4;
   public void init() {
      setLayout(new GridLayout(n, n));
      setFont(new Font("Helvetica", Font.BOLD, 24));
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      for (int i = 0; i < n; i++) {
         for (int j = 0; j < n; j++) {
            int k = i * n + j;
            if (k > 0)
            add(new Button("" + k));
         }
      }
   }
}
```

Если доработать этот пример - получится неплохая игра:




На самом деле, если подобное игровое поле состоит из большого количества активных полей (как например, стандартная игра "Сапер" в Windows), то использовать AWT-кнопки будет уже не рационально, так как множество собственных компонент Windows будут заметно замедлять работу приложения.

- Insets

Класс Insets используется для того, чтобы вставлять в объект Panel границы, напоминающие горизонтальные и вертикальные промежутки между объектами, которые делает диспетчер размещения. Для того, чтобы добиться вставки границ в объект Panel, нужно заместить метод Insets реализацией, возвращающей новый объект Insets с четырьмя целыми значениями, соответствующими ширине верхнего, нижнего, левого и правого краев.

```
public Insets insets() {
   return new Insets(10, 10, 10, 10);
}
```

- CardLayout

Класс CardLayout по своему уникален. Он отличается от других программ управления размещением компонентов тем, что представляет несколько различных вариантов размещения, которые можно сравнить с колодой карт. Колоду можно тасовать так, чтобы в данный момент времени наверху была только одна из карт. Это может быть полезно при создании интерфейсов пользователя, в которых есть необязательные компоненты, включаемые и выключаемые динамически в зависимости от реакции пользователя.

# 6. Окна

- Window

Класс Window во многом напоминает Panel за тем исключением, что он создает свое собственное окно верхнего уровня. Большая часть программистов скорее всего будет использовать не непосредственно класс Window, а его подкласс Frame.

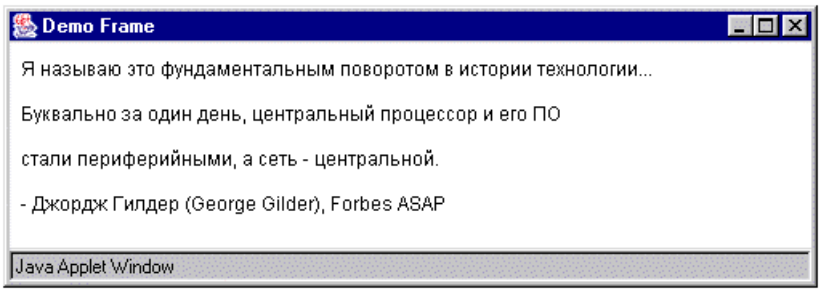
- Frame

Frame - это как раз то, что обычно и считают окном на рабочей поверхности экрана. У объекта Frame есть строка с заголовком, управляющие элементы для изменения размера и линейка меню. Для того чтобы вывести/спрятать изображение объекта Frame, нужно использовать методы show и hide. Ниже приведен пример апплета, который показывает объект Frame с содержащимся в нем компонентом TextArea.

```
/* <applet code = "FrameDemo" width=200 height=200>
   </applet>
```

```
 */
import java.awt.*;
import java.applet.*;

public class FrameDemo extends Applet {
   public void init() {
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));

      Frame f = new Frame("Demo Frame");
      f.setSize(width, height);
      f.setLayout(new FlowLayout(FlowLayout.LEFT));

      f.add(new Label("Я называю это фундаментальным поворотом в истории
технологии..."));
      f.add(new Label("Буквально за один день, центральный процессор и его
ПО"));
      f.add(new Label("стали периферийными, а сеть - центральной."));
      f.add(new Label("- Джордж Гилдер (George Gilder), Forbes ASAP"));

      f.show();
   }
}
```

Вот как выглядит такой фрейм:



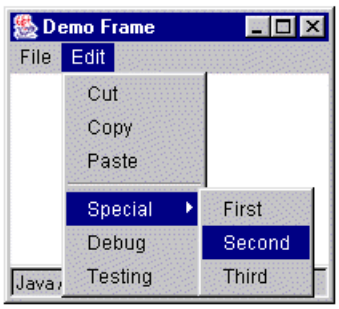

# 7. Меню

С каждым окном верхнего уровня может быть связана линейка меню. Объект MenuBar может включать в себя несколько объектов Menu. Последние, в свою очередь, содержат в себе список вариантов выбора - объектов MenuItem. Menu - подкласс MenuItem, так что объекты Menu также могут включаться в этот список, что позволяет создавать иерархически вложенные подменю. Вот пример, в котором к окну добавлены несколько вложенных меню.

```
/* <applet code = "MenuDemo" width=200 height=200>
   </applet>
 */
import java.awt.*;
import java.applet. *;
```

```
public class MenuDemo extends Applet {
   public void init() {
      int width = Integer.parseInt(getParameter("width"));
      int height = Integer.parseInt(getParameter("height"));
      Frame f = new Frame("Demo Frame");
      f.setSize(width, height);
      MenuBar mbar = new MenuBar();
      f.setMenuBar(mbar);
      Menu file = new Menu("File");
      file.add(new MenuItem("New... "));
      file.add(new MenuItem("Open..."));
      file.add(new MenuItem("Close"));
      file.add(new MenuItem("-"));
      file.add(new MenuItem("Quit..."));
      mbar.add(file);
      Menu edit = new Menu("Edit");
      edit.add(new MenuItem("Cut"));
      edit.add(new MenuItem("Copy"));
      edit.add(new MenuItem("Paste"));
      edit.add(new MenuItem("-"));
      Menu sub = new Menu("Special");
      sub.add(new MenuItem("First"));
      sub.add(new MenuItem("Second"));
      sub.add(new MenuItem("Third"));
      edit.add(sub);
      edit.add(new CheckBoxMenuItem("Debug"));
      edit.add(new CheckBoxMenuItem("Testing"));
      mbar.add(edit);
      f.show();
   }
}
```

Посмотрим на практически классическое меню:




# 8. Обработка событий

Модель обработки событий в AWT представляет собой, по существу, модель обратных вызовов (callback). При создании GUI-элемента ему сообщается, какой метод или методы

он должен вызывать при возникновении в нем определенного события (нажатия кнопки, мыши и т.п.). Эту модель очень легко использовать в C++, поскольку этот язык позволяет оперировать указателями на методы (чтобы определить обратный вызов, необходимо всего лишь передать указатель на функцию). Однако в Java это недопустимо (методы не являются объектами). Поэтому для реализации модели необходимо определить класс, реализующий некоторый специальный интерфейс. Затем можно передать экземпляр такого класса GUI-элементу, обеспечивая таким образом обратный вызов. Когда наступит ожидаемое событие, GUI-элемент вызовет соответствующий метод объекта, определенного ранее.

Модель обработки событий Java используется как в пакете AWT, так и в JavaBeans API. В этой модели разным типам событий соответствуют различные классы Java. Каждое событие является подклассом класса java.util.EventObject. События пакета AWT, которые и рассматриваются в данной главе, являются подклассом java.awt.AWTEvent. Для удобства события различных типов пакета AWT (например, MouseEvent или ActionEvent) помещены в новый пакет java.awt.event.

Для каждого события существует порождающий его объект, который можно получить с помощью метода getSource(), и каждому событию пакета AWT соответствует определенный идентификатор, который позволяет получить метод getID(). Это значение используется для того, чтобы отличать события различных типов, которые могут описываться одним и тем же классом событий. Например, для класса FocusEvent возможны два типа событий: FocusEvent.FOCUS_GAINED и FocusEvent.FOCUS_LOST. Подклассы событий содержат информацию, связанную с данным типом события. Например, в классе MouseEvent существуют методы getX(), getY() и getClickCount (). Этот класс наследует, в числе прочих, и методы getModifiers() и getWhen().

Модель обработки событий Java базируется на концепции слушателя событий. Слушателем события является объект, заинтересованный в получении данного события. В объекте, который порождает событие (в источнике событий), содержится список слушателей, заинтересованных в получении уведомления о том, что данное событие произошло, а также методы, которые позволяют слушателям добавлять или удалять себя из этого списка. Когда источник порождает событие (или когда объект источника зарегистрирует событие, связанное с вводом информации пользователем), он оповещает все объекты слушателей событий о том, что данное событие произошло.

Источник события оповещает объект слушателя путем вызова специального метода и передачи ему объекта события (экземпляра подкласса EventObject). Для того чтобы источник мог вызвать данный метод, он должен быть реализован для каждого слушателя. Это объясняется тем, что все слушатели событий определенного типа должны реализовывать соответствующий интерфейс. Например, объекты слушателей событий ActionEvent должны реализовывать интерфейс ActionListener. В пакете java.awt.event содержатся интерфейсы слушателей для каждого из определенных в нем типов событий (например, для событий MouseEvent здесь определено два интерфейса слушателей: MouseListener и MouseMotionListener). Все интерфейсы слушателей событий являются расширениями интерфейса java.util.EventListener. В этом интерфейсе не определяется ни один из методов, но он играет роль интерфейса-метки, в котором однозначно определены все слушатели событий как таковые.

В интерфейсе слушателя событий может определяться несколько методов. Например, класс событий, подобный MouseEvent, описывает несколько событий, связанных с мышью,

таких как события нажатия и отпускания кнопки мыши. Эти события вызывают различные методы соответствующего слушателя. По установленному соглашению, методам слушателей событий может быть передан один единственный аргумент, являющийся объектом того события, которое соответствует данному слушателю. В этом объекте должна содержаться вся информация, необходимая программе для формирования реакции на данное событие. В таблице 6 приведены определенные в пакете java.awt.event типы событий, соответствующие им слушатели, а также методы, определенные в каждом интерфейсе слушателя.

Таблица 1. Типы событий, слушатели и методы слушателей в Java

| Класс события | Интерфейс слушателя | Методы слушателя |
|---|---|---|
| ActionEvent | ActionListener | actionPerformed() |
| AdjustmentEvent | AdjustmentListener | adjustmentValueChanged() |
| ComponentEvent | ComponentListener | componentHidden()<br>componentMoved()<br>componentResized()<br>componentShown() |
| ContainerEvent | ContainerListener | componentAdded()<br>componentRemoved() |
| FocusEvent | FocusListener | focusGained()<br>focusLost () |
| ItemEvent | ItemListener | itemStateChanged() |
| KeyEvent | KeyListener | keyPressed()<br>keyReleased()<br>keyTyped() |
| MouseEvent | MouseListener | mouseClicked()<br>mouseEntered()<br>mouseExited()<br>mousePressed()<br>mouseReleased() |
| MouseMotionEvent | MouseMotionListener | mouseDragged()<br>mouseMoved() |
| TextEvent | TextListener | textValueChanged() |
| WindowEvent | WindowListener | windowActivated()<br>windowClosed()<br>windowClosing()<br>windowDeactivated()<br>windowDeiconified()<br>windowIconified()<br>windowOpened() |

Для каждого интерфейса слушателей событий, содержащего несколько методов, в пакете java.awt.event определен простой класс-адаптер, который обеспечивает пустое тело для каждого из методов соответствующего интерфейса. Когда нужен только один или два таких метода, иногда проще получить подкласс класса-адаптера, чем реализовать интерфейс самостоятельно. При получении подкласса адаптера требуется лишь переопределить те методы, которые нужны, а при прямой реализации интерфейса необходимо определить все методы, в том числе и ненужные в данной программе. Заранее определенные классы-адаптеры называются так же, как и интерфейсы, которые они реализуют, но в этих названиях Listener заменяется на Adapter: MouseAdapter, WindowAdapter и т.д.

Как только реализован интерфейс слушателя или получены подклассы класса-адаптера, необходимо создать экземпляр нового класса, чтобы определить конкретный объект слушателя событий. Затем этот слушатель должен быть зарегистрирован соответствующим

источником событий. В программах пакета AWT источником событий всегда является какой-нибудь элемент пакета. В методах регистрации слушателей событий используются стандартные соглашения об именах: если источник событий порождает события типа X, в нем существует метод addXListener () для добавления слушателя и метод removeXListener() для его удаления. Одной из приятных особенностей модели обработки событий Java является возможность легко определять типы событий, которые могут порождаться данным элементом. Для этого следует просто просмотреть, какие методы зарегистрированы для его слушателя событий. Например, из описания API для объекта класса Button следует, что он порождает события ActionEvent. В таблице 7 приведен список элементов пакета AWT и событий, которые они порождают.

Таблица 2. Элементы пакета AWT и порождаемые ими события в Java1.1

| Элемент | Порождаемое событие | Значение |
|---|---|---|
| Button | ActionEvent | Пользователь нажал кнопку |
| CheckBox | ItemEvent | Пользователь установил или сбросил флажок |
| CheckBoxMenuItem | ItemEvent | Пользователь установил или сбросил флажок рядом с пунктом меню |
| Choice | ItemEvent | Пользователь выбрал элемент списка или отменил его выбор |
| Component | ComponentEvent | Элемент либо перемещен, либо он стал скрытым, либо видимым |
| | FocusEvent | Элемент получил или потерял фокус ввода |
| | KeyEvent | Пользователь нажал или отпустил клавишу |
| | MouseEvent | Пользователь нажал или отпустил кнопку мыши, либо курсор мыши вошел или покинул область, занимаемую элементом, либо пользователь просто переместил мышь или переместил мышь при нажатой кнопке мыши |
| Container | ContainerEvent | Элемент добавлен в контейнер или удален из него |
| List | ActionEvent | Пользователь выполнил двойной щелчок мыши на элементе списка |
| | ItemEvent | Пользователь выбрал элемент списка или отменил выбор |
| MenuItem | ActionEvent | Пользователь выбрал пункт меню |
| Scrollbar | AdjustmentEvent | Пользователь осуществил прокрутку |
| TextComponent | TextEvent | Пользователь внес изменения в текст элемента |
| TextField | ActionEvent | Пользователь закончил редактирование текста элемента |
| Window | WindowEvent | Окно было открыто, закрыто, представлено в виде пиктограммы, восстановлено или требует восстановления |

## 8.1. Рисование "каракулей" в Java

Классический апплет, в котором используется модель обработки событий Java. В этом примере реализованы интерфейсы MouseListener и MouseMotionListener, регистрирующие себя с помощью своих же методов addMouseListener() и addMouseMotionListener().

```
/* <applet code = "Scribble2" width=200 height=200>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Scribble2 extends Applet implements
MouseListener, MouseMotionListener {
   private int last_x, last_y;

   public void init() {
      // Сообщает данному апплету о том, какие объекты
      // классов MouseListener и MouseMotionListener он должен оповещать
      // о событиях, связанных с мышью и ее перемещением.
      // Поскольку интерфейс реализуется в самом апплете,
      // при этом будут вызываться методы апплета.
      this.addMouseListener(this) ;
      this.addMouseMotionListener(this);
   }

   // Метод интерфейса MouseListener. Вызывается при нажатии
   // пользователем кнопки мыши.
   public void mousePressed(MouseEvent e) {
      last_x = e.getX();
      last_y = e.getY();
   }

   // Метод интерфейса MouseMotionListener. Вызывается при
   // перемещении мыши с нажатой кнопкой.
   public void mouseDragged(MouseEvent e) {
      Graphics g = this.getGraphics();
      int x = e.getX(), y = e.getY();
      g.drawLine(last_x, last_y, x, y);
      last_x = x; last_y = y;
   }

   // Другие, не используемые методы интерфейса MouseListener.
   public void mouseReleased(MouseEvent e) {;}
   public void mouseClicked(MouseEvent e) {;}
   public void mouseEntered(MouseEvent e) {;}
   public void mouseExited(MouseEvent e) {;}
```

```
   // Другой метод интерфейса MouseMotionListener.
   public void mouseMoved(MouseEvent e) {;}
}
```

Экран вроде бы пустой - но на нем можно рисовать:

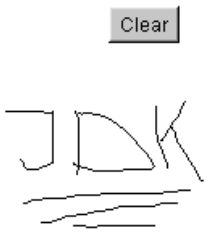

## 8.2. Рисование "каракулей" с использованием встроенных классов

Модель обработки событий Java разработана с учетом того, чтобы хорошо сочетаться с другой особенностью Java: встроенными классами. В следующем примере показано, как изменится данный апплет, если слушатели событий будут реализованы в виде анонимных встроенных классов. Обратите внимание на компактность данного варианта программы. Новая особенность, добавленная в апплет - кнопка Clear. Для этой кнопки зарегистрирован объект ActionListener, а сама она выполняет очистку экрана при наступлении соответствующего события.

```
/* <applet code = "Scribble3" width=200 height=200>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Scribble3 extends Applet {
   int last_x, last_y;

   public void init() {
      // Определяет, создает и регистрирует объект MouseListener.
      this.addMouseListener(new MouseAdapter() {
         public void mousePressed(MouseEvent e) {
            last_x = e.getX(); last_y = e.getY();
         }
      });

      // Определяет, создает и регистрирует объект MouseMotionListener.
      this.addMouseMotionListener(new MouseMotionAdapter() {
         public void mouseDragged(MouseEvent e) {
            Graphics g = getGraphics();
            int x = e.getX(), y= e.getY();
            g.setColor(Color.black);
            g.drawLine(last_x, last_y, x, y);
            last_x = x; last_y = y;
         }
```

```
        });

        // Создает кнопку Clear.
        Button b = new Button("Clear");
        // Определяет, создает и регистрирует объект слушателя
        // для обработки события, связанного с нажатием кнопки.
        b.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              // стирание каракулей
              Graphics g = getGraphics();
              g.setColor(getBackground());
              g.fillRect(0, 0, getSize().width, getSize().height);
           }
        });

        // Добавляет кнопку в апплет.
        this.add(b);
    }
}
```

Теперь апплет выглядит так:




Обратите внимание, что в этот пример порождает 3 вспомогательных класса: Scribble3$1,2,3.

# 9. Заключение

В этой главе изучается построение графического интерфейса пользователя (GUI) с помощью Java, в которой для этой цели предназначена библиотека AWT.

Рассмотрение начинается с апплетов, небольших программ, которые предназначены для работы в браузерах как небольшие части HTML-страниц. Во-первых, необходимо использовать специальный тег, чтобы разместить апплет на странице. В частности, можно указывать специальные параметры, чтобы апплет можно было настраивать без перекомпиляции кода. Во-вторых, рассматриваются этапы жизненного цикла апплета, который отличается от цикла обычного приложения, которое запускается методом main. Наконец, рассматриваются способы рисования в Java – абстрактный класс Graphics, работа с цветами, шрифтами.

Затем описываются стандартные компоненты AWT, которые иерархически упорядочены в дерево наследования с классом Component в вершине. Важным его наследником является класс Container, который может хранить набор компонент. Прямые наследники Component

составляют набор управляющих элементов («контролов» от англ. controls), а наследники Container – набор контейнеров для группировки и расположения компонент. Для упрощения размещения отдельных элементов пользовательского интерфейса применяются менеджеры компоновки (Layout managers).

Один из наследников Container – класс Window, который представляет собой самостоятельное окно в многооконной операционной системе. Два его наследника – Dialog и Frame. Для работы с файлами определен наследник Dialog – FileDialog.

Для построения меню используется свое небольшое дерево наследования с MenuComponent в качестве вершины.

Наконец, излагаются принципы модели событий от пользователя, позволяющей обрабатывать все действия, которые производит клиент, работая с программой. 11 событий и соответствующих им интерфейсов предоставляют все необходимое для написания полноценной GUI-программы.

# 10. Контрольные вопросы

11-1. От какого класса наследуется класс Applet?

a.) java.awt.Panel

11-2. Может ли быть дважды вызван метод init у апплета? Метод start?

a.) Метод init вызывается только один раз при конструировании апплета. Метод start может быть вызван многократно, если пользователь покидал и возвращался на страницу.

11-3. Чем различаются методы paint, update, repaint?

a.) paint определяет внешний вид компоненты, в нем описывается отрисовка всех внешних элементов

update сначала закрашивает всю компоненту фоновым (background) цветом, а затем вызывает paint

repaint не перерисовывает компоненту напрямую, он инициирует вызов метода update через указанный промежуток времени

11-4. Как создать объект класса Color, описывающий чистый синий цвет?

a.) new Color(0,0,255). Также можно воспользоваться константой Color.blue

11-5. Какими параметрами в Java характеризуется шрифт?

a.) Имя семейства шрифта, размер (в пунктах), стиль (обычный, жирный, наклонный).

11-6. Для чего нужен класс FontMetrics?

a.) Поскольку размер шрифта задается в пунктах, а отображение текста делается с помощью шрифтов, которые поддерживаются операционной системой, необходима специальная утилита для вычисления размера

шрифта в пикселах. Это может потребоваться для точного позиционирования текста в компоненте.

Класс FontMetrics предоставляет набор методов для получения отдельных параметров шрифта, таких как ширина слова, высота шрифта и другие.

11-7. Напишите класс-компоненту, у которого по центру рисуется квадрат размерами 10x10.

a.)
```
public class SquareComponent extends Canvas {
   public void paint(Graphics g) {
      g.drawRect(getWidth()-5, getHeight()-5, 10, 10);
   }
}
```

11-8. Как в AWT создаются компоненты чекбокс (check-box)? Радио-кнопка (radio-button)?

a.) чек бокс порождается компонентой Checkbox:

```
Checkbox chbox = new Checkbox("название");
```

Для создания радио-кнопок необходимо связать несколько компонент Checkbox с помощью класса CheckboxGroup:

```
CheckboxGroup group = new CheckboxGroup();
Checkbox  rb1 = new Checkbox("режим 1",group, true);
Checkbox  rb2 = new Checkbox("режим 2",group, false);
```

11-9. В чем разница между компонентами List и Choice?

a.) List отображает несколько элементов списка сразу, а Choise только один. Так же в List может быть выбрано несколько элементов (если установлено свойство multiselect), а в Choice только один.

11-10. Для чего нужны менеджеры компоновки? Исходя из каких параметров они выполняют свою работу?

a.) Для автоматического расположения компонент внутри контейнера. Менеджер компоновки может установить размер и местоположение компоненты, и в дальнейшем не придется проводить дополнительную работу, если изменился размер окна или компонент, или их количество.

При компоновке учитываются следующие параметры:

- размер контейнера
- количество компонент и порядок их следования
- начальный размер и положение компонент
- constraints, устанавливаемый при добавлении компоненты
- дополнительные свойства самого менеджера (отступы между компонентами и т.п.)

11-11. В чем разница между Dialog и Frame?

a.) Основное различие заключается в том, что Frame – самостоятельное окно, а Dialog всегда привязан к Frame. Только Dialog обладает свойством модальности. Только Frame может иметь главное меню. Dialog нельзя минимизировать или максимизировать.

11-12. Какие действия необходимо произвести, чтобы создать компонент и подписаться на событие, которое он генерирует?

a.) Сначала создается сама компонента. Затем создается класс-слушатель, реализующий соответствующий Listener-интерфейс, который будет реагировать на появление события. Наконец, вызывается метод add<…>Listener, который регистрирует слушателя.

11-13. Как узнать, какие события генерирует стандартный компонент?

a.) Необходимо посмотреть, каким методы add<…>Listener в нем объявлены или унаследованы от родительского класса.

# Программирование на Java

## Лекция 12. Потоки выполнения. Синхронизация

### 20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>




Аннотация

Эта лекция завершает рассмотрение ключевых особенностей Java. Последняя тема раскрывает особенности создания многопоточных приложений – такая возможности присутствует в языке начиная с самых первых версий.

Первый вопрос – как на много- и, самое интересное, однопроцессорных машинах выполняются несколько потоков одновремнно, и для чего они нужны в програме. Затем описываются классы, необходимые для создания, запуска и управления потоками в Java. При одновремнной работе с данными из нескольких мест возникает проблема синхронного доступа, блокировок и, как следствие, взаимных блокировок. Изучаются все механизмы, присутствующие в языке, для корректной организации такой логики работы.

# Оглавление

# Лекция 12. Потоки выполнения. Синхронизация

## Содержание лекции.



## 1. Введение

До сих пор во всех рассматриваемых примерах подразумевалось, что в один момент времени исполняется лишь одно выражение, или действие. Однако начиная с самых первых версий, виртуальные машины Java поддерживают многопоточность, т.е. поддержку нескольких потоков исполнения (threads) одновременно.

В данной главе сначала рассматриваются преимущества такого подхода, способы реализации и возможные недостатки.

Затем описываются базовые классы Java, которые позволяют запускать и управлять потоками исполнения.

При одновременном обращении нескольких потоков к одним и тем же данным может возникнуть ситуация, когда результат программы будет зависеть от случайных факторов, таких как временное чередование исполнения операций несколькими потоками. В такой ситуации становятся необходимым механизмы синхронизации, обеспечивающие последовательный, или монопольный, доступ. В Java этой цели служит ключевое слово

synchronized. Предварительно будет рассмотрен подход к организации хранения данных в виртуальной машине.

В заключение рассматриваются методы wait(), notify(), notifyAll() класса Object.

# 2. Многопоточная архитектура

Не претендуя на полноту изложения, рассмотрим общее устройство многопоточной архитектуры, ее достоинства и недостатки.

Реализацию многопоточной архитектуры проще всего представить себе для системы, в которой есть несколько центральных вычислительных процессоров. В этом случае для каждого из них можно выделить задачу, которую он будет выполнять. В результате несколько задач будут обслуживаться одновременно.

Однако возникает вопрос- каким же тогда образом обеспечивается многопоточность в системах с одним центральным процессором, который в принципе выполняет лишь одно вычисление в один момент времени? В таких системах применяется процедура квантования времени (time-slicing). Время разделяется на набольшие интервалы. Перед началом каждого интервала принимается решение, какой именно поток выполнения будет отрабатываться на протяжении этого кванта времени. За счет такого частого переключения между задачами эмулируется многопоточная архитектура.

На самом деле, как правило и для многопроцессорных систем применяется процедура квантования времени. Дело в том, что даже в мощных серверах приложений процессоров не так много (редко бывает больше десяти), а потоков исполнения запускается как правило гораздо больше. Например, операционная система Windows без единого стартованного приложения инициализирует десятки, а то и сотни потоков. Квантование времени позволяет упростить управление выполнением задач на всех процессорах.

Теперь перейдем к вопросу о преимуществах - зачем вообще может потребоваться более одного потока выполнения?

Среди начинающих программистов существует мнение, что многопоточные программы работают быстрее. Рассмотрев способ реализации многопоточности, можно утверждать, что такие программы работают на самом деле медленнее. Действительно, для переключения между задачами на каждом интервале требуется дополнительное время, а ведь они (переключения) происходят довольно часто. Если бы процессор, не отвлекаясь, выполнял задачи последовательно, одну за другой, он закончил бы их заметно быстрее. Стало быть, преимущества заключаются не в этом.

Первый тип приложений, который выигрывает от поддержки многопоточности, предназначен для задач, где действительно требуется выполнять несколько действий одновременно. Например, будет вполне обоснованно ожидать, что сервер общего пользования будет обслуживать несколько клиентов одновременно. Можно легко представить себе пример из сферы обслуживания, когда есть несколько потоков клиентов, и желательно обслуживать их все одновременно.

Другой пример - активные игры, или подобные приложения. Необходимо одновременно опрашивать клавиатуру и другие устройства ввода, чтобы реагировать на действия пользователя. В то же время одновременно необходимо рассчитывать и перерисовывать изменяющееся состояние игрового поля.

Понятно, что в случае отсутствия поддержки многопоточности для реализации подобных приложений потребовалось бы реализовывать квантования времени вручную. Условно говоря - одну секунду проверять состояние клавиатуры, а следующую - пересчитывать и перерисовывать игровое поле. Если сравнить две реализации time-slicing, одну - на низком уровне, выполненную средствами, как правило, операционной системы, другую - выполняемую вручную, на языке высокого уровня, мало подходящего для таких задач, то становится понятным первое и, возможно, главное преимущество многопоточности. Она обеспечивает наиболее эффективную реализацию процедуры квантования времени, существенно облегчая и укорачивая процесс разработки приложения. Код переключения между задачами на Java выглядел бы гораздо более громоздко, чем независимое описание действий для каждого потока независимо.

Следующее преимущество проистекает из того, что компьютер состоит не только из одного или нескольких процессоров. Вычислительное устройство - лишь одно из ресурсов, необходимых для выполнения задач. Всегда есть оперативная память, дисковая подсистема, сетевые подключения, периферия (принтер и т.п.) и другие. Предположим, пользователю требуется распечатать большой документ и скачать большой файл из сети. Очевидно, что обе задачи требуют совсем небольшого участия процессора, а основные необходимые ресурсы, которые будут использоваться на пределе возможностей, у них разные - сетевое подключение и принтер. Значит, если выполнять задачи одновременно, но замедление от организации квантования времени будет незначительным, процессор легко справится с обслуживанием обеих задач. В то же время, если каждая задача по отдельности занимала, скажем, 2 часа, то вполне вероятно, что и при одновременном исполнении потребуется не более тех же 2 часов, а сделано при этом будет гораздо больше.

Если же задачи в основном загружают процессор (например, математические расчеты), то их одновременное исполнение займет в лучшем случае столько же времени, что и последовательное, а скорее всего и больше.

Третье преимущество появляется из-за возможности более гибко управлять выполнением задач. Предположим, пользователь системы, не поддерживающей многопоточность, решил скачать большой файл из сети или произвести сложное вычисление, что занимает, скажем, 2 часа. Запустив задачу на выполнение, он может внезапно обнаружить, что ему нужен не этот, а какой-нибудь другой файл (или вычисление с другими начальными параметрами). Однако если приложение занимается только работой с сетью (вычислениями) и не реагирует на действия пользователя (не обрабатываются данные с устройств ввода, таких как клавиатура или мышь), то он не сможет быстро исправить свою ошибку. Получается, что процессор выполняет большее количество вычислений, но при этом приносит гораздо меньше пользы своему пользователю.

Процедура квантования времени поддерживает приоритеты (priority) задач. В Java приоритет представляется целым числом. Чем больше число, тем выше приоритет. Строгих правил работы с приоритетами нет, каждая реализация может вести себя по-разному на разных платформах. Однако есть общее правило - поток с более высоким приоритетом будет получать большее количество квантов времени на исполнение, и таким образом сможет быстрее выполнять свои действия и реагировать на поступающие данные.

В описанном примере представляется разумным запустить дополнительный поток, отвечающий за взаимодействие с пользователем. Ему можно поставить высокий приоритет, так как в случае бездействия пользователя этот поток практически не будет занимать

ресурсы машины. В случае же активности пользователя необходимо как можно быстрее произвести необходимые действия, чтобы обеспечить максимальную эффективность работы пользователя.

Рассмотрим здесь же еще одно свойство потоков. Раньше, когда рассматривались однопоточные приложения, завершение вычислений однозначно приводило к завершению выполнения программы. Теперь же приложение должно работать видимо до тех пор, пока есть хоть один действующий поток исполнения. В то же время часто бывают нужны обслуживающие потоки, которые не имеют никакого смысла, если они остаются в системе одни. Например, автоматический сборщик мусора в Java запускается в виде фонового (низкоприоритетного) процесса. Его задача - отслеживать объекты, которые уже не используются другими потоками, и затем уничтожать их, освобождая оперативную память. Понятно, что работа одного потока garbage collector'a не имеет никакого смысла.

Такие обслуживающие потоки называют демонами (daemon), это свойство можно установить любому потоку. В итоге приложение выполняется до тех пор, пока есть хотя бы один поток не-демон.

Рассмотрим, как потоки реализованы в Java.

# 3. Базовые классы для работы с потоками

## 3.1. Класс Thread

Поток выполнения в Java представляется экземпляром класса Thread. Для того, чтобы написать свой поток исполнения необходимо наследоваться от этого класса и переопределить метод run(). Например,

```
public class MyThread extends Thread {
  public void run() {
   // некоторое долгое действие, вычисление
   long sum=0;
   for (int i=0; i<1000; i++) {
    sum+=i;
   }
   System.out.println(sum);
  }
}
```

Метод run() содержит действия, которые должны исполнятся в новом потоке исполнения. Чтобы запустить его, необходимо создать экземпляр класса-наследника, и вызвать унаследованный метод start(), который сообщает виртуальной машине, что необходимо запустить новый поток исполнения и начать в нем исполнять метод run().

```
MyThread t = new MyThread();
t.start();
```

В результате чего на консоли появится результат:

```
499500
```

Когда метод run() завершен (в частности, встретилось выражение return) поток выполнения останавливается. Однако, ничто не препятствует записи бесконечного цикла в этом методе. В результате поток не прервет своего исполнения, и будет остановлен только при завершении работы всего приложения.

## 3.2. Интерфейс Runnable

Описанный подход обладает одним недостатком. Поскольку в Java отсутствует множественное наследование, требование наследоваться от Thread может привести к конфликту. Если еще раз посмотреть на приведенный выше пример, то станет понято, что наследование производилось только с целью переопределения метода run(). Поэтому предлагается более простой способ создать свой поток исполнения. Достаточно реализовать интерфейс Runnable, в котором объявлен только один метод - уже знакомый void run(). Запишем пример, приведенный выше, с помощью этого интерфейса:

```
public class MyRunnable implements Runnable {
  public void run() {
   // некоторое долгое действие, вычисление
   long sum=0;
   for (int i=0; i<1000; i++) {
    sum+=i;
   }
   System.out.println(sum);
  }
}
```

Также незначительно меняется процедура запуска потока:

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Если раньше объект, представляющий сам поток выполнения, и объект с методом run(), содержащим полезную функциональность, были объединены в одном экземпляре класса MyThread, то теперь они разделены. Какой из двух подходов удобней, можно свободно решать в каждом конкретном случае.

Подчеркнем, что Runnable не является полной заменой классу Thread, поскольку создание и запуск самого потока исполнения возможно только через метод Thread.start().

## 3.3. Работа с приоритетами

Рассмотрим, как в Java потокам можно назначать приоритеты. Для этого в классе Thread существуют методы getPriority() и setPriority(), а также объявлены три константы:

```
MIN_PRIORITY
MAX_PRIORITY
NORM_PRIORITY
```

Из названия очевидно, что их значения описывают минимальное, максимально и нормальное (по умолчанию) значения приоритета.

Рассмотрим следующий пример:

```
public class ThreadTest implements Runnable {

 public void run() {
  double calc;
  for (int i=0; i<50000; i++) {
   calc=Math.sin(i*i);
   if (i%10000==0) {
    System.out.println(getName()+" counts " + i/10000);
   }
  }
 }

public String getName() {
  return Thread.currentThread().getName();
}

public static void main(String s[]) {
 // Подготовка потоков
  Thread t[] = new Thread[3];
  for (int i=0; i<t.length; i++) {
   t[i]=new Thread(new ThreadTest(), "Thread "+i);
  }

// Запуск потоков
  for (int i=0; i<t.length; i++) {
   t[i].start();
   System.out.println(t[i].getName()+" started");
  }
 }
}
```

В примере используется несколько новых методов класса Thread:

- getName()

  Обратите внимание, что конструктору класса Thread передается два параметра. К реализации Runnable добавляется строка. Это имя потока, которое используется только для упрощения его идентификации. Имена нескольких потоков могут совпадать. Если его не задать, то Java генерирует простую строку вида "Thread-" и номер потока (вычисляется простым счетчиком) Именно это имя возвращается методом getName(). Его можно сменить с помощью метода setName().

- currentThread()

  Этот статический метод позволяет в любом месте кода получить ссылку на объект класса Thread, представляющий текущий поток исполнения.

Результат работы такой программы будет иметь следующий вид:

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 0 counts 0
Thread 1 counts 0
Thread 2 counts 0
Thread 0 counts 1
Thread 1 counts 1
Thread 2 counts 1
Thread 0 counts 2
Thread 2 counts 2
Thread 1 counts 2
Thread 2 counts 3
Thread 0 counts 3
Thread 1 counts 3
Thread 2 counts 4
Thread 0 counts 4
Thread 1 counts 4
```

Можно видеть, что все три потока были запущены один за другим, и начали проводить вычисления. Видно также, что потоки исполняются без определенного порядка, случайным образом. Тем не менее, в среднем они движутся с одной скоростью, никто не отстает и не догоняет.

Введем в программу работу с приоритетам, расставим разные значения для разных потоков и посмотрим, как это скажется на выполнении. Изменяется только метод main()

```
public static void main(String s[]) {
// Подготовка потоков
 Thread t[] = new Thread[3];
 for (int i=0; i<t.length; i++) {
  t[i]=new Thread(new ThreadTest(), "Thread "+i);
  t[i].setPriority(Thread.MIN_PRIORITY +
(Thread.MAX_PRIORITY-Thread.MIN_PRIORITY)/t.length*i);
 }

// Запуск потоков
 for (int i=0; i<t.length; i++) {
  t[i].start();
  System.out.println(t[i].getName()+" started");
 }
}
```

Формула вычисления приоритетов позволяет равномерно распределить все допустимые значения для всех запускаемые потоки. На самом деле, константа минимального приоритета имеет значение 1, максимального - 10, нормального - 5. Так что в простых программах можно явно пользоваться этими величинами и указывать в качестве, например, пониженного приоритета значение 3.

Результатом работы будет:

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 2 counts 0
Thread 2 counts 1
Thread 2 counts 2
Thread 2 counts 3
Thread 2 counts 4
Thread 0 counts 0
Thread 1 counts 0
Thread 1 counts 1
Thread 1 counts 2
Thread 1 counts 3
Thread 1 counts 4
Thread 0 counts 1
Thread 0 counts 2
Thread 0 counts 3
Thread 0 counts 4
```

Потоки, как и раньше, стартуют последовательно. Но затем сразу видно, что чем выше приоритет, тем быстрее отрабатывает поток. Тем не менее весьма показательно, что поток с минимальным приоритетом (Thread 0) тем не менее получил возможность выполнить одно действие раньше, чем отработал поток с более высоким приоритетом (Thread 1). Это говорит о том, что приоритеты не делают систему однопоточной, выполняющей единовременно лишь один поток с наивысшим приоритетом. Напротив, приоритеты позволяют одновременно работать над несколькими задачами с учетом их важности.

Если увеличить параметры метода (выполнять 500.000 вычислений, а не 50.000, и выводить сообщение каждое 1000 вычисление, а не 10.000), то можно будет наглядно увидеть, что все три потока имеют возможность выполнять свои действия одновременно, просто более высокий приоритет позволяет выполнять их чаще.

## 3.4. Демон-потоки

Демон-потоки позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них. Для работы с этим свойством существуют методы setDaemon() и isDaemon().

Рассмотрим следующий пример:

```
public class ThreadTest implements Runnable {
```

```
 // Отдельная группа, в которой будут
 // находится все потоки ThreadTest
public final static ThreadGroup GROUP =
 new ThreadGroup("Daemon demo");

 // Стартовое значение,
 // указывается при создании объекта
private int start;

public ThreadTest(int s) {
   start = (s%2==0)? s: s+1;
  new Thread(GROUP, this, "Thread "+start).start();
 }

 public void run() {
  // Начинаем обратный отсчет
  for (int i=start; i>0; i--) {
   try {
    Thread.sleep(300);
   } catch (InterruptedException e) {}

   // По достижению середины порождаем новый
   // поток с половинным начальным значением
   if (start>2 && i==start/2) {
    new ThreadTest(i);
   }
  }
 }

public static void main(String s[]) {
   new ThreadTest(16);
   new DaemonDemo();
 }
}

public class DaemonDemo extends Thread {
 public DaemonDemo() {
  super("Daemon demo thread");
  setDaemon(true);
  start();
 }

 public void run() {
  Thread threads[]=new Thread[10];
  while (true) {
   // Получаем набор всех потоков из
   // тестовой группы
   int count=ThreadTest.GROUP.activeCount();
   if (threads.length<count) threads =
```

```
new Thread[count+10];
   count=ThreadTest.GROUP.enumerate(threads);

   // Распечатываем имя каждого потока
   for (int i=0; i<count; i++) {
    System.out.print(threads[i].getName()+", ");
   }
   System.out.println();
   try {
    Thread.sleep(300);
   } catch (InterruptedException e) {}
  }
 }
}
```

В этом примере происходит следующее. Потоки ThreadTest имеют некоторое стартовое значение, которое передается им при создании. В методе run() это значение последовательно уменьшается. При достижении половины от начальной величины порождается новый поток с вдвое меньшим начальным значением. По исчерпании счетчика поток останавливается. Метод main() порождает первый поток со стартовым значением 16. В ходе программы, значит, будут дополнительно порождены потоки со значениями 8, 4, 2.

За этим процессом наблюдает демон-поток DaemonDemo. Этот поток регулярно получает список всех существующих потоков ThreadTest и распечатывает их имена для удобства наблюдения.

Результатом программы будет:

```
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8,
Thread 16, Thread 8, Thread 4,
Thread 16, Thread 8, Thread 4,
Thread 8, Thread 4,
Thread 4, Thread 2,
Thread 2,
```

Несмотря на то, что демон-поток никогда не выходит из метода run(), виртуальная машина прекращает работу как только все не-демон-потоки завершаются.

В примере использовалось несколько дополнительных классов и методов, которые еще не были рассмотрены:

- класс ThreadGroup

  Все потоки находятся в группах, представляемых экземплярами класса ThreadGroup. Группа указывается при создании потока. Если группа не была указана, то поток помещается в ту же группу, где находится поток, породивший его.

  Методы activeCount() и enumerate() возвращают количество и полный список соответственно всех потоков в группе.

- sleep()

  Этот статический метод класса Thread приостанавливает выполнение текущего потока на указанное количество миллисекунд. Обратите внимание, что метод требует обработки исключения InterruptedException. Он связан с возможностью вернуть к работе метод, который приостановил свою работу. Например, если поток находится в выполнении метода sleep(), то есть ничего не исполняет на протяжении указанного периода времени, то его можно вывести из этого состояния, вызвав метод interrupt() из другого потока выполнения. В результате, метод sleep() прервется исключением InterruptedException.

Кроме метода sleep() существует еще один статический метод yield() без параметров. Когда поток вызывает его, он временно приостанавливает свою работу и позволяет отработать другим потокам. Один из методов обязательно должен применяться внутри бесконечных циклов ожидания, иначе есть риск, что такой ничего не делающий поток серьезно затормозит работу остальных потоков.

# 4. Синхронизация

При многопоточной архитектуре приложения возможны ситуации, когда несколько потоков будут одновременно работать с одними и теми же данными, используя их значения и присваивая новые. В таком случае результат работы программы становится невозможным предопределить, глядя только на исходный код. Финальные значения переменных будут зависеть от случайных факторов, исходя из того, какой поток какое действие успел сделать первым или последним.

Рассмотрим пример:

```
public class ThreadTest  {

 private int a=1, b=2;
 public void one() {
  a=b;
 }
 public void two() {
  b=a;
 }

 public static void main(String s[]) {
  int a11=0, a22=0, a12=0;
```

```
  for (int i=0; i<1000; i++) {
   final ThreadTest o = new ThreadTest();

   // Запускаем первый поток, который
   // вызывает один метод
new Thread() {
    public void run() {
     o.one();
    }
   }.start();

   // Запускаем второй поток, который
   // вызывает второй метод
   new Thread() {
    public void run() {
     o.two();
    }
   }.start();

   // даем время отработать потокам
try {
    Thread.sleep(100);
   } catch (InterruptedException e) {}

   // анализируем финальные значения
if (o.a==1 && o.b==1) a11++;
   if (o.a==2 && o.b==2) a22++;
   if (o.a!=o.b) a12++;
  }
  System.out.println(a11+" "+a22+" "+a12);
 }
}
```

В этом примере два потока исполнения одновременно обращаются к одному и тому же объекту, вызывая у него два разных метода one() и two(). Эти методы пытаются приравнять два поля класса a и b друг другу, но в разном порядке. Учитывая, что исходные значения полей равны 1 и 2 соответственно, можно было бы ожидать, что после того, как потоки завершат свою работу, поля будут иметь одинаковое значение. Однако понять, какое из двух возможных значений они примут, уже невозможно. Посмотрим на результат программы:

```
135 864 1
```

Первое число показывает, сколько раз из тысячи обе переменные приняли значение 1. Второе число соответствует значению 2. Такое сильное преобладание одного из значений обусловлено последовательностью запусков потоков. Если ее изменить, то и количества случаев с 1 и 2 также меняются местами. Третье же число сообщает, что на тысячу случаев произошел один, когда поля вообще обменялись значениями!

При количестве итераций, равном 10.000, были получены следующие данные, которые подтверждают сделанные выводы:

```
494 9498 8
```

А если убрать задержку перед анализом результатов, то получаемые данные радикально меняются:

```
0 3 997
```

Видимо, потоки просто не успевают отработать.

Итак, наглядно показано, сколь сильно и непредсказуемо может меняться результат работы одной и той же программы, применяющей многопоточную архитектуру. Необходимо учитывать, что в приведенном простом примере задержки создавались вручную методом Thread.sleep(). В реальных сложных системах задержки могут возникать в местах проведения сложных операций, их длины непредсказуемы, и оценить их последствия невозможно.

Для более глубокого понимания принципов многопоточной работы в Java рассмотрим организацию памяти в виртуальной машине для нескольких потоков.

## 4.1. Хранение переменных в памяти

Виртуальная машина поддерживает основное хранилище данных (main storage), в котором сохраняются значения всех переменных и которое используется всеми потоками. Под переменными здесь понимаются поля объектов и классов, а также элементы массивов. Что касается локальных переменных и параметров методов, их значения не могут быть доступы другим потокам, поэтому они не представляют интереса.

Для каждого потока создается его собственная рабочая память (working memory), в которую копируются значения всех переменных перед использованием.

Рассмотрим основные операции, доступные для потоков при работе с памятью:

- use - чтение значения переменной из рабочей памяти потока
- assign - запись значения переменной в рабочую память потока
- read - получение значения переменной из основного хранилища
- load - сохранение значения переменной, прочитанного из основного хранилища, в рабочей памяти
- store - передача значения переменной из рабочей памяти в основное хранилище для будущего сохранения
- write - сохраняет в основном хранилище значение переменной, переданной командой store

Подчеркнем, что перечисленные команды не являются методами каких-либо классов, они не доступны программисту. Сама виртуальная машина использует их для обеспечения корректной работы потоков исполнения.

Поток, работая с переменной, регулярно применяет команды use и assign для использования ее существующего значения и присвоения нового. Кроме этого, должны осуществляться действия по передачи значений из/в основное хранилище. Они выполняются в два этапа.

При получении данных сначала основное хранилище считывает значение командой read, а затем поток сохраняет результат в своей рабочей памяти командой load. Эта пара команд всегда выполняется вместе именно в таком порядке, т.е. нельзя выполнить одну, не выполнив другую. При отправлении данных сначала поток считывает значение из рабочей памяти командой store, а затем основное хранилище сохраняет его командой write. Эта пара команд также всегда выполняется вместе именно в таком порядке, т.е. нельзя выполнить одну, не выполнив другую.

Набор этих правил составлялся с тем, чтобы операции с памятью были достаточно строги для точного анализа их результатов, а с другой стороны правила должны оставлять достаточное пространство для различных технологий оптимизаций (регистры, очереди, кэш и т.д.).

Последовательность команд подчиняется следующим правилам:

- все действия, выполняемые одним потоком строго упорядочены, т.е. выполняются одно за другим
- все действия, выполняемые с одной переменной в основном хранилище памяти, строго упорядочены, т.е. следуют одно за другим

За исключением некоторых дополнительных очевидных правил больше никаких ограничений нет. Например, если поток изменил значение сначала одной, а затем другой переменной, то они эти изменения могут быть переданы в основное хранилище в переставленном порядке.

Поток создается с чистой рабочей памятью и должен загрузить все необходимые переменные из основного хранилища перед использованием. Любая переменная сначала создается в основном хранилище и лишь затем копируется в рабочую память потоков, которые будут ее использовать.

Таким образом, потоки никогда не взаимодействуют друг с другом на прямую, только через главное хранилище.

## 4.2. Модификатор volatile

При объявлении полей объектов и классов может быть указан модификатор volatile. Он устанавливает более строгие правила работы со значениями переменных.

Если поток собирается выполнить команду use для volatile переменной, то требуется, чтобы предыдущим действием над этой переменной было обязательно load, и наоборот - операция load может выполняться только перед use. Таким образом, переменная и главное хранилище всегда имеют самое последнее значение этой переменной.

Аналогично, если поток собирается выполнить команду store для volatile переменной, то требуется, чтобы предыдущим действием над этой переменной было обязательно assign, и наоборот - операция assign может выполняться только если следующей будет store. Таким образом, переменная и главное хранилище всегда имеют самое последнее значение этой переменной.

Наконец, если проводятся операции над несколькими volatile переменными, то передача соответствующих изменений в основное хранилище должно проводится строго в том же порядке.

При работе с обычными переменными компилятор имеет больше пространства для маневра. Например, при благоприятных обстоятельствах может оказаться возможным предсказать значение переменной, заранее вычислить и сохранить его, а затем в нужный момент использовать уже готовым.

Нужно обратить внимание на два 64-разрядных типа double и long. Поскольку многие платформы поддерживают лишь 32-битную память, величины этих типов рассматриваются как две переменные, и все описанные действия делаются независимо для двух половинок таких значений. Конечно, если производитель виртуальной машины считает возможным, он может обеспечить атомарность операций и над этими типами. Для volatile переменных это является обязательным требованием.

## 4.3. Блокировки

В основном хранилище для каждого объекта поддерживается блокировка (lock), над которой можно произвести два действия - установить (lock) и снять (unlock). Только один поток в один момент времени может установить блокировку на некоторый объект. Если до того, как этот поток выполнит операцию unlock, другой поток попытается установить блокировку, его выполнение будет приостановлено до тех пор, пока первый поток не отпустит ее.

Операции lock и unlock накладывают жесткое ограничение на работу с переменными в рабочей памяти потока. После успешно выполненного lock, рабочая память очищается, и все переменные необходимо заново считывать из основного хранилища. Аналогично, перед операцией unlock необходимо все переменные сохранить в основном хранилище.

Важно подчеркнуть, что блокировка является чем-то вроде флага. Если блокировка на объект установлена, это не означает, что этим объектом нельзя пользоваться, что его поля и методы становятся недоступными - это не так. Единственное действие, которое становится невозможным - установить эту же блокировку другому потоку до тех пор, пока первый поток не выполнит unlock.

В Java-программе для того, чтобы воспользоваться механизмом блокировок, существует ключевое слово synchronized. Оно может быть применено в двух вариантах - для объявления synchronized-блока и как модификатор метода. В обоих случаях действие его примерно одинаковое.

Synchronized-блок записывается следующим образом:

```
synchronized (ref) {
  ...
}
```

Прежде чем начать выполнять действия, описанные в этом блоке, поток обязан установить блокировку на объект, на который ссылается переменная ref (поэтому она не может быть null). Если другой поток уже установил блокировку на этот объект, то выполнение первого потока приостанавливается до тех пор, пока не удастся выполнить операцию lock.

После этого блок выполняется. В случае успешного либо не успешного завершения исполнения, производится операция unlock, чтобы освободить объект для других потоков.

Рассмотрим пример:

```
public class ThreadTest  implements Runnable {

 private static ThreadTest shared = new ThreadTest();

 public void process() {
  for (int i=0; i<3; i++) {
   System.out.println (Thread.currentThread().
getName()+" "+i);
   Thread.yield();
  }
 }

 public void run() {
  shared.process();
 }

 public static void main(String s[]) {
  for (int i=0; i<3; i++) {
   new Thread(new ThreadTest(),
"Thread-"+i).start();
  }
 }
}
```

В этом простом примере три потока вызывают метод у одного объекта, чтобы тот распечатал три значения. Результатом будет:

```
Thread-0 0
Thread-1 0
Thread-2 0
Thread-0 1
Thread-2 1
Thread-0 2
Thread-1 1
Thread-2 2
Thread-1 2
```

То есть, все потоки одновременно работают с одним методом одного объекта. Заключим обращение к методу в synchronized-блок:

```
  public void run() {
synchronized (shared) {
shared.process();
}
}
```

Теперь результат будет строго упорядочен:

```
Thread-0 0
Thread-0 1
Thread-0 2
Thread-1 0
Thread-1 1
Thread-1 2
Thread-2 0
Thread-2 1
Thread-2 2
```

Synchronized-методы работают аналогичным образом. Прежде чем начать выполнять их, поток пытается заблокировать объект, у которого вызывается метод. После выполнения блокировка снимается. В предыдущем примере аналогичной упорядоченности можно было добиться, если использовать не synchronized-блок, а объявить метод process() синхронизированным.

Также допустимы static synchronized методы. При их вызове блокировка устанавливается на объект класса Class, отвечающего за тип, у которого вызывается этот метод.

При работе с блокировками всегда надо помнить о возможности появления deadlock - взаимных блокировок, которые приводят к зависанию программы. Если один поток заблокировал один ресурс, и пытается заблокировать второй, а другой поток заблокировал второй и пытается заблокировать первый, то такие потоки уже никогда не выйдут из состояния ожидания.

Рассмотрим простейший пример:

```
public class DeadlockDemo {

 // Два объекта-ресурса
 public final static Object one=new Object(), two=new Object();

 public static void main(String s[]) {
  // Создаем два потока, которые будут
  // конкурировать за доступ к объектам one и two
  Thread t1 = new Thread() {
   public void run() {
    // Блокировка первого объекта
    synchronized(one) {
     Thread.yield();
     // Блокировка второго объекта
     synchronized (two) {
      System.out.println("Success!");
     }
    }
   }
  };
  Thread t2 = new Thread() {
```

```
   public void run() {
    // Блокировка второго объекта
    synchronized(two) {
     Thread.yield();
     // Блокировка первого объекта
     synchronized (one) {
      System.out.println("Success!");
     }
    }
   }
  };

  // Запускаем потоки
  t1.start();
  t2.start();
 }
}
```

Если запустить такую программу, то она никогда не закончит свою работу. Обратите внимание, на вызовы метода yield() в каждом потоке. Они гарантируют, что когда один поток выполнил первую блокировку и переходит к следующей, второй поток находится в таком же состоянии. Легко понять, что в результате оба потока "замрут", не смогут продолжить свое выполнение. Первый поток будет ждать освобождение второго объекта, и наоборот. Именно такая ситуация называется "мертвой блокировкой", или deadlock. Если один из потоков успел бы заблокировать оба объекта, то программа успешно бы выполнилась до конца. Однако многопоточная архитектура не дает никаких гарантий, как именно потоки будут выполняться друг относительно друга. Задержки (которые в примере моделируются вызовами yield()) могут возникать из логики программы (необходимость произвести вычисления), действий пользователя (не сразу нажал кнопку "ОК"), занятости ОС (из-за нехватки физической оперативной памяти пришлось воспользоваться виртуальной), значений приоритетов потоков и так далее.

В Java нет никаких средств распознавания или предотвращения ситуаций deadlock. Также нет способа перед вызовом синхронизированного метода узнать, заблокирован ли уже объект другим потоком. Программист сам должен строить работу программы таким образом, чтобы неразрешимые блокировки не возникали. Например, в рассмотренном примере достаточно было организовать блокировки объектов в одном порядке (всегда сначала первый, затем второй), и программа выполнялась бы всегда успешно.

Опасность возникновения взаимных блокировок заставляет с особенным вниманием относиться к работе с потоками. Например, важно помнить, что если у объекта потока был вызван метод sleep(..), то такой поток будет бездействовать определенное время, но при этом все заблокированные ими объекты будут оставаться недоступными для блокировок со стороны других потоков, а это потенциальный deadlock. Такие ситуации крайне сложно выявить тестированием и отладкой, поэтому вопросам синхронизации надо уделять много времени на этапе проектирования.

# 5. Методы wait(), notify(), notifyAll() класса Object

Наконец, переходим к рассмотрению трех методов класса Object, которое завершает описание механизмов поддержки многопоточности в Java.

Каждый объект в Java имеет не только блокировку для synchronized блоков и методов, но и так называемый wait-set, набор потоков исполнения. Любой поток может вызвать метод wait() любого объекта и таким образом попасть в его wait-set. При этом выполнение такого потока приостанавливается до тех пор, пока другой поток не вызовет у точно этого же объекта метод notifyAll(), который пробуждает все потоки из wait-set. Метод notify() пробуждает один, случайно выбранный поток из этого набора.

Однако применение этих методов связано с одним важным ограничением. Любой из них может быть вызван потоком у объекта только после установления блокировки на этот объект. То есть, либо внутри synchronized-блока с ссылкой на этот объект в качестве аргумента, либо обращение к методам должны быть в синхронизированных методах класса самого объекта. Рассмотрим пример:

```
public class WaitThread implements Runnable {
  private Object shared;

  public WaitThread(Object o) {
   shared=o;
}

public void run() {
 synchronized (shared) {
  try {
  shared.wait();
    } catch (InterruptedException e) {}
  System.out.println("after wait");
 }
}

public static void main(String s[]) {
 Object o = new Object();
 WaitThread w = new WaitThread(o);
 new Thread(w).start();
 try {
  Thread.sleep(100);
} catch (InterruptedException e) {} System.out.println("before notify");
 synchronized (o) {
o.notifyAll();
   }
}
}
```

Результатом программы будет:

```
before notify
after wait
```

Обратите внимание, что метод wait(), также как и sleep(), требует обработки InterruptedException, то есть его выполнение также можно прервать методом interrupt().

В заключение рассмотрим более сложный пример для трех потоков:

```
public class ThreadTest implements Runnable {
  final static private Object shared=new Object();

  private int type;
  public ThreadTest(int i) {
   type=i;
  }

public void run() {
 if (type==1 || type==2) {
  synchronized (shared) {
   try {
    shared.wait();
   } catch (InterruptedException e) {}
   System.out.println("Thread "+type+
" after wait()");
  }
 } else {
  synchronized (shared) {
   shared.notifyAll();
   System.out.println("Thread "+type+
" after notifyAll()");
  }
   }
}

public static void main(String s[]) {
 ThreadTest w1 = new ThreadTest(1);
 new Thread(w1).start();
 try {
  Thread.sleep(100);
 } catch (InterruptedException e) {}

 ThreadTest w2 = new ThreadTest(2);
 new Thread(w2).start();
 try {
  Thread.sleep(100);
 } catch (InterruptedException e) {}
```

```
 ThreadTest w3 = new ThreadTest(3);
 new Thread(w3).start();

}
}
```

Результатом работы программы будет:

```
Thread 3 after notifyAll()
Thread 1 after wait()
Thread 2 after wait()
```

Рассмотрим, что происходило. Во-первых, был запущен поток 1, который тут же вызвал метод wait() и приостановил свое выполнение. Затем то же самое произошло с потоком 2. Далее начинает выполняться поток 3.

Сразу обращает на себя внимание следующий факт. Еще поток 1 вошел в synchronized-блок, а стало быть установил блокировку на объект shared. Но судя по результатам видно, что это не помешало и потоку 2 затем зайти в synchronized-блок, а потом и потоку 3. Причем для последнего это просто необходимо, иначе как можно "разбудить" потоки 1 и 2?

Можно сделать вывод, что потоки, прежде чем приостановить выполнение после вызова метода wait(), отпускают все занятые блокировки. Итак, вызывается метод notifyAll(). Как уже было сказано, все потоки из wait-set возобновляют свою работу. Однако чтобы корректно продолжить исполнение необходимо вернуть блокировку на объект, ведь следующая команда также находится внутри synchronized-блока!

Получается, что даже после вызова notifyAll() все потоки не могут сразу возобновить работу. Лишь один из них сможет вернуть себе блокировку и продолжить работу. Когда он покинет свой synchronized-блок и отпустит объект, второй поток возобновит свою работу и так далее. Если по какой-то причине объект так и не будет освобожден, поток так никогда и не выйдет из метода wait(), даже если будет вызван метод notifyAll(). В рассмотренном примере потоки один за другим смогли возобновить свою работу.

Кроме этого определен метод wait() с параметром, который задает период тайм-аута, по истечении которого поток сам попытается возобновить свою работу. Но начать ему придется все равно с повторного получения блокировки.

# 6. Контрольные вопросы

12-1. Каким образом на однопроцессорной машине исполняются многопоточные приложения?

a.) Операционная система разделяет рабочее время процессора на небольшие интервалы. В начале каждого интервала решается, какая из задач будет выполняться на его протяжении. Поскольку процессор переключается между задачами очень быстро, возникает ощущение, что они выполняются одновременно. Этот прием называется time-slicing.

12-2. Какие преимущества дает многопоточная архитектура?

а.) Основные преимущества:

- Если сама проектируемая система предназначена для выполнения нескольких действий одновременно (например, сервер, обслуживающий нескольких клиентов сразу), то, воспользовавшись встроенной многозадачностью, можно существенно упростить архитектуру системы, повысив ее надежность и быстродействие.
- Если необходимо выполнить несколько действий, которые требуют различных аппаратных и других ресурсов (память, жесткий диск, сеть, принтер, монопольный доступ к БД и т.д.), то одновременное выполнение этих задач потребует меньше времени, чем выполнение их по отдельности.
- Воспользовавшись свойством приоритетов потоков можно настроить систему так, чтобы она была наиболее удобна пользователю. Пусть формально процессор выполнит меньше вычислений, но они будут более полезны клиенту.

12-3. Что такое приоритет потока?

а.) Приоритет – это характеристика, как правило, числовая, которая влияет на распределение интервалов работы процессора между задачами. Предсказать количественный эффект от изменения приоритета в ту или иную сторону нельзя, но задача с большим приоритетом всегда получает больше времени, чем с меньшим приоритетом.

12-4. Что такое демон-поток?

а.) Демон-поток – это обычный поток. Единственно отличие от не-демонов заключается в том, что виртуальная машина закрывается, когда остаются рабочими только демон-потоки.

12-5. Когда закрывается виртуальная машина, выполняющая программу с несколькими потоками исполнения?

а.) Когда остаются рабочими только демон-потоки, либо не остается ни одного потока.

12-6. Для чего служит в Java класс Thread?

а.) Класс Thread служит для запуска потока, изменения его свойств и дальнейшего управления им (приостановка, возобновление работы, остановка и т.д.).

12-7. Поскольку интерфейс Runnable представляет собой альтернативный способ программировать потоки исполнения, можно ли в такой программе обойтись вовсе без класса Thread?

а.) Нет, поскольку для создания и запуска потока всегда необходимо создавать экземпляр класса Thread или его наследника. Также он необходим для дальнейшего управления потоком.

12-8. Что такое локальное и главное хранилища в Java? Чем они различаются?

a.) JVM поддерживает одно главное хранилище, в котором хранятся значения всех переменных, созданных программой. Локальное хранилище создается для каждого потока. После создания оно пустое, и поток должен скопировать те переменные, с которыми работает, в свою локальную память. Взаимодействие между главным и локальными хранилищами строго регламентировано. В частности, любая переменная создается сначала в главной, а затем копируется в локальную память.

12-9. Если один поток начал исполнение synchronized-блока, указав ссылку на некий объект, может ли другой поток обратиться к полю этого объекта? К методу?

a.) К любому полю можно обратиться беспрепятственно, равно как и к не-synchronized методу. Вызов synchronized-метода потребует установки блокировки, а до этого второй поток будет приостановлен.

12-10. Если объявить метод synchronized, то какой эффект будет этим достигнут?

a.) Гарантируется, что в один момент времени только один поток может его выполнять.

12-11. Как работают static synchronized методы?

a.) Аналогично обычным synchronized методам, только блокировка устанавливается не на объект класса, а на объект класса Class, описывающий исходный класс. Таким образом, гарантируется, что в один момент времени только один поток может работать со static synchronized методами класса.

12-12. Почему метод wait требует обработки InterruptedException, а методы notify и notifyAll – нет?

a.) Потому что вызов метода wait приводит к приостановке потока, работу которого можно возобновить, вызвав метод interrupt(), что и приведет к возникновению исключительной ситуации InterruptedException.

Методы notify и notifyAll не приостанавливают работу потока.

12-13. Может ли поток никогда не выйти из метода wait, даже если будет вызван метод notify? notifyAll?

a.) Поскольку метод wait может быть корректно вызван только при наличии блокировки на объект, а после успешного вызова поток эту блокировку освобождает, то для возобновления работы поток должен снова установить блокировку. Если по каким-то причинам она постоянно «занята», то поток никогда не выйдет из метода wait.

12-14. Какой будет результат работы следующего кода?

