Н.А. Вязовик

Программирование на Java

ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Программирование на Java

2-е издание, исправленное

Вязовик Н.А.

Национальный Открытый Университет "ИНТУИТ"
2016

Программирование на Java/ Н.А. Вязовик - М.: Национальный Открытый Университет "ИНТУИТ", 2016

Курс лекций посвящен современному и мощному языку программирования Java. В его рамках дается вводное изложение принципов ООП, необходимое для разработки на Java, основы языка, библиотеки для работы с файлами, сетью, для построения оконного интерфейса пользователя (GUI) и др.

Java изначально появилась на свет как язык для создания небольших приложений для Интернета (апплетов), но со временем развилась как универсальная платформа для создания программного обеспечения, которое работает буквально везде – от мобильных устройств и смарт-карт до мощных серверов.
 Данный курс начинается с изложения истории появления и развития Java. Такие знания позволят лучше понять особенности платформы и спектр существующих продуктов и технологий. Также создание Java является интересным примером истории одного из самых популярных и успешных проектов в компьютерном мире.
 Затем излагаются основные концепции ООП, необходимые для освоения объектно-ориентированного языка программирования Java.
 Ключевые понятия и конструкции языка описываются доступным языком, но, тем не менее, на достаточно глубоком уровне. Детально рассмотрены особенности лексики, системы типов данных, объектной модели. Уделяется особое внимание модификаторам доступа, соглашениям по именованию, преобразованию типов, работе с массивами, обработке ошибок (исключительных ситуаций). Курс завершается рассмотрением базовых библиотек Java, предоставляющих всю необходимую функциональность для создания самых разных приложений – коллекции объектов, работа с файлами, сетью, создание GUI приложений, построение многопоточной архитектуры и многое другое. Описание сетевой библиотеки предваряется изложением основ сетевых протоколов и технологий.

Что такое Java? История создания

Первая лекция начинается с рассказа о событиях, происходивших задолго до официального объявления Java. Хотя эта технология на сегодняшний день разительно отличается от того, как задумывали ее создатели, однако многие особенности берут свое начало от решений, принятых в то время. Будут освещены все основные этапы создания, появления и развития Java. Также в лекции излагаются необходимые базовые знания для разработчиков – основные свойства платформы Java, и почему она является платформой, а не просто языком программирования. Что входит в пакет разработчика, где найти нужную информацию, какие дополнительные продукты предоставляет Sun, чем различаются Java и Java Script – ответы на эти и другие общие вопросы находятся в первой лекции.

Что такое Java?

Что знают о Java обычные пользователи персональных компьютеров и Internet? 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 (кроссплатформенность). 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 Internet, и самое распространенное ее применение - небольшие программы, 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 заявлены, а на самом деле не очень-то работают, а главное - программы на Java исполняются чрезвычайно 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 достаточно много времени для "просто модной технологии". Во-вторых, конференция разработчиков Java One, которая впервые была организована в 1996 году, уже через год собрала более 10000 участников и стала крупнейшей конференцией по созданию программного обеспечения в мире (каждый следующий год число участников росло примерно на 5000). 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 сегодня число программистов на Java оценивается в 3 миллиона.

Было выпущено пять основных версий языка, начиная с 1.0 в 1995 году и заканчивая 1.4 в феврале 2002 года. Следующая версия 1.5 выпущена в 2004 году. Все версии и документацию к ним всегда можно было бесплатно получить на официальном web-сайте Java ссылка: 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. 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 построенных на ней продуктов. Это и "голубой гигант" IBM, и создатель платформы Macintosh фирма Apple, и лидер в области реляционных БД Oracle, и даже главный конкурент фирмы Sun - корпорация Microsoft - лицензировала Java еще в марте 1996 года.

В следующем разделе описывается краткая история зарождения и развития идей, приведших к появлению Java, что поможет понять, чем на самом деле является эта технология, каковы ее свойства и отличительные черты, для чего она предназначена и откуда взялось такое разнообразие мнений о ней.

История создания Java

Если поискать в Internet историю создания Java, выясняется, 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.

Сложности внутри Sun Microsystems

Действительно, события начинают разворачиваться в декабре 1990 года, когда бурное развитие WWW (World Wide Web - "всемирная паутина") никто не мог еще даже предсказать. Тогда компьютерная индустрия была поглощена взлетом персональных компьютеров. К сожалению, фирма Sun Microsystems, занимающая значительную долю рынка серверов и высокопроизводительных станций, по мнению многих сотрудников и независимых экспертов, не могла предложить ничего интересного для обычных пользователей "персоналок" - для них компьютеры от Sun представлялись "слишком сложными, очень некрасивыми и чересчур "тупыми" устройствами".

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, в Sun делается неверно. 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. Он безжалостно раскритиковал новую программную архитектуру NeWS, над которой фирма работала в то время, а также высоко оценил только что объявленную операционную систему NeXTstep. Нотон предложил привлечь профессиональных художников-дизайнеров, чтобы сделать пользовательские интерфейсы Sun более 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 почти всех сотрудников из Window Systems Group (если выполнить предыдущие условия, они будут просто не нужны).

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

МакНили разослал письмо Нотона всему управляющему составу корпорации, а те переслали его своим ведущим специалистам. Откликнулись буквально все, и, по общему мнению, Нотон описал то, о чем все думали, но боялись высказать. 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, а также автором первой реализации текстового редактора EMACS на C. Эти люди имели огромный авторитет в корпорации.

Чтобы не останавливаться на достигнутом, Нотон решил предложить какой-то совершенно новый проект. Он 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 аппаратно-программную платформу; эту платформу нужно воплотить в устройстве, предназначенном для персонального пользования, удобном и простом в обращении - т.е. создать 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 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 несовместимы с продуктами Sun; на первый год группе необходим миллион долларов.

Проект Green

5 декабря 1990 года, в день, когда Нотон должен был перейти в компанию NeXT, Sun сделала ему встречное предложение. Руководство согласилось со всеми его условиями. Поставленная задача - "создать что-нибудь необычайное". 1 февраля 1991 года Патрик Нотон, Джеймс Гослинг и Майк Шеридан (Mike Sheridan) вплотную приступили к реализации проекта, который получил название Green.

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

На ежегодном собрании Sun весной 1991 года Гослинг заметил, что компьютерные чипы получили необычайное распространение, они применяются в видеомагнитофонах, тостерах, даже в дверных ручках гостиниц! 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 будет взаимодействовать с пользователем с помощью анимации, показывая, чем можно управлять и как. Чтобы создать такой прибор, Нотон начал работать над специализированной графической системой, Гослинг взялся за программное обеспечение, а Шеридан занялся бизнес-вопросами.

В апреле 1991 года команда покидает офис Sun, отключаясь даже от внутренней сети корпорации, и въезжает в новое помещение. Закупаются разнообразные бытовые электронные устройства, такие как игровые приставки Nintendo, телевизионные приставки, пульты дистанционного управления, и разработчики играют в различные 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. 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 концерте, где смог воочию наблюдать сложное переплетение проводов, огромное количество колонок и полуавтоматических прожекторов, которые, казалось, согласованно двигаются 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 на 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 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 новый каталог и раздумывая, как его 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, после проведения 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 возможности 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 любого другого телефона, а поскольку редко кого из них можно было застать на рабочем месте, эти слова очень часто громко кричали на весь офис). Для придания интерфейсу большей привлекательности разработчики создали забавного персонажа по имени Дьюк (Duke), который всегда был готов помочь пользователю выполнить его задачу. В дальнейшем он стал спутником Java, счастливым талисманом - его можно встретить 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 состоялась первая демонстрация для Билла Джоя и Скотта МакНили. В ноябре группа снова подключилась к сети Sun по модемной линии. Чем дальше развивался проект, тем больше новых специалистов 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 нарисованного дома, а чтобы управлять им, надо было просто водить по экрану пальцем - никаких специальных средств 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 стоили очень дорого) и MS DOS, даже мышь еще не была обязательным атрибутом персонального компьютера.

Руководители Sun были просто в восторге - появилось отличное оружие против таких могучих конкурентов, как HP, IBM и Microsoft. Новая технология была способна не только демонстрировать мультики. Объектно-ориентированный язык OaK обещал стать достаточно мощным инструментом для написания программ, которые могут работать в сетевом окружении. Его объекты, свободно распространяемые по сети, работали бы на любом устройстве, начиная с персонального компьютера и заканчивая обычными бытовыми видеомагнитофонами и тостерами. На презентациях Нотон представлял области 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 они не обязательно должны были 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, а значит, незаметные и удобные для пользователя.

Компания FirstPerson

Крупные компании-производители, такие как Mitsubishi Electric, France Telecom, Dolby Labs, заинтересовались новой технологией, начались переговоры. Шеридан подготовил бизнес-план с оригинальным названием "Beyond the Green Door" ("За зеленой дверью"), в котором предложил 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 позднее оказалось, что стоимость подобного решения (процессор, память, экран) составляет не менее $50. Производители же бытовой техники не привыкли платить значительные 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 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 импортировать 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 идея интерактивного телевидения оказалась нежизнеспособной. Ожиданиям не суждено было 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 снова приносить прибыль.

World Wide Web

В погоне за призраком интерактивного телевидения многие участники компьютерного рынка 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 Internet. 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 в любой точке планеты. Именно такие перекрестные обращения, используя которые, пользователь мог незаметно для себя посетить множество узлов Internet, 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 бесплатны и доступны для всех желающих. Впервые обычные пользователи персональных компьютеров без всякой специальной подготовки могли пользоваться глобальной сетью не только для решения рабочих вопросов, но и для поиска информации на самые разные темы. Количество документов в пространстве WWW стало расти экспоненциально, и очень скоро сеть Internet 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, каким-то непостижимым образом 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 Internet были произведены именно Sun.

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

По многим причинам этот план не устроил руководство Sun (он не вполне 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 Internet, они поняли, что функциональность тех сетевых приложений, для которых создавался OaK, очень близка к WWW. Билл Джой вспомнил, как он двадцать лет назад принимал участие в разработке UNIX в Беркли и затем эта операционная система получила широчайшее распространение благодаря тому, что ее можно было загрузить по сети бесплатно. Такой принцип бесплатного распространения коммерческих продуктов создал саму WWW, тем же путем 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 Internet поможет воскресить платформу 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 Internet. Гослинг пересматривает программный код платформы, а Нотон берется за написание "убойного" приложения, которое сразу бы продемонстрировало всю мощь OaK для Internet.

В самом деле, эти технологии прекрасно подошли друг другу. Языки программирования всегда играли важную роль в развитии компьютерных технологий. Мэйнфреймы не были особенно полезны, пока не появился Cobol. Благодаря языку Fortran от IBM, компьютеры стали широко применяться для научных вычислений и исследований. Altair BASIC - самый первый продукт от Microsoft - позволил всем программистам-любителям 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 Internet.

Несмотря на то, что к середине 1994 года WWW достиг невиданных размеров (конечно, по меркам того времени), web-страницы по-прежнему были скорее похожи на обычные бумажные издания, чем на интерактивные приложения. По большей части вся работа в сети заключалась в отправке запроса на web-сервер и получении ответа, который содержал обычный статический HTML -файл, отображаемый браузером на стороне клиента. Уже тогда функциональность web-серверов расширялась с помощью CGI (Common Gateway Interface). Эта технология позволяла по запросу клиента запускать на сервере обычную программу 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), клиент мог ждать несколько минут, чтобы лишь увидеть сообщение о том, 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 серверы.

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 свои задачи, нуждаются в различных компьютерах). Следовательно, приложение необходимо развивать сразу в нескольких вариантах, что существенно увеличивает стоимость поддержки. Кроме того, обновление клиентской части означает, что нужно перенастроить все компьютеры компании в кратчайший срок. А ведь обновлениями часто занимаются несколько групп разработчиков.

Попытка придать Internet- браузерам возможности полноценного клиентского приложения встречает еще большие трудности. Во-первых, обычные сложности предельно возрастают - в Internet представлены практически 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 средством их распространения были CD-ROM'ы. Понятно, что для Internet необходимо было серьезно поработать над компактностью кода.

Если оглянуться на историю развития OaK, становится понятно, что эта платформа удивительным образом отвечает всем перечисленным требованиям Internet-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.

Возрождение OaK

Для победного выхода OaK не хватало последнего штриха - браузера, который поддерживал бы 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 Internet, назвали апплетами (applets).

Следующая демонстрация происходила на конференции, где встречались разработчики Internet-приложений и представители индустрии развлечений. Когда Гослинг начал презентацию WebRunner, слушатели не проявили 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 данная функция, возможно, не производит такого впечатления, однако 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 отрезки по размеру. Пример тоже нехитрый, однако наглядно демонстрирующий, что на стороне клиента появилась полноценная программная платформа. Оба эти апплета сейчас являются стандартными примерами и входят в состав Java Development Kit любой версии. Успех демонстрации, которая закончилась бурными

аплодисментами, показал, что OaK и WebRunner способны устроить революцию в Internet, так как все участники конференции по-другому взглянули на возможности, которые предоставляет Всемирная Сеть.

Кстати, в начале 1995 года, когда стало ясно, что официальное объявление уже не за горами, за дело взялись маркетологи. В результате их исследований OaK был переименован в Java, а WebRunner стал называться HotJava. Многие тогда недоумевали, что же послужило поводом для такого решения. Легенда гласит, что Java - это сорт кофе (такой кофе действительно есть), который очень любили программисты. Видимо, похожим образом родилось и название HotJava ("горячая Java "). Тема кофе навсегда останется в названиях и логотипах (технология создания компонентов названа Java Beans - зерна кофе, специальный формат для архивирования файлов с Java -программами JAR - банка с кофе и т.д.), а сам язык критики стали называть "для кофеварок". Впрочем, сейчас все уже привыкли и не задумываются над названием, возможно, на это и было рассчитано (а тем, кто продолжает выражать недовольство, приводят альтернативные варианты, которые рассматривались тогда - Neon, Lyric, Pepper или Silk).

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

Когда это произошло, команда стала подсчитывать случаи загрузки их продукта для просмотра. Вскоре пришлось считать уже сотнями. Затем решили, что если удастся достигнуть 10.000, то это будет просто ошеломляющий успех. Ждать пришлось совсем не так долго, как можно было предположить. Интерес нарастал лавинообразно, после просмотров приходило большое количество писем и мощности Internet-канала стало не хватать. На письма всегда отвечали очень подробно, что

поначалу 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 сотрудник, так как приходило уже по 2-3 тысячи писем в день. Вскоре руководство Sun осознало, что имея такой мощный успех Java не имеет никакого бюджета или плана для рекламы и других акций продвижения на рынок. Первым шагом в этом направлении становится публикация 23 марта 1995 года в газете Sun Jose Mercury News статьи с описанием новой технологии, где был приведен адрес официального сайта ссылка: http://java.sun.com/, который и по сей день является основным источником информации по Java.

Java выходит в свет

Наконец, вся подготовительная работа стала подходить к своему логическому завершению. Официальное объявление Java, уже получившей широкое признание и подающей большие 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 четыре 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 назад, когда перехватил предложение от Time Warner) и Марком Андриссеном (создателем NCSA Mosaic). Эта компания стала 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 технологии Java и HotJava были официально объявлены Sun и тогда же представители компании сообщили, что новая версия самого популярного браузера Netscape Navigator 2.0 будет поддерживать новую технологию. По сути, это означало, что отныне Java становится такой же неотъемлемой частью WWW, как и HTML. Уже второй раз презентация закончилась овацией - победное шествие Java

началось.

История развития Java

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

Браузеры

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

С помощью Java web-страницу можно наполнить не только обычным текстом, но и динамическими элементами - простыми видеовставками типа вращающегося земного шара или Дьюка, машущего рукой (хотя сейчас такие задачи хорошо решает анимированный GIF, а в более сложных случаях - Macromedia Flash); интерактивными элементами типа вращающейся модели химической молекулы; бегущими строками, содержащими, например, биржевые индексы или прогноз погоды.

Но на самом деле Java - это больше, чем украшение HTML. Поскольку это полноценный язык программирования, с его помощью можно создать 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 Internet.

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 так же, скачав апплет и поработав с ним, можно удалить его, 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, и при этом автоматически поддерживаются самые последние версии.

С другой стороны, пользователь уже не привязан к своему основному рабочему месту, в любом Internet-кафе можно открыть нужную web-страницу и начать работу с привычными программами. И все это без каких-либо опасений подцепить вирус. Разработчиков очень заинтересовало, что их программы через день после выпуска могут увидеть пользователи всего мира, 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 браузер Netscape Navigator, начиная с версии 2.0, также поддерживал 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 подписала лицензионное соглашение с Sun, Microsoft оказалась в трудной ситуации.

Internet Explorer 2.0 не вызывал энтузиазма и никто не верил, что он может составить хоть сколько-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 намерении лицензировать Java, а в марте

1996 года соглашение о лицензировании было 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 произошедших событий. Теперь уже очевидно, что Microsoft полностью удалось осуществить свой план. Если Netscape Navigator 3.x еще сохранял лидирующее положение, то Netscape 4.x уже начал уступать Internet Explorer 4.x. Версия NN 5.x так и не вышла, а NN 6.x стал очередным разочарованием для бывших поклонников "Навигатора". Сейчас появилась версия 7.0, однако она не занимает значительной 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 компания боролась с Netscape "нерыночными" средствами. Однако сравним действия конкурентов. Среди многих шагов, предпринятых Microsoft, была и поддержка независимой организации W3C, которая руководила разработкой нового стандарта HTML 3. Вначале компания Netscape считалась локомотивом индустрии, поскольку она постоянно развивала и модернизировала HTML, который изначально вообще-то не предназначался для графического оформления текста. Но Microsoft, вложив большое количество средств 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, созданные 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, готовым к использованию (и от которого, кстати, избавиться нельзя было принципиально).

А каким образом Netscape смог добиться лидирующего положения? В свое время подобными же методами компания пыталась (успешно, в конце концов) вытеснить с рынка NCSA Mosaic. Тогда HTML был не особенно богат интересными возможностями, а потому инновации, поддерживаемые Navigator, сразу привлекали внимание разработчиков и пользователей. Однако такие страницы некорректно отображались в

Mosaic, что заставляло его пользователей задуматься о переходе к продуктам 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 без того небогатые возможности HTML приходилось изощренно подгонять таким образом, чтобы страницы выглядели одинаково в обоих браузерах.

Про HotJava, к сожалению, сказать особенно нечего. Некоторое время Sun поддерживала этот продукт и добавила возможность визуально генерировать web-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 черты 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 кроссплатформенный объектный язык сценариев для корпоративных сетей и Internet. Код Java Script описывается прямо в HTML -тексте (хотя можно 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 использовать эти языки, опыт

программирования не требуется;

- кроссплатформенность. 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 Internet-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 объектно-ориентированного подхода поддерживаются), но позволяет использовать различные объекты, предоставляемые браузером ;
- похожая история появления и развития. Оба языка были объявлены компаниями Sun и Netscape с интервалом в несколько месяцев. Вышедший вскоре после этого Netscape Navigator 2.0 поддерживал обе новые технологии. Возможно, само название Java Script было дано для того, чтобы воспользоваться 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 позиционировался как язык для создания Internet-приложений (апплетов), то сейчас уже очевидно, 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 защиту 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.

Сетевые компьютеры

Когда стало понятно, что новая технология пользуется небывалым спросом, разработчикам захотелось укрепить и развить успех и распространенность Java. Для того чтобы Java не разделила судьбу NeWS (эта оконная система упоминалась в начале лекции, она не получила развития, проиграв X Window), компания Sun старалась наладить сотрудничество с независимыми фирмами для производства различных библиотек, средств разработчика, инструментов. 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 термин "платформа", чем Java отличается от обычного языка программирования.

Как правило, платформой называют сочетание аппаратной архитектуры ("железо"), которая определяется типом используемого процессора (Intel x86, Sun SPARC, PowerPC и др.), с 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 другими возможностями. 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 по-разному поддерживают многие возможности (например, операционная система Mac OS традиционно использует однокнопочную мышь, в то время как Windows изначально рассчитана 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 специфичны для каждой платформы. Так было с самого начала эпохи 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 бы с развитием компьютерной индустрии разработчики не попытались создать универсальную платформу, под которой могли работать все программы. Особенно такому шагу способствовало бурное развитие Глобальной сети Internet, 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 создавался 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 - с помощью виртуальной машины. 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 речь идет о браузере с поддержкой Java, подразумевается, что в нем имеется встроенная виртуальная машина.

Подробнее JVM рассматривается ниже, но необходимо сказать, что разработчики Sun приложили 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, это была "возможно, самая небольшая и быстрая операционная система, поддерживающая Java ". Действительно, разработчики стремились к тому, чтобы обеспечить возможность исполнять Java -приложения на самом широком спектре устройств - сетевые компьютеры, карманные компьютеры (PDA), принтеры, игровые приставки, мобильные телефоны и т.д. 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, продвигающий платформу Java на один уровень вниз - на уровень операционных систем. Предполагалось 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.

Кроме бытовых приборов, компания Sun позиционировала данное решение и для компьютерной индустрии - сетевые компьютеры должны были заменить разнородные платформы персональных рабочих станций. Такой подход хорошо укладывался в основную 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 планету, идея была революционной.

Если же строить многофункциональную сеть, то к ее рабочим станциям предъявляются совсем другие требования - они не должны быть особенно мощными, вычислительные задачи можно переложить на серверы. Это даже более выгодно, так как позволяет централизовать поддержку и обновление программного обеспечения, а также не вынуждает сотрудников быть привязанными 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 очевидных удобств, это начинание было с большим энтузиазмом поддержано индустрией и в силу того, что оно являлось сильнейшим оружием в борьбе с крупнейшим 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, заставляло 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 свела бы на нет преимущества Wintel в случае широкого распространения. Разработчики просто перестали бы задумываться, что находится внутри их рабочей станции, так же, как домашние пользователи не имеют представления, на каких микросхемах собран их мобильный телефон или видеомагнитофон.

Мы уже рассказывали о том, 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 объявила, 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 разработчики Microsoft слегка "улучшили" язык Java, добавив несколько новых ключевых слов и библиотек. Не то что бы это были сверхмощные расширения, однако достаточно привлекательные для того, чтобы значительная часть разработчиков начала ее использовать. К счастью, в Sun быстро осознали всю степень опасности такого шага. Java могла потерять звание универсальной платформы, для которой верен знаменитый девиз "Write once, run everywhere" ("Написано однажды, работает везде"). В таком случае она утратила бы основу своего успеха, превратившись всего лишь в "еще один язык программирования".

Компании Sun удалось 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 тяжба продолжается до сих пор с переменным успехом сторон. Например, Microsoft исключила из виртуальной машины Internet Explorer библиотеку java.rmi, позволяющую создавать распределенные приложения, пытаясь привлечь внимание 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, то, увы, они пока не нашли 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.