```
public abstract class Test implements Runnable {
 private Object lock = new Object();

 public void lock() {
  synchronized (lock) {
```

```
    try {
     lock.wait();
     System.out.println("1");
    } catch (InterruptedException e) {
    }
   }
  }

  public void unlock() {
   synchronized (lock) {
    lock.notify();
    System.out.println("2");
   }
  }

  public static void main(String s[]) {
   new Thread(new Test() {
    public void run() {
     lock();
    }
   }).start();
   new Thread(new Test() {
    public void run() {
     unlock();
    }
   }).start();
  }
 }
```

a.) На консоли появится только число 2, а виртуальная машина никогда не завершит работу, поскольку первый поток никогда не выйдет из метода wait. Хотя второй поток и вызывает метод notify, однако потоки работают с разными объектами (у каждого свое поле lock), что и приводит к описанному эффекту.

# Программирование на Java

## Лекция 13. Пакет java.lang.

### 20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

В этой лекции рассматривается основная библиотека Java – java.lang. В ней содержатся классы Object и Class, классы-обертки для примитивных типов, класс Math, классы для работы со строками String и StringBuffer, системные классы System, Runtime и другие. В этом же пакете находятся типы, уже рассматриваемые ранее – для работы с исключительными ситуациями и потоками исполнения.

# Оглавление

# Лекция 13. Пакет java.lang.

## Содержание лекции.

# 1. Введение

В состав пакета java.lang входят классы, составляющие основу для всех других - из-за чего он является наиболее важным из всех, входящих в Java API. Каждый класс в Java неявным образом (по умолчанию) импортирует все классы этого пакета - можно не писать import java.lang.*;.

Основу пакета составляют классы:

Object - является корневым в иерархии классов. Если при описании класса не указывается родительский, то им считается класс Object. Все объекты, включая массивы, наследуются от этого класса.

Class - экземпляры этого класса представляют классы и интерфейсы в запущенной Java-программе.

String - представляет средства работы с символьными строками

StringBuffer - используется для работы (создания) строк

Number - абстрактный класс, являющийся суперклассом для классов-объектных оберток числовых примитивных типов Java

Character - объектная обертка для типа char

Boolean- объектная обертка для типа boolean

Math - реализует ряд наиболее известных математических функций

Throwable - базовый класс для объектов, представляющих исключения. Любое исключение, которое может быть брошено и, соответственно, перехвачено блоком catch, должно быть унаследовано от Throwable

Thread - представляет вычислительный поток в Java. Виртуальная машина Java позволяет одновременно выполнять несколько потоков

ThreadGroup - позволяет объединять потоки в группу, и производить действия сразу над всеми потоками в ней. Сущствуют ограничения по безопасности на манипуляции с потоками из других групп. операции над потоком могут проводиться только потоком, относящимся к той же группе.

System - содержит полезные поля и методы для работы системного уровня

Runtime - позволяет приложению взаимодействовать с окружением в котором оно запущено

Process - представляет внешнюю программу, запущенную при помощи Runtime

ClassLoader - отвечает за загрузку классов

SecurityManager - для обеспечения безопасности накладывает ограничения на данную среду выполнения программ

Compiler - используется для поддержки Just-in-Time компиляторов

Интерфейсы:

Cloneable - должен быть реализован объектами, которые планируется клонировать с помощью средств JVM (Object.clone())

Runnable - может использоваться в сочетании с классом Thread, позволяя реализовать метод run(), котрый будет вызван при старте данного потока.

Comparable - позволяет упорядочивать (сортировать, сравнивать) объекты каждого класса, реализующего этот интерфейс.

# 2. Object

Класс Object является базовым для всех остальных классов. Он определяет методы, которые поддерживаются любым классом в Java.

Это методы:

1.0) public boolean equals(Object obj) - определяет, являются ли объекты одинаковыми. Если оператор == определяет равенство именно объектных ссылок, то метод equals() определяет, содержат ли объекты одинаковую информацию. Реализация этого метода в классе Object такова, что значение true будет возвращено тогда и только тогда, если объектные ссылки переданного объекта и объекта, у которого вызывается метод совпадают, то есть:

```
public boolean equals(Object obj) {
  return (this == obj);
 }
```

В классах - наследниках, этот метод при необходимости может быть переопределен, что бы отражать действительное равенство объектов. Например, сравнение объектов-оберток целых чисел (класс Integer), должно по всей логике возвращать значение true, если равны значения int чисел, которые обернуты, даже если различаются объектные ссылки на сами объекты класса-обертки Integer.

Метод equals() может быть переопределен любым способом (например, всегда возвращает false, или, наоборот, - true) - компилятор, конечно же, не будет проводить анализ реализации и давать рекомендации, НО: реализация метода предполагается с учетом следующих правил:

1. рефлексивность: для любой объектной ссылки x, вызов x.equals(x) возвращает true
2. симметричность: для любых объектных ссылок x и y, вызов x.equals(y) возвращает true тогда и только тогда, если вызов y.equals(x) возвращает true
3. транзитивность: для любых объектных ссылок x, y и z, если x.equals(y) возвращает true и y.equals(z) возвращает true, тогда вызов x.equals(z) должен вернуть true
4. непротиворечивость: для любых объектных ссылок x и y, многократные последовательные вызовы x.equals(y) возвращают одно и то же значение (либо всегда true либо всегда false)
5. для любой не равной null объектной ссылки x, вызов x.equals(null) должен вернуть значение false

Пример:

```
package demo.lang;
```

```
public class Rectangle {
 public int sideA;
 public int sideB;
public Rectangle(int x, int y) {
 super();
 sideA = x;
 sideB = y;
}
public boolean equals(Object obj) {
 if(!(obj instanceof Rectangle)) return false;
 Rectangle ref = (Rectangle)obj;
 return (((this.sideA==ref.sideA)&&(this.sideB==ref.sideB))||
    (this.sideA==ref.sideB)&&(this.sideB==ref.sideA));
}
public static void main(String[] args) {
 Rectangle r1 = new Rectangle(10,20);
 Rectangle r2 = new Rectangle(10,10);
 Rectangle r3 = new Rectangle(20,10);
 System.out.println("r1.equals(r1) == " + r1.equals(r1));
 System.out.println("r1.equals(r2) == " + r1.equals(r2));
 System.out.println("r1.equals(r3) == " + r1.equals(r3));
 System.out.println("r2.equals(r3) == " + r2.equals(r3));
 System.out.println("r1.equals(null) == " + r1.equals(null));
}
}
```

Запуск этой программы, очевидно, приведет к выводу на экран следующего:

```
r1.equals(r1) == true
r1.equals(r2) == false
r1.equals(r3) == true
r2.equals(r3) == false
r1.equals(null) == false
```

В этом примере метод equals() у класса Retangle был переопределен таким образом, что бы прямоугольники были равны, если их можно наложить друг на друга (геометрическое равенство).

1.1) public int hashCode() - возвращает хеш-код (hash code) для объекта. Хеш-код - это целое число, которое с очень большой вероятностью является уникальным для данного объекта. Это значение используется классом Hashtable для хранения с возможностью быстрой выборки объектов(хеш-код используется для группировки объектов, и при поиске объекта достаточно просмотреть только блок, соответствующий его хеш-коду)

Обычно вполне можно пользоваться стандартной реализацией, но если же будет принято решение изменить метод вычисления хеш-кода, то необходимо убедиться, что в этом случае будет возвращаться одно и то же значение для равных между собой объектов. То есть, если x.equals(y) возвращает true, то хеш-коды x и y должны совпадать, то есть вызовы

x.hashCode() и y.hashCode() должны возвращать одно и то же значение. В противном случае Hashtable будет считать объекты различными, не вызывая метод equals()

Формально правила, которым должна следовать реализация метода, формулируется следующим образом:

a) в одном запуске программы, для одного объекта, при вызове метода hashCode(), должно возвращаться одно и то же int значение, если между этими вызовами НЕ были затронуты данные, используемые для проверки объектов на идентичность в методе equals(). Это число НЕ обязано быть одним и тем же при повторном запуске той же программы, даже если все данные будут идентичны

b) если два объекта идентичны, то есть вызов метода equals(Object) возвращает true, тогда вызов метода hashCode() у каждого из этих двух объектов должен возвращать одно и то же значение

c) если два объекта различны, то есть вызов метода equals(Object) возвращает false, тогда различие их хеш-кодов желательно, но НЕ обязательно. Различие в хеш-кодах для НЕ идентичных объектов нужно только для обеспечения хорошей производительности при использовании этих объектов в хеш-таблицах.

В классе Object реализация метода hashCode() использует для получения результата преобразование внутреннего адреса объекта в памяти, поэтому если не перекрывать эту реализацию, то для разных объектов будут возвращены различные значения.

1.2) public String toString() - возвращает строковое представление объекта. В классе Object этот метод реализован следующим образом:

```
public String toString() {
  return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

То есть возвращает строку, содержащую название класса объекта и его хеш-код.

В классах-наследниках этот метод может быть переопределен для получения более наглядного пользовательского представления объекта. Обычно это значения некоторых полей, характеризующих экземпляр. Например, для книги это может быть название, автор и количество страниц:

```
package demo.lang;

public class Book {
 private String title;
 private String author;
 private int pagesNumber;
public Book(String title, String author, int pagesNumber) {
 super();
 this.title = title;
 this.author = author;
 this.pagesNumber = pagesNumber;
}
public static void main(String[] args) {
```

```
 Book book = new Book("Java2", "Sun", 1000);
 System.out.println("object is: " + book);
}
public String toString(){
 return "Book: " + title + " ( " + author + ", " + pagesNumber + " pages )";
}
}
```

При запуске этой программы, на экран будет выведено следующее:

```
object is: Book: Java2 ( Sun, 1000 pages )
```

То есть, для получения строкового представления объекта book, был вызван его метод toString() (оператор "+" определен для строк таким образом, что все операнды будут сначала приведены к String, для объектов в этом случае вызывается метод toString()).

1.3) wait(), notify(), notifyAll() - эти методы используются для поддержки многопоточности. Они определены с атрибутом final и НЕ могут быть переопределены в классах-наследниках.

1.4) protected void finalize() throws Throwable - этот метод вызывается Java-машиной перед тем, как garbage collection (сборщик мусора) очистит память, занимаемую объектом. Работа сборщика мусора и описание схемы, по которой производится вызов метода finaliuze() приведено в соответствующей главе. Здесь же ограничимся только напоминанием, что объект считается пригодным для сборщика мусора, когда выполняющейся части программы не будет доступно ни одной ссылки на объект. Перед очисткой занимаемой объектом памяти, будет произведен вызов его метода finalize().

Реализация этого метода в классе Object - не производит никаких действий. В классах-наследниках этот метод может быть переопределен для проведения всех необходимых действий по освобождению различных занимаемых ресурсов - закрытия сетевых соединений, файлов и т.д.

1.5) protected native Object clone() throws CloneNotSupportedException - создает копию объекта. Для того, что бы им можно было воспользоваться, объект должен реализовывать интерфейс Cloneable. Этот интерфейс не определяет никаких методов, определение, что класс его реализует - только символизирует, что можно создавать копии объектов этого класса. В классе Object метод clone() реализован таким образом, что будут скопированы только базовые типы и ссылки на объекты. Если же потребуется "глубокое " копирование, то есть скопировать не только ссылки на объекты, но и создать копии объектов - в классах-наследниках метод clone() можно переопределить.

Например, в следующем примере:

```
package demo.lang;
public class BookStorage implements Cloneable{
 public Book[] books;
public BookStorage() {
 books = new Book[2];
 books[0] = new Book("Essential java", "Sun", 500);
 books[1] = new Book("Professional Java", "Sun", 1000);
}
}
```

Если выполнить следующий код:

```
 BookStorage storage1 = new BookStorage();
 try{
  BookStorage storage2 = (BookStorage)storage1.clone();
  storage2.books[2] = new Book("Java 2 Enterprise", "Sun", 2000);
System.out.println("storage1.books[1] = " + storage1.books[1]);
 }catch(CloneNotSupportedException e){
  e.printStackTrace();
 }
```

То изменения затронут и объект storage1 - изменится содержимое массива books. Это произойдет, потому что реализация метода clone() по умолчанию скопирует только ссылку на массив, поэтому изменение его содержимого затронет оба объекта: storage1 и storage2. Если же мы хотим, что бы копировались не только ссылки, имеющиеся в объекте, но объекты, на которые имеются ссылки, необходимо соответствующим образом переопределить метод clone(). В случае данного примера, это можно сделать следующим образом:

```
protected Object clone() throws CloneNotSupportedException{
 BookStorage result = (BookStorage)super.clone();
 result.books = (Book[])books.clone();
 return result;
}
```

Теперь будет создаваться копия не только BookStorage, но и массива books. Это легко проверить, еще раз запустив вышеприведенный код.

1.6) public final native Class getClass() - возвращает объект типа Class, соответствующий классу объекта. Именно этот объект используется при использовании синхронизации статических методов.

# 3. Class

В запущенной программе Java каждому классу соответствует объект типа Class. Этот объект содержит информацию, необходимую для описания класса – поля, методы и т.д.

Класс Class не имеет открытого конструктора – объекты этого класса создаются автоматически Java-машиной по мере загрузки классов и вызовов метода defineClass() загрузчика классов. Получить экземпляр Class для конкретного класса можно воспользовавшись методом forName():

2.1) public static Class forName(String name, boolean initialize, ClassLoader loader) – возвращает объект Class, соответствующий классу или интерфейсу с названием, указанным в name (необходимо указывать полное название класса или интерфейса), используя переданный загрузчик классов. Если в качестве загрузчика классов loader передано значение null, будет взят таковой, который использовался для загрузки вызывающего класса. При этом класс будет инициализирован, только если значение initialize равно true и класс не был инициализирован ранее.

Довольно часто проще и удобнее воспользоваться методом forName(), передав только название класса:

public static Class forName(String className) – при этом будет использоваться загрузчик вызывающего класса, и класс будет инициализирован, если до этого не был.

2.2) public Object newInstance() – создает и возвращает объект класса, который представляется данным экземпляром Class. Создание будет проходить, используя конструктор без параметров. Если такового в классе нет, будет брошено исключение InstantiationException. Это же исключение будет брошено, если объект Class соответствует абстрактному классу, интерфейсу или же по какой-либо другой причине.

2.3) Каждому методу, полю, конструктору класса так же соответствуют объекты, которые можно получить вызовом соответствующих методов на объекте Class:

getMethods(), getFields(), getConstructors(),getDeclaredMethods() и т.д. В результате будут получены объекты, которые отвечают за поля, методы, конструкторы объекта. Их можно использовать для формирования динамических вызовов Java – такой процесс называется отражение (reflection). Классы, используемые для отражения содержатся в пакете java.lang.reflection.

Динамическое создание экземпляров не обязательно всегда сопровождается вызовом методов посредством отражения. Это показано в следующем примере:

```
package demo.lang;
interface Vehicle {
void go();
}
class Automobile implements Vehicle {
public void go() {
  System.out.println("Automobile go!");
}
}
class Truck implements Vehicle {
public Truck(int i) {
  super();
}
public void go() {
  System.out.println("Truck go!");
}
}
public class VehicleStarter {
public static void main(String[] args) {
 Vehicle vehicle;
 String[] vehicleNames = {"demo.lang.Automobile", "demo.lang.Truck",
"demo.lang.Tank"};
 for(int i=0; i<vehicleNames.length; i++){
  try{
   String name = vehicleNames[i];
   System.out.println("look for clas for: " + name);
   Class aClass = Class.forName(name);
```

```
   System.out.println("creating vehicle...");
   vehicle = (Vehicle)aClass.newInstance();
   System.out.println("create vehicle: " + vehicle.getClass());
   vehicle.go();
  }catch(ClassNotFoundException e){
   System.out.println("Exception: " + e.toString());
  }catch(InstantiationException e){
   System.out.println("Exception: " + e.toString());
  }catch(Throwable th){
   System.out.println("Another problem: " + th.toString());
  }
 }
}
}
```

Если запустить эту программу, на экран будет выведено следующее:

```
look for clas for: demo.lang.Automobile
creating vehicle...
create vehicle: class demo.lang.Automobile
Automobile go!
look for clas for: demo.lang.Truck
creating vehicle...
Instantiation exception: java.lang.InstantiationException
look for clas for: demo.lang.Tank
Class not found: java.lang.ClassNotFoundException: demo.lang.Tank
```

Как видим, объект класса Automobile был успешно создан, а дальше с ним работа велась через интерфейс Vehicle. А вот класс Truck был найден, но при создании объекта этого класса, было брошено и, соответственно, обработано исключение java.lang.InstantiationException. Так-как класс java.lang.Tank не был определен, то при попытке получить объект Class, ему соответствующий, было брошено исключение java.lang.ClassNotFoundException.

# 4. Wrapper Classes

Во многих случаях бывает предпочтительней работать именно с объектами, а не примитивными типами. Так, например, при использовании коллекций, просто необходимо значения примитивных типов каким-то образом представлять в виде объектов. Для этих целей и предназначены так называемые классы-обертки. Для каждого примитивного типа Java существует свой класс обертка. Такой класс является неизменяемым (то есть, для изменения значения необходимо создавать новый объект), к тому же имеет атрибут final - от него нельзя наследовать класс. Все классы-обертки (кроме Void) реализуют интерфейс Serializable, поэтому объекты любого (кроме Void) класса-обертки могут быть сериализованы. Все классы-обертки содержат статическое поле TYPE - содержащее объект Class, соответствующий примитивному оборачиваемому типу.

Так же классы-обертки содержат статические методы для обеспечения удобного манипулирования соответствующими примитивными типами, например преобразование к строковому виду.

В таблице 1 описаны примитивные типы и соответствующие им классы обертки:

| Класс-обертка | Примитивный тип |
|---|---|
| Byte | byte |
| Short | short |
| Character | char |
| Integer | int |
| Long | long |
| Float | float |
| Double | double |
| Boolean | boolean |

При этом классы обертки числовых типов - Byte, Short, Integer, Long, Float, Double наследуются от одного класса - Number. В нем содержится код, общий (часть реализована посредством абстрактных методов) для всех классов-оберток числовых типов - примитивного значения в виде byte, short, int, long, float и double.

Все классы-обертки реализуют интерфейс Comparable. Number реализует интерфейс java.io.Serializable, поэтому все объекты классов-оберток примитивных числовых типов могут быть сериализованы.

Все классы-обертки числовых типов имеют метод equals(Object), сравнивающий примитивные значения объектов.

> Стоит быть особо внимательным - результат выполнения (new Integer(1)).equals(new Byte(1)) дает false, хотя сами значения, вокруг которых обернуты объекты - равны. Такой результат получается потому, что во всех таких классах, метод equals() определен таким образом, что сначала производится проверка, совпадают ли типы (классы) значений, и если нет - сразу возвращается false. Например в jdk1.3.1 для Integer метод equals() определен следующим образом:

```
    public boolean equals(java.lang.Object obj) {
    if(obj instanceof java.lang.Integer)
        return value == ((java.lang.Integer)obj).intValue();
    else
        return false;
}
```

Рассмотрим более подробно некоторые из классов-оберток.

## 4.1. Integer

Наиболее часто используемые статические методы.

public static int parseInt(String s) - преобразует в int значение строку, представляющую десятичную запись целого числа

public static int parseInt(String s, int radix) - преобразует в int значение строку, представляющую запись целого числа в системе счисления radix

Оба этих метода могут возбуждать исключение NumberFormatException, если строка, переданная на вход, содержит нецифровые символы.

Не следует путать эти методы, с другой парой похожих методов:

public static Integer valueOf(String s) public static Integer valueOf(String s,int radix)

Эти методы выполняют аналогичную работу, только результат представляют в виде объекта-обертки.

Существует так же два конструктора для создания экземпляров в класса Integer

Integer(String s) - такой метод, принимающий в качестве параметра строку, представляющую значение, имеется для каждого класса

Integer(int i) - аналогично, для каждого класса-обертки числового примитивного типа, имеется конструктор, принимающий значение оборачиваемого примитивного типа

Первый вариант конструктора так же может возбуждать исключение NumberFormatException

pubic static String toString(int i) - используется для преобразования значения типа int в строку

Далее перечислены методы, преобразующие int в строковое восьмеричное, двоичное и шестнадцатеричное представление:

pubic static String toOctalString(int i) - восьмеричное

pubic static String toBinaryString(int i) - двоичное

pubic static String toHexString(int i) - шестнадцатиричное .

Имеется так же две статические константы:

Integer.MIN_VALUE - минимальное int значение

Integer.MAX_VALUE - максимальное int значение.

Аналогичны константы, равные границам соответствующих типов, определены и для всех остальных классов-оберток числовых примитивных типов.

public int intValue();

возвращает значение примитивного типа для данного Integer. Классы-обертки остальных примитивных целочисленных типов: Byte, Short, Long содержат только аналогичные методы и константы, только определенные для соответствующих типов: byte, short, long.

Пример:

```
    public static void main(String[] args) {
        int i = 1;
        byte b = 1;
        String value = "1000";
        Integer iObj = new Integer(i);
        Byte bObj = new Byte(b);
        System.out.println("while i==b is " + (i==b));
        System.out.println("iObj.equals(bObj) is " + iObj.equals(bObj));
        Long lObj = new Long(value);
        System.out.println("lObj = " + lObj.toString());
        Long sum = new Long(lObj.longValue() + iObj.byteValue() +
bObj.shortValue());
        System.out.println("The sum = " + sum.doubleValue());
    }
```

В данном примере произвольным образом используются различные варианты классов-оберток и их методов. Соответственно, в результате выполнения, на экран будет выведено следующее:

```
while i==b is true
iObj.equals(bObj) is false
lObj = 1000
The sum = 1002.0
```

Оставшиеся классы-обертки числовых типов: Float и Double - помимо описанного для целочисленных примитивных типов, дополнительно содержат определения следующих констант:

NEGATIVE_INFINITY - отрицательная бесконечность

POSITIVE_INFINITY - положительная бесконечность

NaN - НЕ числовое значение (неопределенность, комплексное число и т.д.)

Кроме того, несколько иначе трактуется значение MIN_VALUE - вместо наименьшего значения, оно представляет минимальное положительное (строго > 0) значение, которое может быть представлено соответствующим примитивным типом и, соответственно, классом-оберткой.

Кроме классов-оберток для примитивных числовых типов, определены таковые и для остальных примитивных типов Java:

## 4.2. Character

Реализует интерфейс Comparable.

Из конструкторов - имеет только один, принимающий char в качестве параметра.

Кроме стандартных методов equals(), hashCode(), toString() еще из НЕ статических, содержит только два метода:

public char charValue() - возвращает обернутое значение char

public int compareTo(Character anotherCharacter) - сравнивает обернутые значения char как числа, то есть возвращает значение return this.value - anotherCharacter.value;

Так-же, для совместимости с интерфейсом Comparable, метод compareTo() определен с параметром Object:

public int compareTo(Object o) - если переданный объект имеет тип Character, результат будет аналогичен вызову compareTo((Character)o), иначе будет брошено исключение ClassCastException - так-как Character можно сравнивать только с Character.

Статических методов в классе Character довольно много, все их здесь перечислять смысла не имеет. Все они могут быть довольно полезны. Большинство - это методы, принимающие char и проверяющие всевозможные свойства - является ли цифрой, буквой, буквой заглавного или строчного шрифта, может ли с него начинаться переменная в Java, и т.д. Например:

public static boolean isDigit(char c) - проверяет, является ли char цифрой

public static boolean isLetter(char c) - проверяет, является ли char буквой

public static boolean isDigitOrLetter(char c) - проверяет, является ли char цифрой или буквой

public static boolean isIdentifierStart(char c) - проверяет, является ли символ подходящим для того, что бы с него начиналось наименование переменной JAVA

Эти методы возвращают значение истина или ложь, в соответствии с тем выполнен ли критерий проверки.

## 4.3. Boolean

Представляет класс-обертку для примитивного типа boolean.

Реализует интерфейс java.io.Serializable и во всем напоминает аналогичные классы-обертки.

Для получения примитивного типа используется метод booleanValue().

## 4.4. Void

Этот класс-обертка, в отличии от остальных, НЕ реализует интерфейс java.io.Serializable. Он не имеет открытого конструктора. Более того, экземпляр этого класса вообще не может быть получен. Он нужен только для получения ссылки на объект Class, соответствующий void. Эта ссылка представлена статической константой TYPE.

Делая краткое заключение по классам-оберткам, можно сказать что

- каждый примитивный тип имеет соответствующий класс-обертку
- все классы-обертки могут быть сконструированы как с использованием примитивных типов, так и с использованием String, за исключением Character, который может быть сконструирован только по char
- Классы-обертки могут сравниваться с использованием метода equals()
- примитивные типы могут быть извлечены из классов-оберток с помощью соответствующего метода xxxxValue() (например intValue())
- классы-обертки так же являются классами-утилитами, т.е. предоставляют набор статических методов для работы с примитивными типами
- классы-обертки не могут быть модифицированы

# 5. Math

Класс Math состоит из набора статических методов, производящих наиболее популярные математические вычисления и двух констант, имеющих особое значение в математике - это число Пи и экспонента. Часто этот класс еще называют классом-утилитой (Utility class). Так как все методы класса статические нет необходимости создавать экземпляр этого класса - поэтому он и не имеет открытого конструктора. Нельзя так же и унаследовать этот класс, поскольку он объявлен с атрибутом final.

Итак, константы определены следующим образом:

public static final double Matht.PI - задает число Пи

public static final double Matht.E - число e.

> Следует обратить внимание, что тип констант double. При использовании этих констант в вычислениях, результат будет автоматически конвертирован в double, если его явно не привести к другому типу.

В таблице 2 приведены все методы класса. Так же дано их краткое описание

| | | |
|---|---|---|
| static double | abs(double a) | Возвращает абсолютное значение типа double |
| static float | abs(float a) | Возвращает абсолютное значение типа byte |
| static int | abs(int a) | Возвращает абсолютное значение типа int (1) |
| static long | abs(long a) | Возвращает абсолютное значение типа long |
| static double | acos(double a) | Вернет значение арккосинуса угла в диапазоне от 0 до PI |
| static double | asin(double a) | Вернет значение арксинуса угла в диапазоне от -PI/2 до PI/2 |
| static double | atan(double a) | Вернет значение арктангенса угла в диапазоне от -PI/2 до PI/2 |
| static double | ceil(double a) | Возвращает наименьшее целое число которое больше a. (2) |
| static double | floor(double a) | Возвращает целое число которое меньше a. (2) |
| static double | cos(double a) | Возвращает косинус угла (3) |
| static double | IEEEremainder(double a, double b) | Возвращает остаток от деления a/b по стандарту IEEE 754 (* см. пояснение дальше по тексту) |
| static double | sin(double a) | Возвращает косинус угла (3) |
| static double | tan(double a) | Возвращает тангенс угла (3) |
| static double | exp(double a) | Возвращает e в степени числа a |
| static double | log(double a) | Возвращает натуральный логарифм числа a |
| static double | max(double a, double b) | Возвращает наибольшее из двух чисел типа double (4) |
| static float | max(float a, float b) | Возвращает наибольшее из двух чисел типа double (4) |
| static long | max(long a, long b) | Возвращает наибольшее из двух чисел типа long (4) |
| static int | max(int a, int b) | Возвращает наибольшее из двух чисел типа int (4) |
| static double | min(double a, double b) | Возвращает наименьшее из двух чисел типа double (4) |
| static float | min(float a, float b) | Возвращает наименьшее из двух чисел типа double (4) |
| static long | min(long a, long b) | Возвращает наименьшее из двух чисел типа long (4) |
| static int | min(int a, int b) | Возвращает наименьшее из двух чисел типа int (4) |
| static double | pow(double a, double b) | Возвращает a в степени b |
| static double | random() | Возвращает случайное число в диапазоне от 0.0 до 1.0 |
| static double | rint(double a) | Возвращает int число, ближайшее к a |

| | | |
|---|---|---|
| static long | round(double a) | Возвращает значение типа long ближайшее по значению к a. (5) |
| static long | round(double a) | Возвращает значение типа long ближайшее по значению к a. (6) |
| static double | sqrt(double a) | Возвращает положительный квадратный корень числа a |
| static double | toDegrees(double angrad) | Преобразует значение угла из радианов в градусы |
| static double | toRadians(double angdeg) | Преобразует значение угла из градусов в радианы |

Следует обратить внимание, что

1. abs вернет значения типа int, если в качестве параметра будут переданы значения типа byte, short, char.
2. Угол задается в радианах.
3. Если round имеет аргумент double, то возвращается значение типа long, если аргумент типа float, то будет возвращено значение типа int.
4. Операторы и функции для вычисления остатка подробно рассматривались в 3 лекции.

# 6. Строки

## 6.1. String

Этот класс используется в Java для представления строк. Он предназначен для хранения не модифицируемых строк. После того как создан экземпляр этого класса, строка уже не может быть модифицирована. Для создания объекта String можно использовать различные варианты конструктора. Наиболее простой - если содержимое строки известно на этапе компиляции - это написать текст в кавычках:

```
String abc = "abc";
```

Можно использовать и различные варианты конструктора. Наиболее простой из них - конструктор, получающий на входе строковый литерал.

```
String s = new String("immutable");
```

На первый взгляд, эти варианты создания строк отличаются только синтаксисом. На самом деле различие есть, хотя в большинстве случаев его можно и не заметить. Каждый строковый литерал имеет внутреннее представление, как экземпляр класса String. Классы в JAVA могут иметь целый набор таких строк и, когда класс компилируется, экземпляр представляющий литерал, добавляется в этот набор. Однако, если такой литерал уже имеется где-то в другом месте класса, т.е. уже представлен в наборе строковых объектов, то новый экземпляр (фактически копирующий уже существующий) создан не будет. Вместо этого будет создана ссылка на уже имеющийся объект. Т.к. строки не являются модифицируемыми объектами, то это не нанесет никакого урона другим фрагментам программы. С другой стороны, если объект-строка создается с помощью явного вызова

конструктора, то даже если эти строки будут совершенно идентичными, экземпляры класса String будут отличаться.

В объекте String определен метод equals() который сравнивает две строки на предмет идентичности.

Рассмотрим пример

```
public class Test {
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        String s1 = "Hello world !!!";
        String s2 = "Hello world !!!";
        System.out.println("String`s equally = " + (s1.equals(s2)));
        System.out.println("Strings are the same = " + (s1==s2));
    }
}
```

в результате на консоль будет выведено

```
String`s equally = true
Strings are the same = true
```

Теперь несколько модифицируем код

```
public class Test {
    public Test() {
    }
    public static void main(String[] args) {
        Test t = new Test();
        String s1 = "Hello world !!!";
        String s2 = new String("Hello world !!!");
        System.out.println("String`s equally = " + (s1.equals(s2)));
        System.out.println("Strings are the same = " + (s1==s2));
    }
}
```

в результате на консоль будет выведено

```
String`s equally = true
Strings are the same = false
```

В первом примере для создания строк используются строковые литералы, поэтому ссылки s1 и s2 ссылаются на один и тот же объект. Во втором случае применены конструкторы, поэтому, несмотря на то, что строки идентичны переменные ссылаются на разные объекты, которые, в сущности, создаются во время выполнения программы и не находятся во множестве строковых констант, которое создается на момент компиляции.

Следует обратить внимание, что при создании экземпляров строк во время выполнения, они не помещаются в набор строковых констант. Однако, можно явно указать на необходимость поместить, вновь создаваемый экземпляр класса String в этот набор, применив метод intern()

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        String s1 = "Hello world!!!";
        String s2 = new String("Hello world!!!").intern();
        System.out.println("String`s equally = " + (s1.equals(s2)));
        System.out.println("Strings are the same = " + (s1==s2));
    }
}
```

в этом случае на консоль будет выведено

```
String`s equally = true
Strings are the same = true
```

В такой подход экономит занимаемую программой память, что в некоторых случаях может быть существенно. Помимо этого, если мы уверены, что все троки находятся в наборе сформированном при компиляции, для сравнения сток, вместо метода equals() можно использовать оператор ==, который выполняется значительно быстрее.

В JAVA для строк переопределен оператор +. При использовании этого оператора производится конкатенация строк. В классе String так же определен метод

```
public String concat(String s);
```

он возвращает новый объект строку дополненный справа строкой s.

Следует еще раз обратить внимание, что строки являются не модифицируемыми объектами. И если используется оператор конкатенации или какой-либо вспомогательный метод класса String то изменения строки не произойдет, а будет создан новый экземпляр класса String. Так же следует обратить на использование в методах параметров типа String. Не смотря на то, что String является объектом и передается в метод по ссылке, String не модифицируемый объект и все изменения в методе не повлекут изменений исходного объекта.

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        String s = "prefix";
        System.out.println("String before = " + s);
        t.perform(s);
        System.out.println("String after =" + s);
    }
    private void perform(String s){
        System.out.println(s + " suffix");
    }
}
```

```
String before = prefix
prefix suffix
String after =prefix
```

В этом примере видно, что строка s в действительности не изменяется.

Рассмотрим другой пример.

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        String s = " prefix !";
        System.out.println(s);
        s = s.trim();
        System.out.println(s);
        s = s.concat(" suffix");
        System.out.println(s);
    }
}
```

```
prefix !
prefix !
prefix ! suffix
```

В данном случае может сложиться впечатление, что строку можно изменять. В действительности это не так. После выполнения операций trim и concat создается новый объект - строка, ссылка s, будет указывать на новый объект-строку.

Как уже отмечалось ранее, строка состоит из шестнадцатибитовых UNICODE символов. Однако во многих случаях требуется работать с восьмибитовыми символами (ввод/вывод, работа с базой данных и т.д.). Преобразование строки в последовательность байтов (восьмибитовые символы) производится с методами

byte[] getBytes(); - возвращает последовательность байтов, в кодировке принятой по умолчанию. (Как правило зависит от настроек операционной системы)

byte[] getBytes(String encoding); - возвращает последовательность байтов, в кодировке encoding;

Для выполнения обратной операции (преобразования байтов в строку) необходимо сконструировать новый объект-строку.

String(byte[] bytes); - создает строку из последовательности байтов в кодировке принятой по умолчанию;

String(byte[] bytes,String enc); - создает строку из последовательности байтов в указанной кодировке.

## 6.2. StringBuffer

Этот класс используется для создания и модификации строковых выражений, которые после можно превратить в String. Он реализован на основе массива char[] и, в отличии от String, после создания объекта, значение строки, в нем содержащейся может быть изменено.

Рассмотрим наиболее часто используемые конструкторы класса StringBuffer

StringBuffer () - создает пустой StringBuffer

StringBuffer(String s) - в качестве параметра используется объект String

StringBuffer (int capacity) - создает экземпляр класса StringBuffer с заранее заданным размером. Задание размера не означает, что нельзя будет манипулировать со строками, длинной более указанной при создании объекта, а всего лишь говорит о том, что при манипулировании строками, длина которых меньше указанной, не потребуется дополнительного выделения памяти.

Типичный пример использования StringBuffer может быть продемонстрирован следующим кодом:

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        String s = new String("ssssss");
        StringBuffer sb = new StringBuffer("bbbbbb");
        s.concat("-aaa");
        sb.append("-aaa");
        System.out.println(s);
        System.out.println(sb);
    }
}
```

В результате на экран будет выведено следующее:

```
ssssss
bbbbbb-aaa
```

В данном примере можно заметить, что объект String остался неизменным, а объект StringBuffer изменился.

Основные методы, используемые для модификации StringBuffer - это:

public StringBuffer append(String str) - добавляет переданную строку str к уже имеющейся в объекте;

public StringBuffer insert(int offset, String str) - вставка строки, начиная с позиции offset (пропустив offset символов)

Стоит обратить внимание, что оба этих метода имеют варианты, принимающие в качестве параметров различные примитивные типы Java вместо String. При использовании этих методов, значение этого примитивного типа сначала будет представлено строкой, как это произошло бы вызовом метода String.valueOf(), после чего будет вызван этот же метод, передав полученное значение.

Еще один важный момент, связанный с этими методами - они возвращают сам объект, на котором вызываются. Таким образом, возможно их использование в цепочке. Например:

В результате на экран будет выведено:

```
abcdef
```

При передаче экземпляра класса в качестве параметра в метод StringBuffer, так же следует помнить об отличии String и StringBuffer.

```
public class Test {
    public static void main(String[] args) {
        Test t = new Test();
        StringBuffer sb = new StringBuffer("aaa");
        System.out.println("Before = " + sb);
        t.doTest(sb);
        System.out.println("After = " + sb);
    }

    void doTest(StringBuffer theSb){
        theSb.append("-bbb");
    }
}
```

В результате на экран будет выведено следующее:

```
Before = aaa
After = aaa-bbb
```

Т.к. все объекты передаются по ссылке, в методе doTest, при выполнении операций с theSB, будет модифицирован объект, на который ссылается sb. Следует еще раз напомнить, что для String переопределен оператор +, т.е. если + применить экземплярам класса String то будет осуществлена конкатенация строк и, если один из операндов не принадлежит к классу String, то он будет неявно приведен к этому типу. Примитивные типы будут преобразованы в String, как это произошло бы вызовом метода String.valueOf().

Например

```
System.out.println("1" + 5)  выведет на консоль 15
System.out.println(1+ 5)  выведет на консоль 6
```

# 7. Системные классы

Следующие классы, которые стоит рассмотреть, отвечают за выполнение хода программы, это

ClassLoader - загрузчик классов. Содержит методы, необходимые для динамической загрузки новых классов

SecurityManager - менеджер безопасности. Содержит различные методы проверки, допустима ли запрашиваемая операция.

System - содержит набор статических методов, применимых к среде, в которой выполняется приложение. Многие из них присутствуют так же в классе Runtime.

Runtime - позволяет приложению взаимодействовать с окружением в котором оно запущено. Каждому приложению соответствует один экземпляр Runtime.

Process - представляет внешнюю программу, запущенную при помощи Runtime

## 7.1. ClassLoader

Это абстрактный класс, ответственный за загрузку классов. По имени класса он находит либо генерирует данные, которые составляют определение класса. Обычно для этого используется следующая стратегия: название класса преобразуется в название файла - "class file", из которого и считывается вся необходимая информация.

Каждый объект Class содержит ссылку на объект ClassLoader, посредством которого он был загружен.

Для изменения способа загрузки классов, можно реализовать свой загрузчик классов, унаследовав его от ClassLoader. Так, хотя обычно классы загружаются из файлов, однако бывают и другие ситуации. Например, классы могут загружаться через сетевое соединение. Метод defineClass() преобразует массив байт в экземпляр класса Class. Экземпляры полученного таким образом класса могут быть получены, используя метод newInstance() у объекта Class. Методы объектов, полученных с помощью загрузчика классов, могут ссылаться на другие, доступные в запущенном приложении классы. Для получения классов, на которые можно ссылаться, вызывается метод loadClass у загрузчика классов.

Для иллюстрации использования загрузчика классов, приведем пример, как может выглядеть простая реализация загрузчика классов, использующего сетевое соединение:

```
class NetworkClassLoader extends ClassLoader {
    String host;
    int port;
    public NetworkClassLoader(String host, int port) {
        this.host = host;
        this.port = port;
    }
    public Class findClass(String className) {
        byte[] bytes = loadClassData(className);
        return defineClass(className, bytes, 0, bytes.length)
    }
    private byte[] loadClassData(String className) {
       byte[] result = null;
       // open connection, load the class data
       return result;
    }
}
```

В этом примере только показано, что подкласс загрузчика классов доложен определить и реализовать методы findClass() и loadClassData() для загрузки классов. Когда набор байт, образующих класс, загружен, необходимо использовать метод defineClass() для создания класса. Для простоты, в примере приведен только шаблонный код без реализации получения байт из сетевого соединения.

Для получения экземпляров классов, загруженных с помощью этого загрузчика, можно написать код, аналогичный следующему:

```
try{
  ClassLoader loader = new NetworkClassLoader(host, port);
  Object main = loader.loadClass("Main").newInstance();
 }catch(ClassNotFoundException e){
  e.printStackTrace();
 }catch(InstantiationException e){
  e.printStackTrace();
 }catch(IllegalAccessException e){
  e.printStackTrace();
 }
```

Если такой класс не будет найден - будет брошено исключение ClassNotFoundException, если класс будет найден, но произойдет какая-либо ошибка при создании объекта этого класса - будет брошено исключение InstantiationException, и, наконец, если у вызывающего потока не имеется достаточно прав для создания экземпляров этого класса (что будет проверено менеджером безопасности), будет брошено исключение IllegalAccessException.

## 7.2. SecurityManager - менеджер безопасности

С помощью этого класса приложения могут перед выполнением потенциально опасных операций, определить, является ли операция таковой и может ли она быть выполнена в данном контексте.

Класс SecurityManager содержит много методов с именами, начинающимися с приставки check . Эти методы вызываются различными из библиотек Java перед тем как в них будут выполнены потенциально опасные операции. Типичный такой вызов выглядит примерно следующим образом:

```
SecurityManager security = System.getSecurityManager();
if(security != null){
   security.checkX(…);
}
```

Где X - какой-либо запрос на доступ: Access, Read, Write, Connect, Delete, Exec, Listen и так далее.

Предотвращение вызова производится путем бросания исключения - SecurityException, если вызов операции НЕ разрешен ( кроме метода checkTopLevelWindow, который возвращает boolean значение ).

Для установки менеджера безопасности в качестве текущего, вызывается метод setSecurityManager() в классе System. Соответственно, для его получения, нужно вызвать метод getSecurityManager().

В большинстве случаев, если приложение запускается локально - будут разрешены все действия. В основном менеджер безопасности проявляет себя при работе с апплетами - загруженными из сети.

## 7.3. System

Содержит набор полезных статических методов и полей. Экземпляр этого класса НЕ может быть получен. Среди прочих полезных средств, предоставляемых этим классом, особо стоит отметить потоки стандартных ввода и вывода, поток для вывода ошибок; доступ к внешне определенным свойствам; возможность загрузки файлов и библиотек; утилиту для быстрого копирования порций массивов.

Конечно, наиболее широко используемым является стандартный вывод, доступный через переменную System.out. стандартный вывод можно перенаправить в другой поток (файл, массив байт и т.д., главное, что бы это был объект PrintStream):

```
    public static void main(String[] args) {
     System.out.println("Study Java");
     try{
      PrintStream print = new PrintStream(new
FileOutputStream("d:\\file2.txt"));
      System.setOut(print);
      System.out.println("Study well");
     }catch(FileNotFoundException e){
      e.printStackTrace();
     }
    }
```

При запуске этого кода, на экран будет выведено только

```
Study Java
```

И в файл "d:\file2.txt" будет записано

```
Study well
```

Абсолютно аналогично могут быть перенаправлены стандартный ввод System.in вызовом System.setIn(InputStream) и System.err - вызовом System.setErr(PrintStream).

Следующие методы класса System позволяют работать с некоторыми параметрами системы:

public static void runFinalizersOnExit(boolean value) - выставляет, будет ли производиться вызов метода finalize() у всех объектов (у кого еще не вызывался), когда выполнение программы будет окончено

public static native long currentTimeMillis() - возвращает текущее время. Это время представляется как количество миллисекунд, прошедших с 1-го января 1970 года

public static String getProperty(String key) - возвращает значение свойства с названием key. Что бы получить все свойства, какие определены в системе, можно воспользоваться методом

public static java.util.Properties getProperties() - возвращает объект java.util.Properties, в котором содержатся значения всех определенных системных свойств.

## 7.4. Runtime

Каждому приложению Java сопоставляется экземпляр класса Runtime. Этот объект позволяет взаимодействовать с окружением, в котором запущена Java программа. Получить соответствующий приложению объект Runtime можно вызовом статического метода в этом же классе - Runtime.getRuntime().

Объект этого класса позволяет:

public void exit(int status) - осуществляет завершение программы с кодом завершения status (при использовании этого метода особое внимание нужно уделить обработке исключений - выход будет осуществлен моментально, и в конструкциях try-catch-finally управление в finally передано не будет)

public native void gc() - сигнализирует сборщику мусора о необходимости запуска

public native long freeMemory() - возвращает количество свободной памяти. В некоторых случаях это количество может быть увеличено, если вызвать у объекта Runtime метод gc()

public native long totalMemory() - возвращает суммарное количество памяти, выделенное Java машине. Это количество может из изменяться даже в течении одного запуска, что зависит от реализации платформы на которой запущена Java машина. Так-же, не стоит закладываться на объем памяти, занимаемой одним определенным объектом - эта величина так же зависит от реализации Java машины.

public void loadLibrary(String libname) - загружает библиотеку с указанным именем. Обычно загрузка библиотек производится следующим образом: в классе, использующем native реализации методов, добавляется статический инициализатор, например:

```
static { System.loadLibrary("LibFile"); }
```

Таким образом, когда класс будет загружен и инициализирован, необходимый код для реализации native методов так-же будет загружен. Если будет произведено несколько вызовов загрузки библиотеки с одним и тем-же именем - произведен будет только первый, а все остальные будут проигнорированы.

public void load(String filename) - подгружает файл с указанным названием в качестве библиотеки. В принципе, этот метод работает так-же как и метод load(), только принимает в качестве параметра именно название файла, а не библиотеки, тем самым позволяя загрузить любой файл с native кодом.

public void runFinalization() - производит запуск выполнения методов finalize() у всех объектов, этого ожидающих

public Process exec(String command) - в отдельном процессе запускает команду, представленную переданной строкой. Возвращаемый объект Process может быть использован для управления выполнением этого процесса.

## 7.5. Process

Объекты этого класса получаются вызовом метода exec() у объекта Runtime - запускающего отдельный процесс. Объект класса Process может использоваться для управления процессом и получения информации о нем.

Process - абстрактный класс, определяющий, какие методы должны присутствовать в реализациях для конкретных платформ. Так, объекты класса Process дают возможность:

public InputStream getInputStream() - получить поток ввода из процесса(это будет Piped поток, присоединенный к потоку стандартного выводы процесса)

getErrorStream(), getOutputStream() - методы, аналогичные getInputStream(), но получающие, соответственно стандартные потоки - ошибок и вывода

public void destroy() - уничтожает процесс. Все подпроцессы, запущенные из него, так же будут уничтожены.

public int exitValue() - возвращает код завершения процесса. По соглашению, код завершения равный 0 - означает нормальное завершение.

public int waitFor() - вынуждает текущий поток выполнения приостановиться до тех пор, пока не будет завершен процесс, представленный этим экземпляром Process. Возвращает значение кода завершения процесса.

Даже если в приложении Java не будет ни одной ссылки на объект Process - процесс не будет уничтожен, и будет продолжать асинхронно выполняться до своего завершения. Спецификацией не оговаривается механизм, с помощью которого будет выделяться процессорное время на выполнение процессов Process и потоков Java. Поэтому при проектировании программ НЕ стоит закладываться ни на какой из их них, так-как реализации могут отличаться в зависимости от реализации Java машины, под которой приложение будет запущено.

# 8. Потоки исполнения

В Java поддерживает многопоточность. То есть программа может выполняться в нескольких потоках выполнения команд, которым определенным образом выделяется процессорное время. Всевозможное управление ходом выполнения потоков, и организация их взаимодействия производится с помощью классов Runnable, Thread, ThreadGroup.

## 8.1. Runnable

Runnable - это интерфейс, содержащий один единственный метод без параметров: run(). При использовании объектов, реализующих этот интерфейс, при старте нового потока, управление в нем будет передано именно в этот метод. Любой класс, объекты которого планируется запускать отдельным потоком должны реализовывать этот интерфейс.

## 8.2. Thread

Объекты этого класса представляют потоки.

Что бы реализовать выполнение некоторого кода в отдельном потоке, можно пойти двумя путями:

а) Написать класс, реализующий интерфейс Runnable, поместив код для выполнения в метод run(). Создать объект класса Thread, передав конструктору объект этого класса. Запустить выполнение в отдельном потоке, вызвав у объекта Thread метод start().

Пример:

```
public static void main(String[] args) {
    class RunCode implements Runnable{
     public void run(){
      System.out.println("RunCode.run() begins");
      System.out.println("RunCode.run() ends");
     }
    }
    Runnable run = new RunCode();
    Thread thread = new Thread(run);
    thread.start();
    System.out.println("main finish");
}
```

В результате выполнения на экран может быть получено примерно следующее:

```
RunCode.run() begins
main finish
RunCode.run() ends
```

Не обязательно именно в таком порядке - фраза "main finish" может быть выведена как самой первой, так и самой последней. Это обусловлено тем, что в метод run() в объекте класса RunCode выполняется в отдельном потоке. Так-как спецификацией механизм распределения процессорного времени между потоками не ограничен жесткими рамками, то может получиться так, сначала выполнится метод run(), и только после этого управление будет передано главному потоку. А может получиться и наоборот - сначала закончит выполнение главный поток и только потом управление будет передано методу run() объекта RunCode.

b) Написать класс, унаследовав его от Thread. При этом в метод run() поместить код, который должен выполняться в отдельном потоке. После этого достаточно будет создать объект этого класса, и вызвать у него метод start().

Предыдущий пример, тогда будет выглядеть следующим образом:

```
public static void main(String[] args) {
    class RunCode extends Thread{
     public void run(){
      System.out.println("RunCode.run() begins");
      System.out.println("RunCode.run() ends");
     }
    }
    Thread thread = new RunCode();
    thread.start();
    System.out.println("main finish");
}
```

Функциональных различий в использовании двух приведенных путях создания новых потоков нет. Класс Thread реализует интерфейс Runnable, и если объект этого класса был создан без передачи конструктору объекта Runnable, в качестве этого объекта будет использоваться сам объект Thread (через ссылку this).

Итак, для управления выполнением потока, класс Thread имеет следующие методы:

public void start() - производит запуск выполнения нового потока

public final void join() - приостанавливает выполнение потока, из которого был вызван этот метод до тех пор, пока не закончит выполнение поток, у объекта Thread которого был вызван этот метод

public static void yield() - поток, из которого вызван этот метод, временно приостановится(отдаст процессорное время), что бы дать возможность выполняться другим потокам

public static void sleep(long millis) - поток, из которого вызван этот метод, перейдет в состояние "сна" на время millis миллисекунд, после чего сможет продолжить выполнение. При этом нужно учесть, что поток именно сможет продолжить выполнение, а НЕ продолжит. То есть через время millis миллисекунд, этому потоку может быть выделено процессорное время (механизм распределения определяется реализацией Java-машины). Правильней было бы говорить, что поток продолжит выполнение НЕ раньше чем через время millis миллисекунд.

еще несколько методов, которые, объявлены deprecated, и рекомендуется избегать их использовать. Это: suspend()-временно прекратить выполнение, resume() - продолжить выполнение (приостановленное вызовом suspend()), stop() - остановить выполнение потока

При вызове метода stop(), в потоке, который представляет этот объект Thread, будет брошена ошибка ThreadDeath. Это ошибка, класс которой ThreadDeath унаследован от Error. Если ошибка не будет обработана в программе и, соответственно, произойдет прекращение работы потока, в top-level обработчике сообщения о ненормальном завершении выведено не будет, так-как такое завершение рассматривается как нормальное. Если в программе эта ошибка будет перехвачена и обработана (например, произведены некоторые действия по закрытию потоков и т.д.), то очень важно позаботиться о том, что бы эта же ошибка была брошена дальше - что бы поток действительно закончил свое выполнение. Класс ThreadDeath специально унаследован от Error, а не Exception, так-как очень часто используется перехват исключений именно класса Exception, и, таким, образом, поток может продолжить выполнение.

Так же Thread позволяет выставлять такие свойства потока, как:

Name - значение типа String, которое можно использовать для более наглядного обращения с потоками в группе

Daemon - выполнение программы не будет прекращено до тех пор пока выполняется хотя бы один НЕ daemon поток

Priority - определяет приоритет потока. В классе Thread определены константы, задающие минимальное и максимальное значения для приоритетов потока - MIN_PRIORITY и MAX_PRIORITY , а так же значение приоритета по умолчанию - NORM_PRIORITY .

Эти свойства могут быть изменены только до того момента, когда поток будет запущен, то есть, вызван метод start() объекта Thread.

Получить эти значения, можно, конечно же, в любой момент жизни потока - и после его запуска, и после прекращения выполнения. Так же, можно и узнать, в каком состоянии сейчас находится поток - вызовом методов isAlive() - выполняется ли еще, isInterrupted() - прерван ли.

### 8.3. ThreadGroup

Для того, что бы отдельный поток не мог оказаться "невоспитанным" и начать останавливать и прерывать все потоки подряд, введено понятие группы. Поток может оказывать влияние только на потоки, которые находятся в одной с ним группе. Группу потоков представляет класс ThreadGroup. Такая организация позволяет защитить потоки от нежелательного внешнего воздействия. В группе потоков так же могут содержаться другие группы потоков, и так далее, организуя, таким образом, некоторое дерево, в котором каждый объект ThreadGroup, за исключением коневого, имеет родителя.

## 9. Исключения

Подробно механизм использования исключений описан в соответствующей лекции. Здесь остановимся только на том, что базовым классом для всех исключений является класс Throwable. Любой класс, который планируется использовать как исключение, должен явным или неявным образом быть от него унаследованным. Класс Throwable, а так же наиболее значимые его наследники - классы: Error, Exception, RuntimeException содержатся именно в пакете java.lang.

## 10. Заключение

В этой главе Вы получили представление о назначении и возможностях классов, представленных в пакете java.lang. Как Вы теперь знаете, пакет java.lang автоматически импортируется во все Java программы и содержит фундаментальные классы и интерфейсы, которые составляют основу для других пакетов Java.

Были рассмотрены все наиболее важные классы пакета java.lang:
Object, Class – основные классы, представляющие объект и класс объектов;

классы-обертки (Wrapper классы) – так как многие классы работают именно с объектами, иногда бывает необходимо представлять значения примитивных типов Java в виде объектов; в таких случаях используют классы-обертки;

Math – класс, предоставляющий набор статических методов, реализующих наиболее распространенные математические функции;

String и StringBuffer – классы для работы со строками;

System, Runtime, Process, ClassLoader, SecurityManager – системные классы, помогающие взаимодействовать с программным окружением (System, Runtime, Process), загружать классы в JVM (ClassLoader) и управлять безопасностью (SecurityManager);

Thread, ThreadGroup, Runnable – типы, обеспечивающие работу с потоками исполнения в Java;

Throwable, Error, Exception, RuntimeException – базовые классы для всех исключений.

# 11. Контрольные вопросы

13-1. В чем различие между следующими кусками кода:

1.
```
String s1 = "abc";
String s2 = new String("abc");
boolean result = (s1==s2);
```

2.
```
String s1 = new String("abc");
String s2 = new String("abc");
boolean result = (s1.equals(s2));
```

a.) Значение result в первом случае будет равно false, в то время как во втором – true. В первом случае сравниваются значения ссылок s1 и s2, во втором, проверяется, идентичны ли объекты, на которые указывают эти ссылки. При этом s1 и s2 указывают на разные объекты, так как объект s2 был получен применением оператора new, вследствие чего был создан новый объект. В случае если бы s2 был получен тем же способом, что и s1 (то есть String s2 = "abc";), то обе ссылки s1 и s2 указывали бы на один и тот же объект, так как Java не создает новый экземпляр для строковых литералов, если таковой уже имеется. Вместо этого ссылка будет указывать на уже существующий объект.

13-2. Какие условия должны быть выполнены при переопределении метода equals()?

a.) Переопределение метода equals() предполагает выполнение следующих правил:

1. a) рефлексивность: для любой объектной ссылки x, вызов x.equals(x) возвращает true
2. b) симметричность: для любых объектных ссылок x и y, вызов x.equals(y) возвращает true тогда и только тогда, если вызов y.equals(x) возвращает true
3. c) транзитивность: для любых объектных ссылок x, y и z, если x.equals(y) возвращает true и y.equals(z) возвращает true, тогда вызов x.equals(z) должен вернуть true
4. d) непротиворечивость: для любых объектных ссылок x и y, многократные последовательные вызовы x.equals(y) возвращают одно и то же значение (либо всегда true либо всегда false)
5. e) для любой не равной null объектной ссылки x, вызов x.equals(null) должен вернуть значение false

13-3. Как формально формулируются правила, которым должна следовать реализация метода hashCode()? Как реализован этот метод в классе Object?

a.) Для реализаций метода hashCode() должны быть выполнены следующие правила:

1. a) в одном запуске программы, для одного объекта при вызове метода hashCode(), должно возвращаться одно и то же int значение, если между этими вызовами НЕ были затронуты данные, используемые для проверки объектов на идентичность в методе equals(). Это число НЕ обязано быть одним и тем же при повторном запуске той же программы, даже если все данные будут идентичны
2. b) если два объекта идентичны, то есть вызов метода equals(Object) возвращает true, тогда вызов метода hashCode() у каждого из этих двух объектов должен возвращать одно и то же значение
3. c) если два объекта различны, то есть вызов метода equals(Object) возвращает false, тогда различие их хеш-кодов желательно, но НЕ обязательно. Различие в хеш-кодах для НЕ идентичных объектов нужно только для обеспечения хорошей производительности при использовании этих объектов в хеш-таблицах.

В классе Object метод hashCode() возвращает значение адреса, где хранится объект. Такая реализация удовлетворяет всем правилам и переопределение метода hashCode() в подклассах Object не требуется, если не был переопределен метод equals().

13-4. Если был переопределен метод equals(), какой еще метод, возможно, понадобится переопределить соответствующим образом, чтобы объекты рассматриваемого класса могли быть корректно использованы в хэш-таблицах?

a.) Среди правил, которым должна удовлетворять реализация метода hashCode(), есть следующее: “если два объекта идентичны, то есть вызов метода equals(Object) возвращает true, тогда вызов метода hashCode() у каждого из этих двух объектов должен возвращать одно и то же значение”.

Соответственно, если метод equals() был переопределен, и объекты этого класса планируется использовать в хеш-таблицах, то в соответствии с указанным правилом, метод hashCode() понадобится переопределить таким образом, что бы возвращались одинаковые значения для объектов, вызов метода equals() для которых возвращает true.

13-5. Как реализованы в классе Object методы equals(), toString(), hashCode() ?

a.) В классе Object методы equals(), toString() и hashCode() имеют следующие реализации:

equals() – возвращает true, если ссылки на объекты совпадают

toString() – возвращает строку, которая составляется следующим образом: название класса, символ ‘@’, значение, возвращаемое вызовом метода hashCode(), представленное в шестнадцатеричном виде

hashCode() – имеет native реализацию, возвращающую адрес, по которому хранится объект.

13-6. Какие объекты могут быть клонированы?

a.) Если клонирование производится встроенным методом Object.clone(), то такие классы необходимо специальным образом помечать, указывая, что они реализуют интерфейс Cloneable. Кроме этого, класс может переопределить метод clone() собственным образом, и обойтись без этого интерфейса.

13-7. Какие существуют ограничения на использование метода newInstance() объектов типа Class для создания экземпляров соответствующего класса?

a.) Вызов этого метода создает объект класса, который представляется данным экземпляром Class. Создание будет происходить с помощью конструктора без параметров. Соответственно, что бы создание прошло успешно, такой конструктор должен иметься в классе, а сам класс не должен быть абстрактным.

13-8. Для каких примитивных типов Java существуют классы-обертки? Что будет получено в результате выполнения: (new Integer(1)).equals(new Byte(1)) ?

a.) Классы-обертки существуют для всех примитивных типов Java – byte, short, char, int, long, float, double, boolean (обертка существуют даже для «отсутствия типа» – void).

Вызов (new Integer(1)).equals(new Byte(1)) вернет значение false, так как метод equals() в классах-обертках реализован таким образом, что сначала производится сравнение классов сравниваемых объектов, и если они не совпадают – возвращается значение false.

13-9. В чем особенность класса-обертки для void?

a.) Этот класс, в отличие от остальных классов-оберток, НЕ реализует интерфейс java.io.Serializable. Он не имеет открытого конструктора. Более того, экземпляр этого класса вообще не может быть получен.

13-10. Какие модификаторы присутствуют в определении класса Math? Можно ли от него наследоваться? Можно ли создавать экземпляры этого класса?

a.) Класс Math определен с модификаторами public и final. Соответственно, наследование от класса Math не возможно. Кроме того, единственный конструктор этого класса имеет модификатор доступа private и поэтому может быть вызван только из самого класса Math (впрочем, в классе Math нигде нет вызова этого конструктора). Все методы этого класса – статические.

13-11. В чем проявляется сходство String и примитивных типов Java?

a.) Во-первых, для объектов только этого типа существуют соответствующие литералы, которыми можно инициализировать их значения. Во-вторых, только для одного ссылочного типа String определен оператор “+”.

Кроме того, поскольку класс String является final, String-переменные всегда хранят значения точно такого же типа. Объекты класса String являются

неизменяемыми, поэтому значение переменной не может измениться, если ее передать в качестве аргумента при вызове метода.

13-12. Какой класс используется для представления модифицируемых строк?

a.) Если класс String представляет НЕ модифицируемые строки, то класс StringBuffer позволяет изменять значение строки, которую его объект содержит.

13-13. Какие классы и интерфейсы, необходимые для поддержки многопоточности, определены в пакете java.lang?

a.) Основной класс для обеспечения многопоточности – java.lang.Thread, объекты которого соответствуют потокам. Потоки могут быть объединены в группы потоков, что представляется объектами класса java.lang.ThreadGroup. Создать новый поток можно, унаследовав класс от Thread, либо реализовав интерфейс Runnable.

13-14. Классы каких базовых исключений определены в пакете java.lang ?

a.) Все классы базовых исключений содержатся в пакете java.lang – Throwable, Error, Exception, RuntimeException.

# Программирование на Java

## Лекция 14. Пакет java.util

### 20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Эта лекция посвящена пакету java.util, в котором содержится множество вспомогательных классов и интерфейсов. Они настолько удобны, что практически любая программа использует эту библиотеку. Центральную часть в изложении занимает тема контейнеров или коллекций – классов, хранящих упорядоченные ссылки на ряд объектов. Они были серьезно переработаны в ходе создания версии Java2. Также рассматриваются классы для работы с датой, для генерации случайных чисел, для обеспечения поддержки многих национальных языков в приложении и другие.

# Оглавление

# Лекция 14. Пакет java.util

## Содержание лекции.

# 1. Введение

В Java имеется большое количество вспомогательных классов. Далее мы рассмотрим наиболее важные классы пакета java.util.

# 2. Работа с датами и временем

## 2.1. Класс Date

Класс Date изначально предоставлял набор функций для работы с датой - для получения текущего года, месяца и т.д. однако сейчас все эти методы не рекомендованы к использованию и практически всю функциональность для этого предоставляет класс Calendar. Класс Date так же определен в пакете java.sql поэтому желательно указывать полностью квалифицированное имя класса Date.

Существует несколько конструкторов класса Date однако рекомендовано к использованию два

Date() и Date(long date)

второй конструктор использует в качестве параметра значение типа long который указывает на количество миллисекунд прошедшее с 1 Января 1970, 00:00:00 по Гринвичу. Первый конструктор создает дату использует текущее время и дату (т.е. время выполнения конструктора). Фактически это эквивалентно второму варианту new Date(System.current-TimeMillis); Можно уже после создания экземпляра класса Date использовать метод setTime(long time), для того, что бы задать текущее время.

Для сравнения дат служат методы after(Date date), before(Date date) которые возвращают булевское значение в зависимости от того выполнено условие или нет. Метод compareTo(Date anotherDate) возвращает значение типа int которое равно -1 если дата меньше сравниваемой, 1 если больше и 0 если даты равны. Метод toString() представляет строковое представление даты, однако для форматирования даты в виде строк рекомендуется пользоваться классом SimpleDateFormat определенном в пакте java.text

## 2.2. Классы Calendar и GregorianCalendar

Более развитые средства для работы с датами представляет класс Calendar. Calendar является абстрактным классом. Для различных платформ реализуются конкретные подклассы календаря. На данный момент существует реализация Грегорианского календаря - GregorianCalendar. Экземпляр этого класса получается вызовом статического метода getInstance(), который возвращает экземпляр класса Gregorian. Подклассы класса Calendar должны интерпретировать объект Date по разному. В будущем предполагается реализовать так же лунный календарь, используемый в некоторых странах.

Calendar обеспечивает набор методов позволяющих манипулировать различными "частями" даты, т.е. получать и устанавливать дни, месяцы, недели и т.д.

Если при задании параметров календаря упущены некоторые параметры, то для них будут использованы значения по умолчанию для начала отсчета. т.е.

YEAR = 1970, MONTH = JANUARY, DATE = 1 и т.д.

Для считывания, установки манипуляции различных "частей" даты используются методы get(int filed), set(int filed, int value), add(int field, int amount), roll(int field, inr amount), переменная типа int с именем filed указывает на номер поля с которым нужно произвести операцию. Для удобства все эти поля определены в Calendar, как статические константы типа int.

Рассмотрим подробнее порядок выполнения перечисленных методов.

set(int field,int value)

Как уже отмечалось ранее данный метод производит установку какого - либо поля даты. На самом деле после вызова этого метода, немедленного пересчета даты не производится. Пересчет даты будет осуществлен только после вызова методов get(), getTime() или TimeInMillis(). Т.о. последовательная установка нескольких полей, не вызовет не нужных вычислений. Помимо этого появляется еще один интересный эффект. Рассмотрим следующий пример. Предположим, что дата установлена на последний день августа. Необходимо перевести ее на последний день сентября. Если внутреннее представление даты изменялось бы после вызова метода set, то при последовательной установке полей мы получили бы вот такой эффект.

```
public class Test {

    public Test() {
    }
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");
         Calendar cal = Calendar.getInstance();
         cal.set(Calendar.YEAR,2002);
         cal.set(Calendar.MONTH,Calendar.AUGUST);
         cal.set(Calendar.DAY_OF_MONTH,31);
         System.out.println(" Initialy set date:        " +
sdf.format(cal.getTime()));
         cal.set(Calendar.MONTH,Calendar.SEPTEMBER);
         System.out.println(" Date with month changed : " +
sdf.format(cal.getTime()));
         cal.set(Calendar.DAY_OF_MONTH,30);
         System.out.println(" Date with day changed :   " +
sdf.format(cal.getTime()));

    }
}
```

```
 Initialy set date:        2002 August 31 22:57:47
 Date with month changed : 2002 October 01 22:57:47
 Date with day changed :   2002 October 30 22:57:47
```

Как видно, в данном примере, при изменении месяца день месяца остался неизменным и было унаследовано его предыдущее значение. Но т.к. в сентябре 30 дней, то дата автоматически была переведена на 1 октября, и, когда было установлено 30 число, то оно относилось бы уже к октябрю месяцу. В следующем примере считывание даты не производится, соответственно ее вычисление не производится, до тех пор пока все поля не установлены.

```
public class Test {

    public Test() {
    }
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");
         Calendar cal = Calendar.getInstance();
         cal.set(Calendar.YEAR,2002);
         cal.set(Calendar.MONTH,Calendar.AUGUST);
         cal.set(Calendar.DAY_OF_MONTH,31);
         System.out.println(" Initialy set date:                  " +
sdf.format(cal.getTime()));
         cal.set(Calendar.MONTH,Calendar.SEPTEMBER);
         cal.set(Calendar.DAY_OF_MONTH,30);
         System.out.println(" Date with day and month changed :   " +
sdf.format(cal.getTime()));

    }
}
```

```
 Initialy set date:                  2002 August 31 23:03:51
 Date with day and month changed :   2002 September 30 23:03:51
```

add(int field,int delta)

Добавляет некоторое смещение к существующей величине поля. В принципе то же самое можно сделать с помощью set(f, get(f) + delta)

В случае использования метода add следует помнить о двух правилах.

1. Если величина поля изменения выходит за диапазон возможных значений данного поля, то производится деление по модулю данной величины, частное суммируется со следующим по старшинству полем.

2. Если изменяется одно из полей, при этом после изменения младшее по отношению к изменяемому полю, принимает некорректное значение, то оно изменяется, на, которое максимально близко к "старому".

```
public class Test {

    public Test() {
    }
```

```
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");
         Calendar cal = Calendar.getInstance();
         cal.set(Calendar.YEAR,2002);
         cal.set(Calendar.MONTH,Calendar.AUGUST);
         cal.set(Calendar.DAY_OF_MONTH,31);
         cal.set(Calendar.HOUR_OF_DAY,19);
         cal.set(Calendar.MINUTE,30);
         cal.set(Calendar.SECOND,00);
         System.out.println("Current date: " + sdf.format(cal.getTime()));
         cal.add(Calendar.SECOND,75);
         System.out.println("Current date: " + sdf.format(cal.getTime()));
         cal.add(Calendar.MONTH,1);
         System.out.println("Current date: " + sdf.format(cal.getTime()));
    }
}
```

```
Current date: 2002 August 31 19:30:00
Rule 1: 2002 August 31 19:31:15
Rule 2: 2002 September 30 19:31:15
```

roll(int field,int delta)

Добавляет некоторое смещение к существующей величине поля и не производит изменения старших полей. Рассмотрим приведенный ранее пример, но с использованием метода roll

```
public class Test {

    public Test() {
    }
    public static void main(String[] args) {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");
         Calendar cal = Calendar.getInstance();
         cal.set(Calendar.YEAR,2002);
         cal.set(Calendar.MONTH,Calendar.AUGUST);
         cal.set(Calendar.DAY_OF_MONTH,31);
         cal.set(Calendar.HOUR_OF_DAY,19);
         cal.set(Calendar.MINUTE,30);
         cal.set(Calendar.SECOND,00);
         System.out.println("Current date: " + sdf.format(cal.getTime()));
         cal.roll(Calendar.SECOND,75);
         System.out.println("Rule 1: " + sdf.format(cal.getTime()));
         cal.roll(Calendar.MONTH,1);
         System.out.println("Rule 2: " + sdf.format(cal.getTime()));
    }
}
```

```
Current date: 2002 August 31 19:30:00
```

```
Rule 1: 2002 August 31 19:30:15
Rule 2: 2002 September 30 19:30:15
```

Как видно из результатов работы приведенного выше кода, действие правила 1 изменилось, по сравнению с методом add, а правило 2 действует так же.

## 2.3. Класс TimeZone

Класс TimeZone предназначен для совместного использования с классами Calendar и DateFormat. Класс абстрактный, поэтому нельзя создать конкретный экземпляр этого с помощью конструктора. Для этого определен статический метод getDefault(), который возвращает экземпляр класса TimeZone с настройками взятыми из настроек операционной системы под управлением которой работает JVM. Для того, что бы получить экземпляр TimeZone с конкретными настройками, можно воспользоваться статическим методом getTimeZone(String ID), в качестве параметра, которому передается наименование конкретного временного пояса, для которого необходимо получить объект TimeZone. Нигде не определено публичного набора полей определяющих возможный набор параметров для getTimeZone. Вместо этого определен статический метод String[] getAvailableIds() который возвращает массив строк с возможными параметрами для getTimeZone. Можно так определить набор возможных параметров для конкретного временного пояса (рассчитывается относительно Гринвича) String[] getAvailableIds(int offset);

Рассмотрим пример в котором на консоль последовательно выводятся

- временная зона по умолчанию
- список всех возможных временных зон
- список временных зон которые совпадают с текущей временной зоной.

Следует обратить внимание что на консоль выводится полное наименование временной зоны. Именно в таком формате следует передавать параметр для getTimeZone(). Хотя и допускается указание строки, которую возвращает метод getDisplayName(), так же возможно указание временного пояса в формате "GMT-8:00".

```
public class Test  {
    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        TimeZone tz = TimeZone.getDefault();
        int rawOffset = tz.getRawOffset();
        System.out.println("Current TimeZone" + tz.getDisplayName() +
tz.getID() + "\n\n");

        // Dispaly all available TimeZones
        System.out.println("All Available TimeZones \n");
        String[] idArr = tz.getAvailableIDs();
        for(int cnt=0;cnt < idArr.length;cnt++){
            tz = TimeZone.getTimeZone(idArr[cnt]);
           System.out.println(test.padr(tz.getDisplayName() + tz.getID(),64)
```

```
 + "  raw offset=" + tz.getRawOffset() + ";hour offset=(" + tz.getRawOffset()/
 (1000 * 60 * 60 ) + ")");
        }

        // Dispaly all available TimeZones same as for Moscow
        System.out.println("\n\n TimeZones same as for Moscow \n");
        idArr = tz.getAvailableIDs(rawOffset);
        for(int cnt=0;cnt < idArr.length;cnt++){
            tz = TimeZone.getTimeZone(idArr[cnt]);
            System.out.println(test.padr(tz.getDisplayName()+ tz.getID(),64)
 + "  raw offset=" + tz.getRawOffset() + ";hour offset=(" + tz.getRawOffset()/
 (1000 * 60 * 60 ) + ")");
        }

    }

    String padr(String str,int len){
        if(len - str.length() > 0){
            char[] buf = new char[len - str.length()];
            Arrays.fill(buf,' ');
            return str + new String(buf);
        }else{
            return str.substring(0,len);
        }

    }
}
```

```
Current TimeZone Moscow Standard TimeEurope/Moscow

All Available TimeZones
```

… полный список временных зон приведен в приложении XXX …

TimeZones same as for Moscow

```
Eastern African TimeAfrica/Addis_Aba  raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Asmera     raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Dar_es_Sa  raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Djibouti   raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Kampala    raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Khartoum   raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Mogadishu  raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Nairobi    raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Aden         raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Baghdad      raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Bahrain      raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Kuwait       raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Qatar        raw offset=10800000;hour offset=(3)
```

```
Arabia Standard TimeAsia/Riyadh       raw offset=10800000;hour offset=(3)
Eastern African TimeEAT               raw offset=10800000;hour offset=(3)
Moscow Standard TimeEurope/Moscow     raw offset=10800000;hour offset=(3)
Eastern African TimeIndian/Antananar  raw offset=10800000;hour offset=(3)
Eastern African TimeIndian/Comoro     raw offset=10800000;hour offset=(3)
Eastern African TimeIndian/Mayotte    raw offset=10800000;hour offset=(3)
```

## 2.4. Класс SimpleTimeZone

Класс SimpleTimeZone являясь потомком TimeZone реализует его абстрактные методы и предназначен для использования в настройках использующих Григорианский календарь. В большинстве случаев нет необходимости создавать экземпляр этого класса с помощью конструктора. Вместо этого лучше использовать статические методы которые возвращают экземпляр класса TimeZone рассмотренные в предыдущем параграфе. Единственная, пожалуй, причина для использования конструктора - необходимость задания нестандартных правил перехода на зимнее и летнее время.

В классе SimpleTimeZone определено три конструктора. Рассмотрим наиболее полный, с точки зрения функциональности) вариант, который помимо временной зоны задает летнее и зимнее время.

```
public SimpleTimeZone(int rawOffset,
                      String ID,
                      int startMonth,
                      int startDay,
                      int startDayOfWeek,
                      int startTime,
                      int endMonth,
                      int endDay,
                      int endDayOfWeek,
                int endTime)
```

rawOffset - временное смещение относительно гринвича

ID - идентификатор временной зоны. (см. пред.параграф)

startMonth - месяц перехода на летнее время

startDay - день месяца перехода на летнее время*

startDayOfWeek - день недели перехода на летнее время*

startTime - время перехода на летнее время (указывается в миллисекундах)

endMonth - месяц окончания действия летнего времени

endDay - день окончания действия летнего времени*

endDayOfWeek - день недели окончания действия летнего времени*

endTime - время окончания действия летнего времени (указывается в миллисекундах)

Месяц перехода времени начинает отсчитываться с нуля. Для этих целей можно применять константы определенные в классе Calendar, например Calendar.JANUARY

Перевод часов на зимний и летний вариант исчисления времени определяется специальным правительственным указом. Обычно переход на летнее время происходит в 2 часа в последнее воскресенье марта, а переход на зимнее время - в 3 часа в последнее воскресенье октября.

Алгоритм расчета таков

- если startDay 1 и установлен день недели, то будет вычисляться первый день недели startDayOfWeek месяца startMonth ( например первое воскресенье)
- если startDay -1, и установлен день недели, то будет вычисляться последний день недели startDayOfWeek месяца startMonth (например последнее воскресенье)
- если день недели startDayOfWeek установлен в 0, то будет вычисляться число startDay конкретного месяца startMonth
- для того что бы установить день недели после конкретного числа специфицируется отрицательное значение дня недели. Например, что бы указать первый понедельник после 23 февраля используется вот такой набор startDayOfWeek=-MONDAY, start-Month=FEBRUARY, startDay=23
- для того что бы указать последний день недели перед каким-либо числом, указывается отрицательное значение этого числа и отрицательное значение дня недели. Например, для того что бы указать последнюю субботу перед 23 февраля необходимо задать такой набор параметров startDayOfWeek=- -SATURDAY, startMonth=FEBRUARY, startDay=-23
- все вышеперечисленное относится так же и к окончанию действия летнего времени.

Рассмотрим пример получения текущей временной зоны с заданием перехода на зимнее и летнее время для России по умолчанию.

```
public class Test  {
    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        SimpleTimeZone stz = new SimpleTimeZone(
                TimeZone.getDefault().getRawOffset()
               ,TimeZone.getDefault().getID()
               ,Calendar.MARCH
               ,-1
               ,Calendar.SUNDAY
               ,test.getTime(2,0,0,0)
               ,Calendar.OCTOBER
               ,-1
               ,Calendar.SUNDAY
               ,test.getTime(3,0,0,0)
                );
        System.out.println(stz.toString());
    }
    int getTime(int hour,int min,int sec,int ms){
        return hour * 3600000 + min * 60000 + sec * 1000 + ms;
```

```
    }
}
```

```
java.util.SimpleTimeZone[id=Europe/Moscow,offset=10800000,dstSavings=3600000,useDaylight=true,startYear=
0,startMode=2,startMonth=2,startDay=-1,startDayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=
2,endMonth=9,endDay=-1,endDayOfWeek=1,endTime=10800000,endTimeMode=0]
```

# 3. Интерфейс Observer и класс Observable

Интерфейс Observable определяет всего один метод update(Observable o, Object arg), который вызывается когда обозреваемый объект изменяется.

Класс Observer предназначен для поддержки обозреваемого объекта в парадигме MVC (model-view-controller), которая, как и другие проектные решения и шаблоны, рассмотрена в специальной литературе. Этот класса должен быть унаследован, если возникает необходимость в том, отслеживать состояние какого - либо объекта. Обозреваемый объект может иметь несколько обозревателей. Соответственно они должны реализовать интерфейс Observable.

После того как в состоянии обозреваемого объекта что-то меняется, то необходимо вызвать метод notifyObservers, который в свою очередь вызывает методы update у каждого обозревателя.

Порядок в котором вызываются методы update обозревателей заранее не определен. Реализация по умолчанию подразумевает их вызов в порядке регистрации. Регистрация осуществляется с помощью метода addObserver(Observer o); Удаление обозревателя из списка осуществляется с помощью deleteObserver(Observer o). Перед вызовом notifyObservers, необходимо вызвать метод setChanged, который устанавливает признак того, что обозреваемый объект был изменен.

Рассмотрим пример организации взаимодействия классов.

```
public class TestObservable extends java.util.Observable {
    private String name = "";
    public TestObservable(String name) {
        this.name = name;
    }

    public void modify(){
        setChanged();
    }

    public String getName(){
        return name;
    }
}
public class TestObserver  implements java.util.Observer{
    private String name = "";
```

```
    public TestObserver(String name) {
        this.name = name;
    }

    public void update(java.util.Observable o,Object arg){
        String str = "Called update of " + name;
        str += " from " + ((TestObservable)o).getName();
        str += " with argument " + (String)arg;
        System.out.println(str);
    }
}

public class Test {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        TestObservable to = new TestObservable("Observable");
        TestObserver o1 = new TestObserver("Observer 1");
        TestObserver o2 = new TestObserver("Observer 2");
        to.addObserver(o1);
        to.addObserver(o2);
        to.modify();
        to.notifyObservers("Notify argument");
    }
}
```

В результате работы на консоль будет выведено.

```
Called update of Observer 2 from Observable with argument Notify argument
Called update of Observer 1 from Observable with argument Notify argument
```

На практике использовать Observer не всегда удобно, т.к. в Java отсутствует множественное наследование, и Observer необходимо наследовать в самом начале построения иерархии классов. В качестве варианта, можно предложить определить интерфейс, задающий функциональность, сходную с Observer, и реализовать его в нужном вам классе.

# 4. Коллекции

Зачастую в программе необходимо сгруппировать объекты в некую логическую структуру , определение которой производится во время исполнения. Наиболее простой способ сделать это с помощью массивов. Однако, несмотря на то, что это достаточно эффективное решение для многих случаев, оно имеет и ограничения. Так в массиве возможно обращение к его элементу только по его номеру (индексу). Так же необходимо знать количеств объектов организуемых в массив до его создания. Массивы существовали в Java изначально, было определено так же два класса для организации боле эффективной работы с наборами объектов Hashtable и Vector. В JDK 1.2 набор классов поддерживающих работу с коллекциями был существенно расширен.

Следует обратить внимание, что коллекции предназначены для работы с объектами. В то время как, массивы могут содержать как простые типы, так и ссылки на объекты, то классы коллекций содержат только ссылки на объекты. Однако если возникает необходимость использования простых типов в коллекциях, то необходимо использовать для этого классы-обертки.

Существует несколько различных типов классов-коллекций. Все они разработаны, по возможности следуя единой логике и определенным интерфейсам, и там где это возможно, манипулирование ими унифицировано. Однако все коллекции отличаются внутренними механизмами хранения, скоростью доступа к элементам, потребляемой памятью и другими деталями. Например в некоторых коллекциях объекты (так же называемые элементами коллекций), могут быть упорядочены, в некоторых нет. В некоторых типах коллекций допускается дублирование ссылок на объект в нет. Далее мы рассмотрим каждый из классов - коллекций

Классы обеспечивающие манипулирование коллекциями объектов, сведены в пакет java.util

## 4.1. Интерфейсы

### 4.1.1. Интерфейс Collection

Является корнем всей иерархии классов-коллекций. Он определяет базовую функциональность любой коллекции - набор методов которые позволяют добавлять, удалять, выбирать элементы коллекции. Классы которые имплементируют интерфейс Collection, могут содержать дубликаты и пустые (null) значения.

AbstractCollection, являясь абстрактным классом обеспечивает, служит основой для создания конкретных классов коллекций и содержит реализацию некоторых методов определенных в интерфейсе Collection.

### 4.1.2. Интерфейс Set

Классы которые реализуют этот интерфейс не разрешают наличие дубликатов. В коллекции этого типа допускается наличие только одной ссылки типа null. Интерфейс Set расширяет интерфейс Collection т.о. любой класс имплементирующий Set реализует все методы определенные в Collection. Любой объект добавляемый в Set должен реализовать метод equals для того, что бы его можно было сравнить с другими.

AbstractSet являясь абстрактным классом представляет из себя основу для реализации различных вариантов интерфейса Set

### 4.1.3. Интерфейс List

Классы которые реализуют этот интерфейс содержат упорядоченную последовательность объектов (Объекты хранятся в том порядке в котором они были добавлены). D JDK 1.2 был переделан класс Vector, так, что он теперь реализует интерфейс List. Интерфейс List расширяет интерфейс Collection т.о. любой класс имплементирующий List реализует все методы определенные в Collection, в то же время вводятся новые методы которые позволяют добавлять и удалять элементы из списка. List обеспечивает так же ListIterator который позволяет перемещаться как вперед, так и назад, по элементам списка.

AbstractList являясь абстрактным классом представляет из себя основу для реализации различных вариантов интерфейса List

Следует помнить, что в пакете java.awt так же определен класс List используемый для представления списка вариантов. Т.о. что бы избежать путаницы, следует использовать полностью квалифицированное имя java.util.List

### 4.1.4. Интерфейс Map

Классы которые реализуют этот интерфейс хранят набор неупорядоченный набор объектов парами ключ/значение. Каждый ключ должен быть уникальным. Hashtable после модификации в JDK 1.2 реализует интерфейс Map. Порядок следования пар ключ/значение не определен.

Интерфейс Map не расширяет интерфейс Collection. AbstractMap являясь абстрактным классом представляет из себя основу для реализации различных вариантов интерфейса Map

Следует обратить внимание, что List и Set расширяют интерфейс Collection, а Map нет.

### 4.1.5. Интерфейс SortedSet

Этот интерфейс расширяет Set требуя, что бы содержимое набора было упорядочено. Только объект имплементирующие SortedSet могут содержать объекты которые реализуют интерфейс Comparator или могут сравниваться с использованием внешнего объекта реализующего интерфейс comparator.

### 4.1.6. Интерфейс SortedMap

Этот интерфейс расширяет Map требуя, что бы содержимое коллекции было упорядочено по значениям ключей.

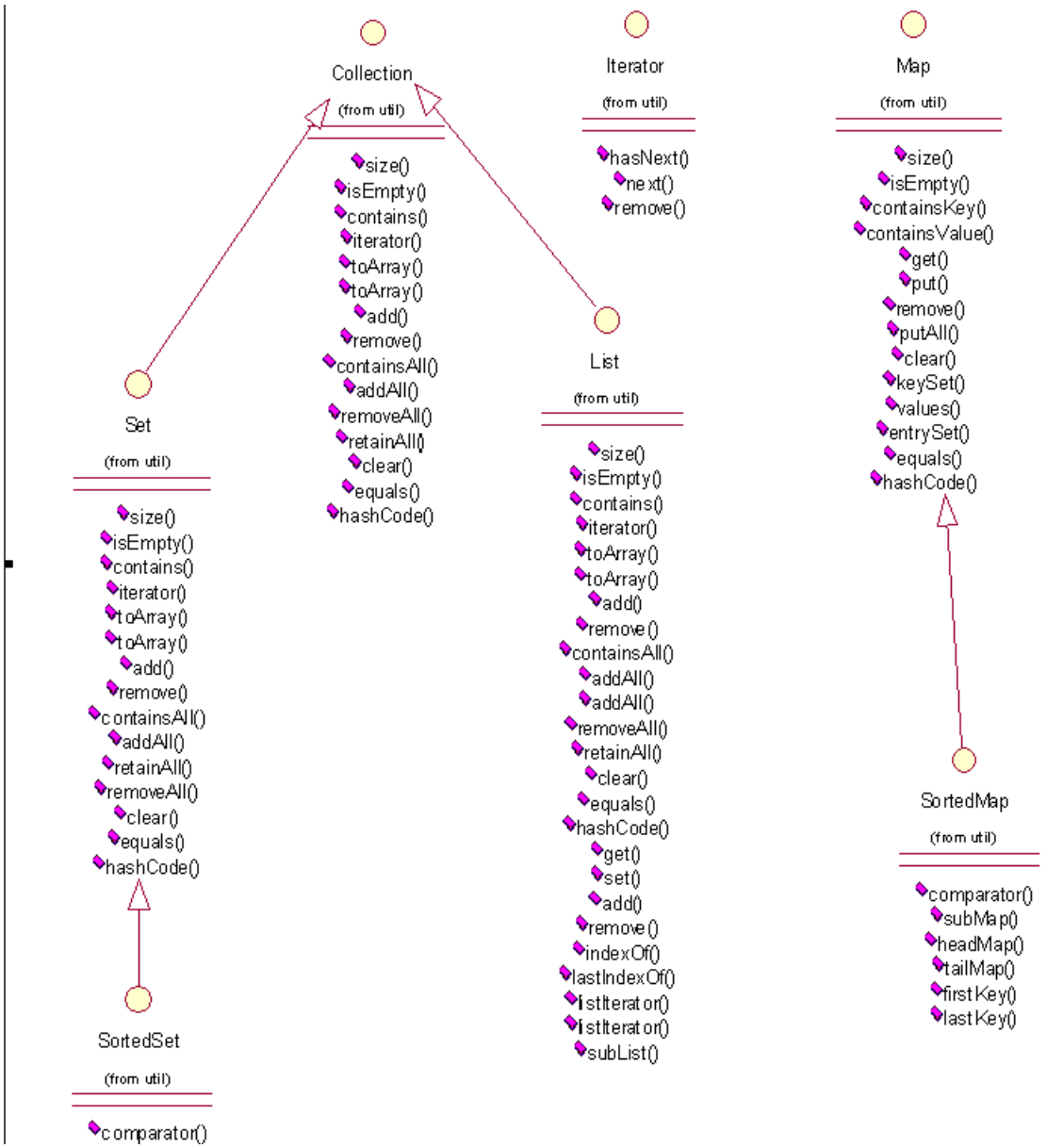
### 4.1.7. Интерфейс Iterator

В Java 1 для перебора элементов коллекции использовался интерфейс Enumeration. В Java 2 для этих целей должны использоваться объекты которые реализуют интерфейс Iterator. Все классы которые реализуют интерфейс Collection должны реализовать метод, iterator, который возвращает объект реализующий интерфейс Iterator. Iterator весьма похож на Enumeration, с тем лишь отличием, что в нем определен метод remove, который позволяет удалить объект из коллекции, для которой Iterator бы создан.

Т.о подводя итог перечислим:

Интерфейсы используемые при работе с коллекциями.

```
java.util.Collection
java.util.Set
java.util.List
java.util.Map
java.util.SortedSet
java.util.SortedMap
java.util.Iterator
```




## 4.2. Абстрактные классы используемые при работе с коллекциями.

java.util.AbstractCollection - этот класс реализует все методы определенные в интерфейсе Collection за исключением iterator и size, т.о. для того что бы создать не модифицируемую коллекцию нужно переопределить эти методы.Для реализации модифицируемой коллекции, необходимо еще переопределить метод public void add(Object o) (в противном случае, при его вызове будет возбуждено исключение UnsupportedOperationException).

Необходимо так же определить два конструктора без аргументов и с аргументом Collection. Первый должен создавать пустую коллекцию, второй коллекцию на основе сеуществующей. Данный класс расширяется классами AbstractList и AbstractSet.

java.util.AbstractList - этот класс расширяет AbstractCollection и реализует интерфейс List. Для реализации создания не модифицируемого списка необходимо имплементировать

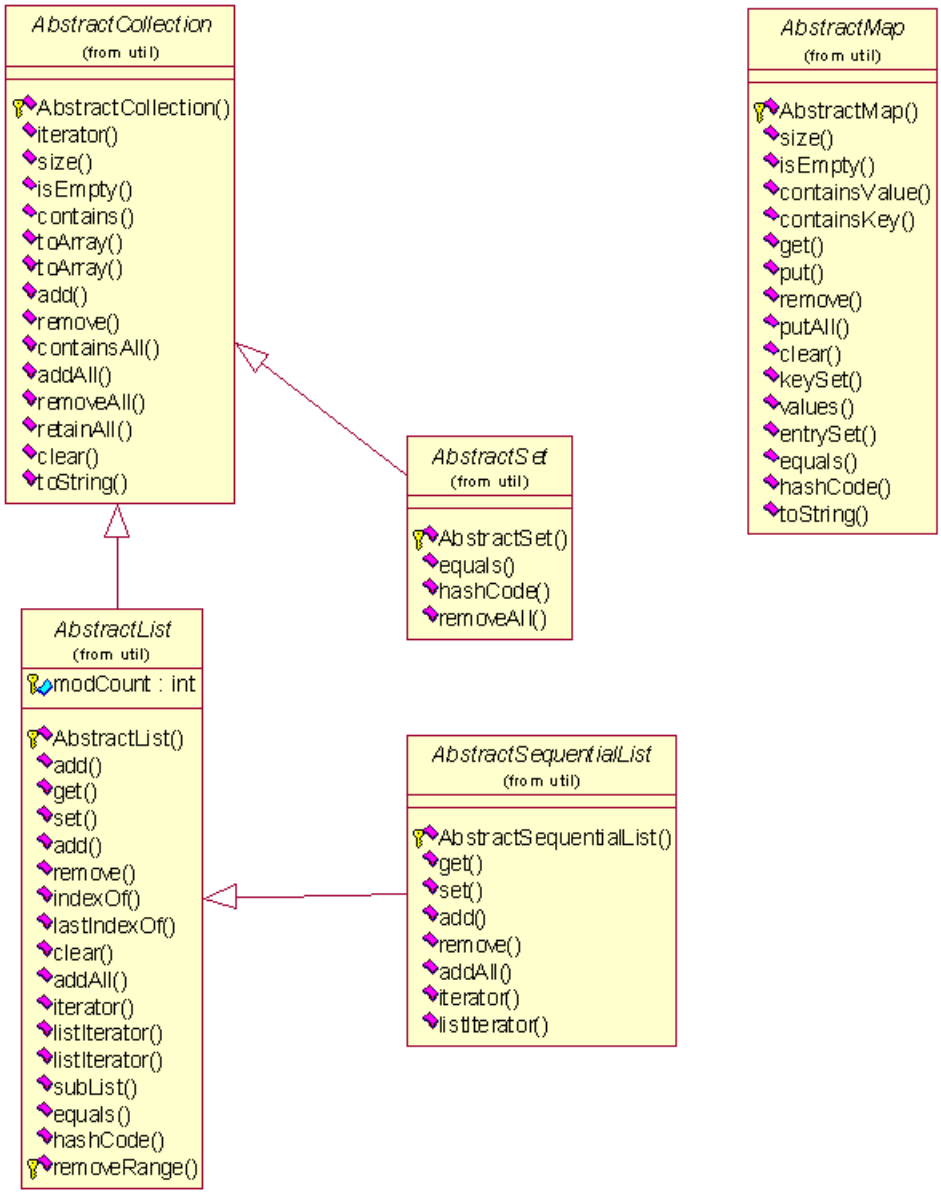
методы public Object get(int index) и public int size(). Для реализации модифицируемого списка необходимо так же реализовать метод public void set(int index,Object element); (в противном случае, при его вызове будет возбуждено исключение UnsupportedOperationException)

В отличии от AbstractCollection в этом случае нет необходимости реализовывать метод iterator, т.к. он уже реализован поверх методов доступа к элементам списка get, set, add, remove.

java.util.AbstratctSet - этот класс расширяет AbstractCollection и реализует основную функциональность определенную в интерфейсе Set. Следует отметить, что этот класс не переопределяет функциональность реализованную в классе AbstractCollection.

java.util.AbstratctMap - этот класс расширяет реализует основную функциональность определенную в интерфейсе Map Для реализации не модифицируемого класса, унаследованного от AbstractMap достаточно определить метод entrySet, который должен возвращать объект приводимый к типу AbstractSet. Этот набор (Set) не должен обеспечивать методов для добавления у даления элементов из набора. Для реализации модифицируемого класса Map необходимо так же переопределить метод put и итератор возвращаемый entrySet().iterator()

java.util.AbstratctSequentialList - этот класс расширяет AbstractList и является основой для класса LinkedList. Основное отличие от AbstractList заключается в том, что этот класс обеспечивает не только последовательный, но и произвольный доступ к элементам списка, с помощью методов get(int index), set(int index, Object element), set(int index, Object element), add(int index, Object element) и remove(int index). Для того что бы реализовать данный класс необходимо переопределить методы listIterator и size. Причем если реализуется не модифицируемый список, для итератора достаточно реализовать методы hasNext, next, hasPrevious, previous и index. Для модифицируемого списка необходимо дополнительно реализовать метод set, а для списков переменной длинны еще и add и remove.




## 4.3. Конкретные классы коллекций

java.util.ArrayList - этот класс расширяет AbstractList и весьма похож на класс Vector. Он так же динамически расширяется как Vector, однако его методы не являются синхронизированными, в следствие чего операции с ним выполняются быстрее. Для того, что бы воспользоваться синхронизированной версией ArrayList, можно применить вот такую конструкцию

```
List l = Collections.synchronizedList(new ArrayList(...));


public class Test {
    public Test() {
```

```
    }

    public static void main(String[] args) {
        Test t = new Test();
        ArrayList al = new ArrayList();
        al.add("Firts element");
        al.add("Second element");
        al.add("Third element");
        Iterator it = al.iterator();
        while(it.hasNext()){
            System.out.println((String)it.next());
        }
        System.out.println("\n");
        al.add(2,"Insertion");
        it = al.iterator();
        while(it.hasNext()){
            System.out.println((String)it.next());
        }
    }
}
```

```
Firts element
Second element
Third element

Firts element
Second element
Insertion
Third element
```

java.util.LinkedList - является реализацией интерфейса List. Он реализует все методы интерфейса List, помимо этого добавляются еще новее методов, которые позволяют добавлять, удалять и получать элементы в конце и начале списка. LinkedList является двухсвязным списком и позволяет перемещаться как от начала в конец списка, так и наоборот.LinkedList удобно использовать для организации стека.

```
public class Test {

    public Test() {
    }

    public static void main(String[] args) {
        Test test = new Test();
        LinkedList ll = new LinkedList();

        ll.add("Element1");
        ll.addFirst("Element2");
        ll.addFirst("Element3");
```

```
        ll.addLast("Element4");
        test.dumpList(ll);

        ll.remove(2);
        test.dumpList(ll);

        String element = (String)ll.getLast();
        ll.remove(element);
        test.dumpList(ll);
    }
    private void dumpList(List list){
        Iterator it = list.iterator();
        System.out.println();
        while(it.hasNext()){
            System.out.println((String)it.next());
        }
    }
}
Element3
Element2
Element1
Element4

Element3
Element2
Element4

Element3
Element2
```

Классы LinkedList и ArrayList имеют схожую функциональность. Однако с точки зрения производительности он отличаются. Так в ArrayList заметно быстрей (примерно на порядок) осуществляются операции прохода по всему списку (итерации) и получения данных. LinkedList почти на порядок быстрее осуществляет операции удаления и добавления новых элеемнтов.

java.util.Hastable - расширяет абстрактный класс Dictionary. D JDK 1.2, класс Hashtable так же реализует интерфейс Map. Hashtable предназначен для хранения объектов в виде пар ключ/значение. Из самого названия следует, что Hashtable использует алгоритм хэшированя для увеличения скорости доступа к данным. Для того что бы выяснить принципы работы данного алгоритма рассмотрим несколько примеров.

Предположим имеется массив строк содержащий названия городов. Для того что бы найти элемент массива содержащий название города, в общем случае необходимо просмотреть весь массив, а если необходимо найти все элементы массива, то для поиска каждого, в среднем потребуется просматривать половину массива. Такой подход может оказать приемлемым только для небольших массивов .

Как уже отмечалось ранее, для того что бы увеличить скорость поиска, используется алгоритм хэширования. Каждый объект в Java унаследован от Object. Как уже отмечалось ранее, Object определено целое число которое уникально идентифицирует экземпляр

класса Object и, соответственно все экземпляры классов унаследованных от Object. Это число возвращает метод hashCode(). Именно это число и используется при сохранении ключа в Hashtable, следующим образом: разделив длину массива предназначенного для хранения ключей на код, получается некое целое число которое служит индексом для хранения ключа в массиве. array.length % hashCode()

Далее, если необходимо добавить новую пару ключ/значение вычисляется новый индекс, если этот индекс совпадает, с уже имеющимся, то создается список ключей, на которой указывает элемент массива ключей. Таким образом, при обратом извлечении ключа, необходимо вычислить индекс массива по тому же алгоритму и получить его. Если ключ в массиве единственный, то используется значение элемента массива, если хранится несколько ключей, то необходимо обойти список и выбрать нужный.

Есть несколько соображений, относящихся к производительности классов, использующих для хранения данных алгоритм хэширования. Размер массива. Если массив будет слишком мал, то связанные списки будут слишком длинными, и скорость поиска будет существенно снижаться, т.к. просмотр элементов списка будет такой же как в обычном массиве Что бы избежать этого задается некий коэффициент заполнения. При заполнении элементов массива в котором хранятся ключи ( или списки ключей) на эту величину, происходит увеличение массива и производится повторное реиндексирование. Таким образом если массив будет слишком мал, то он будет быстро заполнятся и будет производится операция повторного индексирования, которая отнимает достаточно много ресурсов. С другой стороны, если массив сделать большим, то при необходимости просмотреть последовательно все элементы коллекции использующей алгоритм хэширования будет необходимо обработать большое количество пустых элементов массива ключей.

Начальный размер массива и коэффициент загрузки коллекции задаются при конструировании.

Например Hashtable ht = new Hashtable(1000,0.60);

Существует так же конструктор без параметров. который использует значения по умолчанию 101 для размера массива и 0.75 для коэффициента загрузки.

Использование алгоритма хэширования позволяет гарантировать, что скорость доступа к элементам коллекции такого типа будет увеличиваться не линейно, а логарифмически. Таким образом, при частом поиске каких либо значений по ключу имеет смысли использовать коллекции использующие алгоритм хэштрования.

java.util.HashMap,- этот класс расширяет AbstractMap и весьма похож на класс Hashtable. HashMap предназанчен для хранения пар объектов ключ/значение. Как для ключей, так для элементов допускаются значения типа null. Порядок хранения элементов в этой коллекции не совпадает с порядком их добавления. Порядок элементов в коллекции так же может меняться во времени. HashMap обеспечивает постоянное время доступа для операций get и put.

Итерация по всем элементам коллекции пропорциональна ее емкости. Поэтому имеет смысл не устанавливать размер коллекций чрезмерно большим, если достаточно часто придется осуществлять итерацию по элементам.

Методы HashMap не являются синхронизированными. Для того, что бы обеспечить нормальную работу в много потоковом варианте следует использовать либо внешнюю синхронизацию потоков, либо использовать синхронизированный вариант коллекции

Следует еще обратить внимание на разницу между HashMap и Hashtable. Hashtable существует в Java еще с первых релизов . HashMap появился в JDK 1.2. Главное отличие в том, что Hashtable не позволяет сохранять пустые значения, в HashMap это делать можно. Кроме того, методы в Hashtable являются синхронизированными, а в HasMap нет. Кроме этого, следует помнить, что начиная с JDK 1.2 Hashtable реализует интерфейс Map, что может вызвать некоторые проблемы при попытке запус ка приложений использующих более ранние версии JDK.

```
public class Test {

    private class TestObject{
        String text = "";
        public TestObject(String text){
            this.text = text;
        };
        public String getText(){
            return this.text;
        }
        public void setText(String text){
            this.text =  text;
        }
    }

    public Test() {
    }

    public static void main(String[] args) {
        Test t = new Test();
        TestObject to = null;
        HashMap hm = new HashMap();
        hm.put("Key1",t.new TestObject("Value 1"));
        hm.put("Key2",t.new TestObject("Value 2"));
        hm.put("Key3",t.new TestObject("Value 3"));

        to = (TestObject)hm.get("Key1");
       System.out.println("Object value for Key1 = " + to.getText() + "\n");

        System.out.println("Iteration over entrySet");
        Map.Entry entry = null;
       Iterator it = hm.entrySet().iterator(); // Итератор для перебора всех
 точек входа в Map
        while(it.hasNext()){
            entry = (Map.Entry)it.next();
            System.out.println("For key = " + entry.getKey() + " value = " +
 ((TestObject)entry.getValue()).getText());
        }
        System.out.println();
```

```
        System.out.println("Iteration over keySet");
        String key = "";
        it = hm.keySet().iterator(); // Итератор для перебора всех ключей в
Map
        while(it.hasNext()){
            key = (String)it.next();
            System.out.println( "For key = " + key + " value = " +
((TestObject)hm.get(key)).getText());
        }

    }
}

Object value for Key1 = Value 1

Iteration over entrySet
For key = Key3 value = Value 3
For key = Key2 value = Value 2
For key = Key1 value = Value 1

Iteration over keySet
For key = Key3 value = Value 3
For key = Key2 value = Value 2
For key = Key1 value = Value 1
```

java.util.TreeMap - расширяет класс AbstractMap и реализует интерфейс SortedMap.TreeMap содержит ключи в порядке возрастания. Используется либо натуральное сравнение ключей, либо должен быть реализован интерфейс Comparable. Реализация алгоритма поиска обеспечивает логарифмическую зависимость времени выполнения основных операций (containsKey, get, put и remove). Запрещено использование null значений для ключей. При использовании дубликатов ключей ссылка на объект сохраненный с таким же ключом будет утеряна. (см. пример ниже).

```
public class Test {

    public Test() {
    }

    public static void main(String[] args) {
        Test t = new Test();
        TreeMap tm = new TreeMap();
        tm.put("key","String1");
        System.out.println(tm.get("key"));
        tm.put("key","String2");
        System.out.println(tm.get("key"));
    }
}
```

```
String1
String2
```

## 4.4. Класс Collections

Сразу же следует отметить, что не нужно путать класс java.util.Collections с интерфейсом java.util.Collection.