Платформа Java

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

Самое широко известное, и в то же время вызывающее самые бурные споры, свойство — много- или кроссплатформенность. Уже говорилось, что оно достигается за счет использования виртуальной машины JVM, которая является обычной программой, исполняемой операционной системой и предоставляющей Java -приложениям все необходимые возможности. 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 поначалу 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 просто были слишком нетерпеливы. Java только появилась на свет, а первые версии спецификаций были недостаточно исчерпывающими.

Очень скоро специалисты Sun пришли к выводу, что просто свободно публиковать 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). 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 свойством является объектная ориентированность Java, что всегда упоминается во всех статьях и пресс-релизах. Сам объектно-ориентированный подход (ООП) рассматривается в следующей лекции, однако важно подчеркнуть, что в Java практически все реализовано в виде объектов - потоки выполнения (threads) и потоки данных (streams), работа с сетью, работа с изображениями, с пользовательским интерфейсом, обработка ошибок 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 основные особенности. Прежде всего, 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 лекции.

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, при этом причина дефекта может обнаружиться совсем в 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 существует всего 8 типов данных, которые не являются объектами. Они были определены с самой первой версии и никогда не менялись. Это пять целочисленных типов: byte, short, int, long, а также к ним относят символьный char. Затем два дробных типа float и double и, наконец, булевский тип boolean. Такие типы называются простыми, или примитивными (от английского primitive), и они подробно рассматриваются в лекции, посвященной типам данных. Все остальные типы - 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 многим повод думать, что Java - это упрощенный C++ с дополнительными возможностями, такими как garbage collector. Автоматический сборщик мусора (garbage collector) мы рассмотрим 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 по целому ряду причин. Во-первых, они сами на тот момент считали C++ своим основным инструментом. Во-вторых, зачем придумывать что-то новое, когда есть вполне 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 говорилось, C++ принципиально не годился для новых задач, которые поставили себе разработчики из компании Sun, поэтому модель Java была построена заново, причем в соответствии с совсем другими целями. Дальнейшие лекции 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 неоднозначную оценку. Действительно, авторы потрудились 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. Очевидно, что при таком подходе существует две опасности - либо удалить объект, который еще кому-то необходим (и если к нему действительно произойдет обращение, то возникнет ошибка), либо не удалять объект, ставший ненужным, а это означает утечку памяти, то есть программа начинает потреблять все большее 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 легкость освоения новой технологии. Как уже было сказано, ожидалось (и эти ожидания оправдались, в подтверждение правильности выбранного пути!), что Java должна получить максимально широкое применение, даже в тех 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 надежными. Internet стал Всемирной Сетью, поскольку появились непрофессиональные пользователи, а возможность создавать апплеты для них не менее привлекательна. Им требовался простой инструмент для создания надежных приложений.

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

Кроме введения garbage collector, были предприняты и другие шаги для

облегчения разработки. Некоторые из них уже упоминались - отказ от множественного наследования, упрощение синтаксиса и др. Возможность создания многопоточных приложений была реализована в первой же 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 лекциях. 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 приложения, 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 очистка 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 тщательно проверяет, все ли они подчиняются общим правилам безопасности Java и не созданы ли злоумышленниками с помощью каких-то других средств (и не искажены ли 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 (пользователь легко может назначать или отменять конкретные права). Например, апплеты по умолчанию никогда не получат доступ к локальной файловой системе. Такие встроенные ограничения есть 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 защиты, 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 Internet- и интранет (внутренние сети корпораций)-приложений.

Наконец, важная отличительная особенность Java - это его 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 кроссплатформенность ;
- объектная ориентированность, создана эффективная объектная модель;
- привычный синтаксис C/C++;
- встроенная и прозрачная модель безопасности ;
- ориентация на Internet-задачи, сетевые распределенные приложения;
- динамичность, легкость развития и добавления новых возможностей;
- простота освоения.

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.

Основные версии и продукты Java

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 и 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 уже существующими Java -файлами. Компилятор представлен утилитой javac (java compiler). Виртуальная машина реализована программой java. Для тестовых запусков апплетов существует специальная утилита appletviewer. Наконец, для автоматической генерации документации на основе исходного кода прилагается средство javadoc.

Первая версия содержала всего 8 стандартных библиотек:

- `java.lang` - базовые 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 описывается в лекции 11;
- `java.awt.image` - дополнительные классы для работы с изображениями;
- `java.io` - работа с потоками данных (streams) и с файлами;
- `java.net` - работа с сетью.

Таким образом, 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 момент создания курса). 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 подоспела 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, представляет собой 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. Многие части виртуальной машины были оптимизированы и переписаны с использованием Assembler, а не C, как до этого. Кроме того, 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 усовершенствованы поддержка национальных языков и система безопасности.

За первые три недели JDK 1.1 был скачан более 220.000 раз, менее чем через год - более двух 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 того, 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:

- усовершенствованный оконный интерфейс , который получил особое название - Swing ;
- реализация Drag-and-Drop;
- поддержка 2D-графики, более удобная работа с изображениями;
- Accessibility API для пользователей с ограниченными возможностями

и другие функции. 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 компонентов 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 ее и все последующие версии начали называть платформой Java 2 (хотя номера, конечно, по-прежнему отсчитывались 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:

- существенно 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 15 июня 1999 года, спустя полгода после выхода JDK 1.2. 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, в частности, различных спецификаций и библиотек насчитывалось несколько десятков, а потому они нуждались в четкой структуризации. Кроме того, такое разделение облегчало развитие и продвижение на рынок технологии Java.

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

J2EE содержит все необходимое для создания 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, чтобы удовлетворять жестким аппаратным требованиям небольших устройств, таких как карманные компьютеры и сотовые телефоны.

Далее развитие этих технологий происходит разными темпами. Если J2SE уже была доступна 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 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.

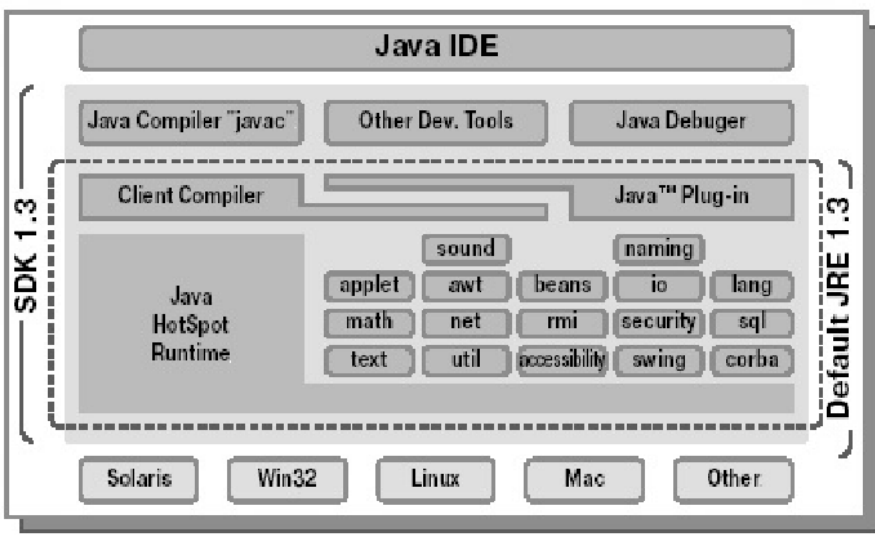
Развитие HotSpot продолжалось более года, пока в начале мая 2000 года высокопроизводительная JVM не вошла в состав новой версии J2SE. В эту версию было внесено еще множество улучшений и исправлений, но именно прогресс в скорости работы стал ключевым изменением нового j2sdk 1.3 (последняя подверсия 1.3.1).

Наконец, последняя на данный момент версия J2SE 1.4 вышла в феврале 2002 года. Она была разработана для более полной поддержки web-сервисов (web services). Поэтому основные изменения коснулись

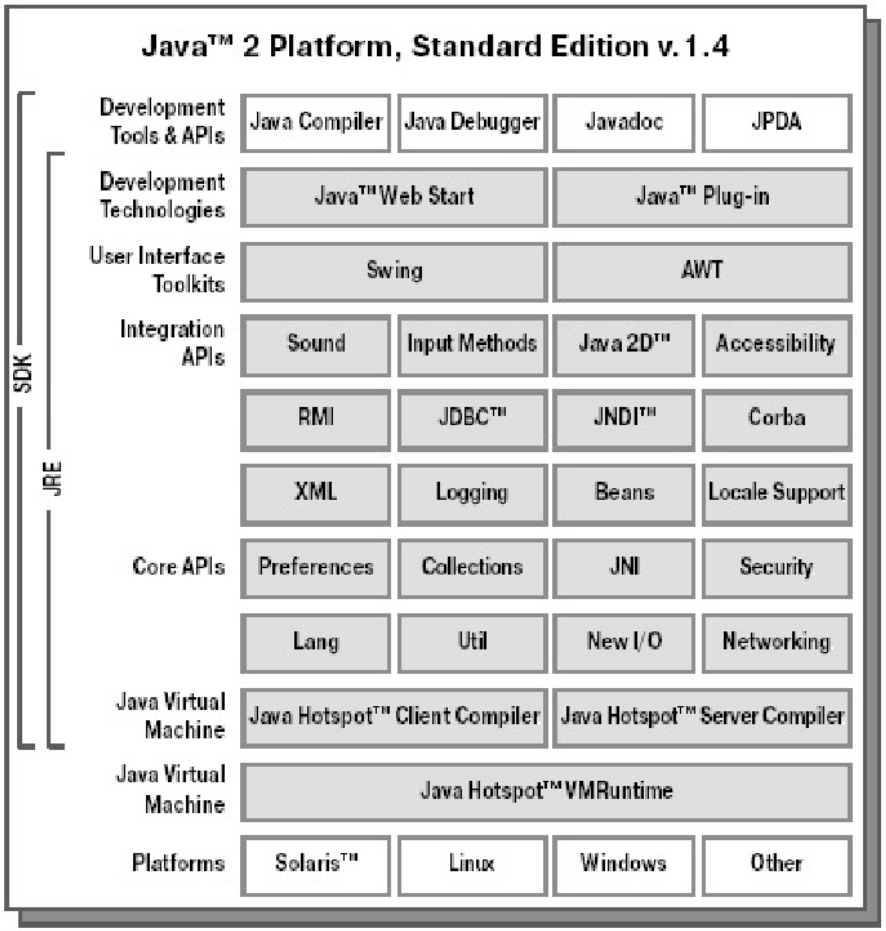
работы с XML (Extensible Markup Language). Другое революционное добавление - выражение assert, позволяющее в отладочном режиме проверять верность условий, что должно серьезно упростить разработку сложных приложений. Наконец, были добавлены классы для работы с регулярными выражениями.

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

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



1.1.Составляющие технологии версии 1.3.



1.2. Составляющие технологии версии 1.4.

Заключение

В этой лекции мы рассказали о том, какая непростая 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 Internet.

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 четырехлетней истории создания и развития, Java была официально представлена миру. Благодаря подписанию лицензионного соглашения с Netscape, это событие стало поистине триумфальным.

Были рассмотрены различные варианты применения Java. Отдельно был описан язык Java Script, который, несмотря на сходство в названии, имеет не так много общего 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.

Основы объектно-ориентированного программирования

В этой лекции излагается основная концепция объектно-ориентированного подхода (ООП) к проектированию программного обеспечения. Поскольку в Java почти все типы (за исключением восьми 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 Internet-ресурсов.

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 любая вычислительная задача будет решена. Эта идея оказалась настолько жизнеспособной, что долгое время доминировала над всем процессом разработки программ. Появились специализированные языки программирования, созданные 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 в данной методологии было понятие алгоритма. Алгоритм - это способ решения вычислительных и других задач, точно описывающий определенную последовательность действий, которые необходимо выполнить для достижения заданной цели. Примерами алгоритмов являются хорошо известные правила нахождения корней квадратного уравнения или системы линейных уравнений.

При увеличении объемов программ для упрощения их разработки появилась необходимость разбивать большие задачи 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, процедурный подход позволил уменьшить объем программного кода за счет написания часто используемых кусков кода в виде процедур и их применения в различных частях программы.

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
  Write('Hello, ');
  WriteLn(name);
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 приходится 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. Одна 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 применяется так называемая блок-

схема . Соответствующая система графических обозначений была зафиксирована в ГОСТ 19.701-90. Пример блок-схемы изображен на рисунке (рис. 2.1).

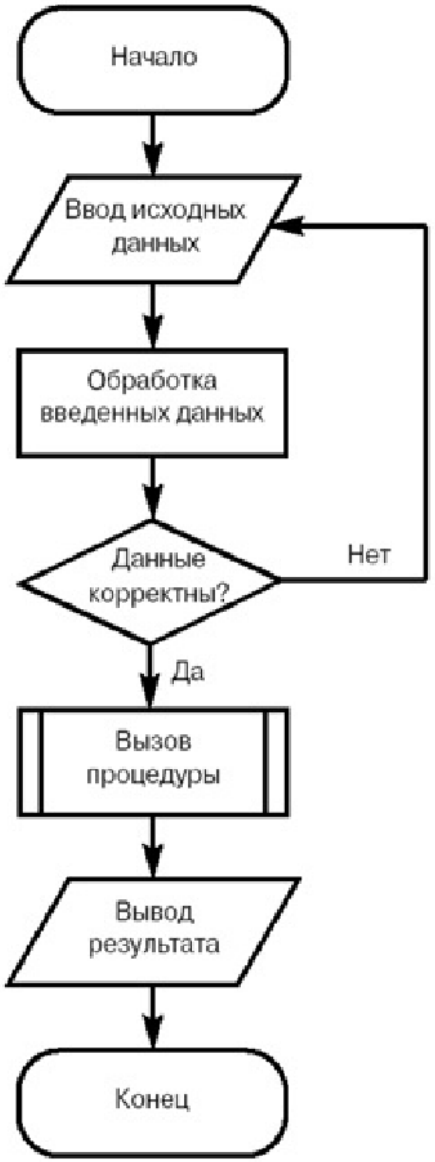


Рис. 2.1. Пример блок-схемы.

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` может заметно осложнить понимание кода. Такие запутанные программы сравнивали с порцией спагетти (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 многие предлагали исключить оператор `goto` из языков программирования. Именно с этого времени отсутствие безусловных переходов стали считать хорошим стилем программирования.

Дальнейшее увеличение программных систем способствовало формированию 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". Пик популярности идей структурного программирования приходится на конец 70-х - начало 80-х годов.

В этот период основным показателем сложности разработки программы считался 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 соответствовать определенным требованиям. 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.

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 для небольших предприятий и организаций была слишком высока.

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 данных и манипулирование ими, 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 независимой от языка 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 2.2 [3]). При объектно-ориентированном подходе эти предметы и понятия заменяются моделями, т.е. определенными формальными конструкциями.

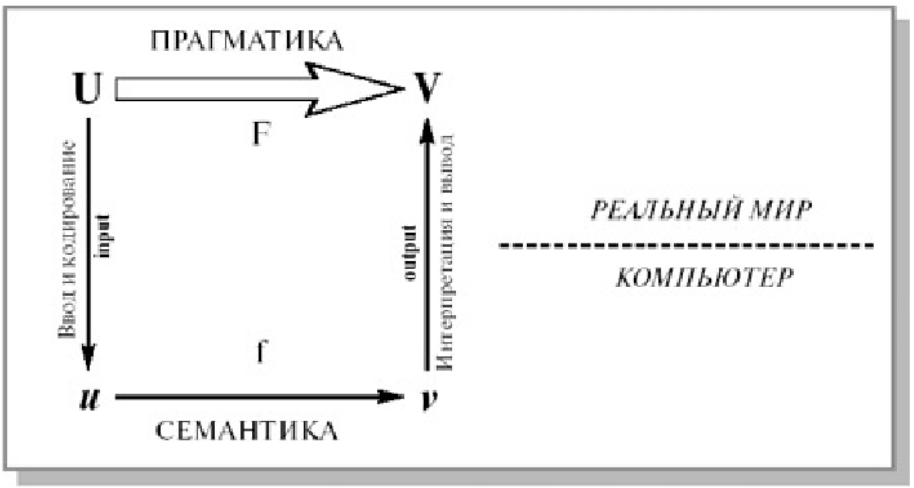


Рис. 2.2. Семантика (смысл программы с точки зрения выполняющего ее компьютера) и прагматика (смысл программы с точки зрения ее пользователей) [3].

Модель содержит не все признаки и свойства представляемого ею предмета или понятия, а только те, которые существенны для разрабатываемой программной системы. Таким образом, 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 обладает такими преимуществами, 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 лекции, 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.

Объекты

По определению будем называть объектом понятие, абстракцию или любой предмет с четко очерченными границами, имеющий 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 и отличительными характеристиками и являющаяся важной в предметной области [2].

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

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"), 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, его моделирующего.

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 момент времени состояние объекта включает в себя перечень (обычно статический) свойств объекта и текущие значения (обычно динамические) этих свойств [2].

Поведение

Для каждого объекта существует 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 того, какие сообщения он может принимать.

Поведение (behavior) - действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния ; видимая извне и воспроизводимая активность объекта [2].

Уникальность

Уникальность - это то, что отличает объект от других объектов. Например, у вас может быть несколько одинаковых монет. Даже если абсолютно все их свойства (атрибуты) одинаковы (год выпуска, номинал и т.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 чаще всего понимается адрес размещения объекта в памяти.

`Identity` (уникальность) объекта состоит в том, что всегда можно определить, указывают 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 имени ссылки на объект. Это неверно, т.к. на один объект может указывать несколько ссылок, и ссылки могут менять свои значения (ссылаться на другие объекты).

Итак, уникальность (identity) - свойство объекта; то, что отличает его от других объектов (автор не согласен с переводом русского издания [2],

поэтому здесь приводится авторский перевод).

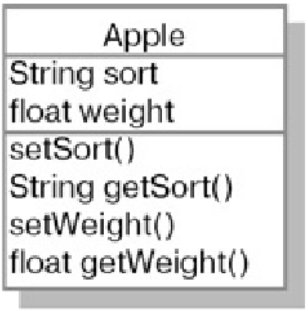
Классы

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 будут использоваться по-разному, 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), широко применяющихся в процедурном программировании; при этом каждый объект имеет тот же смысл, что и экземпляр структуры (переменная или константа соответствующего типа).

Формально класс - это шаблон поведения объектов определенного типа с заданными параметрами, определяющими состояние. Все экземпляры одного класса (объекты, порожденные от одного класса) имеют один и тот же набор свойств и общее поведение, то есть одинаково реагируют на одинаковые сообщения.



В соответствии с UML (Unified Modelling 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 также может 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.

Инкапсуляция

Инкапсуляция (encapsulation) - это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса). При использовании объектно-ориентированного подхода не принято применять прямой доступ к свойствам какого-либо класса из методов других классов. Для доступа к свойствам класса принято задействовать 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 лекции 6. Они позволяют более тонко управлять свойством инкапсуляции.

Открытые члены класса составляют внешний интерфейс объекта. Это та функциональность, которая доступна другим классам. Закрытыми обычно объявляются все свойства класса, а также 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. Это свойство называется модульность.

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.

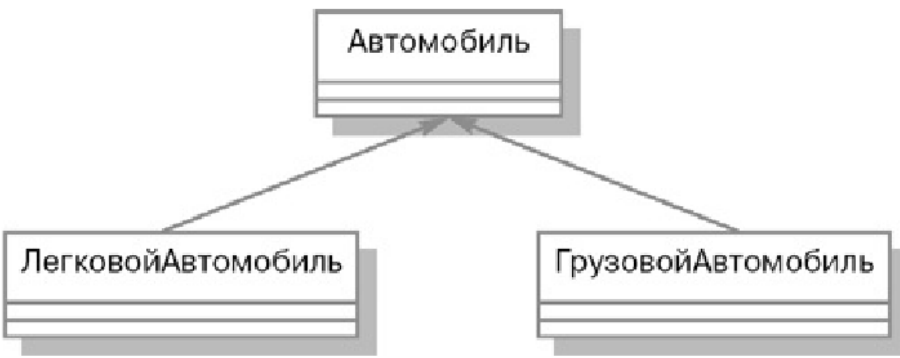
Во-вторых, не составит труда изменить способ хранения данных. Если информация станет храниться не в памяти, а в долговременном хранилище, таком как файловая система или база данных, потребуется изменить лишь ряд методов одного класса, а не вводить эту функциональность во все части системы.

Наконец, программный код, написанный с использованием данного принципа, легче отлаживать. Для того чтобы узнать, кто и когда 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 пришлось бы добавлять вывод отладочной информации во все участки кода, где используется интересующий нас объект.

Наследование

Наследование (inheritance) - это отношение между классами, при котором класс использует структуру или поведение другого класса (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 общую функциональность. Так, оба они имеют 4 колеса, двигатель, могут перемещаться и т.д. Всеми этими свойствами обладает любой автомобиль, независимо 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 конце. 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, созданного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода.

В рассмотренном примере применено одиночное наследование. Некоторый класс также 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 одиночное, за исключением того, что линии наследования соединяют класс-потомок сразу с несколькими суперклассами.

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

В языке Java множественное наследование имеет ограниченную поддержку через интерфейсы и будет рассмотрено в лекции 8.

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 оно означает применительно к объектно-ориентированному программированию, рассмотрим пример.

Предположим, мы хотим создать векторный графический редактор, в котором нам нужно описать 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 код, который при необходимости отобразить рисунок будет последовательно перебирать все примитивы, на момент отрисовки находящиеся на экране, и вызывать метод `draw` у каждого из них. Человек, не знакомый 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[i].draw();
}

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

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

for(int i = 0; i < b.length;i++) {
  if(b[i]!=null) b[i].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 существующий код и добавлять в него определения новых массивов, а также обработку содержащихся в них элементов.

Используя полиморфизм, мы можем значительно упростить реализацию подобной функциональности. Прежде всего, создадим общий родительский класс для всех наших классов. Пусть таким классом будет `Point`. В результате получим иерархию классов, которая изображена на рисунке 2.3.

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[i].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 [2].

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

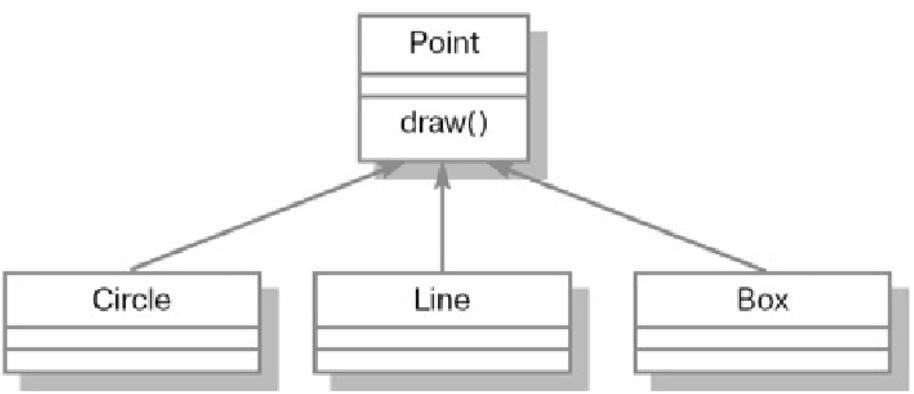


Рис. 2.3. Пример иерархии классов.