Класс Collections является классом-утилитой и содержит несколько вспомогательных методов для работы с классами обеспечивающими различные интерфейсы коллекций. Например для сортировки элементов списков, для поиска элементов в упорядоченных коллекциях и т.д. Но пожалуй наиболее важным свойством этого класса является возможность получения синхронизированный вариантов классов-коллекций. Например для получения синхронизированного варианта Map можно использовать следующий подход.

```
HashMap hm = new HashMap();
…
Map syncMap = Collections.synchronizedMap(hm);
…
```

Как уже отмечалось ранее, начиная с JDK 1.2 класс Vector реализует интерфейс List. Рассмотрим пример сортировки элементов содержащихся в классе Vector.

Следует еще раз напомнить, что в действительности в коллекции содержатся лишь ссылки на объекты, а не их копии.

```
public class Test {

    private class TestObject{
        private String name = "";
        public TestObject(String name){
            this.name = name;
        }
    }

    private class MyComparator implements Comparator{
        public int compare(Object l,Object r){
            String left = (String)l;
            String right = (String)r;
            return -1 * left.compareTo(right);
        }
    }

    public Test() {
    }

    public static void main(String[] args) {
        Test test = new Test();
        Vector v = new Vector();
```

```
        v.add("bbbbb");
        v.add("aaaaa");
        v.add("ccccc");
        System.out.println("Default elements order");
        test.dumpList(v);
        Collections.sort(v);
        System.out.println("Default sorting order");
        test.dumpList(v);
        System.out.println("Reverse sorting order with providing imlicit
comparator");
        Collections.sort(v,test.new MyComparator());
        test.dumpList(v);
    }

    private void dumpList(List l){
        Iterator it = l.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

# 5. Класс Properties

Класс Properties предназначен для хранения набора свойств (параметров). Методы

```
String getProperty(String key)
String getProperty(String key,String defaultValue)
```

позволяют получить свойство из набора.

С помощью метода setProperty(String key, String value) это свойство можно установить.

Метод load(InputStream inStream) позволяет загрузить набор свойств из входного потока (Потоки данных подробно рассматриваются в главе 15). Как правило это текстовый файл в котором хранятся параметры. Параметры представляют собой строки представляющие сбой пары ключ/значение. Предполагается, что по умолчанию используется кодировка ISO 8859-1. Каждая строка должна оканчиваться символами \r,\n или \r\n. Строки из файла будут считываться пока не будет достигнут его конец. Строки состоящие из одних пробелов или начинающиеся со знаков ! или # игнорируются, т.е. их можно трактовать как комментарии. Если строка оканчивается символом /, то следующая строка считается ее продолжением. Первый символ с начала строки, отличающийся от пробела, считается началом ключа. Первый встретившийся пробел, :, = считается окончанием ключа. Все символы окончания ключа при необходимости могут быть включены в название ключа, но при этом перед ними должен стоять символ \. После того как встретился символ окончания ключа, все аналогичные символы будут проигнорированы до начала значения. Оставшаяся часть строки интерпретируется как значение. В строке, состоящей только из символов \t, \n, \r, \\, \", \', \ и \uxxxx, они все распознаются и интерпретируются как одиночные символы. Если встретится сочетание \ и символа конца строки, то следующая строка будет считаться

продолжением текущей, так же будут проигнорированы все пробелы до начала строки-продолжения.

Метод save(OutputStream inStream,String header) сохраняет набор свойств в выходной поток, в виде пригодном для вторичной загрузки с помощью метода load. Символы считающиеся служебными, кодируются так, что бы их можно было считать при вторичной загрузке. Символы в национальной кодировке будут приведены к виду \uxxxx . При сохранении используется кодировка ISO 8859-1. Если указан, header то он будет помещен в начало потока в виде комментария ( т.е. с символом # в начале), далее будет следовать комментарий в котором будет указано время и дата сохранения свойств в потоке.

В классе Properties определено еще метод list(PrintWriter out) который практически идентичен save. Отличается лишь заголовок, который изменить нельзя. Кроме того строки усекаются по ширине. Поэтому этот метод для сохранения Properties не годится.

```
public class Test {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        Properties props = new Properties();
        StringWriter sw = new StringWriter();
        sw.write("Key1 = Vlaue1 \n");
        sw.write("     Key2 : Vlaue2 \r\n");
        sw.write("     Key3  Vlaue3 \n ");
        InputStream is = new ByteArrayInputStream(sw.toString().getBytes());
        try {
            props.load(is);
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
        props.list(System.out);
        props.setProperty("Key1","Modified Value1");
        props.setProperty("Key4","Added Value4");
        props.list(System.out);
    }
}

-- listing properties --
Key3=Vlaue3
Key2=Vlaue2
Key1=Vlaue1
-- listing properties --
Key4=Added Value4
Key3=Vlaue3
Key2=Vlaue2
Key1=Modified Value1
```

# 6. Интерфейс Comparator

В коллекциях многие методы сортировки или сравнения требуют передачи в качестве одного из параметров объекта который реализует интерфейс Comparator. Этот интерфейс определяет единственный метод compare(Object obj1,Object obj2), который, на основании алгоритма определенного пользователем, сравнивает объекты переданные в качестве параметров. Метод compare должен вернуть

```
-1   если obj1 < obj2
0 если obj1 = obj2
1 если obj1 > obj2
```

# 7. Класс Arrays

Статический класс Arrays обеспечивает набор методов для выполнения операций над массивами, такие как поиск, сортировка, сравнение.В Arrays так же определен статический метод public List aList(a[] arr); который возвращает список фиксированного размера основанный на массиве. Изменения в List можно внести изменив данные в массиве.

Обратная операция, т.е. представление какой-либо коллекции в виде массива осуществляется с помощью статического метода Object[] toArray() определенного в классе Collections.

```
public class Test {
    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        String[] arr = {"String 1","String 4","String 2","String 3"};
        test.dumpArray(arr);
        Arrays.sort(arr);
        test.dumpArray(arr);
        int ind = Arrays.binarySearch(arr,"String 4");
        System.out.println("\nIndex of \"String 4\" = " + ind);
    }
    void dumpArray(String arr[]){
        System.out.println();
        for(int cnt=0;cnt < arr.length;cnt++){
            System.out.println(arr[cnt]);
        }
    }
}
```

# 8. Класс StringTokenizer

Этот класс предназначен для разбора строки по лексемам (tokens). Строка которую необходимо разобрать передается в качестве параметра конструктору StringTokenizer(String str). Определено еще два перегруженных конструктора, которым дополнительно можно передать строку-разделитель лексем StringTokenizer(String str,String delim) и признак возврата разделителя лексем StringTokenizer(String str,String delim,Boolean retirnDelims)

Разделителем лексем по умолчанию служит пробел.

```
public class Test {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        String toParse = "word1;word2;word3;word4";
        StringTokenizer st = new StringTokenizer(toParse,";");
        while(st.hasMoreTokens()){
            System.out.println(st.nextToken());
        }
    }
}

word1
word2
word3
word4
```

# 9. Класс BitSet

Класс BitSet предназначен для работы с последовательностями битов. Каждый компонент этой коллекции может принимать булево значение, которое обозначает установлен бит или нет. Содержимое BitSet может быть модифицировано содержимым другого BitSet с использованием операций AND, OR или XOR ( исключающее или)

BitSet имеет текущий размер (количество установленных битов) может динамически изменятся. По умолчанию все биты в наборе устанавливаются в 0 (false). Установка и очистка битов в BitSet осуществляется методами set(int index) и clear(int index)

Метод int length() возвращает "логический" размер набора битов, int size() возвращает количество памяти занимаемой битовой последовательностью BitSet.

```
public class Test {

    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
```

```
        BitSet bs1 = new BitSet();
        BitSet bs2 = new BitSet();
        bs1.set(0);
        bs1.set(2);
        bs1.set(4);
        System.out.println("Length = " + bs1.length() + " size = " +
bs1.size());
        System.out.println(bs1);
        bs2.set(1);
        bs2.set(2);
        bs1.and(bs2);
        System.out.println(bs1);
    }
}
```

```
Length = 5 size = 64
{0, 2, 4}
{2}
```

Проанализировав первую строку вывода на консоль можно сделать вывод, что для внутреннего представления BitSet использует значения типа long.

# 10. Класс Random

Класс Random используется для получения последовательности псевдослучайных чисел. В качестве "зерна" используется 48 битовое число. Если для инициализации Random использовать одно и то же число, будет получена та же самая последовательность псевдослучайных чисел.

В классе Random определено так же несколько методов которые возвращают псевдослучайные величины для примитивных типов Java

Дополнительно следует отметить наличие двух методов double nextGaussian() - возвращает случайное число в диапазоне от 0.0 до 1.0 распределенное по нормальному закону, а void nextBytes(byte[] arr) - заполняет массив arr случайными величинами типа byte.

Пример использования Random

```
public class Test  {
    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        Random r = new Random(100);
        // Generating the same sequence numbers
        for(int cnt=0;cnt<9;cnt++){
            System.out.print(r.nextInt() + " ");
```

```
        }
        System.out.println();
        r = new Random(100);
        for(int cnt=0;cnt<9;cnt++){
            System.out.print(r.nextInt() + " ");
        }
        System.out.println();
        // Generating sequence of bytes
        byte[] randArray = new byte[8];
        r.nextBytes(randArray);
        test.dumpArray(randArray);
    }

    void dumpArray(byte[] arr){
        for(int cnt=0;cnt< arr.length;cnt++){
            System.out.print(arr[cnt]);
        }
        System.out.println();
    }
}
```

```
-1193959466 -1139614796 837415749 -1220615319 -1429538713 118249332 -951589224

-1193959466 -1139614796 837415749 -1220615319 -1429538713 118249332 -951589224

81;-6;-107;77;118;17;93;-98;
```

# 11. Локализация

## 11.1. Класс Locale

Класс Locale предназначен для отображения определенного региона. Под регионом принято понимать не только географическое положение, но так же языковую и культурную среду. Так например для помимо того, что указывается страна Швейцария, можно указать так же и язык французский или немецкий.

Определено два варианта конструкторов в классе Locale

```
Locale(String language, String country)
Locale(String language, String country, String variant)
```

Первые два параметра в обоих конструкторах определяют язык и страну для которой определяется локаль, согласно кодировке ISO. (См приложения XXX1,XXX2). Список поддерживаемых стран и языков можно так же получить с помощью вызова статических методов Locale.getISOLanguages() Locale.getISOCountries() соответственно. Во втором варианте конструктора указан так же строковый параметр variant в котором кодируется информация о платформе. Если здесь необходимо указать дополнительные параметры,

то их необходимо разделить символом подчеркивания, причем более важный параметр должен следовать первым.

Пример использования

```
Locale l = new Locale("ru","RU");
Locale l = new Locale("en","US","WINDOWS");
```

Статический метод getDefault() возвращает текущую локаль, сконструированную на основе настроек операционной системы под управлением которой функционирует JVM.

Для наиболее часто использующихся локалей заданы константы. Например Locale.US или Locale.GERMAN.

После того как экземпляр класса Loale создан, с помощью различных методов можно получить дополнительную информацию о локали.

```
public class Test  {
    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        Locale l = Locale.getDefault();
        System.out.println(l.getCountry() + " " + l.getDisplayCountry() + "
" + l.getISO3Country());
        System.out.println(l.getLanguage() + " " + l.getDisplayLanguage() +
" " + l.getISO3Language());
        System.out.println(l.getVariant() + " " + l.getDisplayVariant());
        l = new Locale("ru","RU","WINDOWS");
        System.out.println(l.getCountry() + " " + l.getDisplayCountry() + "
" + l.getISO3Country());
        System.out.println(l.getLanguage() + " " + l.getDisplayLanguage() +
" " + l.getISO3Language());
        System.out.println(l.getVariant() + " " + l.getDisplayVariant());
    }
}
```

```
US United States USA
en English eng


RU Russia RUS
ru Russian rus
WINDOWS WINDOWS
```

## 11.2. Класс ResourceBundle

Абстрактный Класс ResourceBundle предназначен для хранения объектов специфичных для локали. Например, когда необходимо получить набор строк, зависящих от локали используют ResourceBundle.

Использование ResourceBundle настоятельно рекомендуется, если предполагается использовать программу в многоязыковой среде. С помощью этого класса легко манипулировать, наборами ресурсов зависящих от локалей, легко их менять, добавлять новые и т.д.

Набор ресурсов - это фактически набор классов, имеющих одно базовое имя. Далее наименование класса дополняется наименованием локали, с которой связывается этот класс. Например, если имя базового класса будет MyResources, то для английской локали имя класса будет MyResources_en, для русской - MyResources_ru. Помимо этого может добавляться идентификатор языка, если для данного региона определено несколько языков. Например MyResources_de_CH так будет выглядеть швейцарский вариант немецкого языка. Кроме того можно указать дополнительную признак variant (см. описание Locale

). Так, описанный раннее пример для платформы UNIX будет выглядеть следующим образом: MyResources_de_CH_UNIX

Загрузка объекта для нужной локали производится с помощью статического метода getBundle.

```
ResourceBundle myResources =      ResourceBundle.getBundle("MyResources",
someLocale);
```

Если объект, имя которого в точности совпадает с именем, указанным в getBundle, то будет найден тот, имя которого максимально сходно с запрошенным. Приоритет перебора суффиксов будет такой. В начале ищется файл с указанной локалью, если такого файла нет, то будет подбираться класс с локалью по умолчанию. Если его тоже нет, то будет выбираться класс наименование которого совпадает с базовым классом.

```
baseclass + "_" + language1 + "_" + country1 + "_" + variant1
baseclass + "_" + language1 + "_" + country1 + "_" + variant1 + ".properties"

baseclass + "_" + language1 + "_" + country1
baseclass + "_" + language1 + "_" + country1 + ".properties"
baseclass + "_" + language1
baseclass + "_" + language1 + ".properties"
baseclass + "_" + language2 + "_" + country2 + "_" + variant2
baseclass + "_" + language2 + "_" + country2 + "_" + variant2 + ".properties"

baseclass + "_" + language2 + "_" + country2
baseclass + "_" + language2 + "_" + country2 + ".properties"
baseclass + "_" + language2
baseclass + "_" + language2 + ".properties"
baseclass
```

```
baseclass + ".properties"
```

Индексы 1 2 в данном случае подразумевают затребованную локаль и локаль по умолчанию.

Например если необходимо найти ResourceBundle для локали fr_CH (Швейцарский французский), а локаль по умолчанию en_US при этом название базового класса ResourceBundle MyResources, то порядок поиска подходящего ResourceBundle будет таков.

```
MyResources_fr_CH
MyResources_fr
MyResources_en_US
MyResources_en
MyResources
```

Результатом работы getBundle будет загрузка необходимого класса ресурсов в память, однако данные этого класса могут быть сохранены на диске. Т.о. если нужный класс не будет найден, то к требуемому имени класса будет добавлено расширение ".properties" и будет совершена попытка найти файл с данными на диске.

Следует помнить, что необходимо указывать полностью квалифицированное имя класса ресурсов т.е. имя пакета, имя класса. Кроме того, класс ресурсов должен быть доступен в контексте его вызова (т.е. там где вызывается getResourceBundle), те не быть private и т.д.

Всегда должен создаваться базовый класс без суффиксов, т.е. если вы создается ресурсы с именем MyResource, должен быть в наличии класс MyResource.class

ResourceBundle хранит объекты в виде пар ключ/значение. Как уже отмечалось ранее, класс ResourceBundle абстрактный, поэтому при его наследовании необходимо переопределить методы Enumeration getKeys()

```
public Object handleGetObject(String key)
```

первый метод должен возвращать список всех ключей, которые определены в Resource-Bundle, второй должен возвращать объект связанный с конкретным ключом.

Рассмотрим пример использования ResourceBundle.

```
public class MyResource extends ResourceBundle {

    private Hashtable res = null;
    public MyResource() {
        res = new Hashtable();
        res.put("TestKey","English Variant");
    }

    public Enumeration getKeys() {
        return res.keys();
    }

    protected Object handleGetObject(String key) throws
```

```
java.util.MissingResourceException {
        return res.get(key);
    }
}

public class MyResource_ru_RU extends ResourceBundle{
    private Hashtable res = null;

    public MyResource_ru_RU() {
        res = new Hashtable();
        res.put("TestKey","Русский варинат");
    }

    public Enumeration getKeys() {
        return res.keys();
    }

    protected Object handleGetObject(String key) throws
java.util.MissingResourceException {
        return res.get(key);
    }
}

public class Test  {
    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        ResourceBundle rb =
ResourceBundle.getBundle("experiment.MyResource",Locale.getDefault());
        System.out.println(rb.getString("TestKey"));
        rb = ResourceBundle.getBundle("experiment.MyResource",new
Locale("ru","RU"));
        System.out.println(rb.getString("TestKey"));
    }
}


English Variant
Русский Вариант
```

Кроме того, следует обратить внимание, что ResourceBundle может хранить не только строковые значения. В нем можно хранить также двоичные данные или просто методы, реализующие нужную функциональность в зависимости от локали.

```
public interface Behavior {
  public String getBehavior();
  public String getCapital();
}
```

```
public class EnglishBehavior implements Behavior{
  public EnglishBehavior() {
  }
  public String getBehavior(){
     return "English behavior";
  }
  public String getCapital(){
    return "London";
  }
}

public class RussianBehavior implements Behavior {
  public RussianBehavior() {
  }
  public String getBehavior(){
    return "Русский вариант поведения";
  }
  public String getCapital(){
    return "Москва";
  }
}

public class MyResourceBundle_ru_RU extends ResourceBundle  {
  Hashtable bundle = null;
  public MyResourceBundle_ru_RU() {
    bundle = new Hashtable();
    bundle.put("Bundle description","Набор ресурсов для русской локали");
    bundle.put("Behavior",new RussianBehavior());
  }
  public Enumeration getKeys() {
    return bundle.keys();
  }
  protected Object handleGetObject(String key) throws
java.util.MissingResourceException {
    return bundle.get("key");
  }
}

public class MyResourceBundle_en_EN {
  Hashtable bundle = null;
  public MyResourceBundle_en_EN() {
    bundle = new Hashtable();
    bundle.put("Bundle description","English resource set");
    bundle.put("Behavior",new EnglishBehavior());
  }
  public Enumeration getKeys() {
    return bundle.keys();
  }
  protected Object handleGetObject(String key) throws
```

```
java.util.MissingResourceException {
    return bundle.get("key");
  }
}

public class MyResourceBundle extends ResourceBundle {
  Hashtable bundle = null;
  public MyResourceBundle() {
    bundle = new Hashtable();
    bundle.put("Bundle description","Default resource bundle");
    bundle.put("Behavior",new EnglishBehavior());
  }
  public Enumeration getKeys() {
    return bundle.keys();
  }
  protected Object handleGetObject(String key) throws
java.util.MissingResourceException {
    return bundle.get(key);
  }
}

public class Using {

  public Using() {
  }

  public static void main(String[] args) {
    Using u = new Using();
    ResourceBundle rb =
ResourceBundle.getBundle("lecture.MyResourceBundle",Locale.getDefault());
    System.out.println((String)rb.getObject("Bundle description"));
    rb = ResourceBundle.getBundle("lecture.MyResourceBundle",new
Locale("en","EN"));
    System.out.println((String)rb.getObject("Bundle description"));
    Behavior be = (Behavior)rb.getObject("Behavior");
    System.out.println(be.getBehavior());
    System.out.println(be.getCapital());
  }
}></eg>
    <eg><![CDATA[
Русский набор ресурсов
English  resource bundle
English behavior
London></eg>
    </div2>
    <div2 id="JAVA-LEC14-ListResourceBundle ">
     <head>Классы ListResourceBundle и PropertiesResourceBundle</head>
     <p>У класса <kw>ResourceBundle</kw> определено два прямых потомка
<kw>ListResourceBundle</kw> и <kw>PropertiesResourceBundle</kw>. </p>
```

```
     <p>
     <kw>PropertiesResourceBundle</kw>  хранит набор ресурсов в файле, который
 представляет собой  набор строк.  </p>
     <p>Следует акцентировать внимание, что в случае с
<kw>PropertiesResourceBundle</kw> данные хранятся в обычном текстовом файле
и соответственно можно иметь дело только со строковыми значениями. </p>
     <p>Алгоритм конструирования объекта содержащего набор ресурсов был описан
 в предыдущем параграфе. Во все случаях когда в качестве последнего элемента
 используется <code>.properties</code>, например  <code>baseclass + "_" +
language1 + "_" + country1 + ".properties"</code> речь идет о создании
<kw>ResourceBundle</kw> из файла с наименованием <code>baseclass + "_" +
language1 + "_" + country1</code> и расширением <code>properties. </code>Обычно
 класс <kw>ResourceBundle</kw> помещают в пакет <kw>resources</kw>, а файл
свойств в каталоге <kw>resources</kw>. Тогда для того что инстанциировать
нужный класс необходимо указать полный путь к этому классу(файлу) </p>
     <eg><![CDATA[
getBundle("resources.MyResource",Locale.getDefault());
```

ListResourceBundle хранит набор ресурсов в виде коллекции и является абстрактным классом. Классы которые наследуют ListResourceBundle должны обеспечить

- переопределение метода Object[][] getContents() который возвращает массив ресурсов.
- и собственно двумерный массив содержащий ресурсы.

Рассмотрим пример

```
public class MyResource extends ListResourceBundle {
    Vector v = new Vector();

    Object[][] resources = {
    {"StringKey","String"},
    {"DoubleKey",new Double(0.0)},
    {"VectorKey",v},
    };

    public MyResource(){
        super();
        v.add("Element 1");
        v.add("Element 2");
        v.add("Element 3");
    }

    protected Object[][] getContents() {
        return resources;
    }
}

public class Test  {
    public Test() {
```

```
    }
    public static void main(String[] args) {
        Test test = new Test();
        ResourceBundle rb =
ResourceBundle.getBundle("experiment.MyResource",Locale.getDefault());
        Vector v = (Vector)rb.getObject("VectorKey");
        Iterator it = v.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}

Element 1
Element 2
Element 3
```

Создание ресурсов для локалей отличных от локали по умолчанию, осуществляется так же, как это было показано для ResourceBundle.

Следует заострить внимание на том, что в отличие от PropertiesResourceBundle возможно задание ресурсов не только в виде строк, что требуется несомненно чаще всего, но и виде объектов. Например ничто не мешает сохранить в списке ресурсов скажем объект Image или звуковой трек.Следует избегать однако создания чересчур громоздких объектов содержащих ресурсы т.к. это увеличивает время их загрузки и расходует память.

Далее следует обратить внимание, что если определены как файл ресурсов так и объект с одинаковыми именами, то использован будет объект. Следует это из описания алгоритма поиска необходимого списка ресурсов. (см. выше)

Ключом в ResourceBundle может служить только строковое значение. Если в качестве ключа указать другой объект ( это возможно в случае наследования ResourceBundle, ListResourceBundle), ошибки времени исполнения не возникнет, но данный ресурс будет недоступен. Т.к. в ResourceBundle определены методы получающие в качестве параметра объект типа String.

Например класс MyResource будет откомпилировано нормально.

```
public class MyResource extends ListResourceBundle {
    Vector v = new Vector();

    Object[][] resources = {
    {"Key1","String1"},
    {new Double(1.0),"Double value"}
    };


    protected Object[][] getContents() {
        return resources;
    }
}
```

Однако при попытке получить ресурс использовав в качестве параметра Double получим ошибку компиляции.

```
public class Test  {
    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        ResourceBundle rb =
ResourceBundle.getBundle("experiment.MyResource",Locale.getDefault());
        System.out.println(rb.getString("Key1"));
        System.out.println(rb.getObject(new Double(1.0)));
    }
}
```

# 12. Заключение

- Для работы с датой и временем должны использоваться классы Date , Calendar. Класс Calendar абстрактный, существует конкретная реализация этого класса GregorianCalendar.
- Класс Observer и интерфейс Observable реализуют парадигму MVC и предназначены для уведомления одного объекта об изменении состояния другого.
- Коллекции (Collections) не накладывают ограничений на порядок следования и дублирование элементов.
- Списки (List) поддерживают порядок элементов. (управляется либо самими данными, либо внешними алгоритмами
- Наборы (Set) не допускают дублированных элементов.
- Карты (Maps) используют уникальные ключи, для поиска содержимого.
- Использование массивов делает добавление, удаление и увеличение количества элементов затруднительным.
- Использование связанных (LinkedList) облегчает вставку, удаление и увеличение размера хранилища, но снижает скорость индексированного доступа
- Использование деревьев (Tree) облегчает вставку, удаление и увеличение размера хранилища, снижает скорость индексированного доступа, но увеличивает скорость поиска.
- Использование хэширования облегчает вставку, удаление и увеличение размера хранилища, снижает скорость индексированного доступа, но увеличивает скорость поиска. Однако хэширование требует наличия уникальных ключей для для зпоминания элементов данных
- Класс Properties удобен для хранения наборов параметров в виде пар ключ/значение. Параметры могут сохраняться в потоки (файлы) и загружаться из них.

- Реализация классом интерфейса Comparator позволяет сравнивать экземпляры класса друг с другом и, соответственно, сортировать их, например в коллекциях.
- Arrays является классом-утилитой и обеспечивает набор методов, реализующих различные приемы работы с массивами. Не имеет конструктора.
- StringTokenizer - вспомогательный класс, предназначенный для разбора строк на лексемы.
- При необходимости работать с сущностями, представленными в виде битовых последовательностей, удобно использовать класс BitSet
- Манипуляцию ресурсов, различающихся в зависимости от локализации, удобно осуществлять с помощью классов ResourceBundle, ListResourceBundle, PropertiesResourceBundle. Собственно локаль задается с помощью класса Locale.

# 13. Контрольные вопросы

14-1. Необходимо написать метод, который возвращает случайное число в диапазоне от 0 до 100 кратное 5. Из перечисленных вариантов выберите правильный.

✓a.)
```
public int getRandom5(){
return (int)(Math.random()*20) * 5;
}
```

b.)
```
public int getRandom5(){
 Math m = new Math()
return (int)(m.random()*20) * 5;
}
```

c.)
```
public int getRandom5(){
return (Math.random()*20) * 5;
}
```

Правильный ответ a. Ответ b неверен т.к. производится попытка создать экземпляр класса Math, что вызовет ошибку компиляции т.к. Math не имеет конструктора. Ответ с не верен, потому что Math.random() возвращает случайную величину типа double в диапазоне от 0.0 до 1.0. Соответственно все выражение будет иметь тип double, а возвращаемое значение у функции типа int, следовательно, возникнет ошибка времени компиляции.

14-2. Какое из выражений относительно класса java.lang.Runtime является корректным?

✓a.) Объект Runtime создается при помощи следующего кода
```
Runtime r = Runtime.getRuntime();
```

b.) Метод gc() определенный в Runtime() вызывает начало сборки мусора виртуальной машиной Java

c.) Метод freeMemory() определенный в классе Runtime, освобождает не используемую память.

Правильный ответ a. Ответ b неверен т.к. процедура запуска сборки мусора запускается самой виртуальной машиной, gc() только уведомляет JVM о необходимости эту процедуру начать. Ответ c неверен, т.к. freeMemory возвращает количество свободной памяти, доступной JVM.

14-3. Необходимо написать метод, который получает в качестве параметра значение угла в градусах типа double и вычисляет его косинус. Какой из приведенных вариантов правилен.

a.)
```
double getCos(double angle){
    return Math.cos(angle);
}
```

✓b.)
```
double getCos(double angle){
    return Math.cos(angle * Math.PI / 180);
}
```

c.)
```
double getCos(double angle){
    return Math.cos(angle * PI / 180);
}
```

Правильный ответ b. Ответ a неверен т.к. тригонометрические функции получают в качестве параметра значение угла выраженное в радианах, а не в градусах. Ответ c неверен т.к. константа PI определена в классе Math.

14-4. В JDK 1.2 введены новые классы и интерфейсы, которые позволяют работать с наборами объектов. Отметьте те из них, которые являются интерфейсами.

✓a.) a java.util.List

b.) java.util.TreeMap

c.) java.util.AbstractList

✓d.) java.util.SortedMap

✓e.) java.util.Iterator

f.) java.util.Collections

Правильные ответы a, d, e. TreeMap является конкретным классом, AbstractList абстрактный класс, Collections класс-утилита.

14-5. Какие высказывания относительно java.util.Vector и java.util.Hashtable можно считать корректными

a.) В Vector могут сохранятся ссылки как на объекты так и на примитивные типы.

✓b.) Ссылки на объекты в Vector хранятся в порядке их добавления.

c.) В качестве ключей для Hashtable должны передаваться оъекты типа String

d.) Ссылки на объекты в Hashtable хранятся в порядке их добавления.

✓e.) И Hashtable и Vector являются синхронизированными, для того что бы избежать проблема, когда несколько потоков пытаются получить доступ к одной и той же коллекции.

Правильные ответы b, e. Ответ a неверен т.к. в классе Vector могут храниться лишь ссылки на объекты. Для манипулирования примитивными типами используются соответствующие классы-обертки. Ответ c неверен, т.к. в качестве ключа, в Hashtable может использоваться любой объект не только строки. Ответ d неверен, т.к. порядок следования объектов в Hashtable непредсказуем.

14-6. Приведенный ниже пример кода вызывает ошибку компиляции.

```
    double getCos(double angle){
        return Math.cos(angle);
    }

public static void showStatus(Boolean flag){
 if(flag){
  System.out.println("FIRED")
 }else{
  System.out.println("NOT READY");
 }
}
```

Какое из перечисленных ниже исправлений решит проблему?

a.) Заменить if(flag){ на if(flag.equals(true)

✓b.) Заменить public static void showStatus(Boolean flag){ на public static void showStatus(boolean flag){

✓c.) Заменить if(flag){ на if(flag.booleanValue()){

Правильные ответы b, c. Ответ a неверен т.к. equals метод требует в качестве параметра ссылку на объект, а true есть примитивный тип Boolean.

14-7. Какое значение будет выведено на консоль в существующем фрагменте кода?

```
String str1 = "abc";
String str2 = "abc";
System.out.println(str1 == str2);
```

✓a.) true

a.) false

Правильный ответ a. Т.к. строковые литералы, которыми инициализируются строки str1 и str2 одинаковы, то ссылаться они будут на один и тот же объект.

14-8. Будет ли переменная sb после выполнения кода в строке 2 указывать на тот же самый объект?

```
1. StringBuffer sb = new StringBuffer("abc");
2. sb.append("x");
```

✓a.) Да

b.) Нет

Правильный ответ a. Новый объект создается лишь при манипуляциях с объектом класса String.

14-9. Какой из перечисленных ниже классов имеет набольшее сходство с классом Vector?

a.) TreeSet.

b.) AbstractCollection.

✓c.) c ArrayList

d.) d Hashtable

Правильный ответ c т.к. именно ArrayList наиболее схож по функциональности с Vector. Ответ a неверен т.к. TreeSet использует сортировку. Ответ b неверен, т.к. AbstractCollection абстрактный класс. Ответ d неверен, т.к. Hashtable использует для хранения пары ключ/значение.

14-10. Какой из перечисленный ниже интерфейсов реализует Hashtable?

a.) a SortedMap

✓b.) b Map

c.) c List

d.) d SortedSet

e.) e ни один из перечисленных

Правильный ответ b. Ответы a, c, d не верны, т.к. Hashtable не реализует эти интерфейсы. Ответ e не верен, т.к. в JDK 1.2 Hashtable переделан и реализует интерфейс Map.

# Программирование на Java

## Лекция 15. Пакет java.io

### 20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Эта лекция описывает возможности Java для обмена или передачи информации, что является важной функциональностью для большинства программных систем. Сюда входит работа с файлами, с сетью, долговременное сохранение объектов, обмен данными между потоками исполнения и т.п. Все эти действия базируются на потоках байт (представлены классами InputStream и OutputStream) и потоками символов (Reader и Writer). Библиотека java.io содержит все эти классы и их многочисленных наследников, предоставляющих полезные возможности. Отдельно рассматривается механизм сериализации объектов и работа с файлами.

# Оглавление

# Лекция 15. Пакет java.io

## Содержание лекции.



## 1. Система ввода/вывода. Потоки данных (stream)

Подавляющее большинство программ обменивается данными с внешним миром. Это, безусловно, делают любые сетевые приложения - они передают и получают информацию от других компьютеров и специальных устройств, подключенных к сети. Оказывается удобным точно таким же образом представлять обмен данными между устройствами внутри одной машины. Так, например, программа может считывать данные с клавиатуры и записывать их в файл, или же наоборот - считывать данные из файла и выводить их на

экран. Таким образом, устройства, откуда может производиться считывание информации, могут быть самыми разнообразными - файл, клавиатура, (входящее) сетевое соединение и т.д. То же самое касается и устройств вывода - это может быть файл, экран монитора, принтер, (исходящее) сетевое соединение и т.п. В конечном счете, все данные в компьютерной системе в процессе обработки передаются от устройств ввода к устройствам вывода.

Обычно часть вычислительной платформы, которая отвечает за обмен данным, так и называется - система ввода/вывода. В Java она представлена пакетом java.io (input/output). Реализация системы ввода/вывода осложняется не только широким спектром источников и получателей данных, но еще и различными форматами передачи информации. Ею можно обмениваться в двоичном представлении, символьном или текстовом с применением некоторой кодировки (только для русского языка их насчитывается от 4 штук), или передавать числа в различных представлениях. Доступ к данным может потребоваться как последовательный (например, считывание HTML страницы), так и произвольный (сложная работа с несколькими частями одного файла). Зачастую для повышения производительности применяется буферизация.

В Java для описания работы по вводу/выводу используется специальное понятие поток данных (stream). Поток данных связан с некоторым источником или приемником данных, способных получать или предоставлять информацию. Соответственно, потоки делятся на входные - читающие данные, и на выходные - передающие (записывающие) данные. Введение концепции stream позволяет отделить программу, обменивающуюся информацией одинаковым образом с любыми устройствами, от низкоуровневых операций с такими устройствами ввода/вывода.

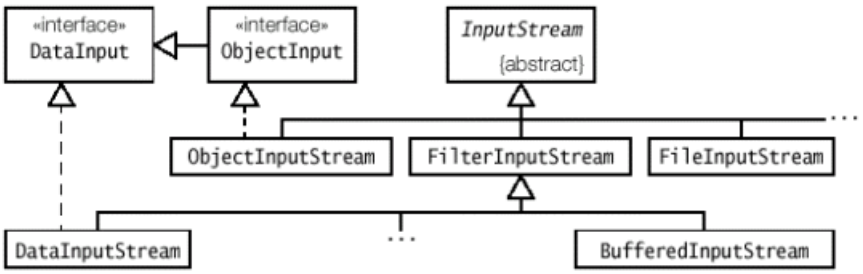
В Java потоки естественным образом представляются объектами. Описывающие их классы как раз и составляют основную часть пакета java.io. Они довольно разнообразны и отвечают за различную функциональность. Все классы разделены на две части - одни осуществляют ввод данных, другие вывод.

Существующие стандартные классы помогают решить большинство типичных задач. Минимальной "порцией" информации является, как известно, бит, принимающий значение 0 или 1 (это понятие также удобно применять на самом низком уровне, где данные передаются электрическим сигналом; условно говоря, 1 представляется прохождением импульса, 0 - его отсутствием). Традиционно используется более крупная единица измерения байт, объединяющая 8 бит. Таким образом, значение, представленное 1 байтом, находится в диапазоне от 0 до 28-1=255, или, если использовать знак, от -128 до +127. Примитивный тип byte в Java в точности соответствует последнему, знаковому диапазону.

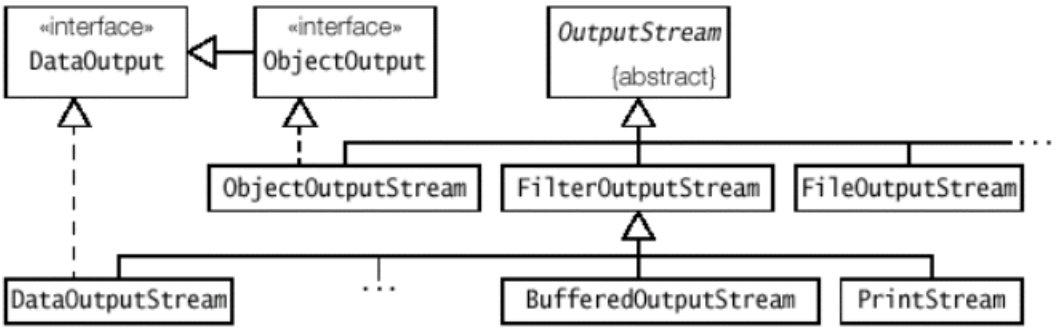
Базовые, наиболее универсальные классы позволяют считывать и записывать информацию именно в виде набора байт. Чтобы их было удобно применять в различных задачах, java.io содержит также классы, преобразующие любые данные в набор байт.

Например, если нужно сохранить результаты вычислений - набор значений типа double - в файл, то их можно сначала легко превратить в набор байт, а затем эти байты записать в файл. Аналогичные действия совершаются и в ситуации, когда требуется сохранить объект (т.е. его состояние) - преобразование в набор байт и последующая их запись в файл. Понятно, что при восстановлении данных в обоих рассмотренных случаях проделываются обратные действия - сначала считывается последовательность байт, а затем она преобразовывается в нужный формат.

На следующих диаграммах представлены иерархии классов ввода/вывода. Как и говорилось, все типы поделены на две группы. Представляющие входные потоки классы наследуются от InputStream:




а выходные - от OutputStream:




## 1.1. Классы InputStream и OutputStream

InputStream - это базовый класс для потоков ввода, т.е. чтения. Соответственно, он описывает базовые методы для работы с байтовыми потоками данных. Эти методы необходимы всем классам, наследующимся от InputStream.

Простейшая операция представлена методом read() (без аргументов). Он является абстрактным, и, соответственно, должен быть определен в классах-наследниках. Этот метод предназначен для считывания ровно одного байта из потока, однако возвращает при этом значение типа int. В случае если считывание произошло успешно, то возвращаемое значение лежит в диапазоне от 0 до 255 и представляет собой полученный байт (значение int содержит 4 байта и получается простым дополнением нулями в двоичном представлении). Обратите внимание, что полученный таким образом байт не обладает знаком и не находится в диапазоне от -128 до +127 как примитивный тип byte в Java.

В случае если достигнут конец потока, то есть в нем больше нет информации для чтения, то возвращаемое значение равно -1.

Если же считать из потока данные не удается из-за каких-то ошибок или сбоев, кидается исключение java.io.IOException. Этот класс наследуется от Exception, т.е. его всегда необходимо явно обрабатывать. Дело в том, что каналы передачи информации, будь то Интернет или, например, жесткий диск, могут давать сбои независимо от того, насколько

хорошо написана программа. А это означает, что нужно быть готовым к ним, чтобы пользователь не потерял нужные данные.

Метод read() - это абстрактный метод, но именно с соблюдением всех этих условий он должен быть реализован в классах-наследниках.

На практике обычно приходится считывать не один, а сразу несколько байт - то есть массив байт. Для этого используется метод read(), где в качестве параметров передается массив byte[]. При выполнении этого метода в цикле производится вызов абстрактного метода read() (определенного без параметров), и результатами заполняется переданный массив. Количество байт, считываемое таким образом, равно длине переданного массива. Но при этом может так получиться, что в потоке данные закончатся еще до того, как будет заполнен весь массив. То есть, возможна ситуация, когда в потоке данных (байт) содержится меньше чем длина массива. Поэтому метод возвращает значение int, указывающее, сколько байт было реально считано. Понятно, что это значение может быть от 0 до величины длины переданного массива.

Если же мы изначально хотим заполнить не весь массив, а только его часть, то для этих целей используется метод read(), которому кроме массива byte[], передаются еще два int значения. Первое - это позиция в массиве, с которой следует начать заполнение, второе - количество байт, которое нужно считать. Такой подход, когда для получения данных передается массив и два int числа - offset (смещение) и length (длина), является довольно распространенным и часто встречается не только в пакете java.io.

При вызове методов read(), возможно возникновение такой ситуации, когда запрашиваемые данные еще не готовы к считыванию. Например, если мы считываем данные, поступающие из сети, и они еще просто не пришли. В таком случае нельзя сказать, что данных больше нет, но и считать тоже нечего - выполнение останавливается на вызове метода read() и получается "зависание".

Что бы узнать, сколько байт в потоке готово к считыванию - используется метод available(). Этот метод возвращает значение типа int, которое показывает, сколько байт в потоке готово к считыванию. При этом не стоит путать это количество байт, готовых к считыванию, с тем количеством байт, которые вообще можно будет считать из этого потока. Метод available() возвращает число - количество байт, именно на данный момент, готовых к считыванию.

Когда работа с входным потоком данных окончена, его следует закрыть. Для этого вызывается метод close(). Этим вызовом будут освобождены все системные ресурсы, связанные с потоком.

Точно так же, как InputStream - это базовый класс для потоков ввода, класс OutputStream - это базовый класс для потоков вывода.

В классе OutputStream, аналогичным образом, определяются три метода write() - один принимающий в качестве параметра int, второй byte[], и третий byte[], плюс два int-числа. Все эти методы возвращают void, то есть ничего не возвращают.

Метод write(int) является абстрактным, и должен быть реализован в классах - наследниках. Этот метод принимает в качестве параметра int, но реально записывает в поток только byte - младшие 8 бит в двоичном представлении. Остальные 24 бита будут проигнорированы. В случае возникновения ошибки этот метод кидает java.io.IOException, как, впрочем, и большинство методов, связанных с вводом-выводом.

Для записи в поток сразу некоторого количества байт, методу write() передается массив байт. Или, если мы хотим записать только часть массива - то передаем массив byte[], и два int-числа - отступ и количество байт для записи. Понятно, что если указать неверные параметры - например отрицательный отступ, отрицательное количество байт для записи, либо если сумма отступ+длина будет больше длины массива - во всех этих случаях кидается исключение IndexOutOfBoundsException.

Реализация потока может быть таковой, что данные записываются не сразу, а хранятся некоторое время в памяти. Например, мы хотим записать в файл какие-то данные, которые мы получаем порциями по 10 байт, и так 200 раз подряд. В таком случае вместо 200 обращений к файлу удобней будет скопить все эти данные в памяти, а потом одним заходом записать все 2000 байт. То есть класс выходного потока может использовать некоторый свой внутренний механизм для буферизации (временного хранения перед отправкой) данных. Что бы убедиться, что данные записаны в поток, а не хранятся в буфере, вызывается метод flush(), определенный в OutputStream. В этом классе его реализация пустая, но если какой-либо из наследников использует буферизацию данных, то этот метод должен быть в нем переопределен.

Когда работа с потоком закончена, его следует закрыть. Для этого вызывается метод close(). Этот метод сначала освобождает буфер, после чего поток закрывается, и освобождаются все системные ресурсы с ним связанные. Закрытый поток не может выполнять операции вывода и не может быть открыт заново. В классе OutputStream реализация метода close() не производит никаких действий.

Итак, классы InputStream и OutputStream определяют необходимые методы для работы с байтовыми потоками данных. Эти классы являются абстрактными. Их задача - определить общий интерфейс для классов, которые получают данные из различных источников. Такими источниками могут быть, например, массив байт, файл, строка и т.д. Все они, или, по крайней мере, наиболее широко используемые, будут рассмотрены далее.

## 1.2. Классы-реализации потоков данных

### 1.2.1. Классы ByteArrayInputStream и ByteArrayOutputStream

Самый естественный и простой источник, откуда можно считывать байты - это, конечно, массив байт. Класс ByteArrayInputStream представляет поток, считывающий данные из массива байт. Этот класс имеет конструктор, которому в качестве параметра передается массив byte[]. Соответственно, при вызове методов read(), возвращаемые данные будут браться именно из этого массива. Например:

```
byte[] bytes = {1,-1,0};
ByteArrayInputStream in = new ByteArrayInputStream(bytes);
int readedInt = in.read(); // readedInt=1
System.out.println("first element read is: " +readedInt);
readedInt = in.read(); // readedInt=255. Однако (byte)readedInt даст значение
 -1
System.out.println("second element read is: " +readedInt);
readedInt = in.read(); // readedInt=0
System.out.println("third element read is: " +readedInt);
```

Если запустить такую программу, на экране отобразится следующее:

```
first element read is: 1
second element read is: 255
third element read is: 0
```

При вызове метода read() данные считывались из массива bytes, переданного в конструктор ByteArrayInputStream. В данном примере второе считанное значение не равно -1, как можно было бы ожидать. Вместо этого оно равно 255. Что бы понять, почему это произошло, нужно вспомнить, что метод read считывает byte, но возвращает значение int - полученное добавлением необходимого числа нулей (в двоичном представлении). Байт, равный -1 в двоичном представлении имеет вид 11111111 и, соответственно, число типа, получаемое приставкой 24-х нулей, равно 255(в десятичной системе). Однако если непосредственно привести его к byte, получим исходное значение.

Аналогично, для записи байт в массив, используется класс ByteArrayOutputStream. Этот класс использует внутри себя объект byte[], куда записывает данные, передаваемые при вызове методов write(). Что бы получить записанные в массив данные, вызывается метод toByteArray(). Пример:

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
out.write(10);
out.write(11);
byte[] bytes = out.toByteArray();
```

В этом примере в результате массив bytes будет состоять из двух элементов: 10 и 11.

Использование классов ByteArrayInputStream и ByteArrayOutputStream может быть очень удобно, когда нужно проверить, что именно записывается в выходной поток. Например, при отладке и тестировании сложных процессов записи и чтения из потоков. Использование этих классов выгодно тем, что можно сразу просмотреть результат, и не нужно создавать ни файл, ни сетевое соединение, ни что-либо еще.

### 1.2.2. Классы FileInputStream и FileOutputStream

Классы FileInputStream используется для чтения данных из файла. Конструктор этого класса в качестве параметра принимает название файла, из которого будет производиться считывание. При указании строки имени файла нужно учитывать, что именно эта строка и будет передана системе, поэтому формат имени файла и пути к нему может различаться на различных платформах. Если при вызове этого конструктора передать строку, указывающую на директорию либо на не существующий файл, то будет брошено java.io.FileNotFoundException.

Понятно, что если объект успешно создан, то при вызове его методов read(), возвращаемые значения будут считываться из указанного файла.

Для записи байт в файл используется класс FileOutputStream. При создании объектов этого класса, то есть при вызовах его конструкторов кроме указания файла, так же можно указать, будут ли данные дописываться в конец файла либо файл будет перезаписан. При этом, если указанный файл не существует, то сразу после создания FileOutputStream он будет создан. После, при вызовах методов write(), передаваемые значения будут записываться

в этот файл. По окончании работы необходимо вызвать метод close(), что бы сообщить системе, что работа по записи файла закончена. Пример:

```
byte[] bytesToWrite = {1, 2, 3};
byte[] bytesReaded = new byte[10];
String fileName = "d:\\test.txt";
try {
// Создать выходной поток
 FileOutputStream outFile = new FileOutputStream(fileName);
 System.out.println("Файл открыт для записи");
 // Записать массив
 outFile.write(bytesToWrite);
 System.out.println("Записано: " + bytesToWrite.length + " байт");
 // По окончании использования должен быть закрыт
 outFile.close();
 System.out.println("Выходной поток закрыт");
 // Создать входной поток
 FileInputStream inFile = new FileInputStream(fileName);
 System.out.println("Файл открыт для чтения");
 // Узнать, сколько байт готово к считыванию
 int bytesAvailable = inFile.available();
 System.out.println("Готово к считыванию: " + bytesAvailable + " байт");
 // Считать в массив
 int count = inFile.read(bytesReaded,0,bytesAvailable);
 System.out.println("Считано: " + count + " байт");
 inFile.close();
 System.out.println("Входной поток закрыт");
} catch (FileNotFoundException e) {
 System.out.println("Невозможно произвести запись в файл: " + fileName);
} catch (IOException e) {
 System.out.println("Ошибка ввода/вывода: " + e.toString());
}
```

При работе с FileInputStream метод available() практически наверняка вернет длину файла, то есть число байт, сколько вообще из него можно считать. Но не стоит закладываться на это при написании программ, которые должны устойчиво работать на различных платформах - метод available() возвращает число, сколько байт может быть на данный момент считано без блокирования. Тот, момент, что в подавляющем большинстве случаев это число и будет длиной файла, является всего лишь частным случаем работы на некоторых платформах.

В данном примере для наглядности закрытие потоков производится сразу же после окончания их использования в основном блоке. Однако считается хорошей практикой закрывать потоки в finally блоке.

```
} finally {
try{inFile.close();}catch(IOException e){};
}
```

Такой подход гарантирует, что поток будет закрыт, и будут освобождены все системные ресурсы с ним связанные.

### 1.2.3. PipedInputStream и PipedOutputStream

Классы PipedInputStream и PipedOutputStream характерны тем, что их объекты всегда используются в паре - к одному объекту PipedInputStream привязывается точно один объект PipedOutputStream. Эти классы могут быть полезны, если необходимо данные и записать и считать в пределах одного выполнения одной программы.

Используются следующим образом: создается по объекту PipedInputStream и PipedOutput-Stream, после чего они могут быть соединены между собой. Один объект PipedOutputStream может быть соединен с ровно одним объектом PipedInputStream и наоборот. Соединенный - означает, что если в объект PipedOutputStream записываются данные, то они могут быть считаны именно в соединенном объекте PipedInputStream. Такое соединение можно обеспечить либо вызовом метода connect() с передачей соответствующего объекта PipedStream, либо передать этот объект еще при вызове конструктора.

В следующем примере показано использование связки PipedInputStream и PipedOutput-Stream.

```
try {
int countRead = 0;
byte[] toRead = new byte[100];
PipedInputStream pipeIn = new PipedInputStream();
PipedOutputStream pipeOut = new PipedOutputStream(pipeIn);
// Считвать в массив, пока он полностью не будет заполнен
while(countRead<toRead.length){
// Записать в поток некоторое количество байт
for(int i=0; i<(Math.random()*10); i++){
pipeOut.write((byte)(Math.random()*127));
}
// Считать из потока доступные данные,
// добавить их к уже считанным.
int willRead = pipeIn.available();
if(willRead+countRead>toRead.length)
//Нужно считать только до предела массива
 willRead = toRead.length-countRead;
 countRead += pipeIn.read(toRead, countRead, willRead);
}
} catch (IOException e) {
pr("Impossible IOException occur: ");
e.printStackTrace();
}
```

Данный пример носит чисто демонстративный характер, так как выгода от использования этих классов в основном проявляется при разработке многопоточного приложения. Так, например, если программа выполняется в нескольких потоках выполнения (поток выполнения - Thread, но перевод на русский язык двух понятий Stream и Thread - совпадает), организовать передачу данных между потоками удобно с помощью объектов классов PipedStream. Для этого достаточно создать связанные объекты классов PipedInputStream

и PipedOutputStream, после чего передать их в соответствующие Thread. Тогда поток выполнения, где производится считывание из потока может содержать подобный код:

```
// inStream - объект класса PipedInputStream
try{
while(true){
byte[] readedBytes = null;
synchronized(inStream){
int bytesAvailable = inStream.available();
 readedBytes = new byte[bytesAvailable];
 inStream.read(readedBytes);
}
// do some work with readedBytes
// …
}catch(IOException e){
/* IOException будет брошено, когда поток inStream либо связанный с ним
PipedOutputStream будет закрыт, после чего будет произведена попытка считывания
 из inStream */
System.out.println("работа с потоком inStream завершена");
}
```

Так как с объектом inStream одновременно могут работать несколько потоков выполнения, блок synchronized(inStream){…} гарантирует, что во время между вызовами inStream.available() и inStream.read(…), ни в каком другом потоке выполнения не будет произведено считывание из inStream. Поэтому вызов inStream.read(readedBytes) не приведет к блокировке, и все данные, готовые к считыванию - будут считаны.

### 1.2.4. StringBufferInputStream

StringBufferInputStream - производит считывание данных, получаемых преобразованием символов строки в байты. По понятным причинам этот класс не имеет аналога в классах выходных потоков.

Если для каких-либо целей, мы хотим считать строку - объект String как набор byte, то можем воспользоваться классом StringBufferInputStream. Например, если мы уже используем потоки, а некоторые данные получили в виде строки, можно организовать считывание данных из нее через единый интерфейс. При создании объекта StringBufferInputStream необходимо конструктору передать объект String. Данные, возвращаемые методом read(), будут браться именно из этого объекта String. При этом символы будут преобразовываться в байты не совсем точно - будут учитываться только младшие 8 бит у символа(в то время как тип char состоит их 2-х байт).

### 1.2.5. SequenceInputStream

Класс SequenceInputStream считывает данные из других двух и более входных потоков. При этом можно указать два потока или их список. Данные будут вычитываться последовательно - сначала все данные из первого потока в списке, потом из второго, и так далее. Конец потока SequenceInputStream будет достигнут только тогда, когда будет достигнут конец потока, последнего в списке.

При создании объекта этого класса в конструктор в качестве параметров передаются объекты InputStream - либо два экземпляра либо Enumeration из них. Когда вызывается метод read(), SequenceInputStream пытается считать байт из текущего входного потока. Если в нем больше нет данных(считанное из него значение равно -1), у этого входного потока вызывается метод close(), и следующий входной поток становится текущим. Так до тех пор, пока последний входной поток не станет текущим, и из него не будут считаны все данные. Тогда, если при считывании обнаруживается, что в текущем входном потоке нет больше данных, и больше нет входных потоков SequenceInputStream возвращет -1, то есть объявляет, что данных больше нет. SequenceInputStream автоматически вызывает вызов метода close() у входных потоков, у которых достигнут конец. Вызов метода close() у SequenceeInputStream закрывает этот поток, предварительно закрыв все содержащиеся в нем входные потоки.

Пример:

```
FileInputStream inFile1 = null;
FileInputStream inFile2 = null;
SequenceInputStream sequenceStream = null;
FileOutputStream outFile = null;
try {
inFile1 = new FileInputStream("file1.txt");
 inFile2 = new FileInputStream("file2.txt");
 sequenceStream = new SequenceInputStream(inFile1, inFile2);
 outFile = new FileOutputStream("file3.txt");
 int readedByte = sequenceStream.read();
 while(readedByte!=-1){
  outFile.write(readedByte);
  readedByte = sequenceStream.read();
 }
} catch (IOException e) {
System.out.println("IOException: " + e.toString());
} finally {
 try{sequenceStream.close();}catch(IOException e){};
 try{outFile.close();}catch(IOException e){};
}
```

В результате выполнения этого примера в файл file3.txt будет записано содержимое файлов file1.txt и file2.txt - сначала полностью file1.txt и потом file2.txt. Закрытие потоков производится в блоке finally. При этом, так как при вызове метода close() может возникнуть IOException, то каждый такой вызов должен быть взят в try-catch блок. Причем в отдельный try-catch блок взят каждый вызов метода close() - для того, что бы все потоки были закрыты независимо от возникновения исключений при закрытии других потоков. При этом нет необходимости закрывать потоки inFile1 и inFile2 - они будут автоматически закрыты при использовании в sequnceStream - либо когда в них закончатся данные, либо при вызове у sequencecStream метода close().

Объект SequenceInputStream можно было создать и другим способом: сначала получить объект Enumeration, содержащий все потоки, и передать этот объект Enumeration в конструктор SequenceInputStream:

```
Vector vector = new Vector();
vector.add(new StringBufferInputStream("Begin file1\n"));
vector.add(new FileInputStream("file1.txt"));
vector.add(new StringBufferInputStream("\nEnd of file1, begin file2\n"));
vector.add(new FileInputStream("file2.txt"));
vector.add(new StringBufferInputStream("\nEnd of file2"));
Enumeration enum = vector.elements();
sequenceStream = new SequenceInputStream(enum);
```

Если заменить в предыдущем примере инициализацию sequenceeStream на приведенную здесь, то в файл file3.txt кроме содержимого файлов file1.txt и file2.txt будут записаны еще и три строки - одна в начале файла, одна между содержимым файлов file1.txt и file2.txt и еще одна дописана в конец file3.txt.

## 1.3. Классы FilterInputStreeam и FilterOutputStream. Их наследники.

Задачи, возникающие при вводе/выводе крайне разнообразны - этот может быть считывание байтов из файлов, объектов из файлов, объектов из массивов, буферизованное считывание строк из массивов. В такой ситуации решение путем простого наследования приводит к возникновению слишком большого числа подклассов. Решение же, когда требуется совмещение нескольких свойств, высокоэффективно в виде надстроек. (В ООП этот паттерн называется адаптер.) Надстройки - наложение дополнительных объектов для получения новых свойств и функций. Таким образом, необходимо создать несколько дополнительных объектов - адаптеров к классам ввода/вывода. В java.io их еще называют фильтрами. При этом надстройка-фильтр, включает в себя интерфейс объекта, на который надстраивается, и поэтому может быть в свою очередь дополнительно быть надстроена.