Примером использования перегрузки методов в языке Java может служить класс `PrintWriter`, который применяется, 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
void println(boolean x)
  // выводит значение булевской
  // переменной (true или false)
void println(String x)
  // выводит строку - значение
  // текстового параметра.
```

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

Типы отношений между классами

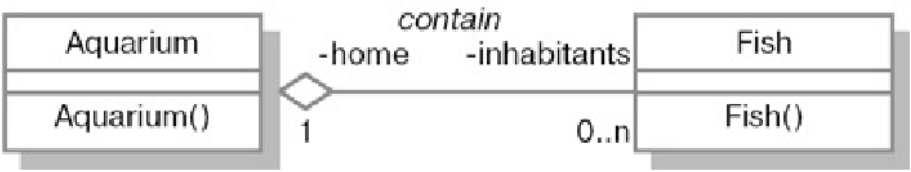
Как правило, любая программа, написанная на объектно-ориентированном языке, представляет собой некоторый набор связанных между собой классов. Можно провести аналогию между написанием программы и строительством дома. Подобно тому, как стена складывается из кирпичей, компьютерная программа с использованием ООП строится из классов. Причем эти классы должны иметь представление друг о друге, для того чтобы сообща выполнять 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 лекции):

- агрегация (`Aggregation`);
- ассоциация (`Association`);
- наследование (`Inheritance`);
- метаклассы (*`Metaclass`*).

Агрегация

Отношение между классами типа "содержит" (contain) 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. Каждый класс имеет свою роль в агрегации, которая указывает, какое место занимает класс в данном отношении. Имя роли не является обязательным элементом обозначений и может отсутствовать на диаграмме. 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 указано 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 также обозначения:

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

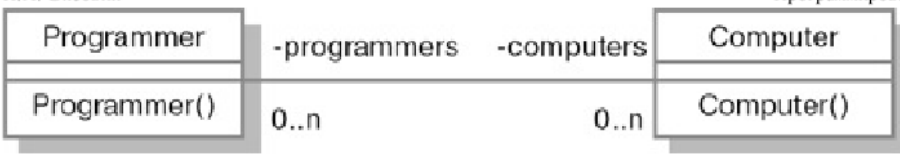
Код, описывающий рассмотренную модель и явление агрегации, может выглядеть, например, следующим образом:

```
// 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() {
   }
}
```

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, 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, в принципе, может не работать с 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() {
  }
}
```

Наследование

Наследование является важным случаем отношений между двумя или более классами. Подробно оно рассматривалось выше.

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 существуют ограничения на именование классов, как описываются поля 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, значит, его можно определить и для класса. Такой шаблон, задающий различные классы, называется метаклассом.

Чтобы представить себе, что такое метакласс, рассмотрим пример некой бюрократической организации. Будем считать, что все классы в такой системе представляют собой 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 только один. Но существуют языки программирования, в которых можно создавать и использовать собственные метаклассы, например язык Python. В частности, функциональность метакласса может быть следующая: при формировании класса он будет просматривать список всех методов в классе и, если имя метода имеет вид `set_XXX` или `get_XXX`, автоматически создавать поле с именем `XXX`, если такого не существует.

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

В языке Java также есть метакласс. Это класс, который так и называется - `Class` (описывает классы), он 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 лекциях.

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 разбить ее на небольшие, четко очерченные части. Чтобы справиться со сложностью, необходимо абстрагироваться от деталей. В этом смысле классы представляют собой весьма удобный инструмент.

- Классы позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что позволяет абстрагироваться от деталей реализации.
- Данные и операции над ними образуют определенную сущность, и они не разносятся по всей программе, как 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 и обновление версий отдельных компонентов.

ООП дает возможность создавать расширяемые системы. Это одно из

основных 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 многоразового использования программного обеспечения не удается добиться из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся компонентов, что позволяет извлечь максимум из многоразового использования компонентов.

- Сокращается время на разработку, которое может быть отдано другим задачам.
- Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
- Когда некий компонент используется сразу несколькими клиентами, улучшения, вносимые в его код, одновременно оказывают положительное влияние и на множество работающих с

ним программ.

- Если программа опирается на стандартные компоненты, ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает использование.

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 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 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, рассредотачивается 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 их намного больше. В коротких методах легче разобраться, но они неудобны тем, что код для обработки сообщения иногда "размазан" по многим маленьким методам.

Инкапсуляцией данных не следует злоупотреблять. Чем больше логики и 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.

Многие считают, что ООП является неэффективным. Как же обстоит дело в действительности? Мы должны проводить четкую 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 их поиска 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++ отправка 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 каждый раз при доступе к данным. Однако если инкапсуляция используется только там, где она необходима (т.е. в тех случаях, когда 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 информацию 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 необходимая дополнительная память выражается в одном указателе для объекта и в одном дескрипторе типа для

класса.

3. Излишняя универсальность. Неэффективность также может означать, что в программе реализованы избыточные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, они становятся мертвым грузом. Это не влияет на время выполнения, но сказывается на размере 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 компоновщику возможность удалять лишние методы. Такие интеллектуальные компоновщики уже существуют 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, надежность программного обеспечения и быстрота его написания часто бывает важнее, чем производительность.

Заключение

В этой лекции мы рассказали 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 - инкапсуляция, 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.

Лексика языка

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.

Кодировка

Технология Java как платформа, изначально спроектированная для Глобальной сети Internet, 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 т.д.), недостаточен. Поэтому для записи текста программы применяется более универсальная кодировка Unicode UTF-16.

Как известно, Unicode UTF-16 представляет символы кодом из 2 байт, описывая, таким образом, 65535 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 строчной, 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 web-сайт стандарта, где можно получить дополнительную информацию,— ссылка: http://www.unicode.org/.

Итак, используя простейшую кодировку ASCII, можно ввести произвольную последовательность символов Unicode. Далее будет показано, 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 ASCII -символов.

Анализ программы

Компилятор, анализируя программу, сразу разделяет ее на:

- пробелы (white spaces);
- комментарии (comments);
- основные лексемы (tokens).

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 при дальнейшей поддержке такого кода.

Для разбиения текста на строки в ASCII используется 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.

Комментарии

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

В Java комментарии бывают двух 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 находиться 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 в 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:
circle./*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 примере

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

описан только 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 во 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 (например, читая описание метода, можно с помощью одного нажатия мыши перейти на описание типов, используемых в качестве аргументов или возвращаемого значения). Однако понятно, что одного названия метода и перечисления его аргументов недостаточно 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 не использовать вообще, но они удобны, когда необходимо форматирование, скажем, в примерах кода.

```
/**
  * Первое предложение - краткое
  * описание метода.
  * <p>
  * Так оформляется пример кода:
  * <blockquote>
  * <pre>
  * if (condition==true) {
  *    x = getWidth();
  *    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 = getWidth();
  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 Internet-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 того, можно описать пакет (так называются библиотеки, или модули, в Java). Для этого необходимо создать специальный файл `package.html`, сохранить в нем комментарий и поместить его в каталог пакета. HTML-текст, содержащийся между тегами `<body>` и `</body>`, будет помещен в документацию, а первое предложение будет использоваться для краткой характеристики этого пакета.

Лексемы

Итак, мы рассмотрели 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 на компилированный код не влияют. 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.

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.

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 следующих лекциях).

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 лекции 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

Также допустимыми являются идентификаторы:

Computer, COLOR_RED, _, aVeryLongNameOfTheMethod

Ключевые слова

Ключевые слова – это зарезервированные слова, состоящие из ASCII -символов и выполняющие различные задачи языка. Вот их полный список (48 слов):

abstract double int strictfp
boolean else interface super

```
break     extends     long       switch
byte      final       native     synchronized
case      finally     new        this
catch     float       package    throw
char      for         private    throws
class     goto        protected  transient
const     if          public     try
continue  implements  return     void
default   import      short      volatile
do        instanceof  static     while
```

Ключевые слова `goto` и `const` зарезервированы, но не используются. Это сделано для того, чтобы компилятор мог правильно отреагировать на их использование в других языках. 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 следующих лекциях.

Литералы

Литералы позволяют задать в программе значения для числовых, символьных и строковых выражений, а также null- литералов . Всего в Java определено 6 видов литералов:

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

Рассмотрим их по отдельности.

Целочисленные литералы

Целочисленные литералы позволяют задавать целочисленные значения в десятичном, восьмеричном и шестнадцатеричном виде. Десятичный 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 начинается с `0x` или `0X` (цифра 0 и латинская ASCII -буква `X` в произвольном регистре). Таким образом, ноль можно записать тремя различными способами:

```
0
0X0
0x0
```

Как обычно, для записи цифр `10 - 15` в шестнадцатеричном формате используются буквы `A`, `B`, `C`, `D`, `E`, `F`, прописные или строчные. 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 -2^{31} до $2^{31}-1$, 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` ($=2^{31}-1$), с унарным оператором - допустимо также `-020000000000` ($= -2^{31}$). Аналогично для шестнадцатеричного формата – от `0x0` до `0x7fffffff` ($=2^{31}-1$), а также `-0x80000000` ($= -2^{31}$).

Тип `long` имеет длину 64 бита, а значит, позволяет хранить значения от -2^{63} до $2^{63}-1$. 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
0777777777777777777777L
0x7fffffffffffffffL
// наименьшие отрицательные значения:
-9223372036854775808L
-01000000000000000000000L
-0x8000000000000000L
```

Другие примеры целочисленных литералов типа `long`:

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

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;

- порядок (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 два возможных значения – латинская ASCII -буква `D` (для типа `double`) или `F` (для типа `float`) в произвольном регистре. Они 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` (используется 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 и еще несколько особых значений, а именно:

- положительная и отрицательная бесконечности (positive/negative infinity);
- значение "не число", Not-a-Number, 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 лекции.

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 того, 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 со 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.

Логические литералы

Логические литералы имеют два возможных значения – `true` и `false`. Эти два зарезервированных слова не являются ключевыми, но также не могут использоваться в качестве идентификатора.

Символьные литералы

Символьные литералы описывают один символ из набора Unicode, заключенный в одиночные кавычки, или апострофы (ASCII -символ single quote, `\u0027`). Например:

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

Также допускается 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
\шестнадцатеричный код  от \u0000 до \u00ff символа
                в шестнадцатеричном 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.

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, можно использовать оператор конкатенации строк +. Примеры строковых литералов:

```
// выражение-константа, составленное из двух
// 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 лекции.

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 лекции.

Разделители

Разделители – это специальные символы, которые используются в служебных целях языка. Назначение каждого из них будет рассмотрено по ходу изложения курса. Вот их полный список:

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

Операторы

Операторы используются в различных операциях – арифметических, логических, битовых, операциях сравнения и 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 лекции приводятся краткие комментарии к ним.

Пример программы

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. Результатом работы будет, как очевидно:

```
Hello, world!
```

Дополнение. Работа с операторами

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

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 сравнения всегда возвращает булевское значение true или false. Оператор присваивания возвращает значение правого операнда. Поэтому обычная опечатка в языке C, когда эти операторы путают:

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

в Java легко устраняется. Поскольку выражение `x=0` имеет числовое значение 0, а не булевское (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;
```

Арифметические операции

Наряду с четырьмя обычными арифметическими операциями `+`, `-`, `*`, `/`, существует 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 два 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`, это 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`, и лишь затем произойдет инкрементация. В результате значение `x` будет равно `2`, а `y` будет равно `1`.

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), а 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
```

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 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 $2^8=256$ 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 два 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
& 00000110
------------
 00000100

   // число 5 в двоичном виде
   // число 6 в двоичном виде

   //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 битовом 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 2^n 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 << 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 << 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
```

Очевидно, 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 2^n. Причем, если для положительных аргументов с ростом `n` результат закономерно стремится к `0`, то для отрицательных предельным значением является `-1`.

Заключение

В этой лекции были рассмотрены основы лексического анализа программ Java. Для их записи применяется универсальная кодировка Unicode, позволяющая использовать любой язык помимо традиционного английского. 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 также можно задать в виде специальной последовательности ASCII -символов.

Во время анализа 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.

Типы данных

Типы данных определяют основные возможности любого языка. Кроме того, Java является строго типизированным языком, а потому четкое понимание модели типов данных очень 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 играет роль метакласса в Java.

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 точное место (строку) и причину ее возникновения, а динамические "баги" (от английского bugs) необходимо сначала выявить с помощью тестирования (что может потребовать значительных усилий), а затем найти место в коде, которое их породило. Поэтому четкое понимание модели типов данных в Java очень 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
- булевые

- 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 сот 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.

Переменные

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

- имя;
- тип;
- значение.

Имя уникально идентифицирует переменную и позволяет обращаться к ней 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 исполнения программы. В частности, можно пользоваться уже объявленными переменными. Далее можно поставить запятую и объявить новую переменную точно такого же типа.

После объявления переменная может применяться 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;
```

Примитивные и ссылочные типы данных

Теперь на примере переменных можно проиллюстрировать различие

между примитивными и ссылочными типами данных. Рассмотрим пример, когда объявляются две переменные одного типа, приравниваются друг другу, а затем значение одной из 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 (неважно, 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 данного значения. Чтобы еще раз подчеркнуть эту особенность, приведем еще один пример:

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

В данном примере происходит преобразование типов (оно подробно рассматривается в соответствующей лекции). 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 разные 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.

Также 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 недоступен программе и является "мусором", то есть без толку 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.

Примитивные типы

Как уже говорилось, существует 8 простых типов данных, которые

делятся на целочисленные (integer), дробные (floating-point) и булевы (boolean).

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, это 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 – понятия взаимооднозначные. 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. В таблице 4.1 сведены данные по всем разобранным типам:

Таблица 4.1. Целочисленные типы данных.

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 10^{19})
`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 были выделены именно 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 булево 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 отдельно мы их рассматривать не будем. Их результат всегда булева типа (`true` или

`false`).

Работа числовых операторов также понятна, к тому же пояснялась в предыдущей лекции. 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 имеет место в том 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 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`, мы видим, что математический результат равен в точности $+2^{31}$, или, в двоичном

формате, `1000 0000 0000 0000 0000 0000 0000 0000` (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. лекцию 3). 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 того, числовые операции в Java обладают еще одной особенностью. Хотя целочисленные типы имеют длину 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.

Никакого способа 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 лекциях. 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 для программиста 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 так же некорректна такая инициализация:

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

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 существует два исключения.

Первое из них – операторы инкрементации и декрементации. Их действие заключается в прибавлении 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 переменной результата 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 того, 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 `%`).

Дробные типы

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

Таблица 4.2. Дробные типы данных.

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 булево 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, которые предусмотрены для целочисленных операторов (оператор деления с остатком `%` рассматривался в предыдущей лекции, 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 лекции 3:

- положительная и отрицательная бесконечности (positive/negative infinity);
- значение "не число", Not-a-Number, 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 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. Так же поступают и в том случае, когда количество десятичных знаков превышает допустимое:

```
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. лекцию 3). 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).

Приведение типов подробно рассматривается в другой лекции, 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, несмотря 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 того, 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`.

Булев тип

Булев 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 булевыми аргументами можно производить следующие операции:

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

Логические операторы `&&` и `||` обсуждались в предыдущей лекции. В операторе с условием `?:` первым аргументом может быть только значение типа `boolean`. Также допускается, чтобы второй и третий аргументы одновременно имели булев тип.

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

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

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

Ссылочные типы

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

Объекты и правила работы с ними

Объект (object) – это экземпляр некоторого класса, или экземпляр массива. Массивы будут подробно рассматриваться в соответствующей лекции. 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 приведенных выше примерах, когда создавались объекты типа `Point`, выражение `new Point (3,5)` означало обращение к конструктору класса `Point`, у которого есть два аргумента типа `int`. Кстати, обязательное объявление такого конструктора в упрощенном объявлении класса отсутствовало. Объявление классов рассматривается в следующих лекциях, 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 ссылок больше нет, 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 три объекта класса `String`. Два объекта порождаются строковыми литералами, третий будет представлять результат конкатенации.

Операция создания объекта – одна из самых ресурсоемких в Java. Поэтому следует избегать ненужных порождений. Поскольку 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 того, 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 ее применения:

```
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 булево значение)
- операции сравнения == и != (возвращают булево 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 ее применения будут приводиться.

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 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 касается 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 том 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 на тот же объект, что и `p1`. Второе же сравнение ложно, несмотря на то, что переменная `p3` ссылается на объект-точку с точно такими же 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 можно сравнивать следующим образом:

```
String s = "abc";
s=s+1;
print(s.equals("abc1"));
```

Операция с условием `?:` работает как обычно и может принимать второй и третий аргументы, если они оба одновременно ссылочного типа. Результат такого оператора также будет иметь объектный тип.

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.

Класс Object

В Java множественное наследование отсутствует. 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()`, возвращающий имя класса:

```
String s = "abc";
Class cl=s.getClass();
System.out.println(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 `false`.

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 убедиться, 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()

Данный 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 различные хэш-коды, но, по крайней мере, необходимо, 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-кодов, основываясь на расположении объекта в памяти.

toString()

Этот метод позволяет получить текстовое описание любого объекта. Создавая новый класс, данный 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()

Данный метод вызывается при уничтожении объекта автоматическим сборщиком мусора (garbage collector). В классе `Object` он ничего не делает, однако в классе-наследнике позволяет 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 удаления объекта JVM сделает сама.

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 с помощью оператора конкатенации строк, который был рассмотрен для каждого типа, как примитивного, так и объектного.

Еще одним важным свойством данного класса является неизменяемость. Это означает, что, породив объект, содержащий некое значение-строку, мы уже не можем изменить данное значение – 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, зачастую компилятор стремится оптимизировать эту работу.

Во-первых, если используется несколько литералов с одинаковым значением, для 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 как 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()`.

Класс Class

Наконец, последний класс, который будет рассмотрен в этой лекции.

Класс `Class` является метаклассом для всех классов Java. Когда JVM загружает файл `.class`, который описывает некоторый тип, в памяти создается объект 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. объект типа `Point`, описывающий точку `(1,2)`;
2. объект класса `Class`, описывающий класс `Point`;
3. объект класса `Class`, описывающий класс `Object`. Поскольку класс `Point` наследуется от `Object`, его описание 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.).

Заключение

Типы данных – одна из ключевых тем курса. Невозможно написать ни одной программы, не используя их. Вот список некоторых операций, 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 будут рассматриваться и дальше по ходу курса. Изучение объектной модели Java даст основу для более подробного изложения объектных типов – обычных и абстрактных классов, интерфейсов и массивов. После 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.

В лекции было рассказано о том, 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 понимание программистами объектной модели.

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. Они также 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 объектов.

Были рассмотрены переменные. Они характеризуются тремя основными параметрами – имя, тип и значение. Любая переменная должна быть объявлена и при этом может быть инициализирована. Возможно использование модификатора `final`.

Примитивные типы состоят из пяти целочисленных, включая символьный тип, двух дробных и одного булевого. 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`.

Имена. Пакеты

В этой лекции рассматриваются 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 возникновения подобных 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.

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 имеются 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. Так же объявляются конструкторы

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.

Имена

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 содержащиеся в нем классы и интерфейсы, 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. 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 предыдущих главах.

Имена и идентификаторы

Теперь, когда мы рассмотрели 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, он становится именем после первого появления в коде в месте объявления.

Во-вторых, существует 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. Пример такого вызова:

```
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, показывающий, 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 != 1) {
      if (num%n==0) {
        continue num;
      }
    }
    System.out.print(num+" ");
  }
```

Результатом будут простые числа меньше `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
```

Мы видим, что здесь 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 со множеством переменных, 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.

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, например, к полю, идет из части кода, попадающей в область видимости его имени, то можно пользоваться простым именем, если нет – 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 выясним, что такое пакеты, как и для чего они используются.

Пакеты

Программа на Java представляет собой набор пакетов (packages). Каждый пакет может включать вложенные пакеты, то есть они образуют иерархическую систему.

Кроме того, пакеты могут содержать классы и интерфейсы и таким образом группируют типы. Это 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 показано 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, заметно усложняет развитие и поддержку такой системы. Используя пакеты, гораздо проще организовать эффективное 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.

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 пакеты `applet`, `awt`, `io`, `lang`, `net`, `util` и не содержал ни одного типа. В пакет `java.awt` входил вложенный пакет `image` и 46 классов и интерфейсов.

Составное имя любого элемента пакета – это составное имя этого пакета плюс простое имя 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 одного из этих пакетов не имеют никакого особенного доступа к типам других пакетов.

Платформенная поддержка пакетов

Простейшим способом организации пакетов и типов является обычная файловая структура. Рассмотрим выразительный пример, когда все пакеты, исходный и бинарный код располагаются в одном каталоге и его подкаталогах.

В этом корневом каталоге 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 корневом каталоге должна быть папка `space`, соответствующая 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 слэш \). Такое преобразование может выполнить 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 каталогов выше, чтобы Java, построив путь из имени пакета, смогла обнаружить нужный файл.

Кроме того, немаловажно, что Java всегда различает регистр идентификаторов, а значит, названия файлов и каталогов должны точно отвечать запрограммированным именам. Хотя в некоторых случаях операционная система может обеспечить доступ, невзирая на регистр, 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 пустой каталог, или заполнить его посторонними файлами, 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 описывающий его файл недоступен текущему пользователю ОС для чтения, для Java этот тип и этот пакет не будут существовать.

Понятно, что далеко не всегда удобно хранить все файлы в одном каталоге. Зачастую классы находятся в разных местах, а некоторые могут даже распространяться в виде архивов, для ускорения загрузки через сеть. Копировать все такие файлы в одну папку было бы крайне затруднительно.

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 каталогам 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 каталоги и содержимое всех архивов "добавляется" к исходному корневому каталогу. 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 указан текущий каталог (представлен точкой). Это делается для того, чтобы поиск всегда начинался с исходного корневого каталога. Конечно, такая запись не является обязательной и делается на усмотрение разработчика.

Несмотря на явные удобства такой конструкции, она таит в себе и опасности. Если разрабатываемые классы хранятся в некотором каталоге и он указан в `classpath` позже, чем некий другой каталог, в котором обнаруживаются одноименные типы, 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 данной переменной среды окружения необходимо относиться с особым 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, можно применять и альтернативные подходы к хранению пакетов и файлов с исходным и бинарным кодом. Например, в качестве такого хранилища может использоваться 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 преобразований из/в файлы.

Модуль компиляции

Модуль компиляции (compilation 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. Если данное выражение отсутствует, значит, 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, эта часть также является необязательной, в том смысле, что в случае ее отсутствия компилятор не выдаст ошибки. Однако никаких `.class` -файлов сгенерировано тоже не будет.

Доступность модулей компиляции определяется поддержкой платформы, т.к. утилиты Java являются обычными программами, которые исполняются операционной системой по общим правилам.

Рассмотрим все три части более подробно.

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;
```

Это одновременно служит объявлением пакета `lang`, вложенного в пакет `java`, и указанием, что объявляемый ниже класс `Object` находится в данном 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.

Импорт-выражения

Как будет рассмотрено ниже, область видимости объявления типа - пакет, в котором он располагается. Это означает, что внутри данного 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, 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 для 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 те, что импортируют неиспользуемые пакеты и типы, но это никак не отразится ни на размере, ни на качестве бинарного кода. Также безразлично, обращаться к типу по его полному имени, или включить его в импортирующее выражение и обращаться по простому имени – результат будет один и тот же.

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 возможен и обычный доступ по полному имени.

Выражение, импортирующее один тип, записывается 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 дублированные 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, дублированные выражения игнорируются. Обратите внимание, что импортировать вложенный пакет нельзя.

Например:

```
// пример вызовет ошибку компиляции
import java.awt.image;
```

Создается впечатление, что теперь мы можем обращаться к типам пакета `java.awt.image` по упрощенному имени, например, `image.ImageFilter`. На самом деле пример вызовет ошибку компиляции, так как данное выражение расценивается как импорт типа, а в пакете `java.awt` отсутствует тип `image`.

Аналогично, выражение

```
import java.awt.*;
```

не делает более доступными классы пакета `java.awt.image`, их необходимо импортировать отдельно.

Поскольку пакет `java.lang` содержит типы, без которых невозможно создать ни одну программу, он неявным образом импортируется в каждый модуль компиляции. Таким образом, все типы из этого пакета доступны по их простым именам без каких-либо дополнительных усилий. Попытка импортировать данный пакет еще раз будет проигнорирована.

Допускается одновременно импортировать пакет и какой-нибудь тип из него:

```
import java.awt.*;
import java.awt.Point;
```

Может возникнуть вопрос, как же лучше поступать – импортировать типы по отдельности или весь пакет сразу? 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 в текущем модуле компиляции;
- наконец, просматриваются выражения, импортирующие пакеты.

Таким образом, если тип явно импортирован, 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 использоваться при создании объекта? Результатом будет:

```
java.awt.Point[x=0,y=0]
```

В соответствии с 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 важные программы в именованных пакетах.

Теперь рассмотрим импорт пакета. Его еще называют "импорт по

требованию", подразумевая, что никакой "загрузки" всех типов импортированного пакета сразу при указании импортирующего выражения не происходит, их полные имена подставляются по мере использования простых имен в коде. Можно импортировать пакет и задействовать только один тип (или даже ни одного) из него.

Изменим рассмотренный 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 выполнять поиск по пакету `java.awt`. Второй объект порождается от класса `Rectangle`, которого не существует в текущем пакете, зато он обнаруживается в `java.awt`.

Также корректен теперь пример:

```
package my_geom;

import java.awt.*;

class Point {
}
```

Таким образом, импорт пакета не препятствует объявлению новых типов 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 того, 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.

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, хранящиеся 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. Данная возможность является вспомогательной и довольно запутанной, поэтому в курсе подробно не рассматривается, 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 этом:

- тип объявлен как `public` и, значит, может использоваться из других пакетов;
- 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 применяется БД, то такое ограничение не должно накладываться.

На практике же программисты зачастую помещают в один модуль

компиляции только 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 том же каталоге. Именно так устроены все стандартные библиотеки Java.

Обратите внимание, что при объявлении классов вполне допускаются перекрестные обращения. В частности, 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 раньше, чем был объявлен. Такое перекрестное применение типов также допускается в случае, если они находятся в разных пакетах. Компилятор должен поддерживать возможность транслировать их одновременно.

Уникальность имен пакетов

Поскольку Java создавался как язык, предназначенный для распространения приложений через Internet, а приложения состоят из

структуры пакетов, 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 Internet-сайт, либо же он работает на организацию, у которой имеется 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, фамилии разработчиков, 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`, `int` (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, 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 Internet-сайте, который дал им имя. Скорее была сделана попытка воспользоваться существующей системой имен вместо того, чтобы создавать новую для именования библиотек.

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 имеет смысл собрать эту информацию в одном месте.

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 пакету `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, 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 доступны 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 недоступна,
// так как блок, в котором она была
// объявлена, уже завершен, а переменная
// x еще недоступна, так как пока не была
// инициализирована
```

Определенные проблемы возникают, когда происходит перекрытие областей видимости и возникает конфликт имен различных конструкций языка.

"Затеняющее" объявление (Shadowing)

Самыми распространенными случаями возникновения конфликта имен является выражение, импортирующее пакет, и объявление локальных переменных, или параметров методов, конструкторов, обработчиков ошибок. Импорт пакета подробно рассматривался в этой главе. Если импортированный и текущий пакеты содержат одноименные типы, то их области пересекаются. Как уже говорилось, предпочтение отдается типу из текущего пакета. Также рассказывалось о том, как эту проблему решать.

Перейдем к проблеме перекрытия имен полей класса и локальных переменных. Пример:

```
class Human {
  int age;
  // возраст
  int getAge() {
    return age;
```

```
    }
    void setAge(int age) {
      age=age; // ???
    }
  }
```

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 называться `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 разные переменные через одно имя успехом не увенчается. 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 в такой ситуации все же обратиться к полю класса. Если доступ по простому имени невозможен, надо воспользоваться составным. Здесь удобнее всего применить специальное ключевое слово `this` (оно будет подробно рассматриваться в следующих главах). Слово `this` имеет значение ссылки на объект, внутри которого оно применяется. Если вызвать метод `setAge()` у объекта класса `Human` и использовать в этом методе слово `this`, то его значение будет ссылкой на данный 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 с помощью тестирования или контрольного просмотра кода.

"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 пол класса `Obscuring` и имя другого типа, находящегося в том же пакете,– `Test`. С помощью этого имени происходит 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 объявление одноименной переменной или типа, имеющее более высокий приоритет. Такое объявление называется "заслоняющим" (obscuring).

Эта проблема скорее всего не возникнет, если следовать соглашениям по именованию элементов языка Java.

Соглашения по именованию

Для того, чтобы код, написанный на Java, было легко читать и понять 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 там, 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 стремиться 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 границы допустимого.)

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 встречается редко, так как имена пакетов и локальных переменных (параметров) начинаются с маленькой буквы, а типов – с большой.

Имена методов должны быть глаголами и обозначать действия, которые совершает данный метод. Имя должно начинаться с маленькой буквы, но может состоять из нескольких слов, причем 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, соответственно, с `get` и `set`, например, для переменной `size` это будут `getSize()` и

`setSize()`;

- метод, возвращающий длину, называется `length()`, например, в классе `String`;
- имя метода, который проверяет булевское условие, начинается с `is`, например, `isVisible()` у компонента графического пользовательского интерфейса;
- метод, который преобразует величину в формат `F`, называется *`toF`*`()`, например, метод `toString()`, который приводит любой объект к строке.

Вообще, рекомендуется везде, где возможно, называть методы похожим образом, как в стандартных классах Java, чтобы они были 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 Internet-сайтов.

Заключение

В этой главе был рассмотрен механизм именования элементов языка. Для того чтобы различные части большой системы не зависели друг от друга, вводится понятие " область видимости имени", вне которой необходимо использовать не простое, а составное имя. Затем были изучены элементы (members), которые могут быть у пакетов и ссылочных типов. Также рассматривалась 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 для более точного разграничения области видимости. Пакет содержит вложенные пакеты и типы (классы и интерфейсы). Вопрос о платформенной поддержке пакетов привел к рассмотрению модулей компиляции как текстовых файлов, поскольку именно в виде файлов и каталогов, 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 изучены 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 возникновения ошибок и двусмысленных ситуаций в программе.

Описание области видимости для различных элементов языка приводит к вопросу о возможных перекрытиях таких областей и, как следствие, о конфликтах имен. Рассматриваются "затеняющие" и "заслоняющие" объявления. Для устранения или уменьшения возможности возникновения таких ситуаций описываются соглашения по именованию для всех элементов языка.

Объявление классов

Центральная тема лекции – 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.

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 данная тема разделена на две лекции.

Эта лекция начинается с продолжения темы прошлой лекции – 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.

Лекция 8 подробно рассматривает особенности объектной модели Java. Вводится понятие интерфейса. Уточняются правила объявления классов и описывается объявление интерфейса.

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 использоваться 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.

Предназначение модификаторов доступа

Очень часто права доступа расцениваются 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 намеренно или по незнанию может присвоить этому полю отрицательное значение, после чего объект станет работать неправильно, могут появиться ошибки. Для защиты такого поля age необходимо объявить его private.

Это довольно распространенная точка зрения, однако нужно признать, что она далека от истины. Основным смыслом разграничения 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, логично предположить, что все или многие из них начнут использовать это поле.

Может получиться так, что целочисленного типа данных будет уже недостаточно и захочется сменить тип поля на дробный. Однако если просто изменить `int` на `double`, 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, подобных этим:

```
Human h = getHuman();   // получаем ссылку
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, 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 данным классом могло бы работать множество программистов и могло 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 совсем несложной операцией, связанной с изменением только в 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 многих местах, 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 того, чтобы можно было 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 кода.

За счет чего была достигнута такая гибкость? Необходимо выделить свойства объекта, которые потребуются будущим пользователям этого класса, и сделать их доступными (в данном случае, `public`). Те же элементы класса, что содержат детали внутренней реализации логики класса, желательно скрывать, чтобы не образовались нежелательные зависимости, которые могут сдерживать 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 установке верных значений полей. 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 игнорируются, 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 можно начать хранить 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 изменить код этих методов, а наружние типы, которые использовали данный 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 применяется 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 применения класса, то его создателям нужно стремиться к тому, чтобы класс был прост в применении, тогда таких проблем не возникнет, ведь программист не станет намеренно 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 два (`public` и `private`), а четыре уровня доступа. Рассмотрим их и весь механизм разграничения доступа в Java более подробно.

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 напрямую 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 случае, если недоступен тип, на основе которого он объявлен.

Все четыре уровня доступа имеют только элементы типов и конструкторы. Это:

- public;
- private;
- protected;
- если не указан ни один из этих трех типов, то уровень доступа определяется по умолчанию (default).

Первые два из них уже были рассмотрены. Последний уровень (доступ по умолчанию) упоминался в прошлой лекции – он допускает 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 приходится 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 допускает 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 открытым):

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 доступны всегда, поэтому у них нет модификаторов доступа (можно сказать, что все они `public`, то есть любой существующий в системе пакет может использоваться из любой точки программы).
- Типы (классы и интерфейсы) верхнего уровня объявления. При их объявлении существует 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 четырьмя возможными значениями уровня доступа. Все элементы интерфейсов являются `public`.

Для типов объявления верхнего уровня нет необходимости во всех четырех 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. Импортировать недоступные типы запрещается.

Проверка уровня доступа проводится компилятором. Обратите внимание на следующие примеры:

```
public class Wheel {
   private double radius;
   public double getRadius() {
      return radius;
   }
}
```

Значение поля `radius` недоступно 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`. Данный метод возвращает экземпляр класса `Child`, который недоступен 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 выполнить 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 `instanceof`), в случае равенства координат возвращалось истинное значение. Обратите внимание на строку, где делается сравнение координат,– для этого приходится обращаться к `private` -полям другого объекта!

Тем не менее, такое действие 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.

Объявление классов

Рассмотрим базовые возможности объявления классов.

Объявление класса состоит из заголовка и тела класса.

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, представляют собой `final` -классы.

После списка модификаторов указывается ключевое слово `class`, а затем имя класса – корректный Java-идентификатор. Таким образом, кратчайшим объявлением класса может быть 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` ;
- существует класс `C`, который является наследником `B`, а `A` является наследником `C` (это правило применяется рекурсивно).

Таким образом можно проследить цепочки наследования на несколько уровней вверх.

Если компилятор обнаруживает, что класс является своим наследником, 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.

Тело класса

Тело класса может содержать объявление элементов (members) класса:

- полей;
- 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 четырех модификаторов доступа. Напоминаем, что соглашения по именованию классов и их элементов обсуждались в прошлой лекции.

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 не-элементы 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 по наследству не передаются.

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.

Объявление полей

Объявление полей начинается с перечисления модификаторов. Возможно применение любого из трех модификаторов доступа, либо никакого вовсе, что означает уровень доступа по умолчанию.

Поле может быть объявлено как `final`, это означает, что оно инициализируется 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. Существует еще два специальных модификатора - *`transient`* и `volatile`. Они будут рассмотрены в соответствующих лекциях.

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 – `0` ;
- для булевского типа – `false` ;
- для ссылочных – `null`.

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`.

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 трех возможных модификаторов доступа. Также допускается использование доступа по умолчанию.

Кроме того, 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`.

Также 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 для Java необходима, 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, таких, как переносимость, безопасность и другие. Поэтому применять JNI следует 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.

Наконец, существует еще один специальный модификатор `synchronized`, который будет рассмотрен в лекции, 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 ввести ключевое слово `final` перед указанием его типа. В этом случае такой параметр не может менять своего значения в теле метода (то есть 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!
}
```

О том, как происходит изменение значений аргументов метода,

рассказано в конце этой лекции.

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 в одном классе друг с другом несовместимы:

```
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 появиться в объявлении тела класса. Можно 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 может 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.

Точно так же в предыдущем примере вторая пара методов различается именем аргументов, которые также не входят в определение сигнатуры и не позволяют определить, какой из 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 лекции.

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 показано в многочисленных примерах в этой и других лекциях. 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 верным:

```
public int get() {
    if (condition) {
        return 5;
    } else {
        return 3;
    }
}
```

Конечно, значение, указанное после слова `return`, должно быть совместимо по типу с объявленным возвращаемым значением (это понятие подробно рассматривается в лекции 7).

В методе без возвращаемого значения (указано `void`) также можно использовать выражение `return` без каких-либо аргументов. Его можно указать в любом месте метода и 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 заметно усложнить понимание логики его работы.

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 недопустимы). Во-вторых, 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. Завершает заголовок конструктора throws-выражение (в примере не использовано, см. лекцию 10 "Исключения"). Оно имеет особую важность для конструкторов, поскольку сгенерировать ошибку – это для конструктора единственный способ 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 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 инициализация объекта, 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 имеется по два 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 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 (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 завершении 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=ny;
        }
  }

  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 такого конструктора нет, 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 все их описать заново.

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 Two.
   // Выражение new Two() ошибочно!
   public Two(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
  }
}
```

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 увеличится на один шаг (скажем, добавится проверка длины на равенство нулю), то такое изменение надо будет внести только в первый конструктор. Такой подход помогает избежать 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.

```
public class Test {
  public Test() {
```

```
        System.out.println("Test()");
    }

    public Test(int x) {
        this();
        System.out.println("Test(int x)");
    }
}
```

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 генерации 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.

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, 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:

- если инициализировать поле при объявлении;
- если инициализировать поле только один раз в инициализаторе объекта (он должен быть записан после объявления поля);
- если инициализировать поле только 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 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)
```

Дополнительные свойства классов

Рассмотрим в этом разделе некоторые особенности работы с классами в Java. Обсуждение данного вопроса будет продолжено в специальной лекции, посвященной объектной модели в Java.

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 лекции 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 существует 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 указываться никакого расширения (`.class` или `.java`) и расположение класса записывается через точку (разделитель имен пакетов), а не с помощью файлового разделителя. Компилятору же, напротив, передается имя и путь к файлу.

Если приведенный выше модуль компиляции сохранен в файле `Test.java`, который лежит в каталоге `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!
```

Параметры методов

Для лучшего понимания работы с параметрами методов в Java необходимо рассмотреть несколько вопросов.

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 изменение 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 doubler(int x) {
  return x+x;
}

public void test() {
  int x=3;
  x=doubler(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 в качестве значения 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, 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)`).

Перегруженные методы

Перегруженными (overloaded) 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 тому или иному методу.

Однако если количество параметров одинаковое, а типы их различаются незначительно, 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");
```

Очевидно, что для первых двух вызовов подходит 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, поэтому возникнет ошибка компиляции. Необходимо, используя явное приведение, указать компилятору, какой метод следует применить:

```
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);
```

Заключение

В этой лекции 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 четыре модификатора доступа, а также возможность их применения для различных элементов языка. Проверка уровня доступа выполняется уже во время 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

конструкторы и инициализаторы 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.

Классы Java мы продолжим рассматривать в следующих лекциях.

Преобразование типов

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 программы. С другой стороны, некоторые безопасные переходы между типами Java позволяет осуществлять неявным для разработчика образом, что может привести к неверному пониманию работы программы. 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 подробнее рассматриваться в следующих лекциях.

Введение

Как уже говорилось, Java является строго типизированным языком, а это 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 a=3;
a = 5+'A'+a;
print("a="+Math.round(a/2F));
```

Рассмотрим, 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 будет преобразован так, чтобы

операция проводилась с точностью в 32 бита. Второй оператор сложения опять потребует преобразования, так как наличие переменной `a` 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 преобразовать в текст, чтобы осуществить дальнейшую конкатенацию строк. Как будет показано 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 преобразования, 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:

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

Здесь во второй строке необходимо провести явное преобразование, чтобы присвоить значение типа `int` переменной типа `byte`. В третьей же строке обратное приведение производится автоматически, неявным для разработчика образом.

Рассмотрим сначала, какие переходы между различными типами можно осуществить.

Виды приведений

В Java предусмотрено 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.

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.

Во-вторых, иногда в Java могут встречаться 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 без труда справится с такой задачей, однако разработчику будет нелегко проследить всю цепочку. В этом случае можно воспользоваться тождественным преобразованием, выполнив приведение к точно такому же типу. Это ничего не изменит в структуре программы, но значительно облегчит чтение кода:

```
print((MyFlatImpl)(getCity().getStreet().getHouse().getFlat()));
```

Преобразование примитивных типов (расширение и сужение)

Очевидно, что следующие четыре вида приведений легко представляются в виде таблицы 7.1.

Таблица 7.1. Виды приведений.

простой тип, расширение	ссылочный тип, расширение
простой тип, сужение	ссылочный тип, сужение

Что все это 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 a=b;
```

В последней строке значение переменной `b` типа `byte` будет преобразовано к типу переменной `a` (то есть, `int`) автоматически, никаких специальных действий для этого предпринимать не нужно.

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`, в отличие от остальных целочисленных типов, является беззнаковым.

Тем не менее, следует помнить, что даже при расширении данные все-таки могут быть в особых случаях искажены. Они уже рассматривались в предыдущей лекции, 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 разрядов.

Повторим этот пример:

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

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 потерять данные.

Следующие 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 действий 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 противоположных чисел (384 и -384) оказался одинаковым. Следовательно, 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 до целочисленного 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 величиной, но в результате округления получилось слишком большое по модулю число для выбранного типа (т.е. для `int` или `long`), то, 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 это, как было описано, просто за счет отбрасывания 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` (31 единица). Отбрасывая старшие биты, получаем для отрицательной бесконечности результат `0`, одинаковый для всех трех 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 `65535`.

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.

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.

Три объявленных класса могут порождать три 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();
```

Обратите внимание, что с помощью такой ссылки `p` можно обращаться лишь к полю `x` созданного объекта. Поле `y` недоступно, 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 вызовет ошибку компиляции.

Аналогично, объекты класса `Child2` обладают полем `z` и полем `x`, полученным по наследству от класса `Parent`. Значит, 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 проходит успешно: 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.

Обратите внимание, что при подобном преобразовании с самим объектом ничего не происходит. Несмотря на то, что, например, поле `y` класса `Child` теперь недоступно, это не означает, что оно исчезло. Такое существенное изменение структуры объекта невозможно. Он был порожден от класса `Child` и сохраняет 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.

Преобразование к строке

Это преобразование уже не раз упоминалось. Любой тип может быть приведен к строке, т.е. к экземпляру класса `String`. Такое 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 снаружи незаметны, 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"`.

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, кроме `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. Это исключает вероятность того, что объект будет совместим с типом `Child2`.

Этим список запрещенных преобразований не исчерпывается. Он довольно велик, и в то же время все варианты достаточно очевидны, поэтому подробно рассматриваться не будут. Желающие могут получить полную информацию из спецификации.

Разумеется, попытка осуществить запрещенное преобразование вызовет ошибку компиляции.

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. Так же 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.

Присвоение значений

Такие ситуации неоднократно применялись в этой лекции 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 a=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 a=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 выполнять преобразование явно:

```
int i=10;
short s=(short)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, 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;
```

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 лекциях 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 те же шаги, что и при приведении в процессе присвоения значений переменным. Если типы образуют запрещенное преобразование, возникнет ошибка.

```
// пример вызовет ошибку компиляции

void calculate(long a) {
  ...
}

void main() {
  calculate(new Long(5));
  // здесь будет ошибка
}
```

Если сужение, то компилятор не сможет осуществить приведение и потребуются явные указания.

```
void calculate(int a) {
  ...
}

void main() {
  long a=5;
  // calculate(a);
```

```
    // сужение! так будет ошибка.
    calculate((int)a);   // 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 прилагать дополнительные усилия, чтобы указать, какой из двух возможных перегруженных методов хотелось бы вызвать.

Аналогичное преобразование потребуется 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 те же 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 a =  5L;
  // приведение при присвоении значения
  get(new Child());
  // приведение при вызове метода
}
```

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 всегда выполняются успешно. Последние два могут стать причиной ошибки исполнения, если значения оказались несовместимыми. Как следствие, выражение `null`

всегда может быть успешно преобразовано к любому ссылочному типу. Но можно 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.

Оператор конкатенации строк

Этот оператор уже рассматривался достаточно подробно. 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 общего ряда.

Правила преобразования уже были подробно описаны в этой лекции, а оператор конкатенации рассматривался в лекции "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
```

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.

Унарное числовое расширение

Это преобразование расширяет примитивные типы `byte`, `short` или `char` до типов `int` по правилам расширения примитивных типов.

Унарное числовое расширение может выполняться при следующих операциях:

- унарные операции `+` и `-` ;
- битовое отрицание `~` ;
- операции битового сдвига `<<`, `>>`, `>>>`.

Операторы сдвига имеют два аргумента, но они расширяются независимо друг от друга, поэтому данное 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 лекциях.

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 выполняться 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 лекциях.

Тип переменной и тип ее значения

Теперь, когда были подробно рассмотрены все примеры преобразований, нужно вернуться 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 иметь.

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 `m` выполнится расширение 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 лишь на объекты, порожденные от тех или иных классов, и далее обсуждаются только свойства данных классов. (Также объекты могут порождаться от массивов, эта тема рассматривается в отдельной лекции.)

Кроме того, 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 значением? 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 гарантированно 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 возможностями 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 никаких изменений с самим объектом не происходит, ограничение порождается используемым способом доступа к этому объекту - переменной типа `Parent`.

Чтобы показать, что объект не потерял никаких свойств, произведем следующее обращение:

```
((Child)p).newChildMethod();
```

Здесь в начале проводится явное сужение к типу `Child`. Во время исполнения программы JVM проверит, совместим ли тип объекта, на который ссылается переменная `p`, 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 коротко обобщим то, что было рассмотрено в данном разделе.

Таблица 4.1. Целочисленные типы данных.