В java.io интерфейс для таких надстроек ввода/вывода предоставляют классы FilterInput-Stream (для входных потоков) и FilterOutputStream (для выходных потоков). Эти классы унаследованы от основных базовых классов ввода/вывода - InputStream и OutputStream соответственно. Конструкторы этих классов принимают в качестве параметра объект InputStream и имеют модификатор доступа protected. Сами же эти классы являются базовыми для надстроек. Поэтому только наследники могут вызывать его(при их создании), передавая переданный им поток. Таким образом обеспечивается некоторый общий интерфейс для надстраиваемых объектов.

Надстройки можно попробовать разделить на два основных типа - одни меняют данные, с которыми может работать поток, другие меняют внутренние механизмы потока: производят буферизацию, позволяют вернуть в поток считанные байты, или же посчитать количество считанных строк.

### 1.3.1. BufferedInputStream и BufferedOutputStream

На практике, при считывании с внешних устройств, ввод данных почти всегда необходимо буферизировать. Для буферизации данных и служат классы BuffereInputStream и BufferedOutputStream.

BufferedInputStream - содержит в себе массив байт, который служит буфером для считываемых данных. То есть, когда байты из потока считываются (вызов метода read()) либо пропускаются(метод skip()), сначала перезаписывается этот буферный массив, при

этом считываются сразу много байт за раз. Так же класс BufferedInputStream добавляет поддержку методов mark() и reset(). Эти методы определены еще в классе InputStream, но их реализация по умолчанию бросает исключение IOException. Метод mark() запоминает точку во входном потоке и метод reset() приводит к тому, что все байты, считанные после наиболее позднего вызова метода mark(), будут считаны заново, прежде чем новые байты будут считываться из содержащегося входного потока.

BufferedOutputStream - при использовании объекта этого класса, запись производится без необходимости обращения к устройству ввода/вывода при записи каждого байта. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и, соответственно, запись в него произойдет, когда буфер будет полностью заполнен. Освобождение буфера с записью байт на устройство вывода можно обеспечить и непосредственно - вызовом метода flush(). Так же буфер будет освобожден непосредственно перед закрытием потока (вызов метода close()). При вызове этого метода также будет закрыт и поток, над которым буфер надстроен.

Следующий пример наглядно демонстрирует повышение скорости считывания данных из файла с использованием буфера.

```
try {
 String fileName = "d:\\file1";
 InputStream inStream = null;
 OutputStream outStream = null;
 //Записать в файл некоторое количество байт
 long timeStart = System.currentTimeMillis();
 outStream = new FileOutputStream(fileName);
 outStream = new BufferedOutputStream(outStream);
 for(int i=1000000; --i>=0;){
  outStream.write(i);
 }
 long time = System.currentTimeMillis() - timeStart;
 System.out.println("Writing durates: " + time + " millisec");
 outStream.close();
 // Определить время считывания без буферизации
 timeStart = System.currentTimeMillis();
 inStream = new FileInputStream(fileName);
 while(inStream.read()!=-1){
 }
 time = System.currentTimeMillis() - timeStart;
 inStream.close();
 System.out.println("Direct read durates " + (time) + " millisec");
 timeStart = System.currentTimeMillis();
 inStream = new FileInputStream(fileName);
 inStream = new BufferedInputStream(inStream);
 while(inStream.read()!=-1){
 }
 time = System.currentTimeMillis() - timeStart;
 inStream.close();
 System.out.println("Buffered read durates " + (time) + " millisec");
```

```
 } catch (IOException e) {
  pr("IOException: " + e.toString());
  e.printStackTrace();
 }
```

Результатом могут быть, например, такие значения:

```
Writing durates: 359 millisec
Direct read durates 6546 millisec
Buffered read durates 250 millisec
```

В данном случае не производилось никаких дополнительных вычислений, занимающих процессорное время, только запись и считывание из файла. При этом считывание с использованием буфера заняло в 10 (!) раз меньше времени, чем аналогичное без буферизации. Для более быстрого выполнения программы запись файл производилась с буферизацией, однако ее влияние на скорость записи нетрудно проверить, убрав из программы строку, создающую BufferedOutputStream. Оставляем это для самостоятельной проверки.

Классы BufferedInputStream и BufferedOutputStream добавляют только внутреннюю логику по обработке запросов, они не добавляют никаких дополнительных методов. Следующие два фильтра как раз добавляют некоторые дополнительные возможности при работе с потоками. Однако, первый из них - LineNumberInputStream объявлен deprecated начиная с версии jdk 1.1, и использовать его не рекомендуется.

### 1.3.2. LineNumberInputStream

Расширяет функциональность InputStream тем, что дополнительно производит подсчет, сколько строк было считано из потока. Номер строки, на которой в данный момент происходит чтение можно узнать вызовом метода getLineNumber(). Так же можно и перейти к определенной строке, вызовом метода setLineNumber(int lineNumber).

Под строкой при этом понимается набор байт, оканчивающийся либо '\n' либо '\r' либо их комбинацией '\r\n' именно в этой последовательности. Аналогичный класс для OutputFilter - отсутствует.

### 1.3.3. PushBackInputStream

Этот фильтр позволяет вернуть во входной поток считанные из него данные. Это действие производится вызовом метода unread(). Понятно, что обеспечивается такая функциональность за счет наличия в классе некоторого буфера - массива байт, который заполняется при считывании из потока. Если будет произведен откат, то потом просто эти данные будут выдаваться еще раз, как будто только были считаны. При конструировании можно указать максимальное количество байт, которое может быть возвращено в поток. Аналогичный класс для OutputFilter - отсутствует.

Следующий класс также является deprecated, и его не рекомендуется применять при написании программ, работающих с различными кодировками. Однако он продолжает активно использоваться в силу исторических причин, так как статические поля out и err класса System ссылаются на объекты именно этого класса, и, поэтому его необходимо знать.

### 1.3.4. PrintStream

Используется для конвертации и записи строк в байтовый поток. В классе PrintStream определен метод print(), принимающий различные примитивные типы java а так же Object. При вызове этих методов передаваемые данные будут сначала превращены в строку вызовом метода String.valueOf(), после чего записаны в поток. То есть, в поток данные будут записаны в представлении, удобном для прочтения человеком. При этом, даже если возникает исключение, то оно обрабатывается внутри метода print и дальше НЕ кидается. При записи строки используется кодировка, определяемая из настроек системы, где выполняется программа. При этом PrintStream не поддерживает Unicode. Вообще же для работы с текстовыми данными следует использовать Writer`ы. Аналогичный класс для InputFilter - отсутствует.

Итак, ряд классов-потоков уже устарел и не рекомендуется к применению. Тем не менее они были описаны в основном для справочного материала и полноты картины. Действительный интерес и практические выгоды от использования предоставляет следующая пара классов.

### 1.3.5. DataInputStream и DataOutputStream

До сих пор речь шла только о считывании и записи в поток данных в виде byte. Для работы с другими примитивными типами данных java, определены интерфейсы DataInput и DataOutput, и существующие их реализации - классы-фильтры DataInputStream и DataOutputStream. Их место в иерархии классов ввода/вывода можно увидеть на диаграммах классов, приведенных на рис.1 и рис.2.

Интерфейсы DataInput и DataOutput определяют, а классы DataInputStream и DataOutput-Stream, соответственно реализуют, методы считывания и записи всех примитивных типов данных. При этом происходит конвертация этих данных в byte и обратно. При этом в поток будут записаны байты, а ответственность за восстановление данных лежит только на разработчике - нужно считывать данные в виде тех же типов, в той же последовательности, как и производилась запись. То есть, можно, конечно, записать несколько раз int или long, а потом считывать их как short или что-нибудь еще - считывание произойдет корректно и никаких предупреждений о возникшей ошибке не возникнет, но результат будет соответствующий - значения, которые никогда не записывались. Это наглядно показано в следующем примере:

```
try {
 ByteArrayOutputStream out = new ByteArrayOutputStream();
 DataOutputStream outData = new DataOutputStream(out);
 outData.writeByte(128); // it would write -128, as casted to byte
 outData.writeInt(128);
 outData.writeLong(128);
 outData.writeDouble(128);
 outData.close();
 byte[] bytes = out.toByteArray();
 InputStream in = new ByteArrayInputStream(bytes);
 DataInputStream inData = new DataInputStream(in);
 System.out.println("Read data in order it was writen: ");
 System.out.println("readByte: " + inData.readByte());
```

```
  System.out.println("readInt: " + inData.readInt());
  System.out.println("readLong: " + inData.readLong());
  System.out.println("readDouble: " + inData.readDouble());
  inData.close();
  System.out.println("read the same data in other order (first byte is
skipped):");
  in = new ByteArrayInputStream(bytes);
  inData = new DataInputStream(in);
  System.out.println("readInt: " + inData.readInt());
  System.out.println("readDouble: " + inData.readDouble());
  System.out.println("readLong: " + inData.readLong());
  inData.close();
 } catch (Exception e) {
  System.out.println("Impossible IOException occurs: " + e.toString());
  e.printStackTrace();
 }
```

Интерфейсы DataInput и DataOutput представляют возможность записи/считывания данных примитивных типов Java. Для аналогичной работы с объектами определены унаследованные от них интерфейсы ObjectInput и ObjectOutput соответственно. В java.io имеются реализации этих интерфейсов - классы ObjectInputStream и ObkectOutputStream. Процесс превращения в байты и обратно для объектов несколько сложнее чем для примитивных типов - записываться могут объекты разных классов, они могут иметь ссылки на другие объекты и т.д. Процесс превращения объекта в набор байт носит название сериализация (Serialization) и, соответственно, для обратного процесса - то есть из набора байт в объект - десериализация.

# 2. Serialization

В java имеется стандартный механизм превращения объекта в набор байт - сериализации. Для того, что бы объект мог быть сериализован, он должен реализовать интерфейс java.io.Serializable(соответствующее объявление должно явно присутствовать в классе объекта или, по правилам наследования, неявно в родительском классе вверх по иерархии). Интерфейс java.io.Serializable не определяет никаких методов. Его присутствие только определяет, что объекты этого класса разрешено сериализовывать. При попытке сериализовать объект, не реализующий этот интерфейс, будет брошено

```
  java.io.NotSerializableException
```

После того, как объект был сериализован, то есть превращен в последовательность байт, его по этой последовательности можно восстановить, при этом восстановление можно проводить на любой машине (вне зависимости от того, где проводилась сериализация), то есть последовательность байт можно передать на другую машину по сети или любым другим образом, и там провести десериализацию. При этом не имеет значения опереционная система под которой запущена Java - например, можно создать объект на машине с ОС Windows, превратить его в последовательность байт, после чего передать их по сети на машину с ОС Unix, где восстановить тот же объект.

Для работы по сериализации в java.io определены интерфейсы ObjectInput, ObjectOutput и реализующие их классы ObjectInputStream и ObjectOutputStream соответственно.

Для сериализации объекта нужен выходной поток OutputStream, который следует передать при конструировании ObjectOutputStream. После чего вызовом метода writeObject() сериализовать объект и записать его в выходной поток. Например:

```
ByteArrayOutputStream os = new ByteArrayOutputStream();
Object objSave = new Integer(1);
ObjectOutputStream oos = new ObjectOutputStream(os);
oos.writeObject(objSave);
```

Что бы просмотреть, во что превратился объект objSave, можно просмотреть содержимое массива

```
byte[] bArray = os.toByteArray();
```

А чтобы получить этот объект, можно десериализовать его из этого массива:

```
ByteArrayInputStream is = new ByteArrayInputStream(bArray);
ObjectInputStream ois = new ObjectInputStream(is);
Object objRead = ois.readObject();
```

Теперь можно убедиться, что считанный объект идентичен записанному:

```
System.out.println("readed object is: " + objRead.toString());
System.out.println("Object equality is: " + (objSave.equals(objRead)));
System.out.println("Reference equality is: " + (objSave==objRead));
```

Результатом выполнения приведенного выше кода будет:

```
readed object is: 1
Object equality is: true
Reference equality is: false
```

Как видим, восстановленный объект не является тем же самым(что очевидно - ведь восстановление могло происходить и на другой машине), но равным сериализованому.

Сериализуемый объект может хранить ссылки на другие объекты, которые в свою очередь так же могут хранить ссылки на другие объекты. И все они тоже должны быть восстановлены при десериализации. При этом, важно, что если несколько ссылок указывают на один и тот же объект, то в восстановленных объектах эти ссылки так же указывали на один и тот же объект. Следующий пример демонстрирует это свойство:

```
package demo.io;
import java.io;
class Point implements java.io.Serializable{
 double x;
 double y;
public Point(double x, double y) {
```

```
  this.x = x;
 this.y = y;
}
public String toString() {
return "("+x+","+y+") reference="+super.toString();
}
}
class Line  implements java.io.Serializable{
 Point point1;
 Point point2;
 int index;
public Line() {
 System.out.println("Constructing empty line");
}
Line(Point p1, Point p2, int index) {
System.out.println("Constructing line: " + index);
  this.point1 = p1;
  this.point2 = p2;
  this.index = index;
}
public int getIndex() {return index;}
public void setIndex(int newIndex) {index = newIndex;}
public void printInfo() {
  System.out.println("Line: " + index);
  System.out.println(" Object reference: " + super.toString());
  System.out.println(" from point "+point1);
  System.out.println(" to point "+point2);
}
}
public class Main {
public static void main(java.lang.String[] args) {
 Point p1 = new Point(1.0,1.0);
 Point p2 = new Point(2.0,2.0);
 Point p3 = new Point(3.0,3.0);
 Line line1 = new Line(p1,p2,1);
 Line line2 = new Line(p2,p3,2);
 System.out.println("line 1 = " + line1);
 System.out.println("line 2 = " + line2);
 String fileName = "d:\\file";
 try{
  // write objects to file
  FileOutputStream os = new FileOutputStream(fileName);
  ObjectOutputStream oos = new ObjectOutputStream(os);
  oos.writeObject(line1);
  oos.writeObject(line2);
  //change state of line 1 and write it again
  line1.setIndex(3);
  //oos.reset();
  oos.writeObject(line1);
```

```
  // close stream
  // it is enought to close only filter stream
  oos.close();
  os.close();
  //read objects
  System.out.println("Read objects:");
  FileInputStream is = new FileInputStream(fileName);
  ObjectInputStream ois = new ObjectInputStream(is);
  while(is.available()>0){
   Line line = (Line)ois.readObject();
   line.printInfo();
  }
  }catch(ClassNotFoundException e){
   e.printStackTrace();
  }catch(IOException e){
   e.printStackTrace();
  }
}
}
```

Первым делом стоит обратить внимание, что метод available() вызывался не у ObjectInput-Stream, а именно у InputStream, над которым объектный поток надстроен. Сделано так потому, что вызов метода available() у ObjectOutputStream возвращает количество байт, определяемое следующим образом: если достигнут конец потока - значение (-1), если поток находится в заблокированном режиме, то есть производит непосредственное считывание байт объекта из содержащегося в нем потока, то возвращается количество - сколько еще должно быть считано для завершения десериализации объекта(например, такое возможно, если идет долгое считывание объекта по сети), если же поток не находится в режиме блокировки, возвращается 0.

Выполнение этой программы приведет к выводу на экран примерно следующего:

```
Constructing line: 1
Constructing line: 2
Read objects:
Line: 1
 Object reference: study.java.Line@2dc5
 from point (1.0,1.0) reference=study.java.Point@322b
 to point (2.0,2.0) reference=study.java.Point@3481
Line: 2
 Object reference: study.java.Line@5214
 from point (2.0,2.0) reference=study.java.Point@3481
 to point (3.0,3.0) reference=study.java.Point@5435
Line: 1
 Object reference: study.java.Line@2dc5
 from point (1.0,1.0) reference=study.java.Point@322b
 to point (2.0,2.0) reference=study.java.Point@3481
```

Можно заметить, что объектные ссылки на Point - point2 у первой линии и point1 у второй линии совпадают - это по-прежнему один и тот же объект. Далее, третий записанный объект

- тот же, что и первый, при этом совпадают объектные ссылки. Но, когда мы записывали его, значение index было присвоено 3, а в десериализованном объекте это значение равно 1. Так произошло потом, что объект уже был сериализован, а в таком случае, что не быть записанным дважды, механизм сериализации некоторым образом для себя помечает, что объект уже записан в граф, и когда в очередной раз попадется ссылка на него, она будет указывать на уже сериализованный объект. Такой механизм необходим, что бы иметь возможность записывать связанные объекты, которые могут иметь перекрестные ссылки. В таких случаях необходимо отслеживать был ли объект уже сериализован, то есть нужно ли его записывать или достаточно указать ссылку на него.

Если класс содержит в качестве полей другие объекты, то эти объекты так же будут сериализовываться и поэтому тоже должны быть сериализуемы. В свою очередь, сериализуемы должны быть и все объекты, содержащиеся в этих сериализуемых объектах и т.д. Полный путь ссылок объекта по всем объектным ссылкам, имеющимся у него и у всех объектов на которые у него имеются ссылки, и т.д. - называется графом исходного объекта.

Поэтому, когда мы повторно записывали line1, состояние этого объекта не было записано, поскольку этот объект уже был помечен, как записанный в граф. Что бы указать, что сеанс сериализации завершен, и мы хотим заново записывать объекты, у ObjectOutputStream нужно вызвать метод reset(). В рассматриваемом примере для этого достаточно убрать комментарий вв строке

```
  //reset();
```

Если теперь запустить программу, то можно заметить, что line1 во второй раз записано полностью. Но при этом объектная ссылка на ее point2 не совпадает с объектной ссылкой point1 у line1. Этого, впрочем, и следовало ожидать - ведь во второй раз line1 была записана уже в новом сеансе сериализации, а значит, и все объекты Point были записаны заново.

Далее, в теле конструкторов Line специально были вставлены строки, выводящие на экран некоторые сообщения, сообщающие о вызове этих конструкторов. Как можно заметить, при десериализации никакие из этих конструкторов не вызывались - даже конструктор по умолчанию, а Point вообще такого не имеет. То есть при десериализации объект просто восстанавливается в том виде в каком он был, а не конструируется заново.

Однако, вопрос, на который следует обратить внимание - что происходит с состоянием объекта, унаследованным от суперкласса. Ведь состояние объекта определяется не только значениями полей, определенными в нем самом, но так же и таковыми, унаследованными от супепркласса. Сериализуемый подтип берет на себя такую ответственность, но только в том случае, если у суперкласса определен конструктор по умолчанию, объявленный с модификатором доступа таким, что будет доступен для вызова из рассматриваемого наследника. Этот конструктор будет вызван при десериализации. В противном случае, во время выполнения будет брошено исключение java.io.InvalidClassException . Что бы проверить, когда вызывается этот конструктор, можно в предыдущем примере добавить объявление класса

```
class AbstractEntity{
  public AbstractEntity(){
   System.out.println("Create Abstract Entity");
}
```

```
}
```

А какому-нибудь классу, скажем Point, указать наследование от этого класса AbstractEntity.

```
class Point extends AbstractEntity implements java.io.Serializable
```

В процессе десериализации, поля НЕ сериализуемых классов (родительских классов, НЕ реализующих интерфейс Serializable) инициируются вызовом конструктора без параметров. Такой конструктор должен быть доступен из сериализуемого их подкласса. Поля сериализуемого класса будут восстановлены из потока.

При обходе графа, может попасться объект, не реализующий интерфейс Serializable. В этом случае будет брошено java.io.NotSerializableException .

Весьма познавательно будет так же попробовать провести следующий эксперимент: выполнить программу из предыдущего примера. После ее выполнения на диске останется сохраненным файл, куда были сериализованы три объекта. Теперь удалим класс Point, то есть нужно сделать так, что бы файл Point.class не был доступен по путям, указанным в переменной CLASSPATH. Если теперь попробовать считать объект из файла "d:\file", при вызове метода readObject() произойдет исключение ClassNotFoundException. Если исключение обработать и попытаться продолжить считывание из ObjectInputStream, будет брошено java.io.StreamCorruptedException. Но из InputStream над которой надстроен ObjectInputStream в таком случае будет считано лишь некоторое количество байт, но это количество, спецификацией не оговаривается и может изменяться в зависимости от реализации Java-машины, поэтому лучше избегать попыток продолжать считывание из такого потока в подобных ситуациях. Получить java.io.StreamCorruptedException можно и более простым путем, для этого достаточно считать произвольное количество байт непосредственно из потока InputStream. В таком случае ObjectInputStream замечает, что окольными путями были считаны данные из потока, и будет брошено исключение java.io.StreamCorruptedException . Однако, если произвести считывание не из InputStream а вызовом того же метода read() у ObjectInputStream, будет возвращено значение (-1), то есть как будто достигнут конец потока. В ObjectOutputStream так же могли быть записаны примитивные типы Java(ведь ObjectOutput наследуется от DataOutput). При попытке считывания объекта, когда следующим на очереди считывания идет некоторый примитивный тип Java, будет брошено исключение java.io.OptionalDataException. Можно рассмотреть еще несколько возможных вариантов некорректного использования ObjectInputStream, но совершенно ясно, что если мы хотим считать именно те данные, которые были записаны в поток, нужно считывать их именно в том порядке, в каком были записаны.

Если классу для сериализации и десериализации требуется своеобразный подход, его можно осуществить путем реализации следующих методов, с точным сохранением сигнатуры:

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException;
private void readObject(java.io.ObjectInputStream in) throws IOException,
ClassNotFoundException;
```

Методом writeObject записываются данные состояния объекта, и методом readObject они, соответственно, считываются. При этом можно вызвать стандартный механизм записи объекта, вызовом метода

```
out.defaultWriteObject();
```

И стандартный механизм считывания вызовом метода

```
  in.defaultReadObject();
```

Этот метод использует информацию из потока что бы присвоить значения полей сохраненного объекта, полям с соответствующими именами в текущем объекте. Использование такой комбинации методов удобно, если ожидается, что со временем классу будут добавлены новые поля.

Возможны ситуации, когда по некоторым причинам, необходимо самостоятельно управлять ходом сериализации и восстановления объекта. Например, опираясь на некоторую замысловатую логику, записывать не все поля, а только некоторые из них. Для такого гибкого управления процессом сериализации предназначен интерфейс java.io.Externalizable .

При использовании этого интерфейса в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию по состоянию экземпляра, должен обеспечить сам класс. Для получения классом полного контроля за форматом и содержанием потока, для объекта и его суперклассов, должны быть реализованы методы writeExternal() и readExternal() интерфейса Externalizable. Эти методы должны полностью координировать сохранение состояния, унаследованное от суперкласса.

При сериализации, сериализуемый объект первым делом проверяется на поддержку интерфейса Externalizable. Если объект реализует Externalizable, вызывается его метод writeExternal . Если объект не поддерживает Externalizable, но реализует Serializable, используется стандартная сериализация ObjectOutputStream. При восстановлении Externalizable объекта, экземпляр создается путем вызова public конструктора без аргументов, после чего вызывается метод readExternal . В противовес этому, объекты Serializable восстанавливаются путем считывания из потока ObjectInputStream.

Метод writeExternal имеет сигнатуру:

```
void writeExternal(ObjectOutput out) throws IOException;
```

Для сохранения состояния, вызываются методы ObjectInput для сохранения как примитивных значений так и объектов, связанных с сериализуемым. Для корректной работы, в соответствующем методе

```
  void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
```

В том же самом порядке эти значения должны быть считаны.

При управлении процессом сериализации не всегда имеет смысл обращаться к реализации интерфейса Externalizable или методов readObject, writeObject. Если вопрос состоит только в том, что нежелательно сохранять и восстанавливать некоторый объекты-члены экземпляра. Обычно подобное требование возникает, когда в объекте хранится некоторая конфиденциальная информация, например пароль. Даже если поле такого объекта описано как private, оно все равно будет сериализовано, то есть записано в поток, а значит, это

значение можно будет прочитать, а при умелом обращении и изменить. Так же может потребоваться пропустить сохранение объекта, десериализация которого все равно не будет иметь смысла в последующем - например сетевое соединение все равно нужно будет устанавливать заново. Один способ пропустить сохранение объекта - реализовать интерфейс Externalizable, или определить методы readObject и writeObject - тогда вообще вся запись и чтение проходят как будет указано. Однако в данном случае можно поступить намного проще - достаточно такое поле объявить с модификатором transient. Например, при сериализации класса Account, приведенном в следующем примере, сериализовываться будут только поля login и name

```
class Account implements java.io.Serializable {
 private String name;
 private String login;
 private transient String password;
 /* Some accessors and mutators for fields
 ...
 */
}
```

Когда объект восстанавливается, таким полям выставляется значение по умолчанию, для объектов это null.

## 2.1. Версии классов

Сериализованный объект может храниться сколь угодно долго, например, если записать его на диск. Тогда, на момент его десериализации может возникнуть такая ситуация, что в его класс уже внесены изменения - добавлены или изменены методы, поля и т.д. Некоторые такие изменения могут изменить класс таким образом, что десериализация станет невозможной. В этом случае попытка десериализации приведет к возникновению InvalidClassException. Например, если сериализовать объект класса User, определенного следующим образом

```
class User implements java.io.Serializable{
 String name;
}
```

После чего модифицировать класс, заменив поле name на два:

```
 class User implements java.io.Serializable{
  protected String firstName;
  protected String lastName;
 }
```

То при попытке десериализовать записанный ранее объект, будет брошено исключение InvalidClassException . Однако этого не произошло бы, если класс изменить следующим образом:

```
class User implements java.io.Serializable{
```

```
 private String name;
 String lastName;
}
```

Для отслеживания таких ситуаций, каждому классу присваивается его идентификатор (ID) версии. Он представляет собой число long (длина 64 бита), полученное при помощи хэш-функции. Для его вычисления используются имена классов, всех реализуемых интерфейсов, всех методов и полей класса. При десериализации объекта, идентификаторы класса и идентификатор, взятый из потока сравниваются.

Изменения, проводимые с классом можно разбить на две группы - совместимые, то есть изменения, которые можно производить в классе, и при этом поддерживать совместимость с ранними версиями, и несовместимые - изменения, нарушающие поддержку с ранней версией класса. Определить к какому типу относится изменение, можно, руководствуясь следующей логикой: так как предполагается, что поздние версии будут поддерживать более ранние, то поздние должны иметь такой набор полей, что в них возможно восстановить поля из старых записей.

Итак, к совместимым относятся следующие изменения:

- добавление поля к классу
- добавление или удаление суперкласса
- изменение модификаторов доступа полей
- удаление у полей модификаторов static или transient
- изменение кода методов, инициализаторов, конструкторов

К несовместимым относятся:

- Удаление поля
- изменение название пакета класса
- изменение типа поля
- добавление к полю экземпляра ключевого слова static или transient
- реализация Serializable вместо Externalizable или наоборот.

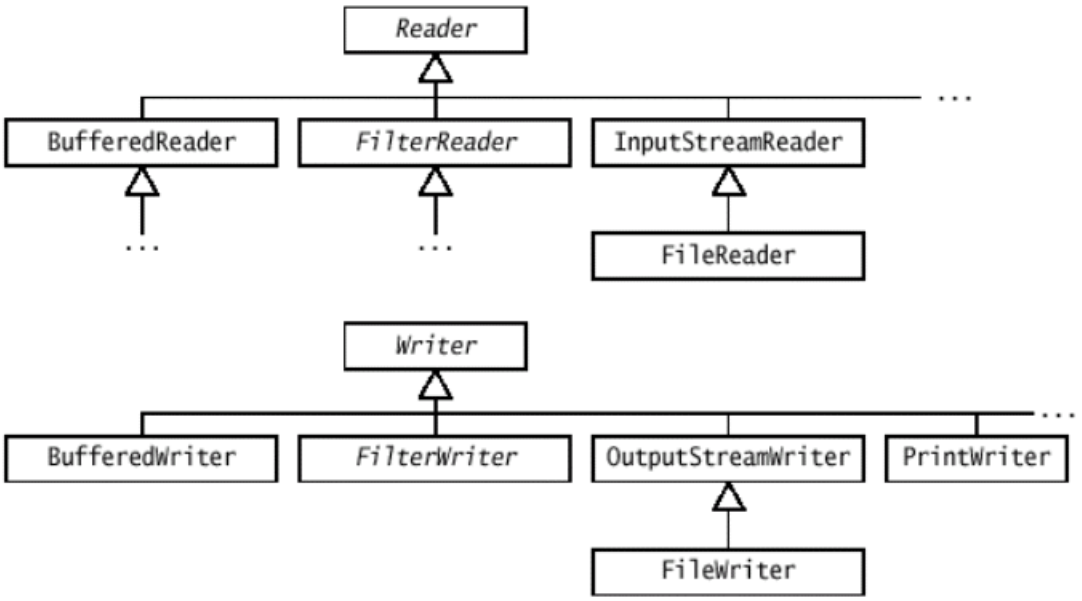
Другими словами, должна остаться возможность восстановить именно те поля, которые были записаны в поток при сериализации.

Обеспечение такой совместимости может стать неприятной задачей, если не позаботиться об этом заранее. В таких случаях сразу следует объявить, что класс реализует интерфейс Externalizable, и потом просто дорабатывать методы readExternalizable и writeExternalizable.

# 3. Классы Reader и Writer. Их наследники.

Рассмотренные классы - наследники InputStream и OutputStream работают с байтовыми данными. При этом, в обращении со строковыми данными поддерживаются 8-ми битные символы и зачастую неверно происходит при интернационализации обращение с 16-ти битными символами Unicode - именно на которых и основан примитивный тип char(символ) в Java. За правильное использование символов в операциях ввода/вывода предназначены

наследники классов Reader (чтение) и Writer (запись). Их иерархия представлена диаграммой на рис.3 .




Эта иерархия очень схожа с аналогичной для байтовых потоков InputStream и OutputStream.

В таблице 1 приведены соответствия классов для байтовых и символьных потоков.

| | |
|---|---|
| InputStream | Reader |
| OutputStream | Writer |
| ByteArrayInputStream | CharArrayReader |
| ByteArrayOutputStream | CharArrayWriter |
| Нет аналога | InputStreamReader |
| Нет аналога | OutputStreamWriter |
| FileInputStream | FileReader |
| FileOutputStream | FileWriter |
| FilterInputStream | FilterReader |
| FilterOutputStream | FilterWriter |
| BufferedInputStream | BufferedReader |
| BufferedOutputStream | BufferedWriter |
| PrintStream | PrintWriter |
| DataInputStream | Нет аналога |
| DataOutputStream | Нет аналога |
| ObjectInputStream | Нет аналога |
| ObjectOutputStream | Нет аналога |
| PipedInputStream | PipedReader |
| PipedOutputStream | PipedWriter |

| StringBufferInputStream | StringReader |
|---|---|
| Нет аналога | StringWriter |
| LineNumberInputStream | LineNumberReader |
| PushBackInputStream | PushBackReader |
| SequenceInputStream | Нет аналога |

Как видно из таблицы, различия крайне незначительны:

Для символьных потоков нет потока, аналогичного потоку SequenceInputStream последовательного считывания, и, конечно же, отсутствует преобразование в символьное представление примитивных типов Java и объектов (DateInput/Output, ObjectInput/Output).

В свою очередь, в символьных потоках присутствует поток записи символьных данных в строку, чего нет для байтовых потоков (скорее в силу излишества такой надобности при использовании байтовых потоков). Кроме того, в символьных потоках имеются классы-мосты, преобразующие символьные потоки в байтовые: InputStreamReader и OutputStreamWriter .

Можно обратить внимание и на остальные различия, которые впрочем, являются незначительными:

- Если классы - основы фильтров для символьных потоков: FilterReader и FilterWriter являются абстрактными, то аналогичные для байтовых: InputStream и OutputStream - не абстрактные.
- Методы: close класса Reader и flush и close класса Writer являются абстрактными, в то время как в соответствующих классах InputStream и OutputStream байтовых потоков они имеют просто пустую реализацию
- BufferedReader наследуется не от фильтра FilterReader, а напрямую от Reader
- LineNumberReader не наследуется от FilterReader, вместо этого он унаследован от BufferedReader, который в свою очередь напрямую унаследован от Reader
- FileReader и FileWriter унаследованы от классов-мостов InputStreamReader и OutputStreamWriter
- Метод available() класса InputStream, отсутствует в классе Reader, он заменен методом ready() - возвращающим вместо количества данных готовых к считыванию, булевое значение - готов ли поток к считыванию (то есть будет ли считывание произведено без блокирования)

В остальном же, использование символьных потоков идентично работе с байтовыми потоками. Так, программный код для записи символьных данных в файл, будет выглядеть примерно следующим образом:

```
String fileName = "d:\\file.txt";
FileWriter fw = null;
BufferedWriter bw = null;
FileReader fr = null;
BufferedReader br = null;
//Строка, которая будет записана в файл
String data = "Some data to be written and readed\n";
```

```
try{
 fw = new FileWriter(fileName);
 bw = new BufferedWriter(fw);
 System.out.println("Write some data to file: " + fileName);
 // Несколько раз записать строку
 for(int i=(int)(Math.random()*10);--i>=0;)bw.write(data);
 bw.close();
 fr = new FileReader(fileName);
 br = new BufferedReader(fr);
 String s = null;
 int count = 0;
 System.out.println("Read data from file: " + fileName);
 // Считывать данные, отображая на экран
 while((s=br.readLine())!=null)
  System.out.println("row " + ++count + " read:" + s);
 br.close();
}catch(Exception e){
 e.printStackTrace();
}
```

Классы-мосты InputStreamReader и OutputStreamWriter имеют возможность производить преобразование символов, используя различные кодировки. Кодировка задается при конструировании потока, путем передачи в конструктор в качестве параметра строки - названия кодировки. При указании названия, которое не соответствует никакой из известных кодировок, будет брошено исключение UnsupportedEncodingException. Вот некоторые из допустимых значений этой строки: "Cp1521", "UTF-8", "8859_1", "8859_2" и т.д..

# 4. Класс StreamTokenizer

Этот класс создается поверх существующего объекта либо InputStream либо Reader. Позволяет работать с данными посредством токенов - кусков данных, выделяемых из потока по определенным свойствам. Процесс разбора данных контролируется через таблицу со множеством флагов котрые могут быть выставлены в различные состояния. StreamTokenizer может распознавать идентификаторы, числа, строки в кавычках и различные стили комментариев. Обычно приложение сначала создает экземпляр этого класса, выставляет синтаксические таблицы, после чего многократно повторяет в цикле вызов метода nextToken, пока не будет возвращено значение StreamTokenizer.TT_EOF.

После вызова метода nextToken поле ttype содержит значение определяющее тип последнего считанного токена. Если последний считанный токен распознан как число, его значение содержится в поле nval. Если же токен распознан как слово, его значение заносится в поле sval.

Способ определения, как разбивать данные на эти куски - настраивается вызовом методов commentChar, ordinaryChar, parseNumbers, quoteChar, resetSyntax, slashSlashComments, slashStarComments, whitespaceChars, wordChars. По умолчанию используются такие настройки, что будут выделяться числа, и тексты(символы с соответствующими кодами) разделяемые переводом строки.

# 5. Работа с файловой системой.

## 5.1. Класс File

Если классы потоков осуществляют реальную запись и чтение данных, то класс File - это вспомогательный инструмент, призванный облегчить обращение с файлами и директориями.

Объект класса File является абстрактным представлением файла и пути к нему. Он устанавливает только соответствие с ним, при этом для создания объекта не важно, существует ли такой файл на диске. После создания можно можно сделать проверку, вызвав метод exists, который возвращает значение true, если файл существует. Создание или удаление объекта класса File никоим образом не отображается на реальных файлах. Этот класс не имеет методов для работы с содержимым файла. Объект File может указывать на директорию (узнать это можно вызовом метода isDirectory), тогда вызовом метода list можно получить список имен (массив String) файлов в ней (если объект File не указывает на директорию - будет возвращен null).

Следующий пример демонстрирует использование объектов класса File.

```
import java.io.*;
public class FileDemo {
 public static void findFiles(File file, FileFilter filter, PrintStream output)
 throws IOException{
  if(file.isDirectory()){
   File[] list = file.listFiles();
   for(int i=list.length; --i>=0;){
    findFiles(list[i], filter, output);
   }
  }else{
   if(filter.accept(file))
    output.println("\t" + file.getCanonicalPath());
  }
 }
 public static void main(String[] args) {
  class NameFilter implements FileFilter{
   private String mask;
   NameFilter(String mask){
    this.mask = mask;
   }
   public boolean accept(File file){
    return (file.getName().indexOf(mask)!=-1)?true:false;
   }
  }
  File pathFile = new File(".");
  String filterString = ".java";
  try{
   FileFilter filter = new NameFilter(filterString);
   findFiles(pathFile, filter, System.out);
  }catch(Exception e){
   e.printStackTrace();
```

```
    }
    System.out.println("work finished");
  }
}
```

При выполнении этой программы на экран будут выведены названия (в каноническом виде) всех файлов, с расширением ".java", содержащихся в текущей директории и всех ее под-директориях. Для определения, что файл имеет расширение ".java" использовался интерфейс FileFilter с реализацией в виде внутреннего класса NameFilter. Интерфейс FileFilter определяет только один метод accept, возвращающий значение, попадает ли переданный файл в условия фильтрации. Помимо этого интерфейса существует еще одна разновидность интерфейса фильтра - FilenameFilter, где метод accept определен несколько иначе: он принимает не объекта файла к проверке, а объект файл директории где находится файл для проверки и строку его названия. Для проверки совпадения с учетом регулярных выражений, нужно соответствующим образом реализовать метод accept. В конкретном приведенном примере можно было обойтись и без использования интерфейсов FileFilter или FilenameFilter. На практике их можно использовать для вызова методов list объектов File - в этих случаях будут возвращены файлы с учетом фильтра.

Так же класс File предоставляет возможность получения некоторой информации про файл:

- Методы canRead и canWrite - возвращается boolean значение, возможно ли будет приложению производить чтение и изменение содержимого из файла соответственно
- exists - возвращается boolean значение, существует ли такой файл на диске
- getName - возвращает строку - имя файла (или директории)
- getParent, getParentName - возвращают директорию, где файл находится в виде строки названия и объекта File соответственно
- getPath - возвращает путь к файлу (при этом в строку преобразуется абстрактный путь, на который указывает объект File)
- isAbsolutely - возвращает boolean значение, является ли абсолютным путь, которым указан файл. Определение, является ли путь абсолютным - зависит от системы, где запущена Java-машина. Так, для Windows - абсолютный путь начинается с указания диска, либо символом '\'. Для Unix - абсолютный путь начинается символом '/'
- isDirectory, isFile - возвращает boolean значение, указывает ли объект на директорию либо файл соответственно
- isHidden - возвращает boolean значение, указывает ли объект на скрытый файл
- lastModified
- length

Так же возможно изменить некоторые свойства файла - методы setReadOnly, setLastModified назначение которых очевидно из названия. Если нужно создать файл на диске - это можно сделать методами createNewFile, mkDir, mkDirs. Соответственно createNewFile создает файл (если таковой еще не существует), mkDir создает директорию если для нее все родительские уже существуют, а mkDir создаст директорию, вместе со всеми необходимыми родительскими.

Можно и удалить файл - для этого предназначены методы delete и deleteOnExit. При вызове метода delete, файл будет удален сразу же, а при вызове deleteOnExit по окончании работы Java-машины(только при корректном завершении работы), при этом отменить запрос не является возможным.

Таким образом класс File дает возможность достаточно полного управления файловой системы.

## 5.2. Класс RandomAccessFile

Этот класс реализует сразу два интерфейса: DataInput и DataOutput и, соответственно, может производить и запись, и чтение всех примитивных типов Java. Эти операции записи и чтения, как следует из названия, производятся с файлами. При этом эти операции можно производить поочередно, вдобавок произвольным образом перемещаясь по файлу - вызовом метода seek(узнать текущее положение указателя в файле можно вызовом метода getFilePointer) .

При создании объекта этого класса конструктору в качестве параметров нужно передать два параметра: файл, и режим работы. Файл, с которым будет проводиться работа указывается либо с помощью String - название файла, либо объектом File ему соответствующим. Режим работы(mode) - представляет собой объект String, который принимает одно из значений: "r"(чтение) либо "rw"(чтение и запись). При этом, если передать, что mode равно "r" и указать несуществующий файл(либо директорию), будет брошено исключение FileNotFoundException. Если указать, что mode равно "rw" и указать несуществующий файл, он будет незамедлительно создан(или же брошено исключение FileNotFoundException если это невозможно осуществить).

После создания объекта RandomAccessFile, можно воспользоваться методами интерфейсов DataInput и DataOutput для проведения с файлом операций считывания и записи. По окончании работы с файлом, его следует закрыть, вызвав метод close.

# 6. Заключение

В данной главе Вы ознакомились с таким важным понятием, как потоки данных. Потоки являются очень эффективным способом решения задач по передаче и получению данных независимо от используемых устройств ввода/вывода. Как Вы теперь знаете, именно в пакете java.io содержатся стандартные классы, решающие задачи обмена данными через наиболее распространенные источники (и приемники) данных. Классы этого пакета эффективно решают задачу как широкого спектра источников и получателей данных, так и различных форматов передачи информации.

Были рассмотрены базовые классы байтовых потоков InputStream и OutputStream, а также символьных потоков Reader и Writer. Все классы потоков явным или неявным образом наследуются от этих классов. Были детально рассмотрены классы потоков, реализующих работу с различными источниками данных – массив байт, символьный массив, файл, pipe (“канал”), строки. Показано назначение и способы эффективного использования фильтров, особое внимание уделено использованию BufferedInputStream и BufferedOutputStream, указано, какие классы являются не рекомендованными к использованию. Показано как можно записать (считать) в поток (из потока) различные примитивные значения Java – классы DataInputStream, DataOutputStream (а так же ObjectInputStream, ObjectOutputStream

и отдельно PrintWriter, BufferedReader). Особое внимание уделено описанию механизма сериализации. Показана возможность чтения данных из потоков с разбиением на токены (слова) – класс StreamTokenizer.

Так же приведены и описаны классы для работы с файловой системой – классы File и RandomAccessFile.

# 7. Контрольные вопросы

15-1. Какие источники могут быть использованы классами стандартных входных потоков java в качестве источника данных? Какие могут быть использованы выходными потоками?

a.) Следующие ресурсы могут быть использованы стандартными потоками в качестве источников данных:

- Файл – представляется объектом класса File
- Массив – представляется массивом byte[] или char[]
- Строка – представляется объектом String
- Сетевое соединение – входной поток получается вызовом getInputStream() у объекта класса java.net.Socket
- “Pipe” – получаемые данные передаются из соединенного объекта PipedOutputStream(PipedWriter)

Выходными потоками могут использоваться эти же самые ресурсы, кроме строк (String). Также, понятно, для сетевого соединения существует различие в получении выходного потока: производится вызовом getOutputStream() у объекта класса Socket.

15-2. Имея два объекта класса File, каким образом будет наиболее корректно узнать, указывают ли они на одну и ту же директорию (и на директорию ли)? Возможно ли только с помощью этих двух объектов удалить директорию? Если да, то как изменится содержимое другого объекта (если они действительно указывают на одну и туже директорию)?

a.) Вызов метода equals() объекта File приведет к лексикографическому сравнению пути файлов. То есть будут сравниваться строки, по которым был создан объект File. Правильным является привести оба объекта к каноническому виду (методы getCanonicalPath() или getCanonicalFile()) и сравнить эти представления. Проверить, указывает ли объект File на директорию, можно вызовом метода isDirectory(). Удалить файл (равно и директорию) можно вызовом метода delete() у объекта File. При этом директория будет удалена только в том случае, если в ней не содержится ни одног³о файл»а или директории. Если объект File указывает на несуществующий файл, то вызов его метода exist() вернет значение false – указывающее, что файл не существует на диске. Именно это и произойдет, если директория, на которую указывает некоторый объект File, будет удалена вызовом delete() через другую ссылку File.

15-3. От какого класса наследуются InputStream и OutputStream? Остальные классы потоков ввода/вывода?

a.) Все входные байтовые потоки наследуются(явно или неявно) от InputStream, а все выходные от OutputStream. Сами же абстрактные классы InputStream и OutputStream унаследованы от Object.

15-4. Если вызвать write(0x01234567) у экземпляра OutputStream, в каком порядке и какие байты будут записаны в выходной поток?

a.) Притом, что метод write() класса OutputStream принимает в качестве параметра значение int, на самом деле в поток будет записан только младший байт – в данном случае 0x67, то есть 16*6+7=103.

15-5. При каких условиях следующий метод вернет значение false?

```
public static boolean test(InputStream is) throws IOException {
 int value = is.read();
 return value == (value &amp;amp; 0xff);
}
```

a.) Выражение value == (value & 0xff) принимает значение true только в случае, если в двоичном представлении значение value содержит единицы только в младшем байте (значение 0xff имеет тип int и равно 255, но не –1). Такой результат будет получен в том случае, если считывание из потока произведено корректно – не был достигнут конец потока, и не было брошено исключение IOException.

15-6. Какие классы предоставляют методы для записи в поток двоичного представления значений примитивных типов Java?

a.) Для записи значений примитивных типов Java, предназначен интерфейс DataOutput. В пакете java.io этот интерфейс реализует класс DataOutputStream. Так же, неявно этот интерфейс реализован классом ObjectOutputStream – этот класс реализует интерфейс ObjectOutput, который расширяет интерфейс DataOutput.

15-7. Если необходимо записать (и после считать) несколько строк в файл (из файла), в каком порядке и какие следует настроить фильтры (и для чтения, и для записи)? Какие из них можно пропустить?

a.) Наиболее распространенным способом является следующая последовательность:

```
new PrintWriter(new BufferedWriter(new
FileWriter("file.txt")))
```

для записи, и, соответственно, для чтения:

```
new BufferedReader(new FileReader("file.txt"))
```

При этом BufferedWriter может быть опущен – он нужен только для обеспечения лучшей производительности путем буферизации (если требуется записать лишь несколько строк, этот класс может быть

действительно лишним). Так же, записать строки можно при желании и используя только лишь FileWriter, а считать - используя FileReader, хотя для удобства желательно воспользоваться классами PrintWriter и BufferedReader (или LineNumberReader – в зависимости от задач).

15-8. Что произойдет при попытке к одному объекту PipedWriter присоединить несколько различных объектов PipedReader? Что произойдет, если несколько раз подряд присоединять один и тот же PipedReader?

a.) К одному объекту PipedWriter может быть присоединен только один объект PipedReader. При попытке присоединить более одного PipedReader, будет брошено исключение new IOException("Already connected") . Это же исключение будет брошено, если подключать более одного раза один и тот же объект PipedReader – метод connect() реализован таким образом, что исключение new IOException("Already connected") будет брошено при попытке вызвать этот метод, когда либо сам объект PipedWriter либо переданный объект PipedReader уже подключен к некоторому объекту PipedWriter.

15-9. Значения каких примитивных типов Java могут быть переданы в качестве параметров методу write() класса Writer? Методу print() класса PrintWriter?

a.) В классе Writer метод write() является перегруженным и определен со следующими параметрами:

- write(char cbuf[])
- write(char cbuf[], int off, int len)
- write(int c)
- write(String str)
- write(String str, int off, int len)

То есть, из примитивных типов может быть передан только int (в случае вызова с передачей byte, char или short – будет произведено приведение к int).

Метод print() класса PrintWriter определен со всеми примитивными типами Java (а так же Object и String).

15-10. Какая кодировка используется классом OutputStreamWriter по умолчанию?

a.) Используемая кодировка зависит от системы, где запущена Java-машина.

15-11. Что будет записано в поток, если вызвать метод print() класса PrintWriter, передав в качестве параметра new File(“d:\\word.txt”) ?

a.) При таком вызове будет вызван метод print(Object obj) и, соответственно, в поток будет записано значение String.valueOf(obj). В свою очередь метод valueOf(Object obj) класса String определен таким образом, что в случае, если переданный объект не равен null, то будет возвращено значение obj.toString() . В данном случае, для объекта File, это будет значение, возвращаемое его методом getPath(), то есть строка, переданная в конструктор – “d:\\word.txt”.

15-12. Какие значения могут быть переданы в конструктор RandomAccessFile для указания режима доступа (чтение/запись)?

a.) Оба конструктора класса RandomAccessFile принимают объект String, указывающий режим доступа к файлу. Значение этой строки может принимать только одно из двух значений – либо “r”(чтение) либо “rw”(чтение и запись). В любом другом случае будет брошено исключение new IllegalArgumentException("mode must be r or rw") (в случае, если переданная строка является null, будет брошено исключение NullPointerException).

15-13. Какое значение следует передать методу seek() объекта RandomAccessFile, чтобы последний байт файла был считан одиночным вызовом read()?

a.) Узнать длину файла, на который указывает объект RandomAccessFile, можно вызовом метода length(). Что бы считать первый байт файла, следует вызвать seek() передав значение 0. соответственно, что бы считать последний байт, нужно методу seek() передать значение randomAccessFile.length()-1 .

15-14. Какие методы объявлены в интерфейсе Serializable?

a.) В интерфейсе java.io.Serializable не объявлено ни одного метода. Классы реализуют этот интерфейс, что бы указать, что объекты этого класса разрешены к сериализации.

15-15. Что произойдет, если записать в файл, используя ObjectOutputStream, значения типов long, int, byte именно в таком порядке, а считать в обратном, используя DataInputStream?

a.) DataInputStream запишет в поток байтовые представления long, int и byte, то есть сначала будут записаны 8 байт, образующие long, потом 4 байта, образующие int, и, наконец, еще один байт. При считывании, будут произведены обратные действия, то есть: для считывания byte будет считан один байт, для считывания int будут считаны следующие 4 байт и для считывания long будут считаны последние 8 байт. Понятно, что в случае, когда считывание производится не в том порядке, в каком была произведена запись, как в данном случае – чтение будет произведено успешно, но полученные значения byte, int и long могут (точнее - будут) отличаться от тех, которые были записаны.

# Программирование на Java

## Лекция 16. Введение в сетевые протоколы

### 20 апреля 2003 года

Авторы документа:

Николай Вязовик (Центр Sun технологий МФТИ) <vyazovick@itc.mipt.ru>
Евгений Жилин (Центр Sun технологий МФТИ) < gene@itc.mipt.ru>



Аннотация

Завершает курс лекция, в которой рассматриваются возможности построения сетевых приложений. Сначала дается краткое введение в сетевые протоколы, семиуровневую модель OSI, стек протоколов TCP/IP и описываются основные утилиты, предоставляемые операционной системой для мониторинга сети. Эти значния необходимы, поскольку библиотека java.net по сути является интерфейсом для работы с этим протоколами. Рассматриваются классы для соединений через высокоуровневые протоколы, протоколы TCP и UDP.

# Оглавление

# Лекция 16. Введение в сетевые протоколы

## Содержание лекции.

# 1. Основы модели OSI

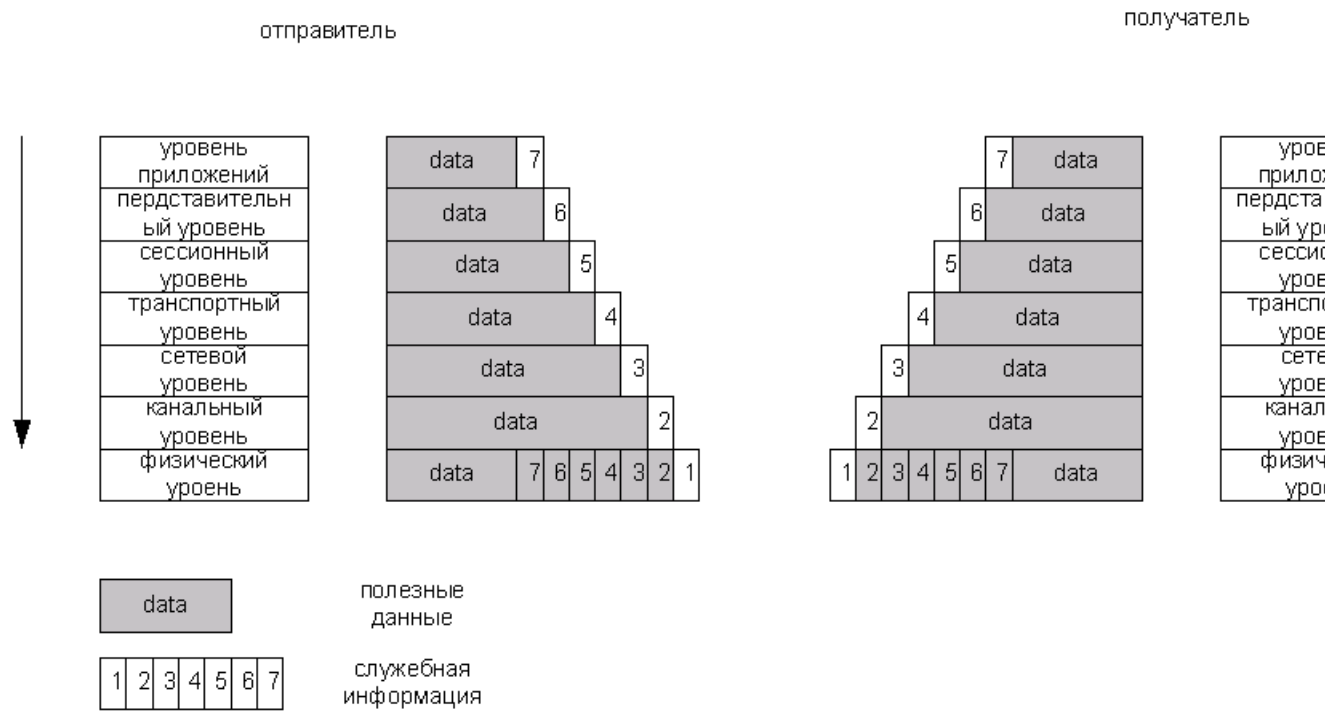
В течение последних нескольких десятилетий размеры и количество сетей значительно выросли. В 80-х годах имелось множество типов сетей. И практически каждая из них была построена на своем типе оборудования и программного обеспечения, зачастую не совместимых между собой. Это приводило к значительным трудностям при попытке соединить несколько различных типов сетей (например, различный тип адресации делал эти попытки практически безнадежными). Эта проблема была рассмотрена Всемирной Организацией по Стандартам (International Organization for Standardization, ISO), и было принято решение разработать модель сети, которая могла бы помочь разработчикам и производителям сетевого оборудования и программного обеспечения работать сообща. В результате в 1984 г. была разработана модель OSI - модель взаимодействия открытых систем (Open Systems Interconnected). Эта модель состоит из семи уровней. Схематично ее можно представить следующим образом:

| Номер уровня | Название уровня | Единица информации |
|---|---|---|
| Layer 7 | Уровень приложений | Данные (data) |
| Layer 6 | Представительский уровень | Данные (data) |
| Layer 5 | Сессионный уровень | Данные (data) |
| Layer 4 | Транспортный уровень | Сегмент (segment) |
| Layer 3 | Сетевой уровень | Пакет (packet) |
| Layer 2 | Уровень передачи данных | Фрейм (frame) |
| Layer 1 | Физический уровень | Бит (bit) |

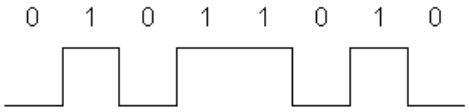
Хотя на сегодняшний день существуют разнообразные модели сети, большинство разработчиков придерживается именно этой общепризнанной схемы. OSI-модель позволяет разделить сетевые функции на уровни и понять, как происходит обмен данными по сети. Каждый уровень взаимодействует только с соседними уровнями, протокол взаимодействия стандартизован. Это позволяет использовать реализации сетевого и программного обеспечения от разных производителей в различных комбинациях. Например, протоколы высокого уровня (HTTP, FTP) не зависят от физических параметров используемой сети.

Рассмотрим процесс передачи информации между двумя компьютерами. Программное обеспечение формирует сообщение на уровне 7 (приложений), состоящее из заголовка и полезных данных. В заголовке содержится служебная информация, которая необходима уровню приложений адресата для обработки требуемой информации (например, это может быть информация о файле, который необходимо передать, или операции, которую необходимо выполнить). После того, как сообщение было сформировано, уровень приложений направляет его вниз на представительский уровень (layer 6). Полученное сообщение, состоящее из служебной информации уровня 7 и полезных данных, для уровня 6 представляется как одно целое сообщение (хотя уровень 6 может считывать служебную информацию уровня 7). Протокол представительского уровня выполняет требуемые действия на основании данных, полученных из заголовка уровня приложений, и добавляет заголовок своего уровня, в котором содержится необходимая информация для соответствующего (6-го) уровня адресата. Полученное в результате сообщение передается далее вниз сеансовому уровню, где также добавляется служебная информация. Дополненное сообщение передается на следующий транспортный уровень и т.д., на каждом последующем уровне (схематично это представлено на рис.1). При этом служебная

информация не обязательно добавляется в начало сообщения. Например, на 3-м уровне служебная информация добавляется к началу и концу сообщения (рис.2). В итоге получается сообщение, содержащее служебную информацию всех 7 уровней. Процесс добавления служебной информации называется инкапсуляцией (encapsulation).




Далее это сообщение передается через сеть в виде битов. Бит - это минимальная порция информации, которая может принимать значение 0 или 1. Таким образом, все сообщение кодируется в виде набора нулей и единиц, например 010110101. В простейшем случае на физическом уровне для передачи формируется электрический сигнал, состоящий их серии электрических импульсов (0 - нет сигнала, 1 - есть сигнал). Именно эта единица принята для измерения скорости передачи информации. Современные сети обычно предоставляют каналы с производительностью в десятки и сотни Кбит/с и Мбит/с.




Получатель на физическом уровне получает сообщение в виде электрического сигнала (рис.3). Далее происходит процесс обратный инкапсуляции - декапсуляция. На каждом

уровне происходит разбор служебной информации. После декапсуляции сообщения на первом уровне (считывания и обработки служебной информации 1го уровня), это сообщение, содержащее служебную информацию второго уровня и данные в виде полезных данных и служебной информации вышестоящих уровней, передается на следующий уровень. На канальном (2-м) уровне снова происходит анализ системной информации, и сообщение передается на следующий уровень. И так до тех пор, пока сообщение не дойдет до уровня приложений, где в виде конечных данных передается принимающему приложению. В качестве примера можно привести обращение браузера к веб-серверу. Приложение клиента - браузер, формирует запрос для получения веб-страницы. Этот запрос передается приложением на уровень 7 и далее последовательно на каждый уровень модели OSI. Достигнув физического уровня, наш первоначальный запрос "обрастает" служебной информацией каждого уровня. После этого он предается по физической сети (кабелям) на сервер в виде электрических импульсов. На сервере происходит разбор соответствующей системной информации на каждом уровне, в результате чего посланный запрос достигает процесса веб-сервера, где обрабатывается и после обработки клиенту отправляется ответ. Процесс отправки ответа аналогичен посылке запроса - за исключением того, что сообщение посылает сервер, а клиент его получает.

Вместе с названием сообщение (message) в стандартах ISO для обозначения единицы данных используют термин протокольный блок данных (Protocol Data Unit, PDU).

Для более глубокого понимания принципов работы сети рассмотрим каждый уровень по отдельности.

# 2. Physical layer (layer 1)

Как видно из общей схемы расположения уровней в модели OSI - физический уровень (physical layer) самый первый. Этот уровень описывает среду передачи данных. Стандартизируются физические устройства, отвечающие за передачу электрических сигналов (разъемы, кабеля и т.д.) и правила формирования этих сигналов. Рассмотрим по порядку все составляющие этого уровня.

Большая часть сетей строится на кабельной структуре (хотя также существуют сети, основанные на передаче информации с помощью, например, радиоволн). На сегодняшний день существуют различные типы кабелей для передачи информации. Наиболее распространенные из них:

- телефонный провод;
- коаксиальный кабель;
- витая пара;
- оптоволокно.

Телефонный кабель начал использоваться для передачи данных со времен первых компьютеров. Главным преимуществом использования телефонных линий - использование существующих линий связи для передачи информации. При использовании телефонных линий можно передавать данные между компьютерами, находящихся на разных материках (также как и передача голоса между людьми, удаленных друг от друга на многие тысячи километров). На сегодняшний день использование телефонных линий также остается популярным. Большинство пользователей, которых устраивает небольшая скорость

передачи данных, могут получить доступ к Интернету со своих домашних компьютеров. Основными недостатками использования телефонного кабеля при передаче данных является небольшая скорость передачи, т.к. соединение происходит не напрямую, а через телефонные станции. При этом требование к качеству передаваемого сигнала при передаче данных значительно выше, чем при передаче "голоса". А т.к. большинство аналоговых АТС не справляется с этой задачей (уровень "шума" или помех и качество сигнала оставляет желать лучшего), то скорость передачи данных очень низкая. Хотя при подключении к современным цифровым АТС можно получить высокую и надежную скорость связи.

Коаксиальный кабель использовался в сетях несколько лет назад, но сегодня встретить сеть, использующую этот кабель, очень сложно. Этот тип кабеля по строению практически идентичен обычному телевизионному коаксиальному кабелю - центральная медная жила отделена слоем изоляции от оплетки, за исключением электрических характеристик (в телевизионном кабеле используется кабель с волновым сопротивлением 75 Ом, в сетевом коаксиальном кабеле - 50 Ом).

Основными недостатками этого кабеля является низкая скорость пропускания (до 10 Мбит/с), подверженность воздействиям внешних помех. Кроме того, подключение компьютеров в таких сетях происходит параллельно, а значит, скорость максимальная возможная скорость пропускания делится на всех пользователей. Но по сравнению с телефонным кабелем коаксиальный кабель позволяет объединять близко расположенные компьютеры, скорость передачи данных и качество связи намного лучше при использовании коаксиального кабеля вместо телефонного.

Витая пара ("twisted pair") - наиболее распространенное средство для передачи данных между компьютерами. В данном типе кабеля используется медный провод, попарно скрученный, что позволяет уменьшить количество помех и наводок при передаче сигнала по самому кабелю, так и при воздействии внешних помех. Существует несколько категорий этого кабеля. Основные из них: Cat 3 - был стандартизирован в 1991 г, электрические характеристики позволяли поддерживать частоты передачи до 16 МГц, использовался для передачи данных и голоса. Более высокая категория, Cat 5 была специально разработана для поддержки высокоскоростных протоколов. Поэтому электрические характеристики лежат в пределах до 100Мгц. На таком типе кабеля работают протоколы передачи данных 10,100,1000 Мбит/с. На сегодняшний день кабель Cat5 практически вытеснил кабель Cat 3. Основным преимуществом витой пары перед телефонными и коаксиальными кабелями - более высокая скорость передачи данных при использовании одно и того же кабеля. Также использование кабеля Cat 5 в большинстве случаев позволяет, не меняя кабельную структуру, повысить производительность сети.

Оптоволокно используется для соединения больших сегментов сети, которые располагаются далеко друг от друга, или в сетях, где требуется большая полоса пропускания, помехоустойчивость. Оптический кабель состоит из центрального проводника света (сердцевины) - стеклянного волокна, окруженного другим слоем стекла - оболочкой, обладающей меньшим показателем преломления, чем сердцевина. Распространяясь по сердцевине, лучи света не выходят за ее пределы, отражаясь от покрывающего слоя оболочки. Световой луч обычно формируется полупроводниковым или диодным лазером. В зависимости от распределения показателя преломления и от величины диаметра сердечника различают:

- одномодовое волокно;
- многомодовое волокно

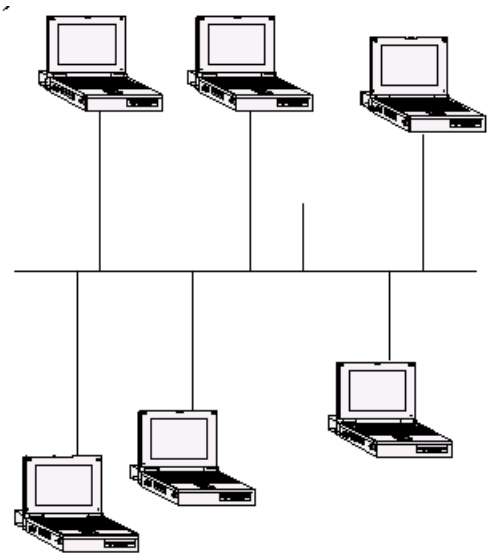
Понятие "мода" описывает режим распространения световых лучей в сердечнике кабеля. В одномодовом кабеле используется проводник очень малого диаметра, соизмеримого с длиной волны света. В многомодовом кабеле используется более широкие сердечники, которые легче изготовить технологически. В этих кабелях в сердечнике одновременно существуют несколько световых лучей, отражающихся от оболочки под разными углами. Угол отражения луча называется модой луча. Оптоволокно обладает следующими преимуществами: устойчивы к электромагнитным помехам, высокие скоростные характеристики на больших расстояниях. Основным недостатком является как дороговизна самого кабеля, так и трудоемкость монтажных работ т.к. все работы выполняются на дорогостоящем высокоточном оборудовании.

Физический уровень также отвечает за преобразование сигналов между различными средами передачи данных. Например, при необходимости соединить сегмент сети, построенной на оптоволокне и витой паре применяют т.н. конверторы (в данном случае они преобразуют световой импульс в электрический).

Сетевой адаптер - устройство, позволяющее обмениваться наборами битов, представленными электрическими сигналами. Сетевой адаптер (сетевая карта, англ. Network adapter) обычно имеет шину для подключения в компьютер ISA или PCI, и соответствующий разъем для подключения к среде передачи данных (например, для витой пары, коаксиала и т.п.).

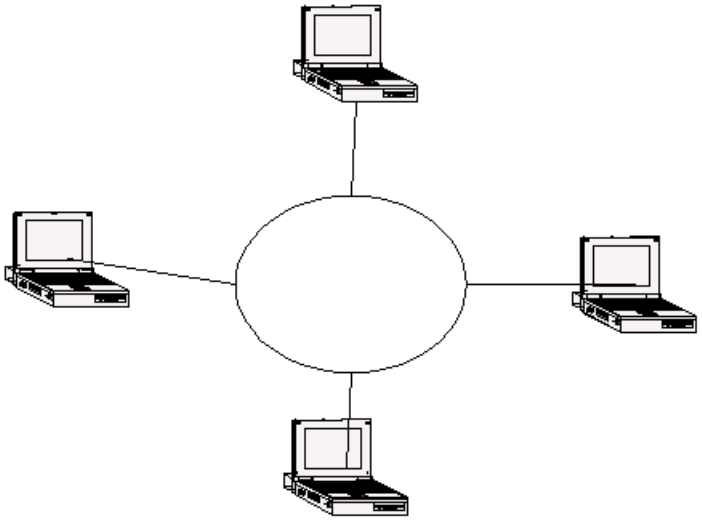
Теперь, когда известно с помощью чего происходит соединение компьютеров в одну сеть, рассмотрим физическую схему соединения компьютеров или другими словами - физическую топологию (структуру локальной сети).

Топология "шина"(bus):



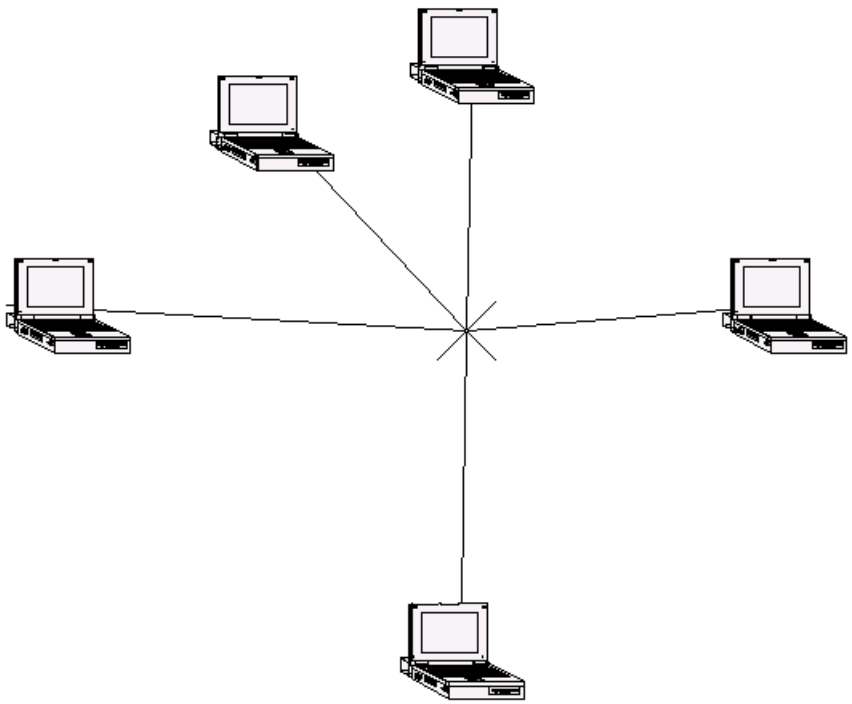
Все компьютеры и сетевые устройства подсоединены к одному проводу, и фактически они напрямую соединены между собой.

Топология "кольцо"(ring):



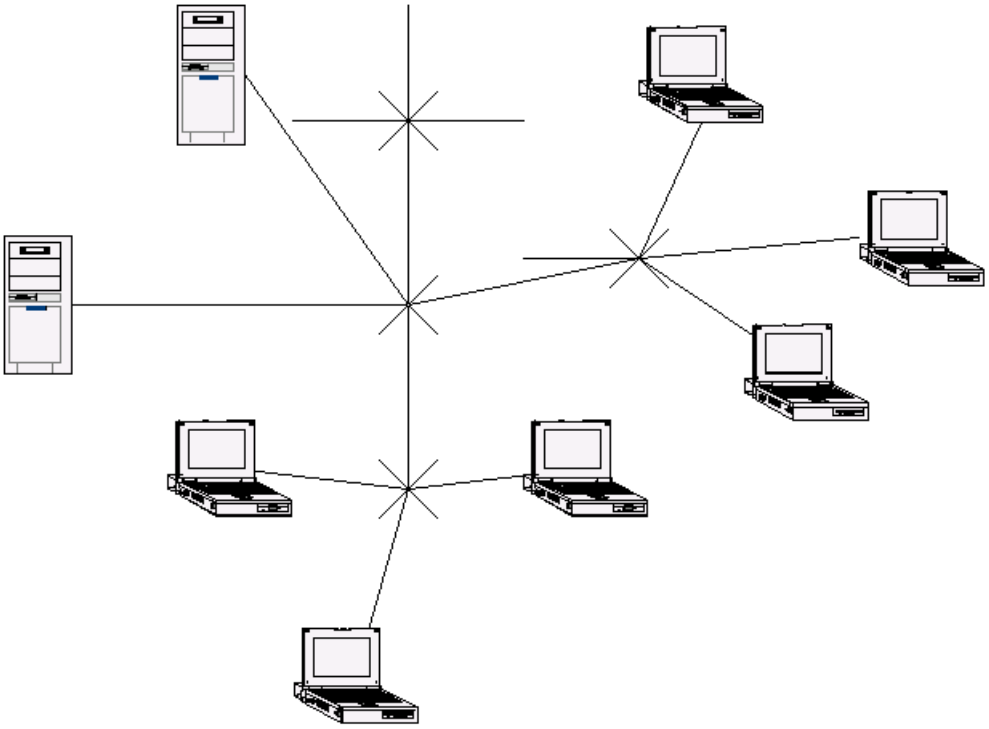
Кольцо состоит из сетевых устройств и кабелей между ними, образующих одно замкнутое кольцо.

Топология "звезда"(star):



Все компьютеры и сетевые устройства подключены к одному центральному устройству.

Топология "расширенная звезда"(extended star):



Такая схема практически аналогична топологии "звезда" за одним исключением. Каждое устройство соединено с локальным центральным устройством, а оно в свою очередь соединено с центром другой "звезды".

# 3. Data layer (layer 2)

На физическом уровне пересылаются просто набор сигналов - битов. При этом не проверяется, что несколько компьютеров могут в одну среду передачи данных одновременно передавать информацию в виде битов. Поэтому одной из задач канального уровня является проверка доступности среды передачи. Также канальный уровень отвечает за доставку фреймов между источником и адресатом в пределах сети с одной топологией. Для обеспечения такой функциональности Data layer разделяют на два подуровня:

- LLC sublayer
- MAC sublayer

LLC отвечает за переход со второго уровня на более высший - 3й сетевой уровень.

MAC отвечает за передачу данных на более низкий уровень - physical layer.

Рассмотрим эти подуровни более подробно.

## 3.1. LLC sublayer.

Этот подуровень был создан для независимости от существующих технологий. Он обеспечивает передачу данных на сетевой (3-й) уровень вне зависимости от физической среды передачи данных. LLC получает данные с сетевого уровня, добавляет в них служебную информацию и передает пакет для последующей инкапсуляции и передачи для требуемой технологии. Например, это может быть Ethernet, Token Ring, Frame Relay.

## 3.2. MAC sublayer.

Этот подуровень обеспечивает доступ к физическому уровню. Как уже говорилось, data layer обеспечивает идентификацию компьютеров в сети. Т.е. у каждого компьютера на data layer есть уникальный адрес, который еще иногда называют физическим адресом или MAC-адресом.

Этот адрес прошит в энерго-незвисимую память сетевой карточки и задается производителем. Длина MAC-адреса 48 бит или 6 байт (каждый байт состоит из 8 бит), которые записываются в шестнадцатеричном формате. Первые 3 байта называются OUI. OUI - Organizational Unique Identifier, Организационный Уникальный Идентификатор - назначается IEEE (Institute of Electrical and Electronic Engineers, Институт инженеров по электротехнике и радиоэлектронике - международная организация, подготавливающая стандарты и спецификации) и обозначает производителя сетевой карты. Остальные 3 байта описывают идентификационный номер самой сетевой карты. Записываться физический адрес может в разных форматах, например: 00:00:B4:90:4C:8C, 00-00-B4-90-4C-8C, 0000.B490.4C8C - это зависит от производителя программного обеспечения.

Рассмотрим, например, адрес 0000.1c12.3456. Здесь 00001c - идентификатор производителя, а 12.3456 - идентификатор сетевой карты.

Такая система адресов гарантирует, что в сети не будет двух компьютеров с одинаковыми физическими адресами.

Рассмотрим более подробно процесс передачи данных на data layer в сети Ethernet. Один компьютер посылает данные другому, используя свой MAC-адрес и MAC-адрес получателя.

Каждый компьютер, получивший это сообщение, проверяет, кому он был адресован. Если MAC-адрес в фрейме и MAC-адрес получившего этот фрейм совпадают, то пакет принимается и передается на вышестоящий уровень для дальнейшей обработки. Если же адрес в пакете не совпадает с адресом сетевой карты, то такой пакет отбрасывается. Если отправитель хочет, что бы его сообщение получили все узлы локальной сети, он отправляет пакет с MAC-адресом получателя в виде FF-FF-FF-FF-FF-FF. Этот адрес используется для широковещания (broadcast), которое примут все сетевые устройства и передадут на следующий уровень.

В сетях Ethernet используется метод разделения среды передачи данных - метод CSMA/CD (carrier sense multiply access/collision detect). Этот метод применяется в сетях с логической общей шиной. Все компьютеры такой сети имеют доступ к общей шине, поэтому она может быть использована для передачи данных между двумя любыми узлами сети. Одновременно все компьютеры сети имеют возможность практически немедленно получить данные, которые любой из компьютеров начал передавать на общую шину. Технология CSMA/CD так же позволяет уменьшать общее кол-во столкновений пакетов (collision - например, когда два компьютера начинают одновременно передавать данные в одну общую шину).

Рассмотрим устройства, используемые для построения сетей в разных топологиях.

При использовании топологии построения сети шина ("bus") все компьютеры посылают фреймы на физический уровень, а дальше по среде передачи данных (например, кабелю) другим компьютерам. Т.е. все устройства, подсоединенные к одной среде, получают все пакеты, которые проходят по сети. Специальных устройств для построения такой сети не используется.

При построении сети на основе топологии Token Ring используется другой принцип. Физически сеть представляет собой замкнутое кольцо. В отличие от сети, работающей на основе Ethernet, здесь используется передача данных по очереди. Т.е. вычисляется, сколько времени может передавать данные одна станция, по истечении этого времени данные начинает передавать другая рабочая станция и т.д.

При построении сети на основе технологии "звезда" нужно использовать кроме сетевых карт в компьютере дополнительное сетевое оборудование в центре, куда подключаются все "лучи звезды". Типичным примером является концентратор (hub). Подключение происходит с помощью кабеля "витая пара". Все компьютеры подключаются к концентратору. Используемая технология Ethernet позволяет снизить количество коллизий с помощью CSMA/CD. Подключенный компьютер посылает данные. Коммутатор транслирует этот фрейм на все порты. Дальше получатель проверяет, кому был послан фрейм и если MAC получателя совпадает с MAC-адресом, то пакет принимается, если нет - отбрасывается. Недостатком концентратора является то, что пользователи сети могут "прослушивать" чужой трафик (в том числе перехватить пароль, если он передается в открытом виде). Также общая максимальная скорость делится на всех подключенных в концентратор. И если скорость передачи данных используется 10 Мбит/с, то в среднем на каждого конкретного пользователя приходится около 2 Мбит/с.

Более дорогим, но и более производительным решением, является использование коммутатора(switch). Коммутатор в отличие от концентратора имеет таблицу MAC-address - Port. Он смотрит у всех фреймов адреса отправителя и получателя. Далее при прохождении фрейма коммутатор просматривает адрес получателя, и если он знает, какому порту соответствует адрес получателя, то он посылает его на этот порт. Если адрес получателя коммутатору не известен, то он отправляет фрейм на все порты, кроме того, с которого этот пакет пришел. Таким образом, получается, что если два компьютера обмениваются данными между собой, то они не забивают своими пакетами другие порты и соответственно их пакеты практически невозможно "подслушать".

# 4. Network layer (layer 3)

В предыдущей главе мы рассмотрели второй уровень в модели OSI. Одним из ограничений 2?го уровня является использование "плоской" модель адресации. При попытке построить большую сеть, используя для идентификации компьютеров MAC-адреса, мы получим огромное количество broadcast-трафика. Протокол, который поддерживается сетевым уровнем, использует иерархическую структуру для уникальной идентификации компьютеров.

Для примера представим себе телефонную сеть. Она также имеет иерархическую адресацию. Например, в номере +7-095-101-12-34 первая цифра обозначает код страны, далее идет код области/города(095), а затем указывается сам телефон (101-12-34). Последний номер также является составным. 101 - это код станции, куда подключен телефон, а 12-34 определяет местоположение телефона. Благодаря такой иерархической

структуре мы можем определить расположение требуемого абонента с наименьшими затратами. Иерархическая адресация для сети также должна позволять передавать данные между разрозненными и удаленными сетями. На сетевом уровне существует несколько протоколов, которые позволяют передавать данные между сетями - IP, IPX. Наиболее распространенным протоколом на сегодняшний день является IP. IPX же практически не используется в публичных сетях, но его можно найти в частных, закрытых сетях.

Устройства, работающие на 3-м уровне, называют роутерами (router). Они соединяют удаленные сети, объединяют территориальные сети (LAN) в глобальные (WAN). Роутер получает пакет с локального устройства или компьютера (в дальнейшем будем просто называть LAN) и смотрит заголовок 3го уровня. На основании полученной информации с 3го уровня роутер принимает решение, что делать с пакетом. Основная задача роутера - выбор пути, по которому нужно передать пакет. Т.к. у нас может существовать множество связей между двумя сетями - еще одна задача роутера выбрать наиболее оптимальный путь для прохождения пакета. Выбор роутером следующего узла сети (следующего hop'а) для доставки его получателю называется "routing the packet". Выбор "next hop", по которому роутер перешлет пакет, может зависеть от многих факторов - загрузки сети, наименьшего пути до получателя, стоимости трафика по различным маршрутам и т.д.

Адресация на сетевом уровне дает возможность роутеру определить путь для доставки пакета получателю через глобальные сети. Было разработано и сейчас существует несколько протоколов сетевого уровня. Один из основных и наиболее распространенный - протокол IP. Рассмотрим более реализацию этого протокола.

При прохождении данных с верхних уровней на нижние на сетевом уровне к ним добавляется дополнительные данные сетевого уровня. В заголовке IP-пакета содержится необходимая для дальнейшей передачи информация - адрес отправителя, адрес получателя(кроме адресов там также содержится служебная информация). Рассмотрим более подробно понятие IP-адрес. IP-адрес представляется 32-х битным бинарным числом, разбитым на 4 части по восемь бит каждая. Логически его разбивают на две части - network и host. Компьютеры с одинаковой частью network IP-адреса в сети могут легко передавать данные между собой, но если они имеют различные network-ID, то даже если они находятся в одном физическом сегменте, обычно они не могут "увидеть" друг друга.

Сетевая часть IP-адреса показывает принадлежность сетевого адреса - какой сети принадлежит адрес. Хост (host) идентифицирует сетевое устройство в этой сети. Т.к. сетевой адрес состоит из 4-х октетов, один, два или три первых октета могут использоваться для определения сетевого адреса. Подобным образом до 3-х октетов может быть использовано для определения host-части. Существует вообще пять классов IP-адресов. Три из них используются в открытых сетях по всему миру (class A, class B, class C).

| | | | | |
|---|---|---|---|---|
| Class A | N | H | H | H |
| Class B | N | N | H | H |
| Class C | N | N | N | H |

## 4.1. Class A

В классе А используется для определения принадлежности адреса к сети первый октет, остальные для определения адреса хоста. Если перевести бинарное число сетевой части в десятичное - то мы получим что адрес сети класса А может быть в диапазоне 0-126(127й

адрес зарезервирован для специального использования). Остальные свободные три октета используются для задания адреса хоста в данной сети. В одной сети может быть использовано до 2^24 адресов (за исключением двух) - получается 16 777 214 возможно использовать в одной сети класса А.

Диапазон адресов 10.0.0.0-10.255.255.255 не используется в публичных сетях. Эти адреса специально зарезервированы для использованя в локальных сетях и не обрабатываются глобальными маршрутизаторами.

## 4.2. Class B

В сети класса В используются первые два октета для определения сети, последние два октета - для определения адреса хоста. Диапазон сети класса В может быть с 128 до 191 (исходя из первых двух октетов). В каждой сети класса В может быть не более 65534 адресов - 216 (за исключением двух адресов).

В этой подсети зарезервированными для локального использования являются следующие адреса: 172.16.0.0-172.31.0.0.

## 4.3. Class CClass DClass E

Диапазон сети класса С определяется первыми тремя октетами. И в десятичном виде эта сеть может начинаться с 192 по 223. Для определения адреса хоста используется последний октет. Таким образом, в сети класса С может быть использовано 28(без двух адресов) или 254 адреса

Зарезервированными для локального использования являются следующие адреса: 192.168.0.0-192.168.255.255.

Этот класс используется для multicast-группы. Диапазон адресов – 224.0.0.0-239.255.255.255.

Этот класс адресов зарезервирован для будущего использования. Диапазон адресов – 240.0.0.0-247.255.255.255.

Два адреса в каждой подсети являются зарезервированными. IP-адрес, оканчивающийся на бинарный ноль, используют для обозначения сетевого адреса. Например для сети класса А адрес сети - 112.0.0.0, и этой сети принадлежит сетевой адрес - 112.2.3.4. Адрес сети используется роутерами для определения маршрута. Второй зарезервированный адрес - броадкаст-адрес (broadcast). Этот адрес используется, когда источник хочет послать данные всем устройствам в сети. В отличии от адреса сети - в адресе броадкаста используется бинарная единица(в октетах, отвечающих за часть хоста). Например, для сети 171.10.0.0 последние 16 бит адреса используется для обозначения хоста, и броадкаст-адрес будет выглядеть как 171.10.255.255. И все устройства в сети 171.10.0.0 получат данные, которые были посланы по адресу 171.10.255.255.

Выше мы рассмотрели разбиение на классы сетей. Но не всегда имеет смысл использовать например сеть класса С когда в ней реально будут использоваться только половина адресов. Для более рационального распределения адресов используются подсети. Адрес подсети включает в себя сетевую часть сети класса А,В или С и т.н. subnet field и часть хоста. Для subnet field выделяется значение из октетов, принадлежащих хосту (т.е. для адреса подсети может быть использовано до 3-х октетов из сети класса А, до 2х из сети класса В, и 1 для С соответственно). Создавая адрес подсети, несколько бит из части

адреса хоста переназначаются в адрес сети. Минимальное значение, которое может быть таким образом занято из адреса хоста - 2 бита. Если занять один бит, то мы получим подсеть, состоящую из двух адресов - адреса подсети и броадкаст-адреса. Максимальное число, которое может быть занято под подсеть из адреса хоста - такое, чтобы в последнем октете для хоста осталось два бита. Разбиение на подсети уменьшает также размеры броадкса-доменов - т.к. в сетях класса A в броадкаст-домене будет порядка 16 миллионов компьютеров. И если каждый пошлет хотя бы по одному броадкаст-адресу то нагрузка на сеть будет очень большой. А т.к. броадкасты не пересылаются роутерами - происходит ограничение броадкаст-домена и сохранения полосы пропускания от ненужного трафика.

Для определения размерности подсети используется маска подсети. Маска подсети определяет, какая часть IP-адреса используется для задания сетевой части, а какая часть для хоста. Маску подсети можно определить следующем образом. Запишем IP-адрес подсети в бинарном виде. Все значения, относящиеся к network-part и subnet-part заменим на 1, все значения, относящиеся к host-part заменим на 0. В результате получим маску подсети.

Например маска подсети для сети класса A будет выглядеть следующим образом: 255.0.0.0, для сети класса B: 255.255.0.0, для сети класса C: 255.255.255.0. Если у нас используется подсеть - то как было сказано выше все значения, относящиеся к сети и подсети будут 1, хосту - 0. Например маска 255.255.255.192 определяет подсеть класса C, кол-во хостов в данной подсети будет равно 64.

Для передачи данных кроме IP-адреса также нужно знать MAC-адрес. Для определения соответствия IP-адресу MAC-адреса существует ARP-протокол(Address Resolution Protocol, протокол определения адресов). ARP-таблица находится в оперативной памяти и периодически обновляется. В этой таблице находятся IP-адреса только локальной сети. Когда источник определяет адрес получателя, источник также смотрим MAC адрес получателя и если он его не находит у себя в ARP-таблице, то он делает запрос для получения MAC-адреса получателя. Протокол RARP (Reverse ARP - обратный ARP) действует наоборот - он известному MAC-адресу сопоставляет IP-адрес. Это необходимо, например, для работы таких проколов, как BOOTP, DHCP. При загрузке по локальной сети посылается broadcast-запрос - противоположный ARP-запросу. Если в ARP-запросе идет опрос "IP-получателя известен, MAC-получателя - ???", то в RARP-запросе "MAC-получателя известен, IP - ???". Поэтому если сервер знает например какому маку должен соответствовать IP-адрес, он отвечает на RARP-запрос и не придется вручную вводить IP-адрес на компьютере (пример такого сервиса - DHCP - особенно это эффективно при большом кол-ве компьютеров).

# 5. Transport layer (layer 4)

Рассмотрим TCP/IP протокол 4го транспортного уровня модели OSI. TCP/IP имеет два протокола - TCP и UDP. TCP обеспечивает виртуальные соединения между пользовательскими приложениями.

Основные характеристики TCP и UDP

| TCP | UDP |
|---|---|
| Для работы устанавливает соединение | Работает без соединений |
| Гарантированная доставка данных | Гарантий доставки нет |

| TCP | UDP |
|---|---|
| Разбивает исходное сообщение на сегменты | Передает сообщения целиком в виде датаграмм |
| На стороне получателя сообщение заново собирается из сегментов | Принимаемые сообщения не объединяются |
| Пересылает заново потерянные сегменты | Подтверждений о доставке нет |
| Контролирует поток сегментов | Никакого контроля потока датаграмм нет |

## 5.1. TCP

TCP/IP представляет собой комбинацию двух уровней TCP и IP. IP - протокол 3го уровня - не обеспечивает гарантированной доставки данных, но обеспечивающий наилучшую доставку через сеть. TCP - протокол 4го уровня - позволяет гарантировать доставку данных. Поэтому совместно они могут предоставить большее количество сервисов. Работа с TCP-соединением состоит из трех фаз. Инициации соединения - инициатор посылает пакет с порядковым номером (sequence number - x). Второй хост, который получает тот пакет, запоминает его и отсылает подтверждение(sequence number + 1 - x+1) и свое собственный sequence number(y). Первый хост снова получает свой sequence number, увеличенный на 1. Если данные дошли корректно (полученный sequence number равен, тому который он отсылал+1(x+1)), то первый хост отсылает sequence number y+1 и второй хост проверяет его верность. Если во всех трех фазах было получено верное значение для sequence number, то соединение считается успешно установленным.

Понятие окна (window) в TCP. При передачи пакета принимающий хост посылает подтверждение о получении пакета. Источник, получив подтверждение, посылает следующий пакет. Более эффективным способом передачи пакетов является использование "окна". В этом случае отсылается подряд несколько пакетов, затем ожидается подтверждение, если подтверждение пришло, посылаем следующую часть пакетов подряд. При этом размер окна может динамически меняться - например размер "окна" равен 5, но хост загружен и размер буфера на текущий момент не позволяет одновременно принять 5 пакетов, об этом он сообщает источник и размер окна уменьшается. Одним из недостатков TCP является то, что установка "виртуального соединения" требует постоянного маршрута между источником и получателем. Это приводит при изменении маршрутной таблице в сети к необходимости заново устанавливать TCP-соединение.

## 5.2. UDP

В отличии от TCP - UDP не гарантирует доставку данных. UDP не устанавливает виртуального соединения, источник просто шлет user datagram получателю. Если данные были некорректно доставлены, или вообще часть пакетов потерялась - UDP не позволяет их восстановить. Запрос на получение данных должен будет выполнен заново. Казалось бы, недостатков у данного протокола довольно много что ставит под сомнение его эффективность. Но есть сервисы, где UDP не заменимо. Например, при передаче потокового аудио-видео если бы мы использовали TCP, то при потере одного пакета у нас будет приостановлена передача данных для передачи потерянного пакета. При использовании UDP потерянный пакет - всего лишь незначительное ухудшение изображения/звука, при этом передача данных не прерывается. Также при использовании UDP нам не обязательно установление виртуального соединения - нам не важен путь, по которому пройдет пакет - например при недоступности самого оптимального маршрута, пакет может быть доставлен

через другой запасной маршрут - при этом последовательность UDP-датаграмм будет сохранена.

Оба протокола TCP и UDP используют порты(port) для прохождения информации на вышестоящие уровни. Использование номера порта позволяет передавать разные данные одновременно. Приложения используют эти порты, часть которых зарезервировано под стандартные приложения. Например, для FTP зарезервирован порт 21. Далее приведен список распределения портов: порты меньше 255 - используются для публичных сервисов, потры из диапазона 255-1023 - назначаются компаниями-разрабочиками для их приложений, номера выше 1023 - не регулируемые.

# 6. Session layer (layer 5)

После того как пакет, разобранный на сетевом уровне, пройдет транспортный уровень - он поступит на session layer. 5й уровень обеспечивает установку, контроль и окончание сессии между приложениями. Уровень сессий координирует приложения, когда они взаимодействуют между двумя хостами. Соединения между двумя компьютерами включает в себя множество мини-"переговоров", обеспечивающих более эффективное соединение двух компьютеров.

# 7. Presentation layer (layer 6)

Уровень представлений отвечает за представление данных в форме, что бы получатель их смог понять. Приведем пример - два человека общаются на разных языках. Что бы друг друга понять, им нужно использовать человека, который понимает оба этих языка. Также уровень представлений - сервис для трансляции данных, которые необходимо передать через сеть. 6-й уровень обеспечивает следующую функциональность: data formatting(presentation), data encryption, data compression. После получения данных с уровня приложений, уровень представлений выполняет одну или все эти функции перед передачей данных на session layer. Приведем пример использования уровня представлений. Например, первый хост использует Extended Binary Coded Decimal Interchange Code (EBCDIC) для представления данных. Второй хост для представления данных использует American Standard Code for Information Interchange (ASCII). Presentation layer обеспечивает взаимодействие двух этих систем с разными типами представления данных. Уровень представлений также отвечает за шифрацию данных - для сохранения частной информации при передаче через публичные сети. За компрессию данных также отвечает представительский уровень. Используя математические алгоритмы для уменьшения объема передаваемых данных. Но это эффективно при использовании каналов связи с маленькой пропускной способностью т.к. использование компрессии может потребовать значительных вычислительных мощностей при больших объемах трафика.

# 8. Application layer (layer 7)

Уровень приложений определяет, какие ресурсы существуют для связи между хостами. Этот уровень не обеспечивает связь со всеми уровнями модели OSI. Он передает данные только на presentation layer. Большинство сетевых приложений можно классифицировать как клиент-серверные приложения. Клиент находится на локальном компьютере и

запрашивает необходимый сервис. Сервер находится на удаленном компьютере и обеспечивает необходимый клиенту сервис. Клиент-серверное приложение работает по следующей схеме: client-request, server-response, client-request, server-response и т.д. Так работает, например браузер - клиент запрашивает URL, а сервер в ответ на этот запрос выдает соответствующую веб-страничку. Примером приложений application layer могут служить:

- telnet - удаленный клиент для работы с сетью
- dns - domain name system
- e-mail - электронные сообщения

Остановимся немного подробнее на DNS. Как известно каждому компьютеру соответствует IP-адрес. И, например, у нас есть web-сервер. И что бы получить к нему доступ мы должны послать запрос на этот сервер. Запрос можно сделать по IP-адресу, например 194.87.0.50 - но запомнить каждый IP-адрес сервера очень затруднительно. Да и сам IP-адрес мало информативен. Для решения этой задачи существует система соответствия имени сервера и IP-адреса. Т.е. например адресу 194.87.0.50 соответствует доменное имя www.ru. ДНС - это иерархическая система. В этой иерархии домены делятся на уровни. Первый уровень обозначает принадлежность домена. Существует много доменов первого уровня - классификация их может быть разная: например .ru, .us, .uk - обозначают принадлежность к стране, .edu - .edu - educational sites, .com - commercial sites, .gov - government sites, .org - non-profit sites, .net - network service. В каждом домене первого уровня может быть множество доменов второго уровня. У домена третьего уровня может быть множество доменов третьего уровня и т.д. За каждую зону отвечает ДНС-сервер. При запросе на сервер у него либо содержится требуемая информация, или он знает какой сервер отвечает за эту ДНС-зону (например сервер test.ru знает про все имена в зоне test.ru, но например у нас есть домен следующего уровня node1.host1.test.ru - основной сервер не обязан знать все имена в домене host1.test.ru, он знает какой сервер отвечает за эту зону и на этот сервер пере направляет запрос)

# 9. Утилиты для работы с сетью

Рассмотрим основные программы, позволяющие читать и изменять сетевые параметры, диагностировать и выявлять ошибки при работе сети. В различных ОС существует свои наборы утилит. Сравним их для 2 систем, например Microsoft Windows NT и Sun Solaris. Какими бы разными не были эти ОС, в каждой из них реализована модель OSI. Естественно программная и аппаратная реализация стека модели OSI у них различается, но формирование, обработка данных на всех уровнях осуществляет по установленному стандарту.

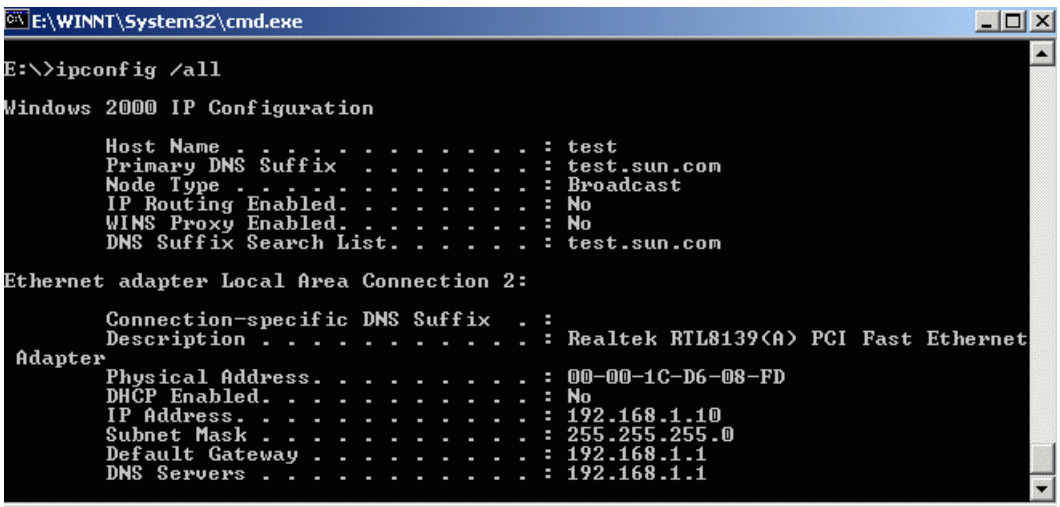
Например, рассмотрим работу файл-сервера под управлением 2 выбранных ОС. Пусть передается файл с файл-сервера Windows на сервер Solaris (будем рассматривать только процесс передачи данных с одного сервера на другой). ОС Windows передает данные на уровень приложения модели OSI. Далее происходит процесс инкапсуляции – на каждом уровне ОС добавляет служебную информацию. Полученное в результате сообщение на физическом уровне в виде электрических импульсов передается по сети получателю – файл-серверу, под управлением ОС Solaris. На этом сервере происходит процесс декапсуляции. Полученное сообщение постепенно “разворачивается”, на каждом уровне считывается соответствующая служебная информация. Т.к. каждый уровень модели OSI

стандартизирован, для потребителей есть возможность использовать совместно оборудование и программное обеспечение различных производителей. В результате файл-сервер под управлением ОС Solaris может получить файл, посланный под управлением ОС другого производителя.

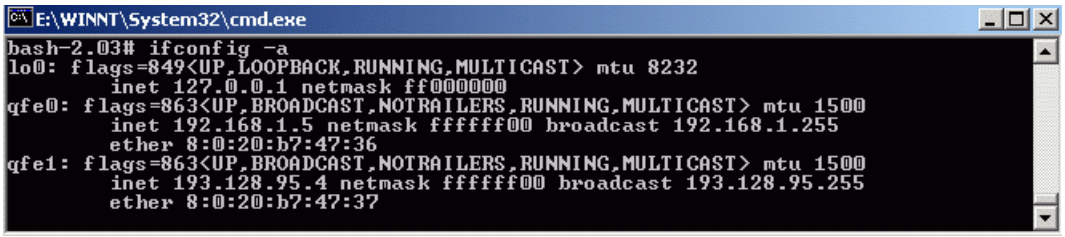
Если бы на серверах применялся различный стандарт передачи данных на физическом уровне (один сервер формирует передачу данных в виде светового импульса, а другой в виде электрических сигналов), то взаимодействие без использования дополнительного оборудования было бы невозможно. Поэтому вводят понятие сете-независмых и сете-зависимых уровней. Три нижних уровня – физический, канальный и сетевой уровни являются сете-зависимыми. Т.е. например смена Ethernet на ATM влечет полную смену протокола физического и канального уровней. Три верхних уровня – приложений, представительский и сессионный - ориентированы на приложения и практически не зависят от физической технологии построения сети. Так переход от Ethernet на FDDI не требует изменений в перечисленных уровнях. Транспортный уровень является “прослойкой” между сете-зависимыми и сете-независимыми уровнями. Он скрывает все детали функционирования нижних уровней от верхних. Это позволяет разработчику приложений не задумываться о технических средствах реализации транспортировки сообщений.

## 9.1. IPCONFIG (IFCONFIG)

Рассмотрим утилиты, которые позволяют просматривать, проверять и изменять сетевые настройки. Обычно сетевые настройки включают информацию 3-го уровня (сетевого) – IP адрес, маску подсети и т.д. Для просмотра сетевых настроек в ОС Windows можно использовать команду ipconfig. Она выдает информацию об IP-адресе, маски подсети (netmask), роутера по умолчанию(default gateway). Задав дополнительный параметр команде ipconfig – all, можно получить более подробную информацию – имя компьютера, имя домена, тип сетевой карты, MAC-адрес и т.д.




В ОС Solaris для получения IP-адреса и прочих сетевых настроек используется команда – ifconfig. Она также показывает название интерфейса, IP-адреса, маску подсети, MAC-адрес.




## 9.2. ARP

Как уже было сказано ранее, в оперативной памяти компьютера находится arp-таблица. В ней содержится MAC-адрес сетевого устройства и соответствующий ему IP-адрес. Для просмотра этой таблички используется команда arp. Например, arp –a выводит все известные MAC-адреса.

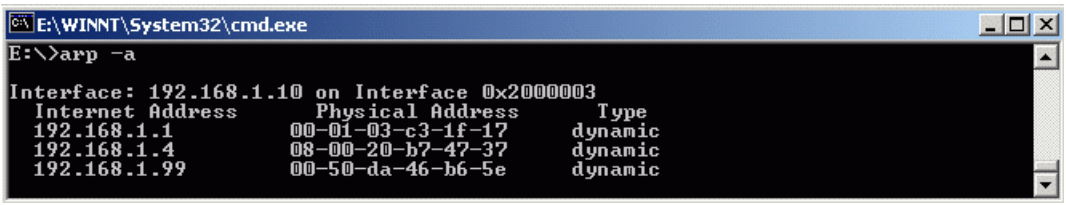



Таблица MAC-адресов хостов хранится в памяти не постоянно. После определенного времени записи из таблицы автоматически удаляются, если к данному IP-адресу не было обращений. Существует вообще два типа записей в ARP-таблице – статический и динамический. Динамическая запись периодически обновляется, при появлении новой пары MAC-IP автоматически добавляется. Статическая запись вносится вручную и существует до тех пор, пока вручную эта запись не будет удалена или компьютер (маршрутизатор) не будет перезагружен.

Если компьютер посылает сообщение на IP-адрес, который неизвестен на транспортном и канальном уровне формируется broadcast-frame – т.н. ARP-запрос. Каждый узел локальной сети получает ARP-запрос и сравнивает IP-адрес указанный в запросе и свой. При совпадении адреса в запросе и получателя формируется ARP-ответ, в котором указывается свой IP-адрес и MAC-адрес. Этот ответ получает машина, посылавшая ARP-запрос и добавляет запись в свою ARP-таблицу. Задержка на получение MAC-адреса составляет порядка нескольких миллисекунд, поэтому для пользователя это будет практически незаметно и задержки при передачи данных не возникнет (в большинстве реализаций сетевой уровень ставит пакет в очередь если неизвестен MAC-адрес, но бывают системы когда формируется ARP-запрос, но сам пакет отбрасывается, и на транспортный уровень возлагаются задачи по повторной передачи пакета).

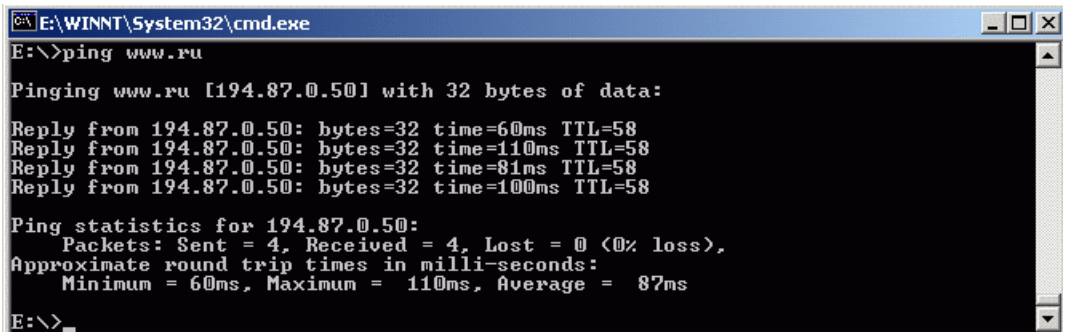
## 9.3. Ping

Для выявления различных неполадок в сети существует несколько утилит, которые позволяют определить, на каком уровне модели OSI произошел сбой или неверная конфигурация сетевых протоколов. Одна из таких утилит – ping. Эта утилита позволяет определить ошибки на сетевом уровне (layer 3), используя протокол ICMP (Internet Control Message Protocol) – протокол межсетевых управляющих сообщений. Формат использования этой утилиты довольно прост: ping 194.87.0.50 (где 194.87.0.50 – IP-адрес удаленного

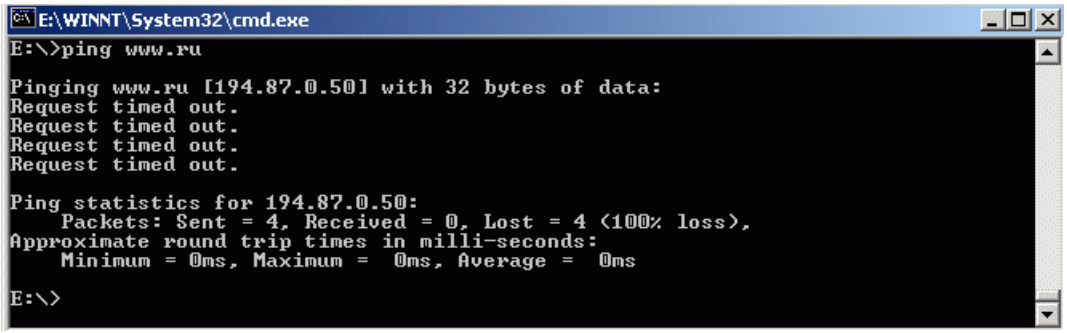
компьютера). Если все работает верно – в результате выводится время задержки прихода ответа от удаленного компьютера и время жизни пакета.

Протокол ICMP находится на стыке двух уровней – сетевого и транспортного. Основной принцип действия этого протокола – формирования ICMP эха-запроса (echo-request) и эха- ответа (echo-reply). Запрос эха и ответ на него может использоваться для проверки достижимости хоста-получателя и его способности отвечать на запросы. Также прохождение эхо-запроса и эхо-ответа проверяет работоспособность основной части транспортной систему, маршрутизацию на машине источника, проверяет работоспособность и корректную маршрутизацию на роутерах между источником и получателем, а также работоспособность и правильность маршрутизации получателя.

Например, если на посланный echo-request возвращается корректный echo-reply от машины, которой был послан запрос – можно сказать что транспортная система работает корректно. И если браузер не может отобразить веб-страницу, то проблема, скорее всего, не в первых трех уровнях модели OSI.




Из примера видно, что по умолчанию размер посылаемого пакета 32 байта, далее вычисляется время задержки ответа, и TTL (time to live – время жизни пакета). В приведенном выше примере показано успешное выполнение команды ping. В случаях, когда запросы echo request посылаются, но echo reply не возвращаются, выводится сообщение об истечении времени запроса.




## 9.4. Traceroute

Утилита traceroute также использует протокол ICMP для определения маршрута прохождения пакета. При отсылке traceroute устанавливает значение TTL последовательно от 1 до 30. Каждый маршрутизатор, через который проходит пакет на пути к назначенному хосту, увеличивает значение TTL на единицу. С помощью TTL происходит предотвращение зацикливание пакета в “петлях” маршрутизации. Например, при выходе маршрутизатора

или линии связи из строя требуется некоторое время для определения, что данный маршрут потерян, и его необходимо обойти.

В это время существует вероятность, что датаграмма будет уничтожена в петле маршрутизации. Чтобы предотвратить потерю датаграммы, поле TTL устанавливается в максимальную величину.

Когда маршрутизатор получает IP датаграмму с TTL равным либо 0, либо 1, он не должен отправлять эту датаграмму дальше. (Принимающий хост должен доставить подобную датаграмму в приложение, так как датаграмма не может быть обработана маршрутизатором. Как правило, системы не должны получать датаграммы с TTL равным 0). Если такую датаграмму получает маршрутизатор, он уничтожает ее и посылает хосту, который ее отправил, ICMP сообщение "время истекло" (time exceeded). Принцип работы traceroute заключается в том, что IP датаграмма, содержащая это ICMP сообщение, имеет в качестве адреса источника IP адрес маршрутизатора.

Теперь мы можем понять, как работает Traceroute. На хост назначения отправляется IP датаграмма с TTL, равным единице. Первый маршрутизатор, который должен обработать датаграмму, уничтожает ее (так как TTL равно 1) и отправляет ICMP сообщение об истечении времени (time exceeded). Таким образом, определяется первый маршрутизатор в маршруте. Затем Traceroute отправляет датаграмму с TTL, равным 2, что позволяет получить IP адрес второго маршрутизатора. Это продолжается до тех пор, пока датаграмма не достигнет хоста назначения. Однако если датаграмма прибыла именно на хост назначения, он не уничтожит ее и не сгенерирует ICMP сообщение об истечении времени, так как датаграмма достигла своего конечного назначения. В ответ генерируется UDP-датаграмма с номером порта, который заведомо не будет обработан приложением (порт выше 30000) и с сообщением port unreachable. При получении такой датаграммы хостом, с которого выполнялась программа traceroute, делается вывод, что или удаленный хост работает корректно, или значение TTL было превышено значения по умолчанию (при выполнении утилиты traceroute TTL по умолчанию равно 30).

Рассмотрим пример выполнения утилиты traceroute

```
traceroute to netserv1.chg.ru (193.233.46.3), 30 hops max, 38 byte packets
 1  n3-core.mipt.ru (194.85.80.1)  1.508 ms  0.617 ms  0.798 ms
 2  mipt-gw-eth0.mipt.ru (193.125.142.177)  2.362 ms  2.666 ms  1.449 ms
 3  msu-mipt-atm0.mipt.ru (212.16.1.1)  5.536 ms  5.993 ms  10.431 ms
 4  M9-LYNX.ATM6-0.11.M9-R2.msu.net (193.232.127.229)  12.994 ms  7.830 ms
6.816 ms
 5  Moscow-BNS045-ATM4-0-3.free.net (147.45.20.37)  12.228 ms  7.041 ms  8.731
 ms
 6  ChgNet-gw.free.net (147.45.20.222)  77.103 ms  75.234 ms  92.334 ms
7 netserv1.chg.ru (193.233.46.3)  96.627 ms  94.714 ms  134.676 ms
```

Первая строка содержит имя и IP-адрес хоста назначения, величина TTL может быть не более 30 и размер посылаемого пакета 38 байт. Последующие строки начинаются с TTL, после чего следует имя хоста или маршрутизатора и его IP адрес. Для каждого значения TTL отправляет 3 датаграммы. Для каждой возвращенной датаграммы вычисляется и выводится время возврата. Если в течение 3-х секунд на каждую из 3-х датаграмм не был получен ответ, то посылается следующая датаграмма, а вместо значения времени выводится звездочка. Время возврата – это время прохождения датаграммы от источника (хоста, выполняющего программу traceroute) до маршрутизатора. Если нас интересует

время, потраченное на каждую пересылку, то необходимо вычесть из значения времени TTL N время TTL N+1. В каждой из операционных систем сетевая часть утилиты реализована практически одинаково, но реализация на уровне приложений различается.

В ОС Solaris используется утилита traceroute. В качестве параметра задается IP-адрес или доменное имя удаленного хоста, связь до которого требуется проверить. В примере, приведенном выше, видно успешное выполнение traceroute и корректную работу сете-зависимых уровней (физический, канальный, сетевой).

В ОС – Windows утилита называется tracert. Используется также как и в ОС Solaris (tracert netserv1.chg.ru). Принципиального различия между утилитами tracert и traceroute нет. Особым отличием traceroute является наличие большей функциональности (например, можно указать, с какого TTL выводить информацию).

В случаях какой-либо неполадки выводится соответствующее сообщение. Например, при недоступности сети на маршрутизаторе выдается сообщение net unreachable:

```
Moscow-BNS045-ATM4-0-3.free.net (147.45.20.37)  947.327 ms !N  996.548 ms !N
  995.257 ms
```

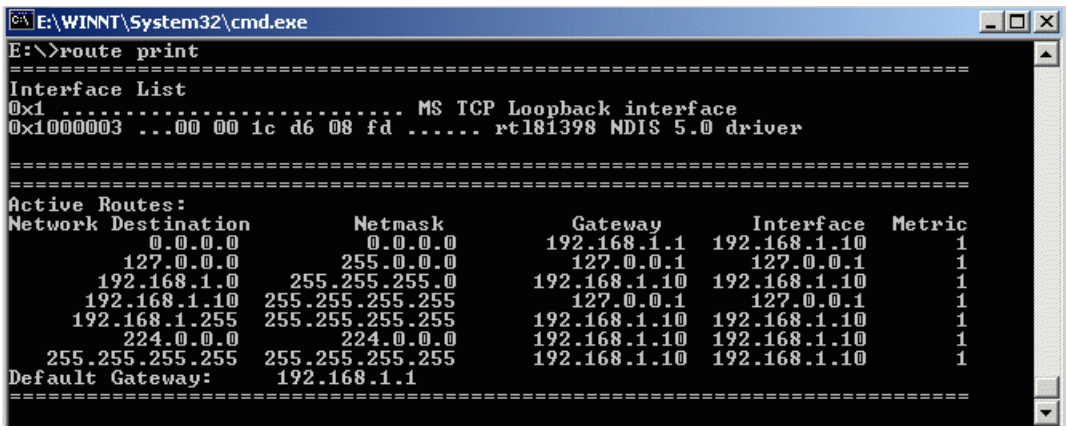
!N – где 147.45.20.37 – маршрутизатор, на котором последующий маршрут недоступен. Если недоступен сам хост: msu-mipt-atm0.mipt.ru (212.16.1.1) 5.536 ms !H 5.993 ms !H 10.431 ms !H. Если в качестве ошибки мы получаем !P - protocol unreachable.

## 9.5. Route

Для просмотра и редактирования таблицы маршрутов используется утилита – route. Типичный пример таблицы маршрутизации на персональном компьютере:

Для ОС Windows:

route print



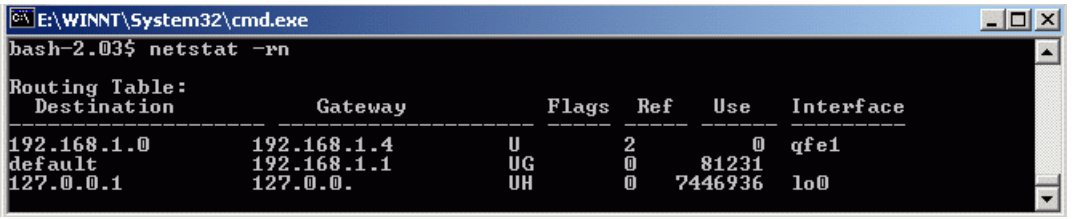

В таблице маршрутизации указывается сеть, маска сети, маршрутизатор, через который доступна эта сеть, интерфейс и метрика маршрута. Из приведенной таблицы видено, что маршрут по умолчанию доступен через маршрутизатор 192.168.1.1. Сеть 192.168.1.0 netmask 255.255.255.0 – является локальной сетью.

При добавлении маршрута можно использовать следующую команду.