Тип переменной	Допустимые типы ее значения
Примитивный	В точности совпадает с типом переменной
Ссылочный	• null • совпадающий с типом переменной

	• классы-наследники от типа переменной
Object	• null • любой ссылочный

Заключение

В этой лекции 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.

Особенным в Java является преобразование к строке.

Затем были рассмотрены все ситуации в программе, где могут происходить преобразования типов. Прежде всего, это присвоение значений, когда преобразование зачастую происходит незаметно для программиста. Вызов метода во многом похож на инициализацию. Явное приведение позволяет осуществить желаемый переход в том 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.

Объектная модель в Java

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, являющиеся альтернативой множественного наследования. Без этих мощных конструкций язык Java был бы неспособен 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.

Статические элементы

До этого момента под полями объекта мы 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:

```
Human.totalCount++;
  // рождение еще одного человека
```

Для удобства разрешено 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 убедиться на следующем примере:

```
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 данного 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 ссылки на объект не требуется.

```
Human.getTotalCount();
```

Хотя для удобства обращения через ссылку разрешены, но принимается во внимание только тип ссылки:

```
Human h=null;
h.getTotalCount();      // два эквивалентных
Human.getTotalCount();  // обращения к 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 это усложняет 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, рассматривавшихся 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 те же 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`, это 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`, а это 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 может не быть ни одного экземпляра `MyClass`. 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
  }
}
```

Ключевые слова this и super

Эти ключевые слова уже упоминались, рассматривались и некоторые случаи их применения. Здесь они будут описаны более подробно.

Если выполнение кода происходит в динамическом контексте, то должен быть объект, ассоциированный с ним. В этом случае ключевое слово `this` возвращает ссылку на данный 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 данный 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 "затеняющих" 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.out.println(c.getValue());
  }
}
```

Вызов переопределенного метода использует механизм полиморфизма, который подробно рассматривается в конце этой лекции. Однако ясно, что результатом выполнения примера будет значение `3`. Невозможно, используя ссылку типа `Child`, получить из метода `getValue()` значение `5`, родительский метод перекрыт и уже недоступен.

Иногда при переопределении бывает полезно воспользоваться результатом работы родительского метода. Предположим, он делал сложные вычисления, а переопределенный метод должен вернуть округленный результат этих вычислений. Понятно, что гораздо удобнее 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.out.println(c.getValue());
  }
}
```

Результатом работы программы будет значение `6`.

Обращаться с помощью ключевого слова `super` к переопределенному методу 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.

Ключевое слово abstract

Следующее важное понятие, которое необходимо рассмотреть,– ключевое слово `abstract`.

Иногда имеет смысл описать только заголовок метода, без его тела, и таким образом объявить, что данный 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 того, существует специальный контейнер, который занимается их отрисовкой. Понятно, что внешний вид каждой компоненты уникален, а значит, соответствующий метод (назовем его `paint()`) будет реализован в разных элементах 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 границы контейнера, и т.д. Каждая фигура может иметь цвет, которым ее надо рисовать, может быть видимой, или невидимой 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, чтобы анализировать, какая именно компонента сейчас обрабатывается, выполнять 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 extends Operation {
  public int calculate(int a, int b) {
    return a+b;
   }
}

// Вычитание
class Subtraction extends Operation {
   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 одинаково.

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 том 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 неабстрактными 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();
  public 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 не

абстрактного 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.

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 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 какая будет функциональность родительского класса, который лишился своего метода?

Все эти проблемы не возникают в том случае, если наследуются только абстрактные методы от нескольких родителей. Даже если унаследовано несколько одинаковых методов, все равно у них нет реализации и можно один раз описать тело метода, которое будет использоваться при вызове любого из этих методов.

Именно так устроены интерфейсы в Java. От них нельзя порождать объекты, но другие классы могут реализовывать их.

Объявление интерфейсов

Объявление интерфейсов очень похоже на упрощенное объявление классов.

Оно начинается с заголовка. Сначала указываются модификаторы. Интерфейс может быть объявлен как `public` и тогда он будет доступен для общего 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 делать этого не рекомендуется, 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 Drawable extends Colorable,
  Resizable {
}
```

Тело интерфейса состоит из объявления элементов, то есть полей-констант и абстрактных методов. Все поля интерфейса должны быть `public final static`, так что эти модификаторы указывать необязательно и даже нежелательно, чтобы не загромождать код. Поскольку поля объявляются финальными, необходимо их сразу инициализировать.

```
public interface Directions {
  int RIGHT=1;
  int LEFT=2;
  int UP=3;
  int DOWN=4;
}
```

Все методы интерфейса являются `public abstract` и эти модификаторы также необязательны.

```
public interface Moveable {
  void moveRight();
  void moveLeft();
  void moveUp();
  void moveDown();
}
```

Как мы видим, описание интерфейса гораздо проще, чем объявление класса.

Реализация интерфейса

Каждый 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 этих методов. Однако если у них 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 нельзя, 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 данного 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`, затем классу.

Наконец, если класс `C1` наследуется от класса `C2`, класс `C2` реализует интерфейс `A1`, а интерфейс `A1` наследуется от интерфейса `A2`, то класс `C1` также реализует интерфейс `A2`.

Все это 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`, это означает, что значения типа `интерфейс` обладают всеми элементами класса `Object`.

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 мнение, 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.

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. Животные делятся на две большие 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 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 из названных групп, 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 отдельно, 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 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.

Полиморфизм

Ранее были рассмотрены правила объявления классов с учетом их наследования. В этой лекции 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 наследующего класса. Рассмотрим, как функционируют классы и объекты в таких ситуациях.

Поля

Начнем с полей, которые могут быть статическими или динамическими. Рассмотрим пример:

```
class Parent {
```

```
    int a=2;
  }
  class Child extends Parent {
    int a=3;
  }
```

Прежде всего, 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 происходит 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. Данное объявление так и называется – "скрывающим" (hiding). Это особый случай перекрытия областей видимости, отличный от "затеняющего" (shadowing) и "заслоняющего" (obscuring) объявлений. Тем не менее, 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, 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. Неважно, 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 со 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 каждый раз определять, с каким из них надо работать.

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 снаружи уже недоступен, 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 произойдет 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 будет:

```
1
```

Данный пример демонстрирует, что переопределение методов должно производиться 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), 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`. Понятно, 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 осуществляется с помощью ссылки типа `Parent`. Именно компилятор выполняет 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 следующих лекциях.

Если абстрактный метод переопределяется неабстрактным, то говорят, что он его реализовал (implements). Как ни странно, 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 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.

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 интерфейсах, кроме:
  - `private` -элементов;
  - "скрытых" элементов (полей и статических методов, скрытых одноименными элементами);
  - переопределенных (динамических) методов.

Во-вторых, продолжим рассматривать взаимосвязь типа переменной и типов ее возможных значений. К случаям, описанным в предыдущей лекции, добавляются еще два. Переменная типа абстрактный класс может ссылаться на объекты, порожденные неабстрактным наследником этого класса. Переменная типа интерфейс может ссылаться на объекты, порожденные от класса, реализующего данный интерфейс.

Сведем эти данные в таблицу.

Таблица 8.1. Взаимосвязь типа переменной и типов ее возможных значений.

Тип переменной	Допустимые типы ее значения
Абстрактный класс	• `null` • неабстрактный наследник
Интерфейс	• `null` • классы, реализующие интерфейс, а именно: • реализующие напрямую (заголовок содержит implements); • наследуемые от реализующих классов; • реализующие наследников этого интерфейса ;

- 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.

Заключение

В этой лекции 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 задействовать 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. Эта конструкция помогает избегать конфликтов имен, а также применяется в конструкторах.

Слово `super` позволяет задействовать 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, пусть 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. Было подробно изучено поведение полей и методов, как статических, так и динамических, при переопределении. Что позволило перейти к вопросу соответствия типов переменной и ее значения.

Массивы

Лекция посвящена описанию массивов в Java. Массивы издавна присутствуют в языках программирования, поскольку при выполнении многих задач приходится оперировать 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 объясняется, 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 описать возможность создания точных копий объектов, порожденных от него.

Массивы как тип данных в Java

В отличие от обычных переменных, которые хранят только 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 легко конвертироваться друг в друга с помощью специальных методов, но все же они не относятся к идентичным типам.

Как уже говорилось, массивы в Java являются объектами (примитивных

типов в Java всего восемь и их количество не меняется), их тип напрямую наследуется от класса `Object`, поэтому все элементы данного класса доступны у объектов-массивов.

Базовый тип также может быть массивом. Таким образом конструируется массив массивов, или многомерный массив.

Работа с любым массивом включает обычные операции, уже описанные для других типов, - объявление, инициализация и т.д. Начнем последовательно изучать их в приложении к массивам.

Объявление массивов

В качестве примера рассмотрим объявление переменной типа "массив, основанный на примитивном типе `int` ":

```
int a[];
```

Как мы видим, сначала указывается базовый тип. Затем идет имя переменной, а пара квадратных скобок указывает на то, что используемый тип является именно массивом. Также допустима запись:

```
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 – 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<5; 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
```

Если бы индекс превысил максимально возможное для такого массива значение, то появилась бы ошибка времени исполнения. Проверка, не выходит ли индекс за допустимые пределы, происходит только во время исполнения программы, т.е. компилятор не пытается выявить эту ошибку даже в таких явных случаях, как:

```
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 создать 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 задействовать `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 рассматривать тип 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, 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!
```

Инициализация массивов

Теперь, когда мы выяснили, 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 убедиться на простом примере:

```
Point p[]=new Point[5];
for (int i=0; i<p.length; i++) {
  System.out.println(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, существует и другой способ создания массивов – инициализаторы. В этом случае ключевое слово `new` не используется, а ставятся фигурные скобки, и в них через запятую перечисляются 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 осуществить 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.

Корректное 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.

Многомерные массивы

Теперь перейдем к рассмотрению многомерных массивов. Так, 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.

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 (один объект) и три массива чисел, каждый длиной `5` (три объекта). Итого, четыре объекта.

В рассмотренном примере три 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 крайняя левая пара, это значение задаст длину 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 применяется столько вложенных фигурных скобок, сколько требуется:

```
int i[][] = {{1,2}, null, {3}, {}};
```

В этом примере порождается четыре объекта. Это, во-первых, массив массивов длиной 4, а во-вторых, три 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.

Класс массива

Поскольку массив является объектным типом данных, можно попытаться представить себе, как выглядело бы 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 эти объявления динамически на основе базового типа и размерности, а 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 играют модификаторы доступа. Класс массива будет иметь такой же уровень доступа, как и базовый тип. То есть 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.

Затем следует имя класса, на котором можно подробно не останавливаться, т.к. к типу массива обращение идет не по его имени, а по имени базового типа и набору квадратных скобок.

Затем нужно указать родительский класс. 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 лекции, а второй будет описан в следующих лекциях.

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, экземпляр типа массив 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
```

Преобразование типов для массивов

Теперь, когда массив введен как полноценный тип данных в Java, рассмотрим, какое влияние он окажет на преобразование 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.

Имейте в виду, 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 трем типам, обратный же переход является сужением и должен производиться 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 останавливаться на этом подробно, однако заметим, что преобразования между типами массивов, основанных на различных

примитивных типах, невозможны 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 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, 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 выполняться 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 выполняет копирование элементов одного массива в другой.

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 такое обращение:

```
c[0].onlyChildMethod();
```

где метод `onlyChildMethod()` определен только в классе `Child`. Данное 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 осуществляет дополнительную проверку перед присвоением значения элементу массива. Необходимо удостовериться, 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 задавать для 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 к ошибке:

```
process(new Child[3]);
```

И это будет как раз ошибка `ArrayStoreException`.

Переменные типа массив и их значения

Завершим описание взаимосвязи типа переменной и типа значений, которые она может хранить.

Как обычно, массивы, основанные на простых и ссылочных типах, мы 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:

1. `null` ;
2. значения точно такого же типа, что и тип переменной;
3. 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.

Таблица Табл. 9.1.. Тип переменной и тип ее значения.

Тип переменной	Допустимые типы ее значения
Массив простых чисел	• null • в точности совпадающий с типом переменной
Массив ссылочных значений	• null • совпадающий с типом переменной • 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 • любой ссылочный, включая массивы

Клонирование

Механизм клонирования, как следует из названия, позволяет порождать новые объекты на основе существующего, которые обладали бы 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, как и выражение

```
x.clone().getClass() == x.getClass()
```

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 – недоступные, но важные для воссоздания состояния исходного объекта;
- в зависимости от логики реализации возможна ситуация, когда не все поля должны копироваться для корректного клонирования; одни могут оказаться лишними, другие 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 возможны два варианта.

Первый вариант: разработчик может в своем классе переопределить этот метод и реализовать его по своему усмотрению, решая перечисленные проблемы так, как того требует логика разрабатываемой системы. Упомянутые условия, которые должны 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. Естественно, перечисленные трудности легко могут быть преодолены самой JVM, ведь она хранит в памяти все свойства объектов.

При выполнении метода `clone()` сначала проверяется, 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 могут быть клонированы. Если проверка не выполняется успешно, метод порождает ошибку `CloneNotSupportedException`.

Если интерфейс `Cloneable` реализован, то порождается новый объект от того же класса, от которого был создан исходный объект. При

этом копирование выполняется 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. 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` ;
- удалено предупреждение об ошибке `CloneNotSupportedException` ;
- результирующий объект может быть модифицирован любым способом, 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=null;
    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)this.p.clone();
  return clone;
}
```

Обратите внимание, что результат метода `Object.clone()` приходится явно приводить к типу `Test`, хотя его реализация гарантирует, что клонированный объект будет порожден именно от этого класса. Однако тип возвращаемого значения в данном 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 того, типом поля клонируемого объекта может быть класс, не реализующий `Cloneable`, что приводило бы к дополнительным проблемам. Как показано в примере, при необходимости дополнительное копирование можно добавить самостоятельно.

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 единица, 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 случаях. Начнем с того, что в первой строке создается двумерный массив, состоящий из двух одномерных. Итого три объекта. Затем, на следующей строке при клонировании будет создан новый двумерный 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 двухмерных массивах. На консоли появится `0`.

Обратите внимание, что если из примера убрать условие `if-else`, так, чтобы отрабатывал первый вариант, а затем 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 уделяться копированию объектных полей, которые по умолчанию копируются только по ссылке.

Заключение

В этой лекции было 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 у созданного массива не может быть изменена. Однако она не входит в определение типа, а потому одна переменная может ссылаться на массивы одного типа с различной длиной.

Создать массив можно 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 массивы. Они, по сути, являются одномерными, основанными на массивах меньшей размерности. Такой подход позволяет единообразно работать с многомерными массивами. Также он дает возможность создавать не только "прямоугольные" массивы, но и массивы 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 делать, реализуя интерфейс `Cloneable`. Поскольку стандартное клонирование порождает только один новый объект, это приводит к особым эффектам при работе с объектными полями классов и массивами.

Операторы и структура кода. Исключения

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. В этой лекции вводятся важные понятия, связанные с данной 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 разделение всех ошибок на проверяемые и непроверяемые компилятором, а также ошибки времени исполнения.

Управление ходом программы

Управление потоком вычислений является фундаментальной основой всего языка программирования. В данной лекции 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, и вместе с тем имеется ряд отличий, которые будут рассмотрены позднее и на которые следует обратить внимание.

Порядок выполнения программы определяется операторами. Операторы могут содержать другие операторы или выражения.

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

- break
- continue

return

Тогда управление будет передано в другое место (в соответствии с правилами обработки этих операторов, которые мы рассмотрим позже).

Нормальное выполнение оператора может быть прервано также при возникновении исключительных ситуаций, которые тоже будут рассмотрены позднее. Явное возбуждение исключительной ситуации с помощью оператора `throw` также 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 `Throwable`, а также все исключения, вызываемые виртуальной машиной Java.

Выражения могут завершаться нормально и преждевременно (аварийно). В данном случае термин "аварийно" вполне применим, т.к. причиной необычной последовательности выполнения выражения 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 том случае, если в операторе имеется вложенный оператор и его завершение происходит ненормально, то так же ненормально завершается оператор, содержащий вложенный (в некоторых случаях это не так, что будет оговариваться особо).

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.

Операторы в блоке выполняются слева направо, сверху вниз. Если все операторы (выражения) в блоке выполняются нормально, то и весь блок выполняется нормально. Если какой-либо оператор (выражение) завершается ненормально, то и весь блок завершается ненормально.

Нельзя объявлять несколько локальных переменных с одинаковыми именами 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. Так, следующий 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);
    }
  }
```

На консоль будет выведено `x = 1`.

То же самое правило применимо к параметрам 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:

```
Local value x = 5
Member value x = 10
```

На следующем примере продемонстрируем, что область видимости локальной переменной ограничена 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 переменной, то выражение исполняется слева направо и его результат присваивается локальной переменной. Использование неинициализированных 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, при которых 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).

Пустой оператор

Точка с запятой (`;`) является пустым оператором. Данная конструкция вполне применима там, где не предполагается выполнение никаких действий. Преждевременное завершение пустого оператора невозможно.

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, допускается обращение из внутренних блоков к меткам, относящимся к внешним.

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 оператор `goto` отсутствует. В некоторых ситуациях (как в рассмотренном примере с вложенными циклами) использование меток вполне оправданно, но, конечно, их применение следует ограничивать лишь самыми необходимыми случаями.

Оператор if

Пожалуй, наиболее распространенной 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. Отметим отличие от языка C, в котором в качестве логического выражения могут использоваться различные типы данных, где отличное от нуля выражение трактуется как истинное значение, а ноль - как ложное. В Java возможно использование только логических выражений.

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")` вполне допустима.

Операторы `if-else` могут каскадироваться.

```
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 если не выполнено предыдущее. Заключительная конструкция `else` относится к самому последнему условию `if` и будет выполнена только в том случае, если ни одно из вышеперечисленных условий не будет истинным. Если хотя бы 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 5");
} else{
   System.out.println("Другое значение");
}
```

Предложение `"Равно 5"` в данном случае напечатано не будет.

Оператор switch

Оператор `switch()` удобно использовать 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 использоваться переменная типа `byte`, `short`, `int`, `char` или `выражение`. Выражение должно в конечном итоге возвращать параметр одного из указанных ранее типов. В операторе `switch` 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 выполняется выражение следующее за этим условием. Если выражение выполнено нормально и нет преждевременного его завершения, то производится выполнение для последующих `case`. Если же выражение, следующее за `case`, завершилось ненормально, 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` также выполнено не будет.

Следует обратить внимание, что, в отличие от многозвенного `if-else`, если какое-либо условие `case` выполнено, то выполнение `switch` не прекратится, а будут выполняться следующие за ним выражения. Если этого необходимо избежать, то после кода следующего за оператором `case` используется оператор `break`, прерывающий дальнейшее выполнение оператора `switch`.

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`.

Рассмотрим пример:

```
int x = 2;
switch(x) {
   case 1:
   case 2:
      System.out.println("Равно 1 или 2");
      break;
   case 3:
   case 4:
      System.out.println("Равно 3 или 4");
      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:   // только константы!
      ...
      break;
}
```

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");
  }
```

недопустима.

Также в конструкции `switch` может быть применен только один оператор `default`.

Управление циклами

В языке Java имеется три основных конструкции управления циклами:

- цикл `while` ;
- цикл `do` ;
- цикл `for`.

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. Этот многократно исполняемый блок называют телом цикла

Операторы `continue` и `break` могут изменять нормальное исполнение тела цикла. Так, если в теле цикла встретился оператор `continue`, то операторы, следующие за ним, будут пропущены и

выполнение цикла начнется сначала. Если `continue` используется с меткой и метка принадлежит к данному `while`, то выполнение его будет аналогичным. Если метка не относится к данному `while`, его выполнение будет прекращено и управление будет передано на оператор, или блок, к которому относится метка.

Если встретился оператор `break`, то выполнение цикла будет прекращено.

Если выполнение блока было прекращено по какой-то другой причине (возникла исключительная ситуация), то выполнение всего цикла будет прекращено по той же причине.

Рассмотрим несколько примеров:

```
public class Test {
  static int x = 5;
  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;
  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` начнет новую итерацию.

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, если на момент начала его выполнения логическое выражение будет истинным. Таким образом, 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.

Цикл do

Основная форма цикла `do` имеет следующий вид:

```
do
   повторяющееся выражение или блок;
while(логическое выражение)
```

Цикл `do` будет выполняться до тех пор, пока логическое выражение будет истинным. В отличие от цикла `while`, этот цикл будет выполнен, как минимум, один раз.

Типичная конструкция цикла `do`:

```
int counter = 0;
do {
   counter ++;
   System.out.println("Counter is "
     + counter);
} while(counter < 5);
```

В остальном выполнение цикла `do` аналогично выполнению цикла `while`, включая использование операторов `break` и `continue`.

Цикл for

Довольно часто бывает необходимо изменять значение какой-либо переменной в заданном диапазоне и выполнять повторяющуюся последовательность операторов с использованием этой переменной. Для выполнения такой 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 круглые скобки 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 `while()`. Тело цикла выполняется до тех пор, пока логическое выражение истинно. Как и в случае с циклом `while()`, тело цикла может не исполниться ни разу. Это происходит, если логическое выражение принимает значение "ложь" до начала выполнения цикла.

Выражение обновления 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.

Разрешается 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 нужно обратить внимание, что область видимости переменной `cnt` будет ограничена телом цикла.

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, мы рассмотрим позже.

Возможно также 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.

Операторы break и continue

В некоторых случаях требуется изменить ход выполнения программы. В традиционных языках программирования для этих целей применяется оператор `goto`, однако в Java он не поддерживается. Для этих целей применяются операторы `break` и `continue`.

Оператор continue

Оператор `continue` может использоваться 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[]  = {....}
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.

Если оператор `continue` будет применен вне контекста оператора цикла, то будет выдана ошибка времени компиляции. В случае использования вложенных циклов оператору `continue`, в качестве адреса перехода, может быть указана метка, относящаяся к одному из этих операторов.

Рассмотрим пример:

```
public class Test {
  public Test() {
  }
  public static void main(String[] args) {
    Test t = new Test();
    for(int i=0; i < 10; i++){
      if(i % 2 == 0) continue;
      System.out.print(" i=" + i);
    }
  }
}
```

В результате работы на консоль будет выведено:

```
i=1 i=3 i=5 i=7 i=9
```

При выполнении условия в строке 7 нормальная последовательность выполнения операторов будет прервана и управление будет передано на начало цикла. Таким образом, на консоль будут выводиться только нечетные значения.

Оператор break

Этот оператор, как и оператор `continue`, изменяет последовательность выполнения, но не возвращает исполнение к началу цикла, а прерывает его.

```
public class Test {
  public Test() {
  }
  public static void main(String[] args) {
    Test t = new Test();
    int [] x = {1,2,4,0,8};
    int y  = 8;
    for(int cnt=0;cnt < x.length;cnt++) {
      if(0 == x[cnt]) break;
      System.out.println("y/x = " + y/x[cnt]);
    }
  }
}
```

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 оператор `break` не содержится.