```
route ADD 157.0.0.0 MASK 255.0.0.0  157.55.80.1
```

157.0.0.0 – удаленная сеть, 255.0.0.0 – маска удаленной сети, 157.55.80.1 – маршрутизатор, через который доступна эта сеть. Примерно такой же синтаксис используется при удалении маршрута: route DELETE 157.0.0.0

В ОС Solaris для просмотра таблицы маршрутизации используется немного другая команда – netstat –r.




Добавление и удаление маршрутов происходит командой route:

route add –net 157.6 157.6.1.20, где 157.6 – сокращенный адрес подсети, а 157.6.1.20 – маршрут, по которому эта сеть доступна. Также удаление маршрутов в таблице маршрутизации: route del –net 157.6

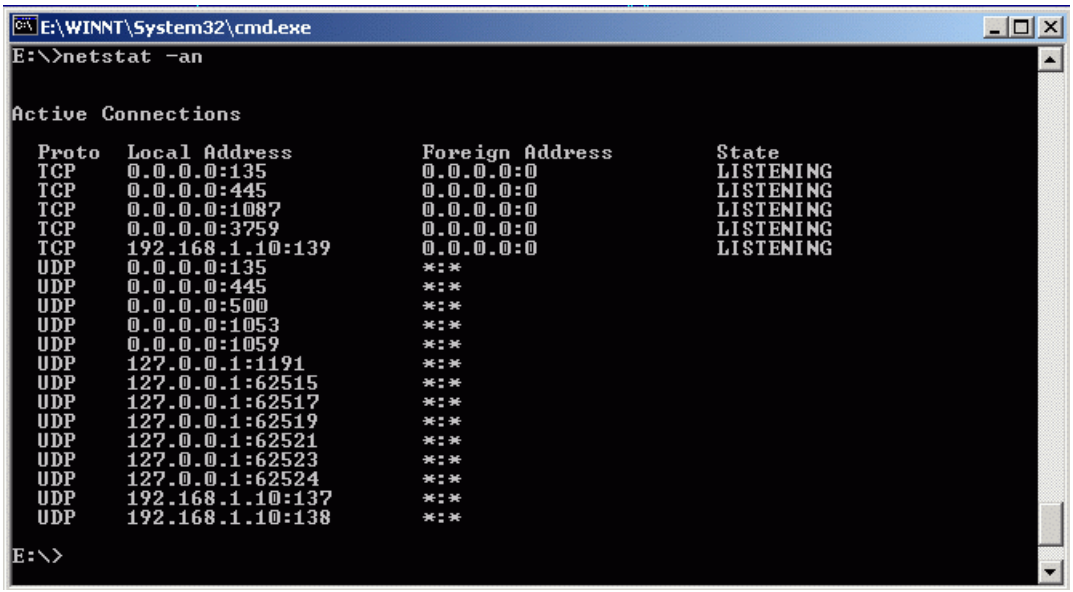
## 9.6. Netstat

Утилита netstat позволяет определить, какие порты открыты и по каким портам происходит передача данных между узлами сети. Например, если запустить веб-браузер и открыть для просмотра web-страницу, то, запустив netstat, можно увидеть следующую строку:

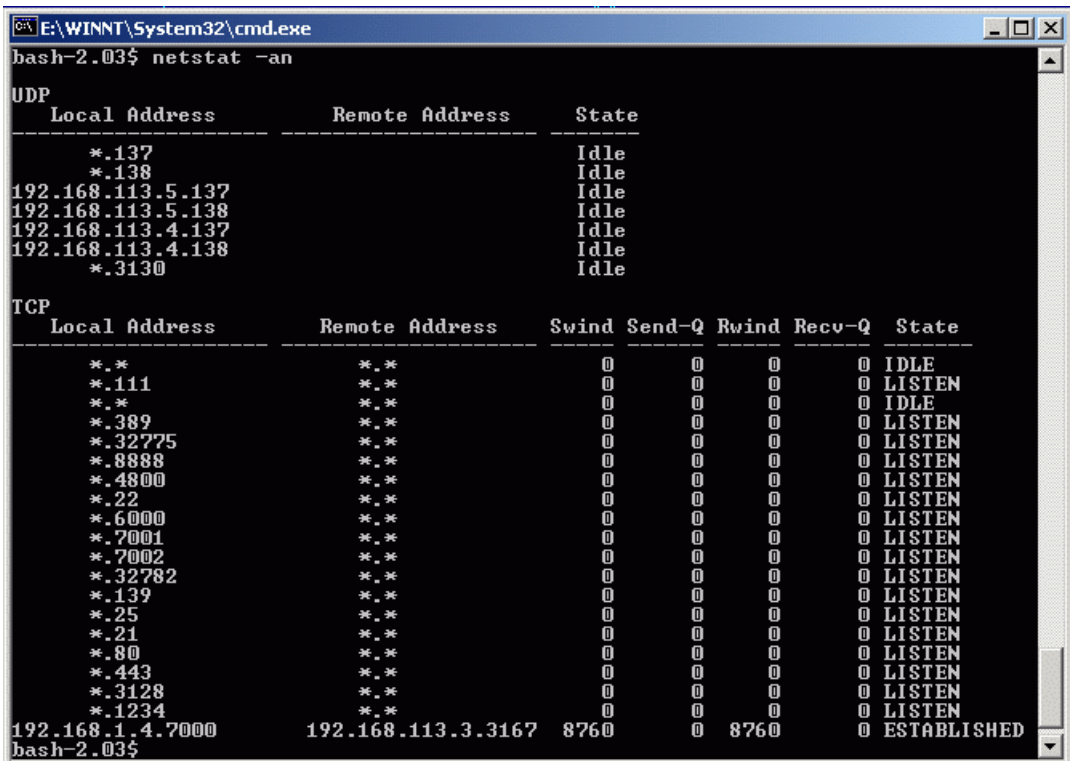
```
TCP    jawa:3687              www.ru:http            ESTABLISHED
```

В проведенном примере первое значение – TCP – тип протокола (может быть tcp,udp), далее идет имя локальной машины и локальный порт, www.ru:http - имя удаленного хоста и порта, к которому производится обращение, ESTABLISHED – показывает, что tcp-соединение установлено.

В ОС Windows командой netstat –an можно получить список всех открытых портов (параметр –n не определяет DNS-имя , а выводит только IP-адрес). Из примера ниже видно, что установленных соединений нет, а все открытые порты находятся в состоянии “прослушивания”, т.е. к этому порту можно обратиться для установки соединения. TCP-порт 139 отвечает за установку Netbios-сессий (например для передачи данных через “сетевое окружение”).




В ОС Solaris для получения информации об используемых портах также используется утилита netstat. Формат вывода практически одинаков.




## 9.7. Задания для практического занятия

1. Выведите информацию об IP-адресе, маске подсети и маршрутизаторе по умолчанию.
2. Выведите arp-таблицу.

3. Выполните утилиту ping. В качестве удаленного компьютера используйте IP-адрес компьютера из вашей локальной подсети; произвольного IP-адреса – например 194.87.0.50. Посмотрите arp-таблицу и обратите внимание на изменения в ней (если они есть).
4. Выполните утилиту traceroute. В качестве удаленного хоста используйте DNS-имя или IP-адрес удаленного сервера или компьютера.
5. Выполните утилиту netstat. Запустите веб-браузер и откройте произвольную интернет-страничку и практически одновременно выполните netstat заново.

# 10. Пакет java.net

Классы для работы с сетью в Java располагаются в пакете java.net, и простейшим из них является класс URL. С его помощью можно сконструировать uniform resource locator (URL), который имеет следующий формат:

```
protocol://host:port/resource
```

Здесь protocol - название протокола, используемого для связи; host - IP-адрес или DNS-имя сервера, к которому производится обращение; port - номер порта сервера (если порт не указан, то используется значение по умолчанию для указанного протокола); resource - имя запрашиваемого ресурса, причем оно может быть составным, например:

```
ftp://myserver.ru/pub/docs/Java/JavaCourse.txt
```

Затем можно воспользоваться методом openStream(), который возвращает InputStream, что позволяет считать содержимое ресурса. Например, следующая программа при помощи LineNumberReader считывает первую страницу сайта http://www.ru и выводит ее на консоль.

```
import java.io.*;
import java.net.*;

public class Net {
 public static void main(String args[]) {
  try {
   URL url = new URL("http://www.ru");
   LineNumberReader r = new LineNumberReader(new
InputStreamReader(url.openStream()));
   String s = r.readLine();
   while (s!=null) {
    System.out.println(s);
    s = r.readLine();
   }
   System.out.println(r.getLineNumber());
   r.close();
  } catch (MalformedURLException e) {
   e.printStackTrace();
  } catch (IOException e) {
   e.printStackTrace();
  }
```

```
  }
}
```

Из примера можно видеть, что работа с сетью, как и работа с потоками, требует дополнительной работы с исключительными ситуациями. Ошибка MalformedURLException появляется в случае, если строка с URL содержит ошибки.

Более функциональным классом является URLConnection, который можно получить с помощью метода класса URL.openConnection(). У этого класса есть два метода - getInput-Stream() (именно с его помощью работает URL.openStream()) и getOutputStream(), который можно использовать для передачи данных на сервер, если он поддерживает такую операцию (многие публичные web-сервера закрыты для таких действий).

Класс URLConnection является абстрактным. Виртуальная машина предоставляет реализации этого класса для каждого протокола, например, в том же пакете java.net определен класс HttpURLConnection. Понятно, что классы URL и URLConnection предоставляют возможности работы через сеть на прикладном уровне с помощью высокоуровневых протоколов.

Пакет java.net также предоставляет доступ к протоколам более низкого уровня - TCP и UDP. Для этого сначала надо ознакомится с классом InetAddress, который является интернет-адресом, или IP. Экземпляры этого класса создаются не с помощью конструкторов, а с помощью статических методов:

```
InetAddress getLocalHost()
InetAddress getByName(String name)
InetAddress[] getAllByName(String name)
```

Первый метод возвращает IP-адрес машины, на которой исполняется Java-программа. Второй метод возвращает адрес сервера, чье имя передается в качестве параметра. Это может быть как DNS-имя, так и числовой IP, записанный в виде текста, например, "67.11.12.101". Наконец третий метод определяет все IP-адреса указанного сервера.

Для работы с TCP-протоколом используются классы Socket и ServerSocket. Первым создается ServerSocket - сокет на стороне сервера. Его простейший конструктор имеет только один параметр - номер порта, на котором будут приниматься входящие запросы. После создания вызывается метод accept(), который приостанавливает выполнение программы и ожидает, пока какой-нибудь клиент не инициализирует соединение. В этом случае работа сервера возобновляется, а метод возвращает экземпляр класса Socket для взаимодействия с клиентом:

```
try {
 ServerSocket ss = new ServerSocket(3456);
 Socket client=ss.accept(); // Метод не возвращает управление, пока не
подключится клиент
} catch (IOException e) {
 e.printStackTrace();
}
```

Клиент для подключения к серверу также используется класс Socket. Его простейший конструктор принимает два параметра - адрес сервера (в виде строки или экземпляра InetAddress) и номер порта. Если сервер принял запрос, то сокет конструируется успешно, и далее можно воспользоваться методами getInputStream() или getOutputStream().

```
try {
 Socket s = new Socket("localhost", 3456);
 InputStream is = s.getInputStream();
 is.read();
} catch (UnknownHostException e) {
 e.printStackTrace();
} catch (IOException e) {
 e.printStackTrace();
}
```

Обратите внимание на обработку исключительной ситуации UnknownHostException, которая будет генерироваться, если виртуальная машина с помощью операционной системы не сможет распознать указанный адрес сервера в случае, если он задан строкой. Если же он задан экземпляром InetAddress, то эту ошибку надо обрабатывать при вызове статических методов этого класса.

На стороне сервера класс Socket используется точно таким же образом - через методы getInputStream() и getOutputStream(). Приведем более полный пример:

```
import java.io.*;
import java.net.*;

public class Server {
 public static void main(String args[]) {
  try {
   ServerSocket ss = new ServerSocket(3456);
   System.out.println("Waiting...");
   Socket client=ss.accept();
   System.out.println("Connected");
   client.getOutputStream().write(10);
   client.close();
   ss.close();
  } catch (IOException e) {
   e.printStackTrace();
  }
 }
}
```

Сервер по запросу клиента отправляет число 10 и завершает работу. Обратите внимание, что при завершении вызываются методы close() для открытых сокетов.

Класс клиента:

```
import java.io.*;
import java.net.*;

public class Client {
 public static void main(String args[]) {
  try {
   Socket s = new Socket("localhost", 3456);
   InputStream is = s.getInputStream();
   System.out.println("Read: "+is.read());
```

```
   s.close();
  } catch (UnknownHostException e) {
   e.printStackTrace();
  } catch (IOException e) {
   e.printStackTrace();
  }
 }
}
```

После запуска сервера, а затем клиента, можно увидеть результат - полученное число 10, после чего обе программы закроются.

Рассмотрим эти классы более подробно. Во-первых, класс ServerSocket имеет конструктор, в который передается кроме номера порта еще и адрес машины. Это может показаться странным, ведь сервер открывается на той же машине, где работает программа, зачем специально указывать ее адрес? Однако если компьютер имеет несколько сетевых интерфейсов (сетевых карточек), то он имеет и несколько сетевых адресов. С помощью такого детализированного конструктора можно указать, по какому именно адресу ожидать подключение. Это должен быть именно локальный адрес машины, иначе возникнет ошибка.

Аналогично класс Socket имеет расширенный конструктор для указания как локального адреса, с которого будет устанавливаться соединение, так и локального порта (иначе операционная система выделяет произвольный свободный порт).

Во-вторых, можно воспользоваться методом setSoTimeout(int timeout) класса ServerSocket, чтобы указать время в миллисекундах, на протяжении которого нужно ожидать подключение клиента. Это позволяет не "зависать" серверу, если никто не пытается начать с ним работать. Таймаут задается в миллисекундах, нулевое значение означает бесконечное время ожидания.

Важно подчеркнуть, что после установления соединения с клиентом сервер выходит из метода accept(), то есть перестает быть готов принимать новые запросы. Однако как правило желательно, чтобы сервер мог работать с несколькими клиентами одновременно. Для этого необходимо при подключении очередного пользователя создавать новый поток исполнения, который будет обслуживать его, а основной поток снова войдет в метод accept(). Приведем пример такого решения:

```
import java.io.*;
import java.net.*;

public class NetServer {
 public static final int PORT = 2500;
 private static final int TIME_SEND_SLEEP = 100;
 private static final int COUNT_TO_SEND = 10;
 private ServerSocket servSocket;

 public static void main(String[] args) {
  NetServer server = new NetServer();
  server.go();
 }

 public NetServer() {
```

```
 try{
  servSocket = new ServerSocket(PORT);
 }catch(IOException e){
  System.err.println("Unable to open Server Socket : " + e.toString());
 }
}

public void go() {

 // Класс-поток для работы с подключившимся клиентом
 class Listener implements Runnable{
  Socket socket;
  public Listener(Socket aSocket){
   socket = aSocket;
  }
  public void run(){
   try{
    System.out.println("Listener started");
    int count = 0;
    OutputStream out = socket.getOutputStream();
    OutputStreamWriter writer = new OutputStreamWriter(out);
    PrintWriter pWriter = new PrintWriter(writer);
    while(count<COUNT_TO_SEND){
     count++;
     pWriter.print(((count>1)?",":"")+ "Say" + count);
     sleeps(TIME_SEND_SLEEP);
    }
    pWriter.close();
   }catch(IOException e){
    System.err.println("Exception : " + e.toString());
   }
  }
 }

 // Основной поток, циклически выполняющий метод accept()
 System.out.println("Server started");
 while(true){
  try{
   Socket socket = servSocket.accept();
   Listener listener = new Listener(socket);
   Thread thread = new Thread(listener);
   thread.start();
  }catch(IOException e){
   System.err.println("IOException : " + e.toString());
  }
 }
}

public void sleeps(long time) {
```

```
  try{
   Thread.sleep(time);
  }catch(InterruptedException e){
  }
 }
}
```

Теперь объявим клиента. Эта программа будет запускать несколько потоков, каждый их которых независимо подключается к серверу, считывает его ответ и выводит на консоль.

```
import java.io.*;
import java.net.*;

public class NetClient implements Runnable{
 public static final int PORT = 2500;
 public static final String HOST = "localhost";
 public static final int CLIENTS_COUNT = 5;
 public static final int READ_BUFFER_SIZE = 10;

 private String name = null;

 public static void main(String[] args) {
  String name = "name";
  for(int i=1; i<=CLIENTS_COUNT; i++){
   NetClient client = new NetClient(name+i);
   Thread thread = new Thread(client);
   thread.start();
  }
 }

 public NetClient(String name) {
  this.name = name;
 }

 public void run() {
  char[] readed = new char[READ_BUFFER_SIZE];
  StringBuffer strBuff = new StringBuffer();
  try{
   Socket socket = new Socket(HOST, PORT);
   InputStream in = socket.getInputStream();
   InputStreamReader reader = new InputStreamReader(in);
   while(true){
    int count = reader.read(readed, 0, READ_BUFFER_SIZE);
    if(count==-1)break;
    strBuff.append(readed, 0, count);
    Thread.yield();
   }
  } catch (UnknownHostException e) {
   e.printStackTrace();
```

```
  } catch (IOException e) {
   e.printStackTrace();
  }
  System.out.println("client " + name + "   read : " + strBuff.toString());
 }
}
```

Теперь рассмотрим UDP. Для работы с этим протоколом и на стороне клиента, и на стороне сервера используется класс DatagramSocket. У него есть следующие конструкторы:

```
DatagramSocket()
DatagramSocket(int port)
DatagramSocket(int port, InetAddress laddr)
```

При вызове первого конструктора сокет открывается на произвольном доступном порту, что уместно для клиента. Конструктор с одним параметром, задающим порт, как правило применяется на серверах, чтобы клиенты знали, на каком порту им нужно пытаться устанавливать соединение. Наконец, последний конструктор необходим для машин, у которых присутствует несколько сетевых интерфейсов.

После открытия сокетов начинается обмен датаграммами. Они представляются экземплярами класса DatagramPacket. При отсылке сообщения применяется следующий конструктор:

```
DatagramPacket(byte[] buf, int length, InetAddress address, int port)
```

Массив содержит данные для отправки (созданный пакет будет иметь длину равную length), а адрес и порт указывают получателя пакета. После этого вызывается метод send() класса DatagramSocket.

```
try {
 DatagramSocket s = new DatagramSocket();
 byte data[]={1, 2, 3};
 InetAddress addr = InetAddress.getByName("localhost");
 DatagramPacket p = new DatagramPacket(data, 3, addr, 3456);
 s.send(p);
 System.out.println("Datagram sent");
 s.close();
} catch (SocketException e) {
 e.printStackTrace();
} catch (UnknownHostException e) {
 e.printStackTrace();
} catch (IOException e) {
 e.printStackTrace();
}
```

Для получения датаграммы также создается экземпляр класса DatagramPacket, но в конструктор передается лишь массив, в который будут записаны данные по получении (также указывается ожидаемая длина пакета). Сокет необходимо создать с указанием порта, иначе скорее всего сообщение просто не дойдет до адресата. Используется метод receive() класса DatagramSocket (аналогично методу ServerSocket.accept() этот метод также

прерывает выполнение потока, пока не придет запрос от клиента). Пример реализации получателя:

```
try {
 DatagramSocket s = new DatagramSocket(3456);
 byte data[]=new byte[3];
 DatagramPacket p = new DatagramPacket(data, 3);
 System.out.println("Waiting...");
 s.receive(p);
 System.out.println("Datagram received: "+data[0]+", "+data[1]+", "+data[2]);
 s.close();
} catch (SocketException e) {
 e.printStackTrace();
} catch (IOException e) {
 e.printStackTrace();
}
```

Если запустить сначала получателя, а затем отправителя, то можно увидеть, что первый напечатает содержимое полученной датаграммы, а потом программы завершат свою работу.

В заключение приведем пример сервера, который получает датаграммы и отправляет их обратно, дописав к ним слово received.

```
import java.io.*;
import java.net.*;

public class DatagramDemoServer {
 public static final int PORT = 2000;
 private static final int LENGTH_RECEIVE = 1;
 private static final byte[] answer = ("received").getBytes();

 private DatagramSocket servSocket = null;
 private boolean keepRunning = true;

 public static void main(String[] args) {
  DatagramDemoServer server = new DatagramDemoServer();
  server.service();
 }

 public DatagramDemoServer() {
  try{
   servSocket = new DatagramSocket(PORT);
  }catch(SocketException e){
   System.err.println("Unable to open socket : " + e.toString());
  }
 }

 protected void service() {
  DatagramPacket datagram;
  InetAddress clientAddr;
```

```
  int clientPort;
  byte[] data;
  while(keepRunning){
   try{
    data = new byte[LENGTH_RECEIVE];
    datagram = new DatagramPacket(data, data.length);
    servSocket.receive(datagram);
    clientAddr = datagram.getAddress();
    clientPort = datagram.getPort();
    data = getSendData(datagram.getData());
   datagram = new DatagramPacket(data, data.length, clientAddr, clientPort);
    servSocket.send(datagram);
   }catch(IOException e){
    System.err.println("I/O Exception : " + e.toString());
   }
  }
 }

 protected byte[] getSendData(byte b[]) {
  byte[] result = new byte[b.length+answer.length];
  System.arraycopy(b, 0, result, 0, b.length);
  System.arraycopy(answer, 0, result, b.length, answer.length);
  return result;
 }
}
```

# 11. Заключение

В данном разделе были рассмотрены теоретические моменты сети как одной большой взаимодействующей системы. Были рассмотрены все уровни модели OSI и их функциональные назначения. Также были рассмотрены основные утилиты, используемые для настройки и обнаружения неисправностей в сети.

# 12. Контрольные вопросы

16-1. Назовите уровни модели OSI.

a.) OSI модель состоит из семи уровней:

7 - уровень приложений

6 - уровень представлений

5- уровень сессий

4- транспортный уровень

3 - сетевой уровень

2 – уровень передачи данных

1 – физический уровень

16-2. Что характеризует физический уровень? Приведите основные типы физической среды передачи данных.

a.) Основная функция физического уровня – передача данных в виде сигнала (электрического, электромагнитного, светового). В простейшем случае конечное сообщение на физическом уровне формируется в виде битов (набор из 0 и 1, например, 010110101). В виде электрического сигнала оно выглядит как “пила”.

Основные типы физической среды передачи данных:

- телефонный провод;
- коаксиальный провод;
- витая пара;
- оптоволокно.

16-3. Опишите основные функции MAC-адреса и LLC-подуровня.

a.) MAC-адрес обеспечивает идентификацию компьютеров в сети. Адресаций является “плоской” – в пределах одного логического сегмента. MAC-sublayer отвечает за связь канального уровня с физическим уровнем. LLC-sublayer отвечает за связь канального уровня с сетевым уровнем. LLC-подуровень обеспечивает независимость передачи данных на сетевой уровень от физической среды, которая используется для передачи данных.

16-4. К какому классу сети относятся следующие IP-адреса?

- 64.12.8.130
- 224.180.224.5
- 172.16.0.1
- 194.86.87.256
- 195.149.20.130

a.) Ответ:

- A;
- D;
- B, private;
- не существует;
- C.

16-5. Каковы основные функции ARP-протокола и RARP-протокола?

a.) Для передачи данных отправителю кроме IP-адреса также нужно знать MAC-адрес получателя. Для определения соответствия IP-адресу MAC-адреса существует ARP-протокол (Address Resolution Protocol). ARP-таблица находится в оперативной памяти и периодически обновляется. В этой таблице находятся IP-адреса компьютеров только из той же

локальной сети. Когда источник определяет MAC-адрес получателя, сначала он обращается к своей arp-таблице, и если не находит его там, то делает широковещательный запрос для определения требуемого адреса.

Протокол RARP действует наоборот – известному MAC-адресу он сопоставляет IP-адрес. Это необходимо, например, для работы таких протоколов, как BOOTP, DHCP. При загрузке по локальной сети посылается broadcast запрос – противоположный arp-запросу. Если в ARP-запросе идет опрос “IP-получателя известен, MAC-получателя - ???”, то в RARP-запросе “MAC-получателя известен, IP - ???”. Например, сервис DHCP занимается динамическим выделением IP-адресов для компьютеров в локальной сети. Новая машина отправляет broadcast запрос со своим MAC-адресом. На сервере хранится таблица соответствия MAC и IP-адресов, поэтому он отвечает на rarp-запрос, и таким образом пользователю не приходится вручную вводить IP-адрес на компьютере (особенно это эффективно при большом количестве машин).

16-6. Что такое маска сети, маска подсети и как она вычисляется?

a.) Для определения размерности подсети используется маска подсети. Она определяет, какие биты в IP-адресе являются сетевой частью (network part), то есть адресуют сеть, а какие – host part (часть, задающая адрес компьютера в сети). Чтобы определить маску сети, нужно записать IP-адрес подсети в двоичном виде. Все биты, относящиеся к network-part заменим на 1, а все значения, относящиеся к host-part, заменим на 0. В результате получим маску сети.

Например маска сети класса A будет выглядеть следующим образом: 255.0.0.0, для сети класса B: 255.255.0.0, для сети класса C: 255.255.255.0. Если используется подсеть, то (по описанному алгоритму) все биты, относящиеся к сети и подсети, будут равны 1, к хосту – 0. Например маска 255.255.255.192 определяет подсеть класса C, в которой кол-во хостов будет равно 64.

16-7. Какие основные различия между протоколами TCP и UDP?

a.) TCP- протокол с образованием постоянного соединения гарантирует доставку каждого пакета. UDP не использует постоянного соединения и не гарантирует доставку сообщения. TCP-протокол используется в сервисах, где важна гарантированная доставка информации – электронная почта, передача HTML-страниц, FTP.

UDP используется в сервисах, где важна своевременная доставка пакета с минимальными затратами: например при транслировании видеоинформации.

16-8. Перечислите основные функции уровня приложений, уровня представлений, уровня приложений.

a.) Уровень приложений определяет, какие ресурсы существуют для связи между сетевыми узлами. Этот уровень не обеспечивает связь со всеми уровнями модели OSI.

Уровень представлений отвечает за представление данных в форме, понятной получателю. Например, для представления данных могут быть использованы такие кодировки, как Extended Binary Coded Decimal Interchange Code (EBCDIC) или American Standard Code for Information Interchange (ASCII). Если компьютеры, между которыми установлено сетевое соединение, используют различные протоколы, то presentation layer обеспечивает их корректное взаимодействие.

Уровень сессий обеспечивает управление диалогом: фиксирует, какая из сторон является активной в настоящий момент, и предоставляет средства синхронизации. Последние позволяют вставлять контрольные точки в длинные передачи, чтобы в случае отказа можно было вернуться назад к последней контрольной точке, а не начинать все сначала. На практике немногие приложения используют сеансовый уровень, и он редко реализуется в виде отдельных протоколов, хотя функции этого уровня часто объединяют с функциями прикладного уровня и реализуют в одном протоколе.

16-9. Ping и traceroute – назначение и различия между этими утилитами.

a.) Основное назначение утилит ping и traceroute – выявление неисправностей в сети. Эти утилиты используют протокол ICMP. Ping определяет работоспособность сети между двумя хостами. Во время работы утилита ping посылает echo-запрос на IP-адрес получателя. Если echo-запрос был корректно доставлен, то в ответ присылается echo-ответ. При корректном получении echo-ответа принимается решение, что удаленный хост доступен. Т.к. echo-ответ был корректно доставлен отправителю, можно сделать вывод что физические, канальные и сетевые уровни на обоих хостах работают корректно, и роутеры корректно пересылают пакеты.

Traceroute использует протокол ICMP для определения маршрута прохождения пакета. Он используется для выявления ошибок в таблицах маршрутизации, доступности подсетей и хостов.

16-10. Как происходит выбор маршрута для передачи данных? Какая утилита позволяет изменять маршрут передачи данных?

a.) Маршрут, по которому необходимо переслать пакет, определяется на основе данных маршрутной таблицы, которая хранится в каждом компьютере. В простейшем случае она содержит только одну запись для работы с удаленными сетями – маршрут по умолчанию. В этом случае все данные, посылаемые в удаленную сеть, отправляются на маршрутизатор по умолчанию, и далее он определяет дальнейший маршрут пересылки пакетов.

Если в маршрутной таблице есть несколько записей, то определяется, к какой подсети принадлежит IP-адрес получателя, и в соответствии с выбранной записью о маршруте происходит пересылка данных. Если в маршрутной таблице не найдена требуемая подсеть, то данные отправляются на “default gateway” – маршрутизатор по умолчанию.

Для просмотра и изменения таблицы маршрутизации используется утилита route.

16-11. Какие действия необходимо предпринять для установления TCP соединения между двумя Java-приложениями? ?

a.) Во-первых, на стороне сервера надо создать экземпляр класса Server-Socket с указанием порта, и затем вызвать у этого объекта метод accept(). При входе в этот метод поток исполнения приостанавливает свою работу в ожидании подключения клиента.

Клиенту необходимо создать экземпляр класса Socket с указанием IP-адреса и порта сервера. После успешного выполнения конструктора на стороне сервера метод accept() вернет экземпляр класса Socket для взаимодействия двух приложений.

16-12. Какие действия необходимо предпринять для обмена данными по UDP протоколу?

a.) Во-первых, отправляющая сторона должна создать экземпляр класса DatagramSocket. Затем поместить в объект DatagramPacket данные, предназначающиеся для отсылки, а также IP-адрес и порт получателя.

Принимающая стороны также создает DatgramSocket с указанием порта, на котором ожидается получение датаграммы. Создается объект Datagram-Packet для сохранения полученных данных, и вызывается метод Datagram-Socket.receive(). Теперь отправитель может вызывать метод Datagram-Socket.send()

16-13. Можно ли с помощью класса URL пересылать данные на сервер?

a.) Нет

b.) Да

✓c.) Да, если сервер позволяет закачивание (upload) данных.

# Глоссарий

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

*100% Pure Java™*
Под термином "100% Pure Java™" понимают Java-программу, которая опирается только на специфицированное поведение Java-платформы, не использует платформно-зависимых (native) методов и не зависит ни от каких программных интерфейсов, кроме Java core API. Программа сертификации служит для того, чтобы определить, является ли данное приложение или апплет 100% Pure Java или нет.

## A

*Abstract Window Toolkit - AWT*
Стандартный пакет графических компонент, реализованных с использованием специфических платформенных методов. Данные компоненты поддерживают лишь то подмножество функциональных возможностей, которое присуще всем платформам.
В значительной степени вытеснен набором компонент Project Swing (см. *Swing-компоненты*).

*abstract*
Ключевое слово языка программирования Java, используемое в определении класса для указания на невозможность создания его экземпляров, но при этом доступного для наследования другими классами. Абстрактный класс может содержать нереализованные (абстрактные) методы, которые должны быть реализованы в его подклассах.

*абстрактный класс (abstract class)*
Класс, который содержит один или более абстрактных методов, вследствие чего нельзя создавать экземпляры данного класса. Абстрактные классы определены таким образом, чтобы другие классы могли расширять и конкретизировать их, реализуя абстрактные методы.

*абстрактный метод (abstract method)*
Метод, не имеющий реализации.

*контроль доступа (access control)*
Технические средства, посредством которых ограничиваются множества пользователей или программ, взаимодействующих с ресурсами, с целью повышения целостности, конфиденциальности и доступности.

*АЦИН (ACID - Atomicity, Consistency, Isolation and Durability)*
Акроним четырех свойств, обеспечиваемых транзакциями: атомарность, целостность, изоляция и надежность.

*активация (activation)*
Процесс передачи корпоративных компонент (enterprise beans) из вторичного устройства хранения данных в память.

*список фактических параметров (actual parameter list)*
Аргументы, определенные в вызове метода (см. *список формальных параметров*).

*альфа-фактор (alpha value)*
Значение, указывающее яркость (или интенсивность) пикселя.

*интерфейс прикладного программирования (API - Application Programming Interface)*
Спецификация, предназначенная для пользователей и описывающая методы доступа к свойствам и состоянию объектов и классов.

*апплет (applet)*
Компонент, который обычно исполняется в Web-браузере или в любой другой программе просмотра апплетов.

*контейнер апплетов (applet container)*
Контейнер, включающий в себя поддержку модели программирования апплетов.

*устройства (appliances)*
Сетевые устройства, такие как принтеры, терминалы с поддержкой технологии Java™ и клиенты, управляемые посредством использования Java Management API (JMAPI).

*компоновщик приложения (application assembler)*
Человек, объединяющий компоненты и модули в большие модули разработки.

*клиентское приложение (application client)*
Клиентская программа первого уровня, исполняемая на собственной виртуальной Java-машине.

*контейнер клиентского приложения (application client container)*
Контейнер, поддерживающий клиентские приложения и обеспечивающий интегрированное представление API платформы J2EE.

*модуль клиентского приложения (application client module)*
Программный модуль, состоящий из одного или более классов и описания клиентского приложения.

*поставщик программных компонент (Application Component Provider)*
Поставщик, который предоставляет классы Java, реализующие методы компонентов, описания JSP-страниц и необходимые дескрипторы.

*модель программирования приложения (Application Programming Model - APM)*
Модель прикладного программирования, которая определяет, как использовать и объединять возможности платформы J2EE для создания прикладных решений в предметной области предприятия.

*аргумент (argument)*
Элемент данных, указанный в вызове метода. Аргумент может быть константой, переменной или выражением.

*массив (array)* Совокупность элементов данных одного типа, в которой позиция каждого элемента однозначно определена целым числом (индексом массива).

*Американский Стандартный Код Обмена Информацией (American standard code for information interchange - ASCII)*
Стандартно, на один символ отводится 7 бит. См. также *Уникод*.

*атомарный (atomic)*
Операция, выполняемая как единый неделимый акт.

*аутентификация (authentication)*
Процесс, посредством которого один объект показывает другому, что он действует от имени определенной идентификационной записи. Платформа J2EE нуждается в трех видах аутентификации: обычной (basic), связанной с формой (form-based) и взаимной (mutual).

*авторизация (authorization)*
См. *управление доступом*.

*ограничение авторизации (authorization constraint)*
Набор ролевых имен, предназначенных для защиты и разрешающих доступ к Web-ресурсам.

## B

*базовая (открытая) аутентификация (basic authentication)*
Метод проверки Web-сервером имени пользователя и пароля, полученных при помощи механизма аутентификации, встроенного в Web-клиент.

*bean-компонент (bean)*
Программный компонент многократного использования. Bean-компоненты могут быть объединены для создания приложения.

*механизм сохранения, управляемый bean-компонентом (bean-managed persistence)*
Механизм, при котором передача данных между переменными экземпляров bean-компонента и основным администратором ресурсов управляется bean-компонентом.

*транзакция, управляемая bean-компонентом (bean-managed transaction)*
Корпоративный компонент (enterprise bean) определяет границы транзакции.

*бинарный оператор (binary operator)*
Знак операции, имеющий два аргумента.

*бит (bit)*
Минимальная единица информации в компьютере. Может принимать значения 0 или 1.

*побитовый оператор (bitwise operator)*
Знак операции, воздействующий на операнды, как на набор битов (0 и 1). Например, бинарные логические операции (&, |, ^), бинарные операции сдвига (<<, >>, >>>) и унарная операция дополнения (~).

*блок (block)*
Любой код на языке программирования Java™, заключенный между двумя фигурными скобками. Например, {x = 1;}.

*булевский (boolean)*
Относится к выражению или переменной, которые могут принимать только два значения: true ("истина") и false ("ложь"). В языке программирования Java™ существует тип boolean, а также литеральные константы true и false.

*ограничительная область (bounding box)*
Прямоугольник наименьшего размера, содержащий указанную геометрическую фигуру. Для растровых объектов включает все заданные пиксели.

*break*
Ключевое слово языка программирования Java™. Оператор break передаёт управление строке, следующей за текущей структурой. Если за break следует метка, то программа продолжает исполнение, начиная с "помеченного" оператора.

*бизнес-логика (business logic)*
Код, реализующий функциональность приложения. В модели Enterprise Java Beans эта логика реализуется при помощи методов корпоративного компонента (enterprise bean).

*бизнес-метод (business-method)*
Метод корпоративного компонента (enterprise bean), реализующий бизнес-логику или правила приложения.

*байт (byte)*
Последовательность из восьми битов. В языке программирования Java™ определен соответствующий тип byte.

*байт-код (bytecode)*
Машинно-независимый код, генерируемый Java-компилятором и исполняемый Java-интерпретатором.

## C

*методы обратной связи (callback methods)*
Метод компонента, вызываемый контейнером для уведомления компонента о важных событиях во время его жизненного цикла.

*вызывающий оператор (caller)*
См. *администратор вызывающего оператора*.

*администратор вызывающего оператора (caller principal)*
Администратор, который идентифицирует объект, вызывающий метод корпоративного компонента (enterprise bean).

*case*
Ключевое слово языка программирования Java (оператор switch), которое определяет набор инструкций, исполняющихся в случае, если значение переключающего выражения совпадает со значением константы (разметки выбирающего предложения), указанной в конструкции case.

*преобразование типа (casting)*
Явное преобразование одного типа данных в другой.

*catch*
Ключевое слово языка программирования Java™, используемое для объявления блока инструкций, которые должны быть выполнены в случае исключительной ситуации или ошибки выполнения, возникающей в предшествующем блоке try.

*char*
Ключевое слово языка программирования Java™, используемое для объявления переменной символьного типа.

*класс (class)*
Тип в языке программирования Java™, определяющий реализацию особого вида объекта. Описание класса определяет экземпляр класса, его переменные и методы. Так же определяются интерфейсы и суперклассы. По умолчанию суперклассом любого класса является Object.

*метод класса (типа) (class method)*
Метод, который вызывается безотносительно ссылки на конкретный объект. Методы класса влияют на класс в целом, а не на конкретный его экземпляр.
Также носит название статического метода. См. также *метод экземпляра*.

*путь к классам (classpath)*
Переменная среды окружения, которая сообщает виртуальной машине Java™ и приложениям Java (например, утилитам, расположенным в директории JDK™ 1.1.X\bin), где находятся библиотеки классов, включая пользовательские библиотеки.
Свойство виртуальной машины (JVM), которое может быть задано с помощью переменной окружения, либо другими способами, например, с помощью опций JVM.

*переменная класса (типа) (class variable)*
Переменная, относящаяся к некоторому классу в целом, а не к отдельным экземплярам класса. Переменные класса являются элементами определения класса.
Также носит название статического поля. См. также *переменная экземпляра*.

*клиент (client)*
В модели соединения "клиент-сервер" клиент - это процесс, который удаленно обращается к ресурсам вычислительного сервера.

*адрес основного кода (codebase)*
При работе с атрибутом code в тэге <APPLET> указывает полный путь к файлу главного класса апплета: code определяет имя файла, а codebase - URL директории, содержащей файл.

*комментарий (comment)*
Поясняющий текст в программе, который игнорируется компилятором. В приложениях Java™ комментарии выделяются символами // или /*...*/.

*фиксация (транзакции) (commit)*
Момент транзакции, когда все изменения, проведенные в процессе транзакции, фиксируются в базе данных.

*единица компиляции (compilation unit)*
Минимальная единица исходного кода, которая может быть откомпилирована. В текущей реализации Java™ единица компиляции - это последовательность определений интерфейсов и классов, которой могут предшествовать объявление пакета и объявления импорта.

*компилятор (compiler)*
Программа, транслирующая исходный код приложения в код, исполняемый компьютером. Java™-компилятор транслирует исходный код, написанный на языке Java, в машинно-независимый код (байт-код) для виртуальной машины Java. См. также *интерпретатор*.

*компонент (component)*
Программный модуль, поддерживаемый контейнером. Компоненты конфигурируемы на стадии разработки. Платформа J2EE определяет четыре вида компонент: корпоративные компоненты (enterprise beans), Web-компоненты, апплеты и клиентские приложения.

*контракт компонента (component contract)*
Набор условий, регулирующих отношения между компонентом и его контейнером. Контракт включает: управление жизненным циклом компонента, контекстный интерфейс, используемый образцом компоненты для получения информации о контейнере или использования возможностей контейнера, и списка функциональных возможностей, которые должен поддерживать каждый контейнер для данной компоненты.

*окружение компонента (component environment)*
Набор требований, определяемых Поставщиком Программных Компонент (Application Component Provider), которые должны быть доступны компоненту J2EE. Записи окружения декларативно определены в описании компонента. Каждый компонент указывает и получает доступ к значениям конфигурации компонента, используя контекст java:comp/env JNDI. Эти значения могут быть объектами, от которых зависит компонент, такими как JDBC DataSource или простыми значениями, такими как налоговая ставка.

*компоновка (compositing)*
Процесс наложения одного изображения на другое, с целью получения единого изображения.

*соединение (connection)*
См. *менеджер соединений*.

*мастер соединения (connection factory)*
См. *менеджер ресурсов мастера соединения*.

*коннектор (connector)*
Стандартный механизм расширения контейнеров для обеспечения взаимодействия с

управленческими информационными системами (Executive Information Systems - EISs). Коннектор специфичен для каждой EIS и состоит из адаптера ресурсов и средств разработки приложений для взаимодействия с EIS. Адаптер ресурсов подключен к контейнеру посредством контрактов системного уровня, определенных в архитектуре коннектора.

*архитектура коннектора (connector architecture)*
Архитектура, предназначенная для интеграции серверов J2EE с управленческими информационными системами (Executive Information Systems - EISs). Архитектура состоит из двух частей: адаптер ресурсов производителя EIS и сервер J2EE, поддерживающий этот адаптер. Данная архитектура определяет набор контрактов, которые должны поддерживаться адаптером ресурсов для подключения к J2EE-серверу. Например, транзакции, обеспечение безопасности, управление ресурсами.

*конструктор (constructor)*
Метод особого рода, создающий объект и инициализирующий его поля. В языке программирования Java имя конструктора совпадает с именем класса. Конструкторы вызываются системой при создании экземпляра объекта (исполнении конструкции new).

*const*
Зарезервированное ключевое слово языка Java™. Однако, не используется текущими версиями Java.

*контейнер (container)*
Сущность, обеспечивающая управление, безопасность, разработку и сервисы выполнения компонент. Кроме того, каждый тип контейнера (EJB, Web, JSP, сервлет, апплет или приложение-клиент) также предоставляет свои специфические сервисы.

*персистенция (сохраняемость), управляемая контейнером (container-managed persistence)*
Механизм, при котором передача данных между экземпляром корпоративной компоненты и менеджером расположенных ниже ресурсов управляется контейнером корпоративной компоненты (enterprise bean).

*транзакция, управляемая контейнером (container-managed transaction)*
Транзакция, границы которой определяются EJB-контейнером. Экземпляр корпоративной компоненты (enterprise bean) должен использовать транзакции, управляемые контейнерами.

*контекстный атрибут (context attribute)*
Объект, встроенный в контекст, ассоциированный с сервлетом.

*continue*
Ключевое слово языка программирования Java, которое обозначает оператор, завершающий текущую итерацию цикла и, если условие повторения выполняется, начинающий исполнение следующей. Если за данным ключевым словом следует метка, continue возобновляет исполнение, начиная с помеченного оператора (то есть начинается следующая итерация цикла, помеченного данной меткой).

*диалоговый режим (conversational state)*
Значения полей сессии bean-компонента плюс транзитивное замыкание объектов, доступных из полей bean-компонента. Транзитивное замыкание bean-компонента определяется в

терминах протокола сериализации языка Java, которые сохранены посредством сериализации экземпляра bean-компонента.

*Common Object Request Broker Architecture - CORBA*
Технология построения распределенных объектных приложений, специфицируемая группой по развитию стандартов объектного программирования (Object Management Group - OMG).

*базовый класс (core class)*
Публичный класс (или интерфейс), являющийся стандартным членом Java™ Platform. Обязательным свойством таких классов является их доступность в любой операционной системе, поддерживающей среду Java. Программой, "полностью" написанной на языке Java, называется программа, использующая только такие классы, и, следовательно, обладающая свойством независимости от платформы. См. также *100% Pure Java™*.

*базовые пакеты (core packages)*
Необходимый набор программных интерфейсов приложений (Application Programming Interfaces - APIs) платформы Java, который поддерживается в любой реализации.

*метод create (create method)*
Метод, определенный в "домашнем" интерфейсе и вызываемый клиентом, для создания корпоративной компоненты (enterprise bean).

*полномочия (credentials)*
Учетная запись с параметрами доступа пользователя, сформированными после его успешной аутентификации.

*критическая секция (critical section)*
Фрагмент программы, в котором поток выполняет действия над общим ресурсом, доступ к которому должен быть монопольным.

*пакет проверки совместимости (Compatibility Test Suite - CTS)*
Набор программ, предназначенных для проверки соответствия продуктов J2EE спецификации платформы J2EE.

## D

*объявление (declaration)*
Выражение, связывающее идентификатор с атрибутами (типом). При необходимости осуществляется выделение памяти (для данных) или выполнение (для методов). См. также *описание*.

*default*
Ключевое слово языка программирования Java™, при необходимости используемое после всех условий case в блоке switch. Если ни одно из проверяемых значений не совпадает со значением оператора switch, то выполняются инструкции, следующие после ключевого слова default.

*описание (definition)*
Объявление, осуществляющее резервирование памяти (для данных) или выполнение (для методов). См. также *объявление*.

*делегирование (delegation)*
Передача (делегирование) функций - способность объекта или потока внутри объекта выполнять под именем клиента запросы к другим удаленным объектам.

*администратор размещения (deployer)*
Человек, устанавливающий модули и приложения J2EE в операционной системе.

*размещение (deployment)*
Процесс установки программного обеспечения в операционную среду.

*дескриптор размещения (deployment descriptor)*
XML-файл, поставляемый с каждым модулем и приложением и описывающий процесс их установки. Дескриптор размещения управляет инструментами для установки модуля или приложения с какими-либо специфичными опциями контейнера, а также описывает особые требования, предъявляемые к конфигурации.

*антирекомендация (deprecation)*
Относится к классам, интерфейсам, конструкторам, методам или полям, которые рекомендуется больше не использовать, и которые могут быть исключены из последующих версий.

*"является потомком" (derived from)*
Класс X "является потомком" класса Y, если класс X расширяет (наследует) класс Y. См. также *подкласс*, *суперкласс*.

*распределенный (distributed)*
Работающий более чем в одном адресном пространстве.

*распределенное приложение (distributed application)*
Приложение, которое составлено из различных компонент, выполняющихся в различных средах, обычно, на разных платформах, соединенных посредством сети. Стандартные распределенные приложения: двухзвенное (клиент/сервер), трехзвенное (клиент/промежуточное программное обеспечение/сервер), n-звенное (клиент/множественное промежуточное программное обеспечение/сервер).

*do*
Ключевое слово языка программирования Java™, используемое для объявления цикла, повторяющего блок инструкций. Условие выхода из цикла определяется ключевым словом while в конце итерации.

*объектная модель документа (Document Object Model - DOM)*
Дерево объектов и интерфейсы для реализации обхода вершин дерева и написания его XML-версии согласно спецификации W3C.

*double*
Ключевое слово языка программирования Java™, используемое для определения переменной типа double.

*двойная точность (double precision)*
Согласно спецификации языка программирования Java™, число с плавающей точкой, занимающее 64 бита данных. См. также *одинарная точность*.

*определение типа документа (Document Type Definition - DTD)*
Описание структуры и свойств XML-файлов.

**E**

*EJB-контейнер (EJB-container)*
Контейнер, содержащий корпоративные компоненты (EJB).

*поставщик EJB-контейнера (EJB Container Provider)*
Производитель, который предоставляет EJB-контейнер.

*EJB-контекст (EJB-context)*
Объект, позволяющий корпоративному компоненту (enterprise bean) запускать сервисы контейнера и получать информацию о программе, вызвавшей клиентский метод.

*"домашний" объект EJB (EJB home object)*
Объект, обеспечивающий жизненный цикл операций (создания, удаления, поиска) для корпоративного компонента (enterprise bean). Класс для "домашнего" объекта EJB порожден инструментами разработки контейнера. "Домашний" объект EJB реализует собственный интерфейс корпоративного компонента (enterprise bean). Клиент ссылается на "домашний" объект EJB, чтобы выполнить операции жизненного цикла EJB-объекта. Для определения местонахождения EJB home object клиент использует JNDI интерфейс.

*EJB-.jar файл (EJB .jar file)*
Архив JAR, который содержит EJB-модуль.

*EJB-модуль (EJB module)*
Программный модуль, который состоит из одного или более корпоративных компонентов (enterprise beans) и дескриптора установки EJB.

*EJB-объект (EJB object)*
Объект, чей класс реализуется удаленным интерфейсом корпоративного компонента (enterprise bean). Клиент никогда не ссылается непосредственно на экземпляр корпоративного компонента, а только на объект EJB. Класс для объекта EJB порожден инструментами разработки, входящими в состав контейнера.

*EJB-сервер (EJB-server)*
Программное обеспечение, предоставляющее службы EJB-контейнеру. Например, EJB-контейнер обычно зависит от менеджера транзакций, который является частью EJB-сервера, предназначенной для двухфазного завершения распределенных транзакций параллельно со

всеми участвующими в ней менеджерами ресурсов. Архитектура J2EE предполагает, что EJB-сервер, содержащий EJB-контейнер, поставляется тем же производителем, поэтому не существует соглашения между двумя этими сущностями. EJB-сервер может содержать один и более контейнеров EJB.

*поставщик EJB-сервера (EJB Server Provider)*
Производитель, который поставляет EJB-сервер.

*ресурс управленческой информационной системы (ресурс УИС) (EIS resource)*
Сущность, предоставляющая функциональные возможности УИС клиентам системы. Например, запись или множество записей в системе базы данных, бизнес-объект в ERP-системе (системе, предназначенная для планирования и управления ресурсами предприятий), программа транзакции в системе обработки транзакций.

*else*
Ключевое слово языка программирования Java™, используемое для выполнения блока инструкций в том случае, если проверка условия в операторе if дала отрицательный результат (false).

*технология EmbeddedJava™ (Embedded Java™ technology)*
Технология, которая впервые появилась в рамках Java 2 Platform. Распространение этой технологии ограничено лицензионным соглашением, которое позволяет владельцу лицензии использовать определенные Java-технологии для создания и внедрения встроенных приложений.

*инкапсуляция (encapsulation)*
Локализация (упрятывание) части данных в пределах класса. Поскольку объекты инкапсулируют данные и реализацию, пользователь может рассматривать объект как черный ящик, предоставляющий услуги. Переменные и методы экземпляров класса могут добавляться, удаляться или изменяться, но до тех пор, пока услуги, предоставляемые объектом, не изменяются, нет необходимости переписывать код, использующий данный объект. См. также *переменная экземпляра класса* и *метод экземпляра класса*.

*корпоративный компонент (enterprise bean)*
Компонент, реализующий бизнес-задачу или бизнес-объект, либо сущность или сессионный компонент.

*управленческая информационная система - УИС (Enterprise Information System - EIS)*
Приложения, которые содержат существующую систему управления информацией всей компании. Эти приложения обеспечивают информационную инфраструктуру предприятия. УИС предлагает определенный набор услуг своим клиентам. Эти услуги представлены клиентам как локальные и/или удаленные интерфейсы. Примеры УИС: ERP-система (система, предназначенная для планирования и управления ресурсами предприятий), универсальная система обработки транзакций, существующая система баз данных.

*поставщик корпоративных компонент (Enterprise Bean Provider)*
Прикладной программист, разрабатывающий классы корпоративных компонентов (enterprise beans), удаленные и локальные интерфейсы, файлы дескрипторов установки и упаковывает их в EJB-.jar файл.

*Enterprise JavaBeans™ - EJB*
Архитектура, предназначенная для развития и установки объектно-ориентированных, распределенных, корпоративных приложений. Приложения, написанные с использованием архитектуры Enterprise JavaBeans™, являются масштабируемыми, многопользовательскими и безопасными.

*компонент управления данными (entity bean)*
Корпоративный компонент (enterprise bean), который отображает постоянные данные, хранящиеся в базе данных. Сущность идентифицируется первичным ключом. Если контейнер, которому принадлежит сущность, разрушается, сущность, ее первичный ключ и все удаленные ссылки сохраняются.

*ebXML*
ebXML основан на стандарте XML (Extensive Markup Language) и предназначен для поставки модульного набора спецификаций, позволяющего предприятиям различного масштаба и географического положения вести дела посредством Internet. ebXML снабжает компании и организации стандартным методом обмена деловой информацией, ведения торговых отношений, передачи данных, а также определения и регистрации бизнес процессов.

*исключительная ситуация (exception)*
Ситуация, возникающая во время работы программы вследствие неудачного выполнения операции, или запрограммированная с использованием оператора throw. Если в программе не предусмотрена реакция на возникшую ситуацию, дальнейшее исполнение программы невозможно. Язык программирования Java™ поддерживает обработку исключительных ситуаций при помощи ключевых слов try, catch и throw. См. также *обработчик исключительных ситуаций*.

*обработчик исключительных ситуаций (exception handler)*
Блок кода, реагирующий на определенный тип исключительных ситуаций. Если исключительная ситуация произошла вследствие ошибки, после которой программа может возобновить работу, то программа продолжает выполнение после обработчика исключительных ситуаций.

*исполняемый код (executable context)*
Программа, которая выполняется из HTML-файла. См. также *апплет*.

*"расширяет" (extends)*
Класс X может "расширять" класс Y с целью добавления функциональности (при помощи добавления полей и методов классу Y или переопределения методов класса Y). В таком случае говорят, что класс X является подклассом класса Y. Один интерфейс "расширяет" другой при помощи добавления методов. См. также *"произошел из"*.

## F

*метод поиска (finder method)*
Метод, определенный в "домашнем" интерфейсе и вызываемый клиентом для нахождения сущности.

*первая поставка пользователям (First Customer Ship - FCS)*
Дата выпуска/передачи продукта пользователю.

*поле (field)*
Элемент класса. Пока не определено обратное, поле не является статическим.

*терминальный - final*
Ключевое слово языка программирования Java™. Объявление класса с модификатором final означает, что класс нельзя расширить или наследовать. Final-методы не могут переопределяться, final-переменные не могут изменять свое начальное значение.

*finally*
Ключевое слово языка программирования Java™, задающее блок операций, которые будут выполнены независимо от того, произошла исключительная ситуация или ошибка в блоке, определенном с ключевым словом try, или нет.

*float*
Ключевое слово языка программирования Java™, используемое для определения числа с плавающей точкой.

*for*
Ключевое слово языка программирования Java™, используемое для объявления цикла, повторяющего блок инструкций. Программист определяет выполняемые операции, условия выхода из цикла, а также некоторые начальные действия перед началом первой итерации (инициализацию служебных переменных и пр.).

*аутентификация, связанная с формой (form-based authentication)*
Аутентификация, при которой Web-сервер выдает специализированную форму для входа в систему.

*FTP - протокол (File Transfer Protocol - FTP)*
Internet-протокол, основанный на TCP\IP. Предназначен для передачи файлов между хост-компьютерами. См. также *TCP\IP*.

*список формальных параметров (formal parameter list)*
Параметры, заданные в описании метода. См. также *список фактических параметров*.

## G

*"сборка мусора" (garbage collection)*
Автоматическое обнаружение и освобождение памяти, которая больше не используется. Система управления и слежения за правильностью выполнения программы (Java™ runtime system) выполняет "сборку мусора" таким образом, чтобы программисту не требовалось напрямую вызывать методы освобождения памяти. При необходимости программист может запустить сборщик мусора явно, путем вызова специального метода. В этом случае сборка мусора будет выполняться синхронно.

*goto*
Зарезервированное слово языка программирования Java™. Однако, в текущих версиях языка не используется.

*группа (group)*
Совокупность пользователей в пределах данного домена политики безопасности.

*графический интерфейс пользователя (Graphical User Interface - GUI)*
Графический интерфейс, предназначенный для удобства использования некоторых программ.

## H

*манипулятор (handle)*
Объект, используемый для уникальной идентификации корпоративного компонента (enterprise bean). Клиент может сериализовать манипулятор, а затем десериализовать его для получения ссылки на корпоративный компонент (enterprise bean).

*шестнадцатеричный (hexadecimal)*
Система счисления с основанием 16. Знаки 0-9 и a-f представляют цифры от 0 до 15. В программе, написанной на языке Java™, шестнадцатеричные числа должны начинаться с символов 0x. См. также *восьмеричный*.

*иерархия (hierarchy)*
Классификация соотношений, в которой каждый элемент, кроме верхнего (называемого корнем), является специализированным видом элемента, расположенного над ним. Каждый элемент может иметь один или несколько элементов, находящихся ниже него в иерархии. В иерархии классов Java™, образуемой при наследовании, корнем является класс Object.

*"домашний" интерфейс (home interface)*
Один из двух интерфейсов для корпоративного компонента (enterprise bean). "Домашний" интерфейс может определять несколько методов для создания и удаления корпоративного компонента (enterprise bean). Для сессионных компонентов (session beans) "домашний" интерфейс определяет методы создания и удаления, тогда как для сущностей (entity beans) - методы создания, нахождения и удаления.

*"домашний" манипулятор (home handle)*
Объект, используемый для получения ссылки на "домашний" интерфейс. "Домашний" манипулятор может быть сериализован и записан в ПЗУ, а также десериализован для получения ссылки.

*Браузер HotJava™ (HotJava™ Browser)*
Легко настраиваемый Web-браузер, разработанный компанией Sun Microsystems и написанный на языке программирования Java™.

*язык разметки гипертекста (HyperText Markup Language - HTML)*
Файловый формат для гипертекстовых страниц в Internet, основанный на SGML. Очень прост и разрешает внедрение изображений, звуков, видео, полей форм, а также простое форматирование текста. Ссылки на другие объекты внедряются с помощью URL.

*протокол передачи гипертекстовых файлов (HyperText Transfer Protocol - HTTP)*
Internet-протокол, основанный на TCP/IP и предназначенный для доступа к гипертекстовым объектам с удаленного компьютера. См. также *TCP/IP*.

*протокол защищенной передачи гипертекстов (HyperText Transmission Protocol, Secure - HTTPS)*
HTTP в совокупности с SSL (Secure Sockets Layer) - протоколом защищенных сокетов.

## I

*язык описания интерфейсов (Interface Definition Language - IDL)*
Интерфейсы прикладного программирования (APIs), написанные на языке Java™, которые обеспечивают функциональную совместимость на основе стандартов и способность взаимодействия с CORBA (Common Object Request Broker Architecture).

*идентификатор (identifier)*
Имя элемента (переменной, класса, метода и проч.) программы, написанной на языке Java™.

*Internet InterORB Protocol - IIOP*
Протокол, определяющий передачу сообщений между сетевыми объектами по протоколам TCP/IP и используемый для коммуникаций между брокерами объектных запросов CORBA.

*if*
Ключевое слово языка программирования Java™, предназначенное для проверки условия и выполнения блока инструкций, если проверка дала положительный результат (true).

*заимствование прав (impersonation)*
Процесс, при котором одна сущность принимает идентичность и привилегии другой сущности без ее уведомления о том, что произошло делегирование. Заимствование прав - это случай простого делегирования.

*implements*
Ключевое слово языка программирования Java™, включаемое при необходимости в объявление класса, для определения интерфейсов, которые реализует данный класс.

*import*
Ключевое слово языка программирования Java™, определяющее классы или целые пакеты, на которые может ссылаться программа, без включения названий пакетов в ссылку.

*наследование (inheritance)*
Концепция классов, автоматически включающих все переменные и методы, определенные в супертипе. См. также *суперкласс*, *подкласс*.

*параметр инициализации (initialization parameter)*
Параметр, который инициализирует контекст, связанный с сервлетом.

*экземпляр (instance)*
Объект класса. В программах, написанных на языке Java™, экземпляр класса создается при помощи оператора new, за которым следует имя класса-типа.

*метод экземпляра (instance method)*
Любой метод, который вызван от имени экземпляра класса. Также называется просто метод. См. *метод класса*.

*переменная экземпляра (instance variable)*
Любой элемент данных, связанный с объектом. Каждый экземпляр класса имеет собственную копию переменной, определенной в классе. Также носит название поля. См. *переменная класса*.

*instanceof*
Ключевое слово языка программирования Java™, имеющее два аргумента и проверяющее, является ли тип первого аргумента преобразуемым к типу второго аргумента.

*int*
Ключевое слово языка программирования Java™, используемое для определения переменной целого типа (integer).

*interface*
Ключевое слово языка программирования Java™, используемое для определения набора методов и постоянных значений (класса специального вида). Интерфейс в дальнейшем может реализовываться классами, которые определяют этот интерфейс с ключевым словом implements.

*Интернет, международная компьютерная сеть (Internet)*
Огромная сеть, состоящая из миллионов компьютеров многих организаций и стран всего мира. Физически сеть Internet составлена из большого количества сетей, причем данные передаются при помощи единого набора протоколов.

*протокол сети Интернет (Internet Protocol - IP)*
Основной протокол Internet. Осуществляет ненадежную доставку индивидуальных пакетов от одного компьютера другому. Не дает никаких гарантий о том, будет ли доставлен пакет, сколько времени займет доставка, придут ли групповые пакеты в том порядке, в котором они были посланы. Протоколы, надстроенные над IP, добавляют понятия связности и надежности. См. также *TCP/IP*.

*интерпретатор (interpreter)*
Программа, которая поочередно декодирует и исполняет каждую инструкцию кода. Интерпретатор Java™ декодирует и выполняет код для виртуальной машины Java*. См. также *компилятор*, система *поддержки и выполнения программ*.

*независимый поставщик ПО (Independent Software Vendor - ISV)*
Фирма, разрабатывающая программное обеспечение для платформы, созданной другой организацией.

## J

*приложение J2EE (J2EE application)*
Любой составной модуль, поддерживающий функциональность J2EE. Это может быть один модуль или группа модулей, упакованных в .war-файл, содержащий дескриптор размещения J2EE-приложения. Приложения J2EE обычно проектируются так, что они распределены по нескольким вычислительным уровням.

*продукт J2EE (J2EE product)*
Продукт разработки, соответствующий спецификации платформы J2EE.

*поставщик продуктов J2EE (J2EE Product Provider)*
Производитель, поставляющий продукты J2EE.

*сервер J2EE (J2EE server)*
Исполняемая часть продукта J2EE. Сервер J2EE обеспечивает Web и/или EJB-контейнеры.

*JAIN*
См. *Java APIs for Integrated Networks™*.

*JAR-файлы (Java ARchive files (.jar))*
Файловый формат, используемый для группирования нескольких файлов в один.

*файловый формат JAR (JAR file format)*
Платформно-независимый файловый формат, соединяющий несколько файлов в один. Составные апплеты, написанные на языке Java™, и необходимые для них компоненты (.class-файлы, изображения, звуковые и другие файлы) могут быть упакованы в JAR-файл и затем загружены в браузер за одну HTTP-транзакцию. Данный формат также поддерживает сжатие файлов и цифровые подписи.

*Java™*
Торговая марка компании Sun для ряда технологий по созданию и безопасной работе программного обеспечения как в автономных, так и в сетевых средах.

*Java™ Application Environment - JAE*
Версия исходного кода программного обеспечения Java Development Kit (JDK™).

*Java APIs for Integrated Networks™ - JAIN*
Дает возможность быстрой разработки продуктов и сервисов, использующих сетевые взаимодействия, на платформе Java.

*платформа J2SE (Java™ 2 Platform, Standard Edition (J2SE platform))*
Ядро платформы технологии Java.

*платформа J2EE (Java™ 2 Platform, Enterprise Edition (J2EE platform))*
Среда для разработки корпоративных приложений. Платформа J2EE состоит из набора услуг, интерфейсов прикладного программирования (APIs) и протоколов, обеспечивающих функциональные возможности для разработки многоуровневых Web-приложений.

*Java™ 2 SDK, Enterprise Edition*
Реализация платформы J2EE компании Sun. Данная реализация включает описание работы платформы J2EE.

*JavaBeans™*
Модель переносимых платформонезависимых компонентов многократного использования.

*Java Blend™*
Продукт, позволяющий разработчику упростить разработку приложений баз данных при помощи отображения записей баз данных в объекты языка Java™ и Java-объектов - в базы данных.

*Java Card™ API*
API для систем с минимальным набором ресурсов, в частности для смарт-карт. Среда прикладных программных средств, соответствующая стандарту ISO 7816-4 для интеллектуальных карт.

*JavaCheck™* Инструмент для проверки соответствия приложений и апплетов языка Java.

*JavaChip™*
Процессор фирмы Sun, выполняющий байт-код виртуальной машины Java*. На компьютере с процессором JavaChip™ байт-код минует эмуляцию Java-машины на какой-либо платформе, исполняясь непосредственно аппаратным процессором.

*Java™ Compatibility Kit (JCK)*
Набор испытательных программ и инструментальных средств, используемых для проверки совместимости реализации платформы Java соответствующим спецификациям Java-платформ, а также эталонным реализациям программ Java.
TCK для Java 2 Standard Edition. См. *TCK*.

*интерфейс JDBC (Java Database Connectivity - JDBC™)*
Промышленный стандарт для независимого от базы данных взаимодействия Java™-платформы и широкого диапазона баз данных. JDBC™ определяет API для доступа к базам данных из Java-приложений.

*Java Developer Connection*
Служба, предназначенная для индивидуальных разработчиков, предоставляющая интерактивное обучение, скидки на продукты, специальные статьи, информацию об ошибках, а также возможности раннего доступа к новым продуктам.

*комплект разработчика для Java (Java Development Kit - JDK™)*
Среда программирования для написания апплетов и приложений в языке программирования Java.

*Java™ Electronic Commerce Framework*
Структурированная архитектура для разработки приложений электронной коммерции в языке программирования Java™.

*Java™ Enterprise API*
Данный API упрощает создание крупномасштабных приложений и приложений баз данных, которые совместно используют мультимедийные данные с другими приложениями в пределах организации или через Internet. В семействе Java™ Enterprise API разработано четыре интерфейса прикладного программирования.

*библиотека базовых классов Java (Java™ Foundation Classes - JFC)*
Дополнительные библиотеки классов графического пользовательского интерфейса (Graphical User Interface - GUI), добавляемые к абстрактному оконному интерфейсу (Abstract Window Toolkit - AWT).

*Java™ IDL*
Технология, обеспечивающая функциональную совместимость и способность к взаимодействию CORBA и J2EE-платформы. Эти возможности позволяют приложениям J2EE вызывать операции удаленных сетевых служб, используя OMG IDL и IIOP.

*Java™ Interface Definition Language (IDL)*
API, написанные на языке программирования Java, которые обеспечивают функциональную совместимость и способность к взаимодействию с CORBA (Common Object Request Broker Architecture).

*JavaMail™*
API для отправки и получения сообщений.

*Java™ Media APIs*
Набор интерфейсов прикладного программирования (APIs) для интеграции аудио и видео файлов, двумерных шрифтов, графики и изображений, а также трехмерных моделей и телефонии.

*Java™ Media Framework*
Ядро структуры поддерживает часы для синхронизации различных потоков (например, аудио и видео сигналов). Standard extension framework позволяет пользователям организовать потоки аудио и видео информации.

*Java™ Message Service - JMS*
Интерфейс прикладного программирования (API) для использования корпоративных систем передачи сообщений, таких как IBM MQ Series, TIBCO Rendezvous и т.д.

*Java Naming and Directory Interface™ (JNDI)*
Стандартный API к корпоративной службе каталогов.

*операционная система Java (JavaOS™)*
Операционная система, основанная на технологии Java™ и оптимизированная для работы на различных платформах. Операционная среда JavaOS™ обеспечивает работу Java-приложений непосредственно на аппаратных платформах, минуя ведущую операционную систему.

*JavaPlan™*
Инструмент для объектно-ориентированного проектирования и схематического изображения, написанный на языке программирования Java™.

*платформа Java™ (Java™ Platform)*
Состоит из языка Java, предназначенного для написания программ, набора интерфейсов прикладного программирования (APIs), библиотек классов, других программ, используемых при разработке и компиляции, программ проверки ошибок, а также виртуальной машины Java, которая загружает и выполняет файлы классов.
Кроме того, платформа Java подчиняется набору требований совместимости для гарантии непротиворечивых и совместимых реализаций. Реализации, выполняющие требования совместимости, могут приобретать квалификацию заданной марки совместимости фирмы Sun.
Java 2 - настоящее поколение платформы Java.

*издания платформы Java (Java™ Platform Editions)*
"Издание" платформы Java - это окончательная и согласованная версия платформы Java, которая обеспечивает функциональные возможности, необходимые в широком секторе рынка.
Издание составлено из двух видов API: (i) "основные пакеты", которые являются необходимыми для всех реализаций данного издания платформы; (ii) "дополнительные пакеты", которые доступны данному изданию платформы и могут поддерживаться совместимыми реализациями.
Существует три различных издания платформы Java:

- Java 2 Platform, Enterprise Edition:
  Издание платформы Java, используемое на предприятиях и предназначенное для разработки и развития многоуровневых приложений, управляемых центральным сервером.
- Java 2 Platform, Standard Edition:
  Издание платформы Java, предназначенное для разработки, развития и управления межплатформенными универсальными приложениями.
- Java 2 Platform, Micro Edition:
  Издание платформы Java, предназначенное для малых автономных потребителей, с целью разработки, развития и управления приложений, варьирующихся от смарт-карт до обычных вычислительных устройств.

*Java™ Remote Method Invocation (RMI)*
Распределенная объектная модель для Java-программы, в которой методы и удаленные объекты, написанные на языке программирования Java, могут быть вызваны другой виртуальной машиной Java, возможно, расположенной на другом компьютере.

*Java™ Runtime Environment (JRE)*
Подмножество комплекта разработчика Java (Java Developer Kit - JDK™) для конечных пользователей и разработчиков, которые хотят самостоятельно настроить оперативные средства управления работой программы (run-time environment). Оперативные средства включают в себя виртуальную машину Java*, классы ядра Java и вспомогательные файлы.

*JavaSafe™*
Инструмент для отслеживания и управления изменениями исходного файла, написанного на языке программирования Java.

*язык сценариев JavaScript (JavaScript™)*
Язык сценариев, который используется как браузерами, так и Web-серверами. Подобно всем языкам сценариев, предназначен главным образом для интеграции компонент и пользовательского ввода.

*Java Server Pages™ (JSP)*
Расширяемая Web-технология, использующая данные шаблона, заказные элементы, языки сценариев и серверные объекты Java для предоставления клиенту динамического содержания. Обычно, данные шаблона - это HTML или XML элементы, а клиент - это Web-браузер.

*JAX*
JAX относится к набору интерфейсов прикладного программирования (APIs), который предназначен для управления различными операциями, включающими данные, определенные посредством XML. Сюда входят такие операции как синтаксический анализ XML, регистрация в репозиториях ebXML или UDDI, обмен сообщениями между приложениями, привязка данных и удаленный вызов процедур.

*действие Java Server Pages™ (JSP) (JSP action)*
JSP элемент, который может действовать на неявные объекты и другие серверные объекты или определять новые переменные создания сценария. Действия придерживаются синтаксиса XML для элементов с начальным тэгом, телом и конечным тэгом; если тело пусто, может использоваться синтаксис пустого тэга. Тэг должен использовать префикс.

*стандартное действие Java Server Pages™ (JSP action, standard)*
Действие, которое определено спецификацией JSP и всегда доступно JSP-файлу без импортирования.

*специальное действие Java Server Pages™ (JSP action, custom)*
Действие, описанное с помощью тэгов и набора классов Java, включенных в страницу JSP с помощью тэговых дескрипторов.

*приложение JSP (JSP application)*
Автономное Web- приложение, написанное с использованием технологии Java Server Pages и включающее JSP файлы, сервлеты, HTML файлы, изображения, апплеты и компоненты JavaBeans.

*контейнер JSP (JSP container)*
Контейнер, предоставляющий те же услуги, что и сервлет-контейнер, а также механизм интерпретации и переработки JSP страниц в сервлеты.

*распределенный контейнер JSP (JSP container, distributed)*
JSP контейнер, который может запускать Web-приложения, помеченные как распределенные и выполняемые одновременно на нескольких виртуальных машинах Java. При этом виртуальные машины могут быть запущены, как на одном, так и на разных компьютерах.

*объявление JSP (JSP declaration)*
Элемент сценариев JSP, который объявляет методы и переменные в JSP файле.

*директива JSP (JSP directive)*
Элемент JSP, который дает команды JSP контейнеру и интерпретируется во время трансляции.

*элемент JSP (JSP element)*
Часть JSP страницы, которая распознается JSP транслятором. Элемент JSP может быть директивой, действием или элементом сценария.

*выражение JSP (JSP expression)*
Элемент сценария, который содержит допустимое выражение языка сценариев, вычисленное, преобразованное в строку и помещенное в неявный выходной объект.

*файл JSP (JSP file)*
Файл с расширением .jsp, который создается разработчиком при помощи стандартных тэгов HTML, основных JSP тэгов и инструкций языка сценариев, для отображения динамических страниц в Web-браузере.

*страница JSP (JSP page)*
Текстовый документ, использующий фиксированные шаблонные данные и элементы JSP. Описывает, как обработать запрос.

*элемент сценария JSP (JSP scripting element)*
Объявление JSP, скриптлет или выражение, синтаксис которого определен спецификацией JSP и содержимое которого написано согласно языку сценариев, используемому в странице JSP. Спецификация JSP описывает синтаксис и семантику для того случая, когда атрибут языка страницы - java.

*скриптлет JSP (JSP scriptlet)*
Элемент сценария JSP, содержащий любой фрагмент кода, допустимый в том языке сценариев, который используется на JSP странице. Спецификация JSP определяет, что является допустимым скриптлетом для того случая, когда атрибут языка страницы - java.

*тэг JSP (JSP tag)*
Текст между левой и правой угловыми скобками, который используется в файлах JSP, как часть элемента JSP. В отличие от данных, тэг является элементом разметки документа, т.к. он выделен угловыми скобками.

*библиотека тэгов JSP (JSP tag library)*
Совокупность тэгов, описанных посредством библиотечных дескрипторов и классов Java. Библиотека тэгов JSP может импортироваться в любой JSP файл и использоваться с различными языками сценариев.

*Java Studio™*
Инструментальный комплекс для построения Java-программ из готовых компонент, основанный на визуальном стиле проектирования и предназначенный для непрограммистов.

*технологии Java (Java™ Technologies)*
Ряд технологий по созданию и безопасной работе программного обеспечения, как в автономных, так и в сетевых средах.

*Java™ Transaction API (JTA)*
API, позволяющий приложениям и сервлетам J2EE иметь доступ к транзакциям.

*Java™ Transaction Service (JTS)*
Определяет реализацию менеджера транзакций, который поддерживает JTA и осуществляет Java отображение спецификации OMG Object Transaction Service (OTS) 1.1 на уровень ниже API.

*виртуальная машина Java* (Java™ virtual machine - JVM*)*
Программный "механизм выполнения", который безопасно выполняет байт-коды файлов классов Java на микропроцессоре (компьютера или другого электронного устройства).
Механизм выполнения Java HotSpot - это высокоэффективный механизм для среды выполнения Java программ, который представляет собой адаптивный компилятор, динамически оптимизирующий работу приложений.
Виртуальная машина KJava - малогабаритная, высоко оптимизированная основа среды выполнения программ (runtime environment) в Java 2 Platform, Micro Edition. KJava разработана на основе виртуальной машины Java и предназначена для малых устройств связи. Ее размер составляет от 30 до 128 Кб, в зависимости от функциональных возможностей устройства.
Виртуальная машина Java Card - малогабаритная, высоко оптимизированная основа среды выполнения программ (runtime environment) в Java 2 Platform, Micro Edition. KJava разработана на основе виртуальной машины Java и предназначена для смарт-карт и других устройств с ограниченной памятью (порядка 24Кб ПЗУ, 16Кб EEPROM и 512б ОЗУ).

*Web-сервер Java (Java Web Server™)*
Удобное, открытое, легко администрируемое, безопасное, платформонезависимое решение для ускорения и простоты разработки Internet/Intranet Web-страниц. Обеспечивает немедленное повышение производительности для трудоемких, полнофункциональных серверных приложений Java.

*Java Workshop™*
Законченный набор инструментов, интегрированный в единую среду программирования с использованием Java технологий. Программное обеспечение Java Workshop использует модульную структуру, которая позволяет легко подключать новые инструментальные средства.

*Java™ wallet*
Пользовательский интерфейс, основанный на Java™ Electronic Commerce Framework, предназначенный для осуществления интерактивных покупок, передачи данных и выполнения административных функций.

*JavaSpaces™*
Технология, содержащая механизмы распределенной работы и обмена данными при программировании на языке Java™.

*JavaSoft™*
Ранее филиал компании Sun Microsystems, в настоящее время известный как подразделение Sun Microsystems Java Software.

*JDBC™*
См. *интерфейс JDBC (Java Database Connectivity - JDBC™)*.

*JDK™*
См. *комплект разработчика для Java (Java Development Kit - JDK™)*.

*JFC*
См. *библиотека базовых классов Java (Java™ Foundation Classes - JFC)*.

*технология Jini (Jini™ Technology)*
Набор интерфейсов прикладного программирования (APIs) Java, которые могут быть включены в дополнительный пакет к любому изданию платформы Java 2. Jini позволяет обеспечивать совместную работу в сети различных устройств и служб, а также устраняет необходимость системного или сетевого административного вмешательства пользователя. В настоящее время технология Jini является дополнительным пакетом, доступным любой Java платформе.

*Java™ Management API - JMAPI*
Совокупность Java классов и интерфейсов, позволяющих разработчику создавать системные, сетевые и служебные приложения.

*JMS*
См. *Java™ Message Service*.

*интерфейс JNDI*
См. *Java Naming and Directory Interface™*.

*стандарт JPEG (Joint Photographic Experts Group - JPEG)*
Стандарт сжатия файлов изображений, установленный группой экспертов по машинной обработке фотографических изображений. Сильное сжатие достигается ценой внесения искажений в изображение, которые почти всегда являются незаметными.

*JRE*
См. *Java™ Runtime Environment*.

*компилятор JIT (Just-in-time (JIT) Compiler)*
Компилятор, динамически ("на лету") преобразующий байт-код в объектный код целевой платформы. Его применение приводит к значительному ускорению выполнения Java-программы.

*JVM*
См. *виртуальная машина Java**.

## K

*ключевое слово (keyword)*
Слово, которое зарезервировано языком программирования Java™, и поэтому не может быть именем переменной или метода.

## L

*лексический (lexical)*
Соответствующий уровню лексем (идентификаторов, чисел, изображений строк и т.д.) языка программирования. Лексический анализ - фаза компилятора, на которой последовательность символов файла исходного текста преобразуется в последовательность лексем.

*компоновщик (linker)*
Модуль, который формирует запускаемую, законченную программу из составных модулей машинного кода. Компоновщик Java™ создает работоспособную программу из откомпилированных классов. См. также *компилятор*, *интерпретатор*, *система поддержки исполнения программ*.

*литерал (literal)*
Основное представление любого целого, символьного значения или значения с плавающей запятой. Например, 3.0 - литерал с плавающей запятой двойной точности, "a" - символьный литерал.

*локальная переменная (local variable)*
Элемент данных, известный в пределах блока, но недоступный вне блока. Например, любая переменная, определенная внутри метода, является локальной и не может использоваться вне этого метода.

*long*
Ключевое слово языка программирования Java™, используемое для определения переменных типа long.

## M

*элемент (member)*
Поле или метод класса. Пока не оговорено обратное, член не является статическим.

*метод (method)*
Функция, определенная в классе. См. также *метод экземпляра*, *метод класса*. Пока не оговорено обратное, метод не является статическим.

*разрешение на вызов метода (method permission)*
Разрешение на вызов указанной группы методов "домашнего" или удаленного интерфейса корпоративного компонента (enterprise bean).

*модуль (module)*
Программный модуль, который состоит из одного или более компонентов J2EE, принадлежащих контейнеру одного типа, и из дескриптора размещения этого типа. Существует три типа модулей: EJB, Web и клиентские приложения. Модули можно использовать автономно или собирать в приложения.

*Mosaic*
Программа с несложным графическим интерфейсом (GUI), предназначенная для простого

доступа к данным, хранящимся в Internet. Данные могут быть обычными файлами или гипертекстовыми документами. Mosaic была разработана в NCSA.

*многопоточный (multithreaded)*
Описывает программу, спроектированную и реализованную в виде совокупности параллельных потоков управления (threads), синхронизированных между собой по общим ресурсам и событиям. См. также *поток*.

*взаимная аутентификация (mutual authentication)*
Процесс, при котором клиент использует сертификат с открытым ключом для установления своей идентичности и поддерживает защиту своего контекста.

## N

*native*
Ключевое слово языка программирования Java™, используемое в объявлении метода для указания на то, что метод реализован не в файле Java, а на другом языке.

*Национальный центр по приложениям для суперкомпьютеров (National Center for Supercomputer Applications - NCSA)*
Исследовательский центр, в котором была разработана программа Mosaic.

*new*
Ключевое слово языка программирования Java™, используемое для создания нового экземпляра класса.

*null*
Тип null имеет единственное значение - "пустую" ссылку, представленную литералом null, который сформирован из символов ASCII. Литерал null всегда имеет тип null.

## O

*объект (object)*
Основной компоновочный блок объектно-ориентированных программ. Каждый объект программного модуля состоит из данных (переменные экземпляра) и функциональных возможностей (методы экземпляра). См. также *класс*.

*объектно-ориентированное проектирование (object-oriented design)*
Метод проектирования программного обеспечения, позволяющий моделировать абстрактные или реальные объекты при помощи классов и объектов.

*восьмеричный (octal)*
Система счисления с основанием 8. Знаки 0-7 представляют цифры. В программе, написанной на языке Java™, восьмеричные числа должны начинаться с символа 0. См. также *шестнадцатеричный*.

*открытое сетевое окружение (Open Net Environment - ONE)*
Сетевая среда, которая была разработана компанией Sun Microsystems и поддержана многими

ведущими производителями. Описывает всестороннюю архитектуру для создания, сборки и использования сетевых услуг. ONE платформонезависима и полностью основана на открытых стандартах. Sun ONE - конкретная реализация данной архитектуры, разработанная компаниями Sun и iPlanet.

*дополнительный пакет (Optional Package)*
Набор API в издании платформы Java, который может быть доступным или поддерживаться в совместимой реализации.
По мере необходимости, дополнительные пакеты могут становиться необходимыми в издании.

*посредник запросов к объектам (Object Request Broker - ORB)*
Библиотека, позволяющая объектам CORBA определять местонахождение и устанавливать связь друг с другом.

*принцип (OS principal)*
Свойство (principal), присущее той операционной системе, на которой запущена платформа J2EE.

*Object Transaction Service (OTS)*
Интерфейсы, позволяющие объектам CORBA участвовать в транзакциях.

*перегрузка (операций) (overloading)*
Использование одного идентификатора для ссылки на разные элементы в одной области действия. В языке программирования Java™ можно перегружать методы, однако, нельзя перегружать переменные или операторы.

*замещение (overriding)*
Означает другую реализацию метода в подклассе класса, первоначально определившего метод.

## P

*пакет (package)*
Группа типов. Пакеты объявляются при помощи ключевого слова package.

*пассивация (passivation)*
Процесс передачи корпоративного компонента (enterprise bean) из памяти во вторичное устройство хранения данных. См. также *активация*.

*одноранговые (peer)*
В организации сетей любые функциональные единицы, находящиеся на одном уровне.

*persistence*
Протокол передачи состояния сущности (entity bean) между переменными ее экземпляра и базой данных.

*PersonalJava™*
Среда выполнения Java для сетевых приложений на персональных устройствах потребителя (домашнего, мобильного или офисного использования).

*пиксель (pixel)*
Элемент площади изображения, например, экрана монитора или напечатанной страницы. Каждый пиксель является индивидуально доступным.

*Portable Object Adapter - POA*
Стандарт CORBA для создания серверных приложений, которые переносимы между различными посредниками запросов к объектам (ORBs).

*интерфейс переносимой операционной системы (Portable Operating System Interface - POSIX)*
Стандарт, который определяет языковой интерфейс между операционной системой UNIX и прикладными программами посредством минимального набора поддерживаемых функций.

*первичный ключ (primary key)*
Объект, уникально идентифицирующий сущность (entity bean).

*принцип (principal)*
Отличительная черта, присвоенная сущности в результате аутентификации.

*private*
Ключевое слово языка программирования Java™, используемое при объявлении метода или переменной. Указывает на то, что к методу или переменной класса не могут обращаться элементы других классов.

*привилегия (privilege)*
Атрибут защиты, который не имеет свойства уникальности и который может быть использован несколькими principals. Пример привилегии - группа.

*процесс (process)*
Виртуальное адресное пространство, содержащее один или более потоков.

*свойство (property)*
Характеристика объекта, которую может устанавливать пользователь. Например, цвет окна.

*профили (Profiles)*
Наборы Java API, которые служат дополнением к изданиям платформ Java и обеспечивают дополнительные возможности. Профили также могут включать другие определенные профили. Реализация профиля нуждается в издании Java Platform для создания законченной разработки и среды ее использования на целевом вертикальном рынке. Каждый профиль подчиняется связанному набору требований совместимости.
Профили могут использоваться одним или несколькими изданиями.
Примеры профилей в Java 2 Platform, Micro Edition: PersonalJava™, Java Card™.

*protected*
Ключевое слово языка программирования Java™, используемое при объявлении метода или

переменной. Указывает на то, что к методу или переменной класса могут обращаться другие элементы данного класса, его подклассов или классов из того же пакета.

*private*
Ключевое слово языка программирования Java™, используемое при объявлении метода или переменной. Указывает на то, что к методу или переменной класса могут обращаться элементы других классов.

## Q

## R

*растр (raster)*
Двумерная прямоугольная сетка пикселей.

*область (realm)*
См. *область политики безопасности*. Строка, проходящая базовую аутентификацию, как часть HTTP-запроса, которая определяет пространство защиты. Защищенные ресурсы на сервере могут находиться в разных пространствах защиты, каждое со своей схемой аутентификации и/или базой данных авторизации.

*реентерабельный корпоративный компонент (re-entrant enterprise bean)*
Корпоративный компонент, который может обрабатывать несколько одновременных или вложенных вызовов, не пересекающихся друг с другом.

*ссылка (reference)*
Элемент данных, значение которого является адресом памяти.

*ссылочная реализация (Reference Implementation - RI)*
Прототип реализации спецификации Java-технологии. Является неотъемлемой частью любой Java-технологии и служит доказательством того, что данная Java-технология может быть реализована на практике.

*удаленный интерфейс (remote interface)*
Один из двух интерфейсов корпоративной компоненты (enterprise bean). Удаленный интерфейс определяет бизнес-методы, вызываемые клиентом.

*метод удаления (remove method)*
Метод, определенный в "домашнем" интерфейсе и вызываемый клиентом для уничтожения корпоративной компоненты (enterprise bean).

*адаптер ресурсов (resource adapter)*
Программный драйвер системного уровня, используемый EJB-контейнером или клиентским приложением для связи с EIS. Адаптер ресурсов специфичен для каждой EIS. Он представляет собой библиотеку, и используется в пределах адресного пространства сервера или клиента, которые используют адаптер. Адаптер ресурсов подключается к контейнеру. Прикладные компоненты, содержащиеся в контейнере, используют API (представленный адаптером) или инструментально созданные высокоуровневые абстракции для доступа к

низкоуровневой EIS. Адаптер ресурса и EJB-контейнер взаимодействуют для того, чтобы обеспечить низкоуровневые механизмы - транзакции, безопасность, организация связного пула - для связи с EIS.

*менеджер ресурсов (resource manager)*
Обеспечивает клиенту доступ к набору общедоступных ресурсов. Менеджер ресурсов участвует в транзакциях, которые управляются и координируются менеджером транзакций. Менеджер ресурсов обычно находится в другом адресном пространстве или на другой машине. Замечание: к EIS обращаются, как к менеджеру ресурсов, когда она упомянута в контексте управления ресурсами и транзакциями.

*соединение с менеджером ресурсов (resource manager connection)*
Объект, представляющий собой сеанс связи с менеджером ресурсов.

*мастер соединения с менеджером ресурсов (resource manager connection factory)*
Объект, используемый для создания сеанса связи с менеджером ресурсов.

*return*
Ключевое слово языка программирования Java™, используемое для окончания выполнения метода. За ним может следовать значение, возвращаемое методом.

*запрос на улучшение (Request for Enhancement - RFE)*

*RMI*
См. *Java™ Remote Method Invocation*.

*роль разработки (role (development))*
Функция, выполняемая человеком в стадии развития приложения, разрабатываемого с помощью технологии J2EE. Примеры ролей: поставщик программных компонент (Application Component Provider), компоновщик приложения (application assembler), "разместитель" (deployer), поставщик платформы J2EE (J2EE Platform Provider), поставщик EJB-контейнера (EJB Container Provider), поставщик EJB-сервера (EJB Server Provider), поставщик Web-контейнера (Web Container Provider), поставщик Web-сервера (Web Server Provider), поставщик инструментов (Tool Provider), системный администратор (System Administrator).

*роль безопасности (role (security))*
Абстрактная логическая группировка пользователей, осуществляемая компоновщиком приложения. Когда приложение установлено, роли ставятся в соответствие идентичностям безопасности, таким как principals или группы.

*распределение ролей (role mapping)*
Процесс объединения групп, распознанных контейнером, в роли безопасности, которые определены в дескрипторе размещения. Роли безопасности должны быть составлены "разместителем" перед установкой контейнера на сервер.

*откат (rollback)*
Момент транзакции, когда все изменения, вносимые в базу данных, отменяются.

*корень (root)*
Элемент иерархии, из которого происходят все элементы. Ни один элемент не может находиться в иерархии выше "корня". См. также *иерархия*, *класс*, *пакет*.

*вызов удаленной процедуры (Remote Procedure Call - RPC)*
Протокол, позволяющий приложениям вызывать процедуры, физически расположенные в другой части сети.

*система поддержки исполнения программ (runtime system)*
Программная среда, в которой могут работать программы, откомпилированные для виртуальной машины Java™*. Система поддержки исполнения включает весь код, необходимый для того, чтобы загружать программы, написанные на Java, динамически связывать "родные" методы, управлять памятью, обрабатывать исключительные ситуации, а также реализацию виртуальной машины Java.

## S

*простой API для XML (Simple API for XML - SAX)*
Управляемый событиями, последовательный механизм доступа к XML документам.

*Sandbox*
Содержит множество взаимодействующих системных компонент, начиная от менеджеров безопасности, которые выполняются как часть приложения, и заканчивая мерами безопасности, встроенными непосредственно в виртуальную машину Java* и язык Java. Sandbox гарантирует, что ненадежные и, возможно, злонамеренные приложения не смогут получить доступ к системным ресурсам.

*область действия (scope)*
Характеристика идентификатора, определяющая, где может использоваться данный идентификатор. Большинство идентификаторов в языке программирования Java имеют либо локальную область действия, либо область действия - класс. Для переменных экземпляров и классов, а также методов областью действия является класс; они могут использоваться вне класса и его подклассов, только если перед ними ставится имя экземпляра класса или (в случае переменных и методов класса) имя самого класса. Все другие переменные объявляются внутри методов и имеют локальную область действия; они могут использоваться только внутри блока.

*протокол безопасных соединений (Secure Socket Layer - SSL)*
Протокол, который позволяет шифровать сообщения, передаваемые между Web-браузером и сервером в целях безопасности.

*атрибуты безопасности (security attributes)*
Набор свойств, связанных с principal. Атрибуты безопасности могут быть связаны с principal посредством протокола аутентификации и/или поставщиком продуктов J2EE (J2EE Product Provider).

*ограничения безопасности (security constraint)*
Декларативный способ аннотирования необходимой защиты содержимого сети. Ограничение

безопасности состоит из совокупности Web-ресурсов, ограничения авторизации, ограничения пользовательских данных.

*контекст безопасности (security context)*
Объект, включающий разделяемые свойства для описания безопасности сущностей.

*право безопасности (security permission)*
Механизм, определенный J2SE и используемый платформой J2EE для определения программных ограничений, наложенных на поставщиков программных компонент (Application Component Providers).

*набор прав безопасности (security permission set)*
Минимальный набор прав безопасности, обеспечиваемый поставщиком программных компонент (Application Component Provider) для выполнения каждого типа компонент.

*область политики безопасности (security policy domain)*
Область, в переделах которой определена и приведена в исполнение администратором политика безопасности. Область политики безопасности имеет следующие характеристики:

1. обладает набором пользователей (или principals);
2. использует хорошо определенный протокол аутентификации пользователей (или principals);
3. может иметь группы для упрощения настройки политики безопасности.

*security role*
См. *роль безопасности (role(security))*.

*область технологии безопасности (security technology domain)*
Область, в пределах которой для осуществления политики безопасности используется один и тот же механизм. В пределах единой области технологии безопасности может существовать несколько областей политики безопасности.

*server principal*
Принцип ОС, в соответствии с которым работает сервер.

*сервлет (servlet)*
Java программа, которая расширяет функциональные возможности Web-сервера, динамически генерируя содержание и взаимодействуя с Web-клиентами при помощи принципа запрос-ответ.

*контейнер сервлета (servlet container)*
Контейнер, обеспечивающий сетевые службы, при помощи которых посылаются запросы и ответы, декодируются запросы и форматируются ответы. Все контейнеры сервлетов должны поддерживать HTTP-протокол, но могут также поддерживать дополнительные протоколы, например, HTTPS.

*распределенный контейнер сервлета (servlet container, distributed)*
Контейнер сервлета, запускающий Web-приложения, которые помечены как распределенные

и выполняются на нескольких виртуальных машинах Java. При этом виртуальные машины могут быть запущены, как на одном, так и на разных компьютерах.

*контекст сервлета (servlet context)*
Объект, содержащий представление (вид) Web-приложения, в котором запущен сервлет. Используя контекст, сервлет может вести журнал событий, получать URL-ссылки на ресурсы, а также устанавливать и хранить атрибуты, которые могут использоваться другими сервлетами в приложении.

*отображение сервлета (servlet mapping)*
Определяет связь между структурой URL и сервлетом. Используется для отображения запросов в сервлеты. Если контейнер, обрабатывающий запрос, является JSP-контейнером, то неявно отображается URL, содержащий расширение .jsp.

*сессия (session)*
Объект, используемый сервлетом, для прослеживания взаимодействий пользователя с Web-приложением при помощи множества HTTP-запросов.

*сессионный компонент (session bean)*
Корпоративный компонент (enterprise bean), который создается клиентом и обычно существует только в течение одной клиент-серверной сессии. Сессионный компонент выполняет для клиента вычислительные операции и организует доступ к базе данных. В случае сбоя системы сессионный компонент не восстанавливается. Объекты сессионных компонент могут не менять своего состояния или поддерживать диалоговое состояние в процессе выполнения методов и транзакций. Если объект поддерживает состояние, то EJB-контейнер управляет этим состоянием, если объект должен быть удален из памяти. Однако объекты сессионного компонента должны управлять собственными хранимыми данными.

*short*
Ключевое слово языка программирования Java™, используемое для определения переменных типа short.

*одинарная точность (single precision)*
В спецификации языка Java описывает число с плавающей запятой, занимающее 32 бита данных. См. также *двойная точность*.

*стандартный язык обобщенной разметки (Standardized Generalized Markup Language - SGML)*
Стандарт ISO/ANSI/ECMA для определения структуры и управления содержимым любого электронного документа.

*Smart Web Services*
Расширяют основную концепцию сетевых услуг, добавляя пользовательский контекст, и способны модифицировать свои действия, чтобы следить за изменениями текущего состояния клиента. Сюда входят классические дескрипторы "кто, что, когда, где, почему", которые объединяются для определения пользовательского контекста в данный момент.

*The Simple Object Access Protocol - SOAP*
Использует основанное на XML структурирование данных и HTTP для того, чтобы

определить стандартизованные методы для вызова методов объектов, распределенных в различных средах по всей Internet.

*язык структурированных запросов (Structured Query Language - SQL)*
Стандартизованный язык реляционных баз данных, предназначенный для определения объектов баз данных и манипулирования данными.

*сессионный компонент с диалоговым состоянием (stateful session bean)*
Сессионный компонент с диалоговым состоянием.

*сессионный компонент, не имеющий состояния (stateless session bean)*
Сессионный компонент, не имеющий состояния. Все экземпляры такого компонента идентичны.

*static*
Ключевое слово языка программирования Java™, используемое для определения переменной класса (типа). Классы обеспечивают только одну копию таких переменных, не зависимо от того, сколько экземпляров класса было создано. Слово static также может использоваться при определении метода класса. Такие статические методы вызываются от имени не экземпляра объекта, а его типа (класса), и могут, в свою очередь, оперировать только статическими переменными.

*статическое поле (static field)*
Другое название переменной класса.

*статический метод (static method)*
Другое название метода класса.

*поток (stream)*
Последовательность байтов данных, пересылаемых от отправителя к получателю. Существует две основные категории потоков, поэтому пакет java.io включает два абстрактных класса (InputStream и OutputStream).

*подмассив (subarray)*
Массив, содержащийся в другом массиве.

*подкласс (subclass)*
Класс, который "произведен" из другого класса. См. также *суперкласс*, *супертип*.

*подтип (subtype)*
Если тип X "расширяет" или реализует тип Y, то X - подтип типа Y. См. также *супертип*.

*суперкласс (superclass)*
Класс, из которого "произведены" другие классы. См. также *подкласс*, *подтип*.

*super*
Ключевое слово языка программирования Java™, используемое для доступа к членам класса, наследуемого классом, из которого производится вызов.

*супертип (supertype)*
Все интерфейсы или классы, расширяемые или реализуемые данным типом. См. также *подтип*, *суперкласс*.

*switch*
Ключевое слово языка программирования Java™, используемое для определения переменной, которая в дальнейшем может быть использована ключевым словом case для выполнения блока инструкций.

*Swing*
Кодовое название совокупности графических компонентов, которые выполняются на любой платформе, поддерживающей виртуальную машину Java™. Данные компоненты могут обеспечивать большие функциональные возможности, вследствие того, что они целиком написаны на языке Java™. См. также *AWT*.

*synchronized*
Ключевое слово языка программирования Java™, которое при применении к методу или блоку кода, гарантирует, что данный код будет выполняться не более чем одним потоком одновременно.

*системный администратор (System Administrator)*
Человек, ответственный за конфигурирование и администрирование компьютеров компании, сетей и систем программного обеспечения.

## T

*TCP/IP (TCP/IP)*
Протокол управления передачей данных, основанный на IP. Internet-протокол, который обеспечивает надежную транспортировку потоков данных между компьютерами. См. также *IP*.

*Испытатель Совместимости Технологии (TCK) (Technology Compatibility Kit - TCK)*
Набор тестов, инструментов, сопутствующих утилит и документации, которые позволяют реализатору Спецификации той или иной технологии определить, соответствует ли его реализация заданной Спецификации. Любая Java-технология состоит из Спецификации, TCK и ссылочной реализации (reference implementation).

*"тонкий" клиент (Thin Client)*
Система, использующая очень упрощенную операционную систему, не требующая локального системного администрирования и выполняющая приложения, загруженные из сети.

*this*
Ключевое слово языка Java™, которое может использоваться для ссылки на экземпляр класса, в котором используется эта ссылка. Слово this может использоваться для доступа, как к полям, так и к методам класса.

*процесс (поток) (thread)*
Основная единица выполнения программы. Процесс может иметь несколько потоков, работающих одновременно и выполняющих различные задачи, такие как, ожидание события или исполнение трудоемкой по времени работы, окончание которой не требуется для дальнейшего выполнения программы. После выполнения потоком своей работы он приостанавливается или уничтожается. См. также *процесс*.

*throw*
Ключевое слово языка Java ™, которое позволяет пользователю сгенерировать исключительную ситуацию или любой класс, реализующий "throwable" интерфейс (т.е. интерфейс, позволяющий классу генерировать прерывания).

*throws*
Ключевое слово языка Java™, используемое в описании метода и определяющее, какие исключительные ситуации не обрабатываются внутри метода, а передаются на следующий, более высокий уровень программы.

*поставщик инструментов (Tool Provider)*
Организация или поставщик программного обеспечения, предоставляющий инструменты, используемые для разработки, компоновки и внедрения J2EE приложений.

*транзакция (transaction)*
Минимальная единица работы по изменению данных. Транзакция включает в себя одну или несколько программных инструкций, которые могут выполниться только все вместе. Если хотя бы одна из инструкций не выполняется, происходит возврат в исходное состояние (откат). Транзакции разрешают одновременный доступ нескольким пользователям к одним и тем же данным.

*атрибут транзакции (transaction attribute)*
Значение, определяемое в дескрипторе внедряемой корпоративной компоненты (enterprise bean), которое используется в контейнере EJB для управления областью действия транзакции при вызове методов компоненты. Атрибут транзакции может принимать следующие значения: Required, RequiresNew, Supports, NotSupported, Mandatory, Never ("требуется", "требуется для новых", "поддерживается", "не поддерживается", "принудительно", "никогда").

*уровень изоляции транзакции (transaction isolation level)*
Степень видимости промежуточного состояния модифицируемых транзакцией данных для других параллельных транзакций и данных, модифицируемых другими транзакциями, для данной транзакции.

*менеджер транзакции (transaction manager)*
Обеспечивает сервисы и управляющие функции, требуемые для поддержки разграничения транзакции, управления ресурсами транзакции, синхронизации и прохождения содержания транзакции.

*transient*
Ключевое слово языка Java™, которое определяет, что поле не является частью сериализуемой формы объекта. Когда объект сериализуется, значения его transient полей не

включаются в представление сериализации, в то время как значения не transient полей включаются.

*try*
Ключевое слово языка Java™, определяющее блок операторов, которые могут генерировать исключительные ситуации Java. При возникновении исключительной ситуации необязательный catch блок может обработать определенные исключительные ситуации, сгенерированные внутри блока try. Также необязательный блок finally будет выполнен независимо от того, генерировалась исключительная ситуация или нет.

*type*
Класс или интерфейс.

## U

*UDDI*
Проект "Универсальное обнаружение и интеграция описаний" (The Universal Description Discovery and Integration - UDDI) обеспечивает глобальный, публичный, основанный на XML, онлайновый бизнес-регистр, в котором пользователи регистрируют и представляют свои web службы. UDDI представляет собой Internet-версию желтых страниц в справочнике телефонов.

*уникод (Unicode)*
16-битная кодовая таблица, определенная ISO 10646. См. также *ASCII*. Все исходные коды программной среды Java™ написаны в Unicode.

*унифицированный идентификатор ресурсов (Uniform Resource Identifier - URI)*
Компактная строка символов для идентификации абстрактного или физического ресурса. URI может быть или URL или URN. URL и URN представляют собой конкретные сущности, которые действительно существуют; URI является абстрактным суперклассом.

*унифицированный указатель информационного ресурса (Uniform Resource Locator - URL)*
Стандарт для записи текстовой ссылки на произвольные данные в WWW. URL выглядит следующим образом: "протокол://хост/локальная_информация", где протокол определяет конкретный протокол, используемый для доступа к объекту (например HTTP или FTP), хост определяет Internet-имя хоста, на котором объект находится, и локальная_информация - строка (часто имя файла), передаваемая обработчику протокола на удаленном хосте.

*URL path*
URL, передаваемый запросом HTTP для вызова сервлета. URL состоит из Context Path + Servlet Path + PathInfo, где Context Path является префиксом пути, ассоциируемым с контекстом сервлета. Этот сервлет является частью контекста. Если этот контекст является контекстом по умолчанию, находящимся в основном пространстве имен URL Web-сервера, префикс пути является пустой строкой. В противном случае префикс пути начинается с символа /, но не заканчивается символом /. Servlet Path - это участок пути, который прямо соответствует отображению, активизировавшему данный запрос. Этот путь начинается с символа /. PathInfo - это часть пути запроса, не являющаяся частью Context Path или Servlet Path.

*унифицированное имя ресурса (URN)*
Уникальный идентификатор, который идентифицирует сущность, но не показывает, где она расположена. Система может использовать URN для локального поиска сущности перед попыткой найти ее в Web. Этот идентификатор также позволяет изменение Web-ссылки при сохранении возможности нахождения сущности.

*ограничитель данных пользователя (user data constraint)*
Определяет, как должны быть защищены данные между клиентом и контейнером. Защита может быть предотвращением несанкционированного изменения данных либо предотвращением перехвата данных.

## V

*переменная (variable)*
Элемент данных, имеющий идентифицирующее его имя. Каждая переменная имеет тип (например, int или Object) и область видимости. См. также *переменная класса*, *переменная экземпляра*, *локальная переменная*.

*виртуальная машина (virtual machine)*
Абстрактная спецификация для вычислительного устройства, которое может быть реализовано различным образом - программно или аппаратно. Вы компилируете последовательность команд виртуальной машины точно так же, как будто вы компилируете последовательность команд микропроцессора. Виртуальная машина Java™ состоит из байтовых команд, набора регистров, стека, "кучи" со сборкой "мусора" и областью для хранения методов.

*словарь (vocabulary)*
Обычно компьютерные программы пишутся и компилируются в машинные коды, которые прямо зависят от операционной системы, управляющей микропроцессором в компьютере. Java-платформа смягчает эту зависимость, обеспечивая модель, по которой программы пишутся, компилируются и могут быть переданы по сети и выполнены в любом месте, где присутствует полностью совместимая виртуальная машина.
Эта модель обеспечивает дополнительное преимущество в повышении безопасности. Во-первых, потому что программы могут быть проверены виртуальной машиной после того, как они были переданы по сети. И, во-вторых, потому что виртуальная машина может запустить программу в защищенном пространстве, которое предотвратит определенные разрушающие действия.
Разработчики программного обеспечения выбирают платформу Java потому, что это уменьшает стоимость и время написания и поддержки программного кода. Им больше не требуется переписывать программы для функционирования на различных компьютерах с различными операционными системами и микропроцессорами. Внедрение приложений Java технологии в компаниях и организациях выгодно, потому что это минимизирует стоимость покупки и модификации различных версий приложений для различных типов компьютеров и серверов внутри их сетей.

*void*
Ключевое слово языка Java™, используемое в описании метода для указания на то, что метод

не возвращает никакого значения. void может также использоваться как нефункциональный оператор.

*volatile*
Ключевое слово языка Java™, используемое в описания переменной для указания на то, что переменная модифицируется асинхронно несколькими совместно выполняющимися потоками.

## W

*wait*
Команда UNIX(r), которая будет ожидать завершения всех фоновых процессов и отчета об их статусе завершения.

*Web-приложение, распределенное(Web application, distributable)*
Web-приложение, использующее технологию J2EE и написанное таким образом, что оно может быть внедрено в Web контейнер, распределенный по нескольким виртуальным машинам Java, работающих на одном хосте или на различных хостах. Дескриптор "размещения" для такого приложения использует распределенный элемент.

*Web-компонент (Web component)*
Компонент, обеспечивающий сервисы в ответ на запросы, сервлеты или JSP страницы.

*Web-контейнер (Web container)*
Контейнер, обеспечивающий исполнение Web компонентов, удовлетворяющих правилам J2EE архитектуры. Эти правила определяют среду времени выполнения (runtime environment) для Web-компонентов, включая безопасность, совместную работу, управление жизненным циклом, транзакции, внедрение и другие службы. Контейнер, обеспечивающий такие же службы как JSP контейнер и интегрированный взгляд на API платформы J2EE. Web контейнер обеспечивается Web или J2EE сервером.

*Web-контейнер, распределенный (Web container, distributed)*
Web-контейнер, который может запускать Web-приложение, помеченное как распределенное, которое выполняется на нескольких виртуальных машинах Java, работающих на одном или на различных хостах.

*Web-сервер (Web server)*
Программа, обеспечивающая службы для доступа в Internet, Intranet, или Extranet. Web-сервер содержит Web-сайты, обеспечивая поддержку HTTP и других протоколов и выполняет серверные программы (такие как CGI-скрипты или сервлеты), которые выполняют определенные функции. Например, Web-контейнер обычно основан на Web-сервере для обеспечения обработки сообщений HTTP. Архитектура J2EE предполагает, что Web-контейнер предоставляется Web-сервером от одного и того же поставщика, т.е. не определяя правила между этими двумя сущностями. Web-сервер может предоставлять один или много Web-контейнеров.

*Web-службы (Web Services)*
Свободно собранные программные компоненты, способные взаимодействовать между собой

по многочисленным сетям для предоставления определенного результата конечному пользователю. Во время работы они используют развивающуюся группу стандартов, которые определяют их (служб) описание и взаимодействие, таких как SOAP (простой протокол доступа к объектам), UDDI (универсальное обнаружение и интеграция описаний), XML (открытый язык меток), WSDL (язык описания Web-служб).

*while*
Ключевое слово языка Java™, используемое для определения цикла, который повторяет блок инструкций. Условие продолжения цикла указывается как часть оператора while.

*общедоступные для чтения файлы (world readable files)*
Файлы в файловой системе, которые могут быть просмотрены (прочтены) любым пользователем. Например: файлы, расположенные на Web-серверах могут быть просмотрены пользователями Internet, если права файлов были установлены как "доступные для чтения".

*wrapper*
Объект, который инкапсулирует и уполномочивает другой объект на изменение своего интерфейса или поведения определенным образом.

*язык описания Web-сервисов (Web Services Description Language - WSDL)*
XML язык, который используется для описания Web-сервиса и для определения способа общения с Web-сервисом.

*World Wide Web - WWW*
"Всемирная паутина". Сеть систем и данных в них, т.е. часть Internet. См. также *Internet*.

## X

*расширяемый язык меток (Extensible Markup Language - XML)*
Расширяемый язык разметки текстов (или текстовых документов). Дескрипторы размещения J2EE выражены при помощи XML.

## Y

## Z