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 решает этой задачи, однако в Java можно именовать блок кода и явно указать операторам, к какому из них относится выполняемое действие. Делается это 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, чтобы прервать выполнение обоих циклов, используется метка:

```
...
int array[][] = {:..};
outerLoop: for(int i=0;i<5;i++) {
   for(j=0;j<4; j++){
      ...
      if(array[i][j] == caseValue)
            break outerLoop;
      ...
   }
```

```
}
...
```

Оператор `break` также может использоваться с именованными блоками.

Между операторами `break` и `continue` есть еще одно существенное отличие. Оператор `break` может использоваться 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. То есть 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, то есть управление будет передано на оператор (выражение), следующий непосредственно за закрывающей фигурной скобкой.

Метки используют пространство имен, отличное от пространства имен классов и методов.

Так, следующий пример кода будет вполне работоспособным:

```
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test t = new Test();
      t.test();
   }
   void test() {
      Test: {
         test: for(int i =0;true;i++) {
```

```
            if(i % 2 == 0) continue test;
            if(i > 10) break Test;
            System.out.print(i + " ");
          }
        }
    }
}
```

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 применять 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: {
      ...
   }
   ...
}
```

Оператор return

Этот оператор предназначен для возврата управления из вызываемого метода в вызывающий. Если в последовательности операторов выполняется `return`, то управление немедленно (если это не оговорено особо) передается в вызывающий метод. Оператор `return` может иметь, а может и не иметь аргументов. Если метод не возвращает значений (объявлен как `void`), то в этом и только этом случае выражение `return` применяется без аргументов. Если возвращаемое значение есть, то `return` обязательно должен применяться с аргументом, чье значение и будет возвращено.

В качестве аргумента `return` может использоваться выражение

```
return (x*y +10) /11;
```

В этом случае сначала будет выполнено выражение, а затем результат его выполнения будет передан в вызывающий метод. Если выражение будет завершено ненормально, то и оператор `return` будет завершен ненормально. Например, если во время выполнения выражения в операторе `return` возникнет исключение, то никакого значения метод не вернет, будет обрабатываться ошибка.

В методе может быть более одного оператора `return`.

Оператор synchronized

Этот оператор применяется для исключения взаимного влияния нескольких потоков при выполнении кода, он будет подробно рассмотрен в лекции 12, посвященной потокам исполнения.

Ошибки при работе программы. Исключения (Exceptions)

При выполнении программы 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 те или

иные 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 изящным, но и более надежным и простым для понимания.

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` Этот оператор применяется для явного порождения ошибки. Очевидно, что и здесь можно 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 попытаемся остановить поток выполнения (вызвав метод `stop()`), нам не удастся предсказать, при выполнении какого именно выражения этот поток остановится.

Таким образом, все ошибки в Java делятся на синхронные и асинхронные. С первыми сравнительно проще работать, так как принципиально возможно найти точное место в коде, которое является причиной возникновения исключительной ситуации. Конечно, Java является строгим языком в том смысле, что все выражения до точки сбоя обязательно будут выполнены, и 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 трудно выявить причины сбоев в виртуальной машине. Это могут быть ошибки 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 не рекомендуется. В лекции 12 рассматриваются 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 данный метод был вызван;
- конструктор, то управление будет передаваться туда, где попытались создать объект (как правило, применяя оператор `new`);
- статический инициализатор, то управление будет передано туда, где произошло первое обращение к классу, потребовавшее его инициализации.

Допускается создание собственных классов исключительных ситуаций. Осуществляется это с помощью механизма наследования, то есть класс пользовательской исключительной ситуации должен быть унаследован от класса `Throwable`, или его потомков.

Обработка исключительных ситуаций

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 оператора `catch`, ассоциированного с данным оператором `try`.

Если в пределах `try` возникает исключительная ситуация, то далее выполнение кода производится по одному из перечисленных ниже сценариев.

Возникла исключительная ситуация, класс которой указан в качестве параметра одного из блоков `catch`. В этом случае производится выполнение блока кода, ассоциированного с данным `catch` (заключенного в фигурные скобки). Далее, если код в этом блоке завершается нормально, то и весь оператор `try` завершается нормально и управление передается на оператор (выражение), следующий за закрывающей фигурной скобкой последнего `catch`. Если код в `catch` завершается не штатно, то и весь `try` завершается нештатно по той же причине.

Если возникла исключительная ситуация, класс которой не указан в качестве аргумента ни в одном `catch`, то выполнение всего `try` завершается нештатно.

Конструкция try-catch-finally

Оператор `finally` предназначен для того, чтобы обеспечить гарантированное выполнение какого-либо фрагмента кода. Вне зависимости от того, возникла ли исключительная ситуация в блоке `try`, задан ли подходящий блок `catch`, не возникла ли ошибка в самом блоке `catch`,- все равно блок `finally` будет в конце концов исполнен.

Последовательность выполнения такой конструкции следующая: если оператор `try` выполнен нормально, то будет выполнен блок `finally`. В свою очередь, если оператор `finally` выполняется нормально, то и весь оператор `try` выполняется нормально.

Если во время выполнения блока `try` возникает исключение и существует оператор `catch`, который перехватывает данный тип исключения, происходит выполнение связанного с `catch` блока. Если блок `catch` выполняется нормально, либо ненормально, все равно затем выполняется блок `finally`. Если блок `finally` завершается нормально, то оператор `try` завершается так же, как завершился блок `catch`.

Если в списке операторов `catch` не находится такого, который обработал бы возникшее исключение, то все равно выполняется блок `finally`. В этом случае, если `finally` завершится нормально, весь `try` завершится ненормально по той же причине, по которой было нарушено исполнение `try`.

Во всех случаях, если блок `finally` завершается ненормально, то весь `try` завершится ненормально по той же причине.

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();
    }
```

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. Еще раз отметим, 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 должен быть выполнен, хотя сама исключительная ситуация обработана не будет и будет передана для обработки на более высокий уровень иерархии.

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 вызывающему методу и обработку исключительной ситуации должен произвести он. Если исключительная ситуация возникла в коде самого высокого уровня (например, методе `main()`), то управление будет передано исполняющей системе Java и выполнение программы будет прекращено (более точно - будет остановлен поток исполнения, в котором произошла такая ошибка).

Использование оператора throw

Помимо того, что предопределенная исключительная ситуация может быть возбуждена исполняющей системой Java, программист сам может явно породить ошибку. 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, что в качестве параметра методу может быть передано только неотрицательное значение; если это условие не выполнено, то с помощью оператора `throw` порождается исключительная ситуация. (Для успешной компиляции также требуется в заголовке метода указать `throws Exception` - это выражение рассматривается ниже.)

Метод должен делегировать обработку исключительной ситуации вызвавшему его коду. Для этого в сигнатуре метода применяется ключевое слово `throws`, после которого должны быть перечислены через запятую все исключительные ситуации, которые может вызывать данный метод. То есть 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");
   }
```

```
}
...
```

Таким образом, создание исключительной ситуации в программе выполняется с помощью оператора `throw` с аргументом, значение которого может быть приведено к типу `Throwable`.

В некоторых случаях после обработки исключительной ситуации может возникнуть необходимость передать информацию о ней в вызывающий код.

В этом случае ошибка появляется 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`, так как здесь содержится оператор, который может это исключение перехватить. То есть произойдет неявная передача управления на соответствующий блок `catch`.

Проверяемые и непроверяемые исключения

Все исключительные ситуации можно разделить на две категории: проверяемые (checked) и непроверяемые (unchecked).

Все исключения, порождаемые от `Throwable`, можно разбить на три группы. Они определяются тремя базовыми типами: наследниками `Throwable` - классами `Error` и `Exception`, а также наследником `Exception - RuntimeException`.

Ошибки, порожденные от `Exception` (и не являющиеся наследниками `RuntimeException`), являются проверяемыми. Т.е. во время компиляции проверяется, предусмотрена ли обработка возможных исключительных ситуаций. Как правило, это ошибки, связанные 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. Таким образом повышается надежность программы, ее устойчивость при возможных сбоях.

Исключения, порожденные от `RuntimeException`, являются непроверяемыми и компилятор не требует обязательной их обработки.

Как правило, это ошибки программы, которые при правильном кодировании возникать не должны (например, `IndexOutOfBoundsException` - выход за границы массива, `java.lang.ArithmeticException` - деление на ноль). Поэтому, чтобы не загромождать программу, компилятор оставляет на усмотрение программиста обработку таких исключений с помощью блоков `try-`

`catch`.

Исключения, порожденные от `Error`, также не являются проверяемыми. Они предназначены для того, чтобы уведомить приложение о возникновении фатальной ситуации, которую программным способом устранить практически невозможно (хотя формально обработчик допускается). Они могут свидетельствовать об ошибках программы, но, как правило, это неустранимые проблемы на уровне JVM. В качестве примера можно привести `StackOverflowError` (переполнение стека), `OutOfMemoryError` (нехватка памяти).

Если в конструкции обработки исключений используется несколько операторов `catch`, классы исключений нужно перечислять в них последовательно, от менее общих к более общим. Рассмотрим два примера:

```
try {
   ...
}
catch(Exception e) {
   ...
}
catch(IOException ioe) {
   ...
}
catch(UserException ue) {
   ...
}
```

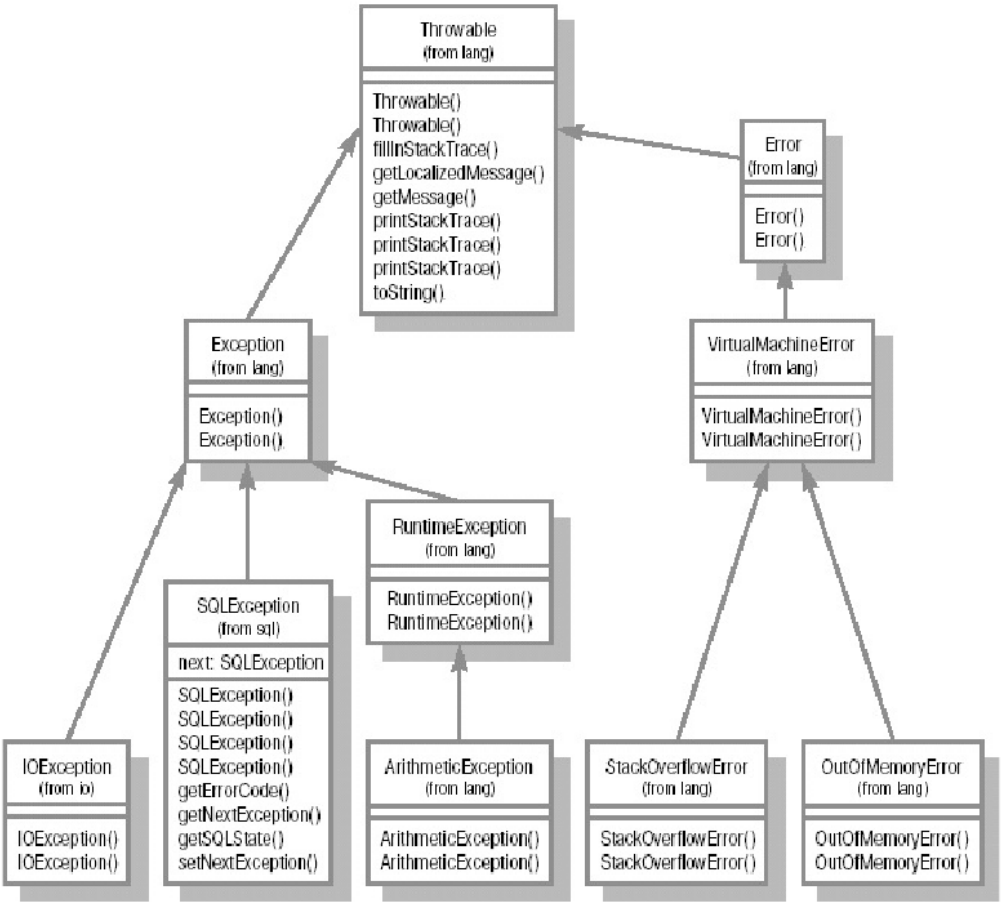


Рис. 10.1. Иерархия классов стандартных исключений.

В данном примере при возникновении исключительной ситуации (класс, порожденный 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(UserException ue) {
  ...
```

```
}
catch(IOException ioe) {
  ...
}
catch(Exception e) {
  ...
}
```

В этом случае будет выполняться 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.

Создание пользовательских классов исключений

Как уже отмечалось, допускается создание собственных классов исключений. Для этого достаточно создать свой класс, унаследовав его от любого наследника `java.lang.Throwable` (или от самого `Throwable`).

Пример:

```
public class UserException extends Exception {
  public UserException() {
    super();
  }
  public UserException(String descry) {
    super(descry);
  }
}
```

Соответственно, данное исключение будет создаваться следующим образом:

```
throw new UserException(
```

```
    "Дополнительное описание");
```

Переопределение методов и исключения

При переопределении методов следует помнить, что если переопределяемый метод объявляет список возможных исключений, то переопределяющий метод не может расширять этот список, но может его сужать. 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 LegalThree 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, так как переопределение метода `method()` верное (список ошибок не изменился);
- определение класса `LegalTwo` будет корректным, так как переопределение метода `method()` верное (новый метод не может выбрасывать ошибок, а значит, не расширяет список возможных ошибок старого метода);
- определение класса `LegalThree` будет корректным, так как переопределение метода `method()` будет верным (новый метод может создавать исключения, которые являются подклассами исключения, возбуждаемого в старом методе, то есть список сузился);
- определение класса `IllegalOne` будет некорректным, так как переопределение метода `method()` неверно (`IllegalAccessException` не является подклассом `IOException`, список расширился);
- определение класса `IllegalTwo` будет некорректным: хотя заголовок `method()` объявлен верно (список не расширился), в теле метода бросается исключение, не указанное в `throws`.

Особые случаи

Во время исполнения кода могут возникать ситуации, которые почти 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 handle stack 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 handle stack 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 handle stack 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 handle stack trace
```

Так как при вторичном возбуждении используется один и тот же объект `Exception`, стек в обоих случаях будет содержать одну и ту же последовательность вызовов. То есть 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: anotherBogus.file (The system cannot find the pa
      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, хотя последовательность вызовов одна и та же, в вызываемом и вызывающем методах обрабатываются разные объекты исключений.

Заключение

В данной лекции рассмотрены основные языковые конструкции.

Для организации циклов в Java предназначены три основных конструкции: `while`, `do`, `for`. Для изменения порядка выполнения операторов применяются `continue` и `break` (с меткой или без). Также существуют два оператора ветвления: `if` и `switch`.

Важной темой является обработка ошибок, поскольку без нее не обходится ни одна программа, ведь причиной сбоев может служить не только ошибка программиста, но и внешние события, например, разрыв сетевого соединения. Основной конструкцией обработки исключительных ситуаций является `try-catch-finally`. Для явной инициализации исключительной ситуации служит ключевое слово `throw`.

Ошибки делятся на проверяемые и непроверяемые. Чтобы повысить надежность программы, компилятор требует обработки исключений, классы которых наследуются от `Exception`, кроме классов-наследников `RuntimeException`. Предполагается, что такие ошибки могут возникать не столько по ошибке разработчика, сколько по внешним неконтролируемым причинам.

Классы, унаследованные от `RuntimeException`, описывают программные сбои. Ожидается, что программист сведет вероятность таких ошибок к минимуму, а потому, чтобы не загромождать код, они являются непроверяемыми, компилятор оставляет обработку на усмотрение разработчика. Ошибки-наследники `Error` свидетельствуют о фатальных сбоях, поэтому их также необязательно обрабатывать.

Методы, код которых может порождать проверяемые исключения, должны либо сами их обрабатывать, либо в заголовке метода должно быть указано ключевое слово `throws` с перечислением необрабатываемых проверяемых исключений. На непроверяемые ошибки это правило не распространяется.

Переопределенный (`overridden`) метод не может расширять список возможных исключений исходного метода.

Пакет java.awt

Эта лекция начинает рассмотрение базовых библиотек Java, которые являются неотъемлемой частью языка и входят в его спецификацию, а именно описывается пакет java.awt, предоставляющий технологию AWT для создания графического (оконного) интерфейса пользователя – GUI. Ни одна современная программа, предназначенная для пользователя, не обходится без удобного, понятного, в идеале – красивого пользовательского интерфейса. С самой первой версии в Java существует специальная технология для создания GUI. Она называется AWT, Abstract Window Toolkit. Именно о ней пойдет речь в этой лекции. Пакет java.awt претерпел, пожалуй, больше всего изменений с развитием версий Java. Мы рассмотрим дерево компонентов, доступных программисту, специальную модель сообщений, позволяющую гибко обрабатывать пользовательские действия, и другие особенности AWT – работа с цветами, шрифтами, отрисовка графических примитивов, менеджеры компоновки и т.д. Хотя технология AWT включает в себя гораздо больше, чем можно изложить в рамках одной лекции, здесь собраны все необходимые сведения для создания полноценного оконного интерфейса.

Введение

Поскольку Java-приложения предназначены для работы на разнообразных платформах, реализация графического пользовательского интерфейса (GUI) должна быть либо одинаковой для любой платформы, либо, напротив, программа должна иметь вид, типичный для данной операционной системы. В силу ряда причин, для основной библиотеки по созданию GUI был выбран второй подход. Во-первых, это лишний раз показывало гибкость Java – действительно, пользователи разных платформ могли работать с одним и тем же Java-приложением, не меняя своих привычек. Во-вторых, такая реализация обеспечивала большую производительность, поскольку была основана на возможностях операционной системы. В частности, это означало и более компактный, простой, а значит, и более надежный код.

Библиотеку назвали AWT – Abstract Window Toolkit. Слово abstract в названии указывает, что все стандартные компоненты не являются

самостоятельными, а работают в связке с соответствующими элементами операционной системы.

Дерево компонентов

Component

Абстрактный класс `Component` является базовым для всех компонентов AWT и описывает их основные свойства. Визуальный компонент в AWT имеет прямоугольную форму, может быть отображен на экране и может взаимодействовать с пользователем.

Рассмотрим основные свойства этого класса.

Положение

Положение компонента описывается двумя целыми числами (тип `int`) `x` и `y`. В Java (как и во многих языках программирования) ось `x` проходит традиционно – горизонтально, направлена вправо, а ось `y` – вертикально, но направлена вниз, а не вверх, как принято в математике.

Для описания положения компонента предназначен специальный класс – `Point` (точка). В этом классе определено два `public int` поля `x` и `y`, а также множество конструкторов и вспомогательных методов для работы с ними. Класс `Point` применяется во многих типах AWT, где надо задать точку на плоскости.

Для компонента эта точка задает положение левого верхнего угла.

Установить положение компонента можно с помощью метода `setLocation()`, который может принимать в качестве аргументов пару целых чисел, либо `Point`. Узнать текущее положение можно с помощью метода `getLocation()`, возвращающего Point, либо с помощью методов `getX()` и `getY()`, которые появились с версии Java 1.2.

Размер

Как было сказано, компонент AWT имеет прямоугольную форму, а потому его размер описывается также двумя целочисленными параметрами – `width` (ширина) и `height` (высота). Для описания размера существует специальный класс `Dimension` (размер), в котором определено два `public int` поля `width` и `height`, а также вспомогательные методы.

Установить размер компонента можно с помощью метода `setSize`, который может принимать в качестве аргументов пару целых чисел, либо `Dimension`. Узнать текущие размеры можно с помощью метода `getSize()`, возвращающего `Dimension`, либо с помощью методов `getWidth()` и `getHeight()`, которые появились с версии Java 1.2.

Совместно положение и размер компонента задают его границы. Область, занимаемую компонентом, можно описать либо четырьмя числами (`x, y, width, height`), либо экземплярами классов `Point` и `Dimension`, либо специальным классом `Rectangle` (прямоугольник). Как легко догадаться, в этом классе определено четыре `public int` поля, с которыми можно работать и в виде пары объектов `Point` и `Dimension`.

Задать границу объекта можно с помощью метода `setBounds`, который может принимать четыре числа, либо `Rectangle`. Узнать текущее значение можно с помощью метода `getBounds()`, возвращающего `Rectangle`.

Видимость

Существующий компонент может быть как виден пользователю, так и быть скрытым. Это свойство описывается булевским параметром `visible`. Методы для управления – `setVisible`, принимающий булевский параметр, и `isVisible`, возвращающий текущее значение.

Разумеется, невидимый компонент не может взаимодействовать с пользователем.

Доступность

Даже если компонент отображается на экране и виден пользователю, он может не взаимодействовать с ним. В результате события от клавиатуры или мыши не будут получаться и обрабатываться компонентом. Такой компонент называется `disabled`. Если же компонент активен, его называют `enabled`. Как правило, компонент некоторым образом меняет свой внешний вид, когда становится недоступным (например, становится серым, менее заметным), но, вообще говоря, это необязательно (хотя очень удобно для пользователя).

Для изменения этого свойства применяется метод `setEnabled`, принимающий булевский параметр (`true` соответствует `enabled`, `false` – `disabled`), а для получения текущего значения – `isEnabled`.

Цвета

Разумеется, для построения современного графического интерфейса пользователя необходима работа с цветами.

Компонент обладает двумя свойствами, описывающими цвета, – foreground и background цвета. Первое свойство задает, каким цветом выводить надписи, рисовать линии и т.д. Второе – задает цвет фона, которым закрашивается вся область, занимаемая компонентом, перед тем, как прорисовывается внешний вид.

Для задания цвета в AWT используется специальный класс `Color`. Этот класс обладает довольно обширной функциональностью, поэтому рассмотрим основные характеристики.

Цвет задается 3 целочисленными характеристиками, соответствующими модели RGB, – красный, зеленый, синий. Каждая из них может иметь значение от 0 до 255 (тем не менее, их тип определен как `int`). В результате `(0, 0, 0)` соответствует черному, а `(255, 255, 255)` – белому.

Класс `Color` является неизменяемым, то есть, создав экземпляр,

соответствующий какому-либо цвету, изменить параметры RGB уже невозможно. Это позволяет объявить в классе `Color` ряд констант, описывающих базовые цвета: белый, черный, красный, желтый и так далее. Например, вместо того, чтобы задавать синий цвет числовыми параметрами `(0, 0, 255)`, можно воспользоваться константами `Color.blue` или `Color.BLUE` (второй вариант появился в более поздних версиях).

Для работы со свойством компонента foreground применяют методы `setForeground` и `getForeground`, а для background – `setBackground` и `getBackground`.

Шрифт

Раз изображение компонента может включать в себя надписи, необходимо свойство, описывающее шрифт для их прорисовки.

Для задания шрифта в AWT существует специальный класс `Font`, который включает в себя три параметра – имя шрифта, размер и стиль.

Имя шрифта задает внешний стиль отображения символов. Имена претерпели ряд изменений с развитием Java. В версии 1.0 требовалось, чтобы JVM поддерживала следующие шрифты: `TimesRoman`, `Helvetica`, `Courier`. Могут поддерживаться и другие семейства, это зависит от деталей реализации конкретной виртуальной машины. Чтобы узнать полный список во время исполнения программы, можно воспользоваться методом утилитного класса `Toolkit`. Экземпляры этого класса нельзя создать вручную, поэтому полностью такой запрос будет выглядеть следующим образом:

```
Toolkit.getDefaultToolkit().getFontList()
```

В результате будет возвращен массив строк-имен семейств поддерживаемых шрифтов.

В Java 1.1 три обязательных имени были объявлены *`depreciated`*. Вместо них был введен новый список, который содержал более универсальные названия, не зависящие от конкретной операционной

системы: `Serif`, `SansSerif`, *`Monospaced`*.

В Java 2 библиотека AWT была существенно пересмотрена и дополнена. Чтобы устранить неоднозначности с разной поддержкой шрифтов на разных платформах, было произведено разделение на логические и физические шрифты. Вторая группа определяется возможностями операционной системы, это те же шрифты, которые могут использовать другие программы, запущенные на этой платформе.

Первая группа состоит из 5 обязательных семейств (добавились `Dialog` и `DialogInput`). JVM устанавливает соответствие между ними и наиболее подходящими из доступных физических шрифтов.

Метод `getFontList` класса `Toolkit` был объявлен *`deprecated`*. Теперь получить список всех доступных физических шрифтов можно следующим образом:

```
GraphicsEnvironment.
  getLocalGraphicsEnvironment().
  getAvailableFontFamilyNames()
```

Класс `Font` является неизменяемым. После создания можно узнать заданное логическое имя (метод `getName`) и соответствующее ему физическое имя семейства (метод `getFamily`).

Вернемся к остальным параметрам, необходимым для создания экземпляра `Font`. Размер шрифта определяет, очевидно, величину символов. Однако конкретные значения измеряются не в пикселах, а в условных единицах (как и во многих текстовых редакторах). Для разных семейств шрифтов символы одинакового размера могут иметь различную ширину и высоту, измеренную в пикселах.

Как и в случае имени шрифта, программист может указать любое значение размера, а JVM поставит ему в соответствие максимально близкий из доступных.

Наконец, последний параметр – стиль. Этот параметр определяет, будет ли шрифт жирным, наклонным и т.д. Если никакие из этих свойств не требуются, указывается `Font.PLAIN` (параметр имеет тип `int` и в классе `Font` определен набор констант для удобства работы с ним).

Значение `Font.BOLD` задает жирный шрифт, а `Font.ITALIC` – наклонный. Для сочетания этих свойств (жирный наклонный шрифт) необходимо произвести логическое сложение: `Font.BOLD|Font.ITALIC`.

Для работы с этим свойством класса `Component` предназначены методы `setFont` и `getFont`.

Итак, мы рассмотрели основные свойства класса `Component`. Как легко видеть, все они предназначены для описания графического представления компонента, то есть отображения на экране.

Существует еще одно важное свойство другого характера. Очевидно, что практически всегда пользовательский интерфейс состоит из более чем одного компонента. В больших приложениях их обычно гораздо больше. Для удобства организации работы с ними компоненты объединяются в контейнеры. В AWT существует класс, который так и называется – `Container`. Его рассмотрение – наша следующая тема. Важно отметить, что компонент может находиться лишь в одном контейнере – при попытке добавить его в другой он удаляется из первого. Рассматриваемое свойство как раз и отвечает за связь компонента с контейнером. Свойство называется `parent`. Благодаря ему компонент всегда "знает", в каком контейнере он находится.

Container

Контейнер описывается классом `Container`, который является наследником `Component`, а значит, обладает всеми свойствами графического компонента. Однако основная его задача – группировать другие компоненты. Для этого в нем объявлен целый ряд методов. Для добавления служит метод `add`, для удаления – `remove` и `removeAll` (последний удаляет все компоненты).

Добавляемые компоненты хранятся в упорядоченном списке, поэтому для удаления можно указать либо ссылку на компонент, который и будет удален, либо его порядковый номер в контейнере. Также определены методы для получения компонент, присутствующих в контейнере, – все они довольно очевидны, поэтому перечислим их с краткими

пояснениями:

- `getComponent(int n)` – возвращает компонент с указанным порядковым номером;
- `getComponents()` – возвращает все компоненты в виде массива;
- `getComponentCount()` – возвращает количество компонент;
- `getComponentAt(int x, int y)` или (`Point p`) – возвращает компонент, который включает в себя указанную точку;
- `findComponentAt(int x, int y)` или (`Point p`) – возвращает видимый компонент, включающий в себя указанную точку.

Мы уже знаем, что положение компонента (`location`) задается координатами левого верхнего угла. Важно, что эти значения отсчитываются от левого верхнего угла контейнера, который таким образом является центром системы координат для каждого находящегося в нем компонента. Если важно расположение компонента на экране безотносительно его контейнера, можно воспользоваться методом `getLocationOnScreen`.

Благодаря наследованию контейнер также имеет свойство `size`. Этот размер задается независимо от размера и положения вложенных компонент. Таким образом, компоненты могут располагаться частично или полностью за пределами своего контейнера (что это означает, будет рассмотрено ниже, но принципиально это допустимо).

Раз контейнер наследуется от `Component`, он сам является компонентом, а значит, может быть добавлен в другой, вышестоящий контейнер. В то же время компонент может находиться лишь в одном контейнере. Это означает, что все элементы сложного пользовательского интерфейса объединяются в иерархическое дерево. Такая организация не только облегчает операции над ними, но и задает основные свойства всей работы AWT. Одним из них является принцип отрисовки компонентов.

Алгоритм отрисовки

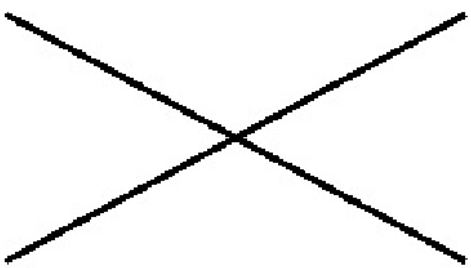
Начнем с отрисовки отдельного компонента – что определяет его внешний вид?

Для этой задачи предназначен метод `paint`. Этот метод вызывается каждый раз, когда необходимо отобразить компонент на экране. У него есть один аргумент, тип которого – абстрактный класс `Graphics`. В этом классе определено множество методов для отрисовки простейших графических элементов – линий, прямоугольников и многоугольников, окружностей и овалов, текста, картинок и т.д.

Наследники класса `Component` переопределяют метод `paint` и, пользуясь методами `Graphics`, задают алгоритм прорисовки своего внешнего вида:

```
public void paint(Graphics g) {
  g.drawLine(0, 0, getWidth(), getHeight());
  g.drawLine(0, getHeight(), getWidth(), 0);
}
```

В этом примере компонент будет отображаться двумя линиями, проходящими по его диагоналям:



Методы класса Graphics для отрисовки

Рассмотрим обзорно методы класса `Graphics`, предназначенные для отрисовки.

drawLine(x1, y1, x2, y2)

Этот метод отображает линию толщиной в 1 пиксел, проходящую из точки (`x1`, `y1`) в (`x2`, `y2`). Именно он использовался в предыдущем примере.

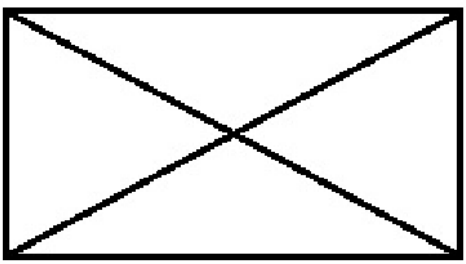
drawRect(int x, int y, int width, int height)

Этот метод отображает прямоугольник, чей левый верхний угол находится в точке (`x`, `y`), а ширина и высота равняются `width` и `height` соответственно. Правая сторона пройдет по линии `x+width`, а нижняя – `y+height`.

Предположим, мы хотим дополнить предыдущий пример рисованием рамки вокруг компонента (периметр). Понятно, что левый верхний угол находится в точке `(0, 0)`. Если ширина компонента равна, например, 100 пикселам, то координата `x` пробегает значения от 0 до 99. Это означает, что ширина и высота рисуемого прямоугольника должны быть уменьшены на единицу. На самом деле по той же причине в предыдущем примере такое уменьшение на единицу должно присутствовать и в остальных методах:

```
public void paint(Graphics g) {
  g.drawLine(0,0,getWidth()-1, getHeight()-1);
  g.drawLine(0,getHeight()-1, getWidth()-1,0);
  g.drawRect(0,0,getWidth()-1, getHeight()-1);
}
```

В результате компонент примет следующий вид:



fillRect(int x, int y, int width, int height)

Этот метод закрашивает прямоугольник. Левая и правая стороны прямоугольника проходят по линиям `x` и `x+width-1` соответственно, а верхняя и нижняя – `y` и `y+height-1` соответственно. Таким образом, чтобы зарисовать все пикселы компонента, необходимо передать следующие аргументы:

```
g.fillRect(0, 0, getWidth(), getHeight());
```

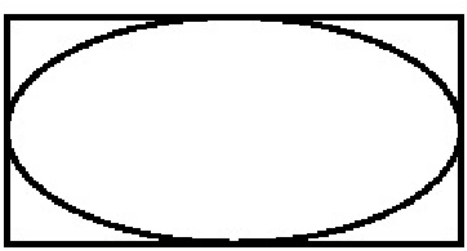
drawOval(int x, int y, int width, int height)

Этот метод рисует овал, вписанный в прямоугольник, задаваемый указанными параметрами. Очевидно, что если прямоугольник имеет равные стороны (т.е. является квадратом), овал становится окружностью.

Снова для того, чтобы вписать овал в границы компонента, необходимо вычесть по единице из ширины и высоты:

```
g.drawRect(0, 0, getWidth()-1, getHeight()-1);
g.drawOval(0, 0, getWidth()-1, getHeight()-1);
```

Результат:



fillOval(int x, int y, int width, int height)

Этот метод закрашивает указанный овал.

drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)

Этот метод рисует дугу – часть овала, задаваемого первыми четырьмя параметрами. Дуга начинается с угла `startAngle` и имеет угловой размер `arcAngle`. Начальный угол соответствует направлению часовой стрелки, указывающей на 3 часа. Угловой размер отсчитывается против часовой стрелки. Таким образом, размер в 90 градусов соответствует дуге в четверть овала (верхнюю правую). Углы "растянуты" в соответствии с размером прямоугольника. В результате, например, угловой размер в 45 градусов всегда задает границу дуги по линии, проходящей из центра прямоугольника в его правый верхний угол.

fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)

Этот метод закрашивает сектор, ограниченный дугой, задаваемой параметрами.

drawString(String text, int x, int y)

Этот метод выводит на экран текст, задаваемый первым параметром. Точка `(x, y)` задает позицию самого левого символа. Для наглядности приведем пример:

```
g.drawString("abcdefgh", 15, 15);
g.drawLine(15, 15, 115, 15);
```

Результатом будет:

Состояние Graphics

Экземпляр класса `Graphics` хранит параметры, необходимые для отрисовки. Рассмотрим их по порядку.

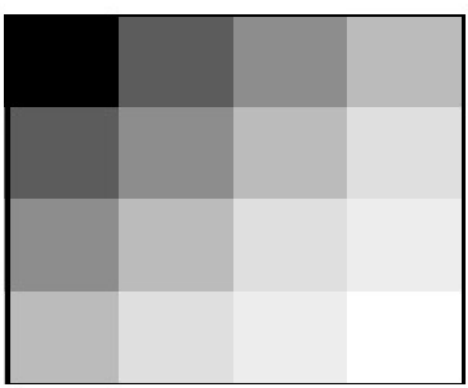
Цвет

Очевидно, что для отрисовки линий, овалов, текста и т.д. необходимо использовать тот или иной цвет. По умолчанию он задается свойством foreground компонента. В любой момент его можно изменить с помощью метода `setColor`.

Рассмотрим пример:

```
public void paint(Graphics g) {
  for (int i=0; i<4; i++) {
    for (int j=0; j<4; j++) {
      int c = (int)((i+j)*255/6);
      g.setColor(new Color(c, c, c));
      g.fillRect(i*getWidth()/4, j*getHeight()/4, getWidth()/4, getHeight()/4);
    }
  }
}
```

В результате компонент будет иметь следующий вид:



Узнать текущее значение цвета для отрисовки можно с помощью метода `getColor`.

Шрифт

Метод `drawString` не имеет аргумента, задающего шрифт для вывода текста на экран. Этот параметр также является частью состояния `Graphics`. Его значение по умолчанию задается соответствующим свойством компонента, однако может быть изменено с помощью метода `setFont`. Для получения текущего значения служит метод `getFont`.

Clip (ограничитель)

Хотя методы класса `Graphics` могут принимать любые значения аргументов, задающих значения координат (в пределах типа `int`), существует дополнительный ограничитель – `clip`. Любые изменения вне этого ограничителя на экране появляться не будут. Например, если вызвать метод `drawLine(-100, -100, 1000, 1000)`, то на компоненте отобразится лишь часть линии, которая помещается в его границы.

Размеры ограничителя можно изменять. Метод `clipRect(int x, int y, int width, int height)` вычисляет пересечение указанного прямоугольника и текущей области `clip`. Результат станет новым ограничителем. Таким образом, этот метод может только сужать область `clip`. Метод `setClip(int x, int y, int width, int height)` устанавливает ограничитель произвольно в форме прямоугольника. Метод `getClipBounds` возвращает текущее значение в виде объекта `Rectangle`.

При появлении приложения на экране каждый видимый компонент должен быть отрисован полностью. Поэтому при первом вызове метода `paint`, как правило, область `clip` совпадает с границами компонента. Однако при дальнейшей работе это не всегда так.

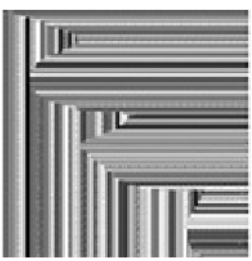
Рассмотрим следующий пример:

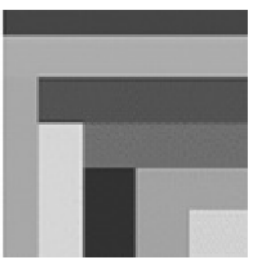
```
public void paint(Graphics g) {
  Color c = new Color(
   (int)(Math.random()*255),
```

```
            (int)(Math.random()*255),
            (int)(Math.random()*255));
        g.setColor(c);
        //g.setClip(null);
        g.fillRect(0, 0, getWidth(), getHeight());
    }
```

Как видно из кода, при каждом вызове метода `paint` генерируется новое значение цвета, после чего этим цветом закрашивается весь компонент. Однако поскольку в `Graphics` есть ограничитель, закрашена будет только область `clip`, что позволит ее увидеть.

После запуска программы компонент будет полностью окрашен одним цветом. Если теперь с помощью мыши "взять" окно какого-нибудь другого приложения и медленно "провести" им поверх компонента, то он окрасится примерно таким образом (левая картинка):





Если же провести быстро, то получится картинка, подобная правой в примере выше. Хорошо видно, что компонент перерисовывается не полностью, а частями. Ограничитель выставляется в соответствии с той областью, которая оказалась "повреждена" и нуждается в перерисовке. Для сложных компонентов можно ввести логику, которая, используя `clip`, будет отрисовывать не все элементы, а только некоторые из них, для увеличения производительности.

В примере закомментирована одна строка, в которой передается значение `null` в метод `setClip`. Такой вызов снимает все ограничения, поэтому компонента каждый раз будет перекрашиваться полностью, меняя при этом цвет. Однако никаким образом нельзя изменить состояние пикселов вне компонента – ограничитель не может быть шире, чем границы компонента.

Методы repaint и update

Кроме `paint` в классе `Component` объявлены еще два метода, отвечающие за прорисовку компонента. Как было рассмотрено, вызов `paint` инициируется операционной системой, если возникает необходимость перерисовать окно приложения, или часть его. Однако может потребоваться обновить внешний вид, руководствуясь программной логикой. Например, отобразить результат операции вычисления, или работы с сетью. Можно изменить состояние компонента (значение его полей), но операционная система не отследит такое изменение и не инициирует перерисовку.

Для программной инициализации перерисовки компонента служит метод `repaint`. Конечно, у него нет аргумента типа `Graphics`, поскольку программист не должен создавать экземпляры этого класса (точнее, его наследников, ведь `Graphics` – абстрактный класс). Метод `repaint` можно вызывать без аргументов. В этом случае компонент будет перерисован максимально быстро. Можно указать аргумент типа `long` – количество миллисекунд. Система инициализирует перерисовку спустя указанное время. Можно указать четыре числа типа `int` (`x, y, width, height`), задавая прямоугольную область компонента, которая нуждается в перерисовке. Наконец, можно указать все 5 параметров – и задержку по времени, и область перерисовки.

Если перерисовка инициируется приложением, то система вызывает не метод `paint`, а метод `update`. У него уже есть аргумент типа `Graphics` и по умолчанию он лишь закрашивает всю область компонента фоновым цветом (свойство background), а затем вызывает метод `paint`. Зачем же было вводить этот дополнительный метод, если можно было сразу вызвать `paint`? Дело в том, что поскольку перерисовка инициируется приложением, для сложных компонентов становится возможной некоторая оптимизация обновления внешнего вида. Например, если изменение заключается в появлении нового графического элемента, то можно избежать повторной перерисовки остальных элементов – переопределить метод `update` и реализовать в нем отображение одного только нового элемента. Если же компонент имеет простую структуру, можно оставить метод `update` без изменений.

Прорисовка контейнера

Теперь, когда известно, как работает прорисовка компонента, перейдем к рассмотрению контейнера. Для его корректного отображения необходимо выполнить два действия. Во-первых, нарисовать сам контейнер, ведь он является наследником компоненты, а значит, имеет метод `paint`, который может быть переопределен для задания особенного внешнего вида такого контейнера. Во-вторых, инициировать отрисовку всех компонентов, вложенных в него.

Первый шаг ничем не отличается от прорисовки обычного компонента. Как правило, контейнер не содержит никаких особых элементов отображения, ведь основную его площадь занимают вложенные компоненты. Поэтому перейдем ко второму шагу.

Если контейнер не пустой, значит, в нем есть одна или несколько компонент. Они будут отрисованы последовательно в том порядке, в каком были добавлены. Однако недостаточно просто в цикле вызвать метод `paint` для каждого компонента.

Во-первых, если компонента невидима (свойство `visible` выставлено в `false`), то, очевидно, метод `paint` у нее вызываться не должен.

Во-вторых, центр координат компонента находится в левом верхнем углу его контейнера, а у контейнера – в левом верхнем углу его контейнера. Таким образом, при переходе от отрисовки контейнера к отрисовке лежащего в нем компонента необходимо изменить (перенести) центр системы координат.

Затем необходимо установить `clip` в соответствии с размером очередного компонента. Необходимо выставить значения по умолчанию для цвета и шрифта, тем более что предыдущий компонент мог изменить их непредсказуемым образом.

В итоге получается более удобным создать новый экземпляр `Graphics` для каждого компонента. Для этого существует метод `create`, который порождает копию `Graphics`, причем ему можно передать аргументы (`int x, int y, int width, int`

height). В результате у нового Graphics будет смещен центр координат в точку (x, y), а clip -область будет получена пересечением существующего ограничителя с прямоугольником (0, 0, width, height) (в новых координатах). Метод create создает копию без изменения этих параметров.

Такие копии бывает удобно порождать и в рамках одного вызова метода paint, если в нем описан слишком сложный алгоритм. После использования такого объекта Graphics его необходимо особым образом освобождать – вызовом метода dispose(). Если необходимо только сместить точку отсчета координат, можно использовать метод translate (int x, int y).

Таким образом, контейнер своим методом paint отрисовывает себя и все вложенные в него компоненты. Если какие-то из них, в свою очередь, являются контейнерами, то процесс иерархически продолжается вглубь. В итоге весь AWT интерфейс, каким бы сложным он ни был, состоит из дерева контейнеров и компонент, отрисовка которых начинается с самого верхнего контейнера и по ветвям развивается вглубь до каждого видимого компонента.

Отдельный интерес представляет этот самый верхний контейнер. Как правило, это окно операционной системы, одновременно являющееся контейнером для Java-компонент. Именно операционная система инициализирует процесс отрисовки, отвечает за сворачивание и разворачивание окна, изменение его размера и так далее. Со стороны Java для работы с окном используется класс Window, который является наследником Container и рассматривается ниже.

Наследники класса Component

Теперь, когда рассмотрены основные принципы работы классов Component и Container, рассмотрим их наследников, с помощью которых и строится функциональный пользовательский интерфейс.

Начнем с наследников класса Component.

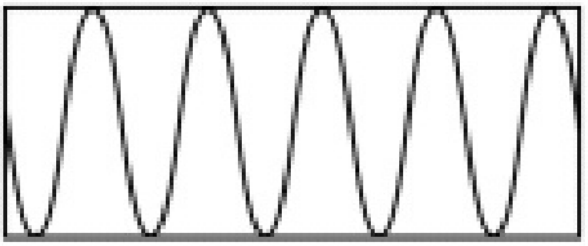
Класс Canvas

Класс `Canvas` является простейшим наследником `Component`. Он не добавляет никакой новой функциональности, но именно его нужно использовать в качестве суперкласса для создания пользовательского компонента с некоторым нестандартным внешним видом.

Ниже приведен пример определения компонента, который отображает график функции `sin(x)`:

```
public class SinCanvas extends Canvas {

  public void paint(Graphics g) {
    int height = getHeight(),
        width = getWidth();
    // Вычисляем масштаб таким образом,
        // чтобы на компоненте всегда умещалось
        // 5 периодов
    double k=2*Math.PI*5/width;
    int sy = calcY(0, width, height, k);
    for (int i=1; i<width; i++) {
      int nsy = calcY(i, width, height, k);
      g.drawLine(i-1, sy, i, nsy);
      sy=nsy;
    }
  }

  // метод, вычисляющий значение функции
  // для отображения на экране
  private int calcY(int x, int width,
                    int height, double k) {
    double dx = (x-width/2.)*k;
    return (int)(height/2.*(1-Math.sin(dx)));
  }
}
```

Как видно из примера, достаточно лишь переопределить метод `paint`. Вот как выглядит такой компонент:



Класс Label

Как понятно из названия, этот компонент отображает надпись. Соответственно, и его основной конструктор принимает один аргумент типа `String` – текст надписи. С помощью стандартных свойств класса `Component` – шрифт, цвет, фоновый цвет – можно менять вид надписи. Текст можно сменить и после создания `Label` с помощью метода `setText`.

Обратите внимание, что при этом компонент сам обновляет свой вид на экране. Такой особенностью обладают все стандартные компоненты AWT.

Класс Button

Этот компонент позволяет добавить в интерфейс стандартные кнопки. Основной конструктор принимает в качестве аргумента `String` – надпись на кнопке. Как обрабатывать нажатие на кнопку и другие пользовательские события, рассматривается ниже.

Классы Checkbox и CheckboxGroup

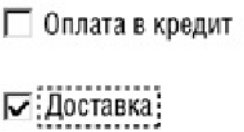
Компонент `Checkbox` имеет два способа применения.

Когда он используется сам по себе, он представляет `checkbox` – элемент, который может быть выделен или нет (например, нужна доставка для оформляемой покупки или нет). В этом случае в конструктор передается лишь текст – подпись к `checkbox`.

Рассмотрим пример, в котором в теле контейнера добавляется два `checkbox`:

```
Checkbox payment = new
      Checkbox("Оплата в кредит");
payment.setBounds(10, 10, 120, 20);
add(payment);
Checkbox delivery = new Checkbox("Доставка");
delivery.setBounds(10, 30, 120, 20);
add(delivery);
```

Ниже приведен внешний вид такого контейнера:



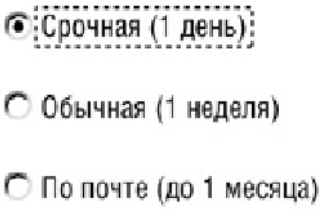
Обратите внимание, что размер `Checkbox` должен быть достаточным для размещения не только поля для "галочки", но и для подписи.

Второй способ применения компонент `Checkbox` предназначен для организации "переключателей" (*`radio buttons`*). В этом случае несколько экземпляров объединяются в группу, причем лишь один из переключателей может быть выбран. В роли такой группы выступает класс `CheckboxGroup`. Он не является визуальным, то есть никак не отображается на экране. Его задача – логически объединить несколько `Checkbox`. Группу, к которой принадлежит переключатель, можно указывать в конструкторе:

```
CheckboxGroup delivery = new CheckboxGroup();
Checkbox fast = new Checkbox(
      "Срочная (1 день)", delivery, true);
fast.setBounds(10, 10, 150, 20);
add(fast);
Checkbox normal = new Checkbox(
      "Обычная (1 неделя)", delivery, false);
normal.setBounds(10, 30, 150, 20);
add(normal);
```

```
Checkbox postal = new Checkbox(
        "По почте (до 1 месяца)",
         delivery, false);
postal.setBounds(10, 50, 150, 20);
add(postal);
```

Ниже приведен внешний вид такого контейнера:



В примере при вызове конструктора класса `Checkbox` помимо текста подписи и группы, указывается состояние переключателя (булевский параметр). Обратите внимание на изменение внешнего вида компонента (форма поля сменилась с квадратной на круглую, как и принято в традиционных `GUI`).

Классы Choice и List

Компонент `Choice` служит для выбора пользователем одного из нескольких возможных вариантов (выпадающий список). Рассмотрим пример:

```
Choice color = new Choice();
color.add("Белый");
color.add("Зеленый");
color.add("Синий");
color.add("Черный");
add(color);
```

В обычном состоянии компонент отображает только выбранный вариант. В процессе выбора отображается весь набор вариантов. На рисунке представлен выпадающий список в обоих состояниях:

Обратите внимание, что для компонента `Choice` всегда есть выбранный элемент.

Компонент `List`, подобно `Choice`, предоставляет пользователю возможность выбирать варианты из списка предложенных. Отличие заключается в том, что `List` отображает сразу несколько вариантов. Количество задается в конструкторе:

```
List accessories = new List(3);
accessories.add("Чехол");
accessories.add("Наушники");
accessories.add("Аккумулятор");
accessories.add("Блок питания");
add(accessories);
```

Вот как выглядит такой компонент (верхняя часть рисунка):

В списке находится 4 варианта. Однако в конструктор был передан параметр 3, поэтому только 3 из них видны на экране. С помощью полосы прокрутки можно выбрать остальные варианты.

Рисунок иллюстрирует еще одно свойство `List` – возможность

выбрать сразу несколько из предложенных вариантов. Для этого надо либо в конструкторе вторым параметром передать булевское значение `true` (`false` соответствует выбору только одного элемента), либо воспользоваться методом `setMultipleMode`.

Классы TextComponent, TextField, TextArea

Класс `TextComponent` является наследником `Component` и базовым классом для компонент, работающих с текстом,– `TextField` и `TextArea`.

`TextField` позволяет вводить и редактировать одну строку текста. Различные методы позволяют управлять содержимым этого поля ввода:

```
TextField tf = new TextField();
tf.setText("Enter your name");
tf.selectAll();
add(tf);
```

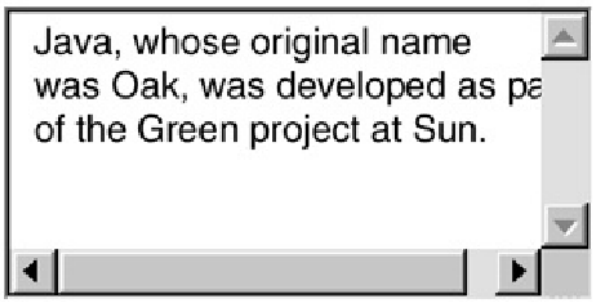
Вот как будет выглядеть этот компонент:

В коде вторая строка устанавливает значение текста в поле ввода (метод `getText` позволяет получить текущее значение). Затем весь текст выделяется (есть методы, позволяющие выделить часть текста).

Для любой текстовой компоненты можно задать особый режим. В базовом классе `Component` определено свойство `enabled`, которое, если выставлено в `false`, блокирует все пользовательские события. Для текстовой компоненты вводится новое свойство – `editable` (можно редактировать), методы для работы с ним – `isEditable` и `setEditable`. Если текст нельзя редактировать, но компонент доступен, то пользователь может выделить часть, или весь текст, и, например, скопировать его в буфер.

`TextField` обладает еще одним свойством. Все хорошо знакомы с полем ввода для пароля – вводимые символы не отображаются, вместо них появляется один и тот же символ. Для `TextField` его можно установить с помощью метода `setEchoChar` (например, `setEchoChar('*')`).

`TextArea` позволяет вводить и просматривать многострочный текст. В конструктор передается количество строк и столбцов, которые определяют размер компонента (вычисляется на основе средней ширины символа). Эти параметры не ограничивают длину вводимого текста – при необходимости появляются полосы прокрутки:



Класс Scrollbar

Класс `Scrollbar` позволяет работать с полосами прокрутки, которые используются для перемещения внутренней области от начальной до конечной позиции. Полоса может быть расположена горизонтально или вертикально. Стрелки на каждом из ее концов служат для перемещения "на один шаг" в соответствующем направлении. "Взявшись" курсором мыши за бегунок, можно переместить его в любую позицию. С помощью кликов мыши по полосе прокрутки, но вне положения бегунка, можно делать перемещение "на страницу" вверх или вниз. Все эти действия хорошо знакомы по многим пользовательским интерфейсам, например, `Windows`. Они полностью поддерживаются компонентом `Scrollbar`.

Конструктор позволяет задавать ориентацию полосы прокрутки — для этого предусмотрены константы `VERTICAL` и `HORIZONTAL`. Кроме того, с помощью конструктора можно задать начальное положение

бегунка, размер "страницы", а также минимальное и максимальное значения, в пределах которых линейка прокрутки может изменять параметр. Для получения и установки текущего состояния полосы прокрутки используются методы `getValue` и `setValue`. Ниже приведен пример, в котором создается и вертикальный, и горизонтальный `Scrollbar`.

```
int height = getHeight(), width = getWidth();
int thickness = 16;
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);
hs.setBounds(0, height - thickness,
          width - thickness, thickness);
vs.setBounds(width - thickness, 0, thickness,
          height - thickness);
```

В этом примере скроллируется, конечно, пустая область:

Наследники Container

Теперь перейдем к рассмотрению стандартных контейнеров AWT.

Класс Panel

Подобно тому, как `Canvas` служит базовым классом для создания своих компонент с особым внешним видом, класс `Panel` является суперклассом для новых контейнеров с особой работой с вложенными компонентами. Впрочем, поскольку `Panel` класс не абстрактный, его можно использовать для иерархической организации сложного пользовательского интерфейса, группируя компоненты в такие простейшие контейнеры.

Класс ScrollPane

Выше был рассмотрен компонент `Scrollbar`, предназначенный для полосы прокрутки. Если стоит задача, например, показать пользователю график некоторой функции с возможностью просмотра для изучения различных областей, необходимо создать две полосы прокрутки, правильно их установить и в дальнейшем обрабатывать все действия пользователя, вычислять новое положение видимой области, перерисовывать график и т.д.

В большинстве случаев все эти задачи может взять на себя контейнер `ScrollPane`. Этот контейнер обладает рядом особенностей. Во-первых, в него можно поместить лишь одну компоненту – при добавлении новой старая удаляется. Во-вторых, отличается работа с вложенным компонентом, чьи границы выходят за границы самого контейнера. Как мы рассматривали раньше, "выступающие" области никогда не будут отображены на экране. В контейнере `ScrollPane` в этом случае появляются полосы прокрутки (горизонтальная или вертикальная), с помощью которых можно промотать видимую область и таким образом увидеть весь компонент полностью. При этом не нужно предпринимать никаких дополнительных действий – надо лишь добавить компонент в `ScrollPane`.

Может вызвать удивление, почему разрешается добавление лишь одного компонента. А если нужно проматывать более сложную конструкцию? Здесь и проявляется польза класса `Panel`. Все элементы собираются в этот простейший контейнер, который, в свою очередь, добавляется в `ScrollPane`.

Конструктор этого класса может принимать параметр, задающий логику

появления полос прокрутки – они могут быть видимы всегда, появляться по мере необходимости, либо не появляться никогда.

Класс Window

Из опыта работы с оконными графическими интерфейсами современных операционных систем мы привыкли к тому, что каждое приложение обладает одним или несколькими окнами. Класс `Window` служит базовым классом для всех окон, порождаемых из Java. Разумеется, он также является интерфейсом к соответствующему окну операционной системы, которая обслуживает окна всех приложений.

Как правило, используется один из двух наследников `Window` – классы `Frame` и `Dialog`, которые будут рассмотрены следующими. Однако экземпляры `Window` не обладают ни рамкой, ни кнопками закрытия или минимизации окна, а потому зачастую используются как заставки (так называемые splash screen).

Конструктор `Window` требует в качестве аргумента ссылку на `Window` или `Frame`. Другими словами, базовые окна не являются самостоятельными, они привязываются к другим окнам.

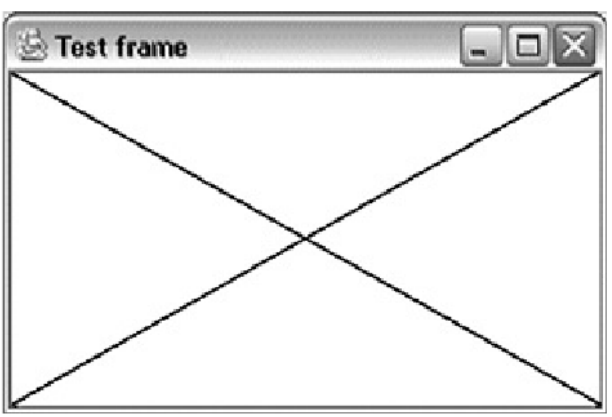
Классы Frame и Dialog

Класс `Frame` предназначен для создания полнофункциональных окон приложений – с полосой заголовка, рамкой, кнопками закрытия, минимизации и максимизации окна. Поскольку `Frame`, как правило, является главным окном приложения, он создается невидимым, чтобы можно было настроить все его параметры, добавить все вложенные контейнеры и компоненты и лишь затем отобразить его в подготовленном виде. Конструктор принимает текстовый параметр – заголовок фрейма.

Рассмотрим пример организации работы с фреймом, который отображает компонент из первого примера лекции ("Алгоритм отрисовки").

```
public class TestCanvas extends Canvas {
  public void paint(Graphics g) {
    g.drawLine(0, 0, getWidth(), getHeight());
    g.drawLine(0, getHeight(),getWidth(), 0);
  }

  public static void main(String arg[]) {
    Frame f = new Frame("Test frame");
    f.setSize(400, 300);
    f.add(new TestCanvas());
    f.setVisible(true);
  }
}
```

Окно запущенной программы будет выглядеть следующим образом:



Обратите внимание, что это окно не будет закрываться по нажатию правой верхней кнопки в заголовке. Причина будет разъяснена ниже.

Если класс `Frame` предназначен для создания основного окна приложения, то экземпляры класса `Dialog` позволяют открывать дополнительные окна для взаимодействия с пользователем. Это может потребоваться, например, для вывода критического сообщения, для ввода параметров и т.д.. Окно диалога обладает стандартным оформлением – полоса заголовка, рамка. В правой части полосы заголовка присутствует лишь одна кнопка – закрытия окна.

Поскольку `Dialog` является несамостоятельным окном, в конструктор

необходимо передать ссылку на родительский фрейм или окно другого диалога. Также можно задать заголовок окна. Как и `Frame`, диалоговое окно создается изначально невидимым.

Важным свойством диалогового окна является модальность. Если диалог модальный, то при его появлении на экране блокируются все пользовательские события, приходящие в родительское окно такого диалога.

Класс FileDialog

Класс `FileDialog` является модальным диалогом (наследником `Dialog`) и позволяет легко организовать работу с файлами. Этот класс предназначен и для открытия файла (open file), и для сохранения (save file). Окно диалога имеет внешний вид, принятый для текущей операционной системы.

Конструктор принимает в качестве параметров ссылку на родительский фрейм, заголовок окна и режим работы. Для задания режима в классе определены две константы – `LOAD` и `SAVE`.

После создания диалога `FileDialog` его необходимо сделать видимым. Затем пользователь делает свой выбор. После закрытия диалога результат можно узнать с помощью методов `getDirectory` (для получения полного имени каталога) и `getFile` (для получения имени файла). Если пользователь нажал кнопку "Отмена" ("Cancel"), то будут возвращены значения `null`.

Обработка пользовательских событий

Весь предыдущий раздел "Дерево компонентов" был посвящен заданию внешнего вида пользовательского интерфейса. Однако до сих пор он был статическим. Перейдем теперь к рассмотрению правил обработки различных событий, которые могут возникать как результат действий пользователя, и не только.

Модель обработки событий построена на основе стандартного шаблона проектирования ООП `Observer/Observable`. В качестве

наблюдаемого объекта выступает тот или иной компонент AWT. Для него можно задать один или несколько классов-наблюдателей. В AWT они называются слушателями (listener) и описываются специальными интерфейсами, название которых оканчивается на слово `Listener`. Когда с наблюдаемым объектом что-то происходит, создается объект "событие" (event), который "посылается" всем слушателям. Так слушатель узнает, например, о действии пользователя и может на него отреагировать.

Каждое событие является подклассом класса `java.util.EventObject`. События пакета AWT, которые и рассматриваются в данной лекции, являются подклассами `java.awt.AWTEvent`. Для удобства классы различных событий и интерфейсы слушателей помещены в отдельный пакет `java.awt.event`.

Прежде, чем углубляться в особенности событий, рассмотрим, как они применяются на практике, на примере простейшего события – `ActionEvent`.

Событие ActionEvent

Рассмотрим появление события `ActionEvent` на примере нажатия на кнопку.

Предположим, в нашем приложении создается кнопка сохранения файла:

```
Button save = new Button("Save");
add(save);
```

Теперь, когда окно приложения с этой кнопкой появится на экране, пользователь сможет нажать ее. В результате AWT сгенерирует `ActionEvent`. Чтобы получить и обработать его, необходимо зарегистрировать слушателя. Название нужного интерфейса прямо следует из названия события – `ActionListener`. В нем всего один метод (в некоторых слушателях их несколько), который имеет один аргумент – `ActionEvent`.

Объявим класс, который реализует этот интерфейс:

```
class SaveButtonListener
    implements ActionListener {
  private Frame parent;
  public SaveButtonListener(Frame parentFrame)
  {
    parent = parentFrame;
  }
  public void actionPerformed(ActionEvent e)
  {
    FileDialog fd = new FileDialog(parent,
              "Save file", FileDialog.SAVE);
    fd.setVisible(true);
    System.out.println(fd.getDirectory()+"/"+
                  fd.getFile());
  }
}
```

Конструктор класса требует в качестве параметра ссылку на родительский фрейм, без которого не удастся создать `FileDialog`. В методе `actionPerformed` класса `ActionListener` описываются действия, которые необходимо предпринять по нажатию пользователем на кнопку. А именно, открывается файловый диалог, с помощью которого определяется путь сохранения файла. Для нашего примера достаточно вывести этот путь на консоль.

Следующий шаг – регистрация слушателя. Название соответствующего метода снова прямо следует из названия интерфейса – `addActionListener`.

```
save.addActionListener(
    new SaveButtonListener(frame));
```

Все необходимое для обработки нажатия пользователем на кнопку сделано. Ниже приведен полный листинг программы:

```
import java.awt.*;
import java.awt.event.*;
```

```
public class Test {
  public static void main(String args[]) {
    Frame frame = new Frame("Test Action");
    frame.setSize(400, 300);
    Panel p = new Panel();
    frame.add(p);
    Button save = new Button("Save");
    save.addActionListener(
        new SaveButtonListener(frame));
    p.add(save);

    frame.setVisible(true);
   }
}

class SaveButtonListener
    implements ActionListener {
  private Frame parent;
  public SaveButtonListener(Frame parentFrame)
  {
    parent = parentFrame;
  }
  public void actionPerformed(ActionEvent e)
  {
    FileDialog fd = new FileDialog(parent,
          "Save file", FileDialog.SAVE);
    fd.setVisible(true);
    System.out.println(fd.getDirectory()+
                       fd.getFile());
  }
}
```

После запуска программы появится фрейм с одной кнопкой "`Save`". Если нажать на нее, откроется файловый диалог. После выбора файла на консоли отображается полный путь к нему.

События AWT

Итак, для каждого события AWT определен класс `XXEvent`, интерфейс `XXListener`, а в компоненте-источнике событий – метод для регистрации слушателя `addXXListener`.

Совсем не обязательно, чтобы одно событие могло порождаться лишь одним компонентом как результат какого-то одного действия пользователя. Например, рассмотренный `ActionEvent` генерируется после нажатия на кнопку (`Button`), после нажатия клавиши `Enter` в поле ввода текста (`TextField`), при двойном щелчке мыши по элементу списка (`List`) и т.д. Узнать, какие события генерирует тот или иной компонент, можно по наличию методов `addXXListener`.

Многие слушатели, в отличие от `ActionListener`, имеют более одного метода для различных видов событий. Например, `MouseMotionListener` наблюдает за движением мыши и имеет два метода – `mouseMoved` (обычное движение) и `mouseDragged` (перемещение с нажатой кнопкой мыши). Иногда бывает необходимо работать лишь с одним методом, остальные приходится объявлять и оставлять пустыми. Чтобы избежать этой бесполезной работы, в пакете `java.awt.event` объявлены вспомогательные классы-адаптеры, например, `MouseMotionAdapter` (название прямо следует из названия слушателя). Эти классы наследуются от `Object` и реализуют соответствующий интерфейс. Адаптер – абстрактный класс, но абстрактных методов в нем нет, они все объявлены пустыми. От такого класса можно наследоваться и переопределить только те методы, которые нужны для приложения.

Классы сообщений (`event`) содержат вспомогательную информацию для обработки события. Метод `getSource()` возвращает объект-источник события. Конкретные наследники `AWTEvent` могут иметь дополнительные методы. Например, `MouseEvent` сообщает о нажатии кнопки мыши, а его методы `getX` и `getY` возвращают координаты точки, где это событие произошло.

Наряду с методом `addXXListener` важную роль играет `removeXXListener`. Поскольку в Java ненужные объекты удаляются из памяти автоматическим сборщиком мусора, который подсчитывает ссылки на объекты, важно следить за тем, чтобы не оставалось ссылок

на ненужные объекты. Если слушатель уже выполнил свою роль и более не нужен, то явно в программе может не остаться ссылок на него, однако компонент будет хранить его в своем списке слушателей. Чтобы дать сработать garbage collector, необходимо воспользоваться методом `removeXXListener`.

Рассмотрим обзорно все события AWT и соответствующих им слушателей, определенных в Java начиная с версии 1.1.

MouseMotionListener и MouseEvent

Это событие рассматривалось выше в примере. Оно отвечает за перемещение курсора мыши. Соответствующий слушатель имеет два метода – `mouseMoved` для обычного перемещения и `mouseDragged` для перемещения с нажатой кнопкой мыши. Обратите внимание, что этот слушатель работает не с событием `MouseMotionEvent` (такого класса нет), а с `MouseEvent`, как и `MouseListener`.

MouseListener и MouseEvent

Этот слушатель имеет методы `mouseEntered` и `mouseExited`. Первый вызывается, когда курсор мыши появляется над компонентом, а второй – когда выходит из его границ.

Для обработки нажатий кнопки мыши служат три метода: `mousePressed`, `mouseReleased` и `mouseClicked`. Если пользователь нажал, а затем отпустил кнопку, то слушатель получит все три события в указанном порядке. Если щелчков было несколько, то метод `getClickCount` класса `MouseEvent` вернет количество. Как уже указывалось, методы `getX` и `getY` возвращают координаты точки, где произошло событие. Чтобы определить, какая кнопка мыши была нажата, нужно воспользоваться методом `getModifiers` и сравнить результат с константами:

```
(event.getModifiers() &
   MouseEvent.BUTTON1_MASK)!=0
```

Как правило, первая кнопка соответствует левой кнопке мыши.

KeyListener и KeyEvent

Этот слушатель отслеживает нажатие клавиш клавиатуры и имеет три метода: `keyTyped`, `keyPressed`, `keyReleased`. Первый отвечает за ввод очередного `Unicode` -символа с клавиатуры. Метод `keyPressed` сигнализирует о нажатии, а `keyReleased` – об отпускании некоторой клавиши. Взаимосвязь между этими событиями может быть нетривиальной. Например, если пользователь нажмет и будет удерживать клавишу `Shift` и в это время нажмет клавишу `"A"`, произойдет одно событие типа `keyTyped` и несколько `keyPressed/Released`. Если пользователь нажмет и будет удерживать, например, пробел, то после первого `keyPressed` будет многократно вызван метод `keyTyped`, а после отпускания – `keyReleased`.

В классе `KeyEvent` определено множество констант, которые позволяют точно идентифицировать, какая клавиша была нажата и в каком состоянии находились служебные клавиши (`Ctrl`, `Alt`, `Shift` и так далее).

FocusListener и FocusEvent

В каждом приложении один из компонентов обладает фокусом и может получать события от клавиатуры. Фокус можно переместить, например, щелкнув мышкой по другому компоненту, либо нажав клавишу `Tab`.

Интерфейс `FocusListener` содержит два метода – `focusGained` и `focusLost` (получен/потерян).

TextListener и TextEvent

Компоненты-наследники `TextComponent` отвечают за ввод текста и порождают `TextEvent`. Слушатель имеет один метод `textValueChanged`. С его помощью можно отслеживать каждое

изменение текста, чтобы, например, выдавать пользователю подсказку, основываясь на первых введенных символах.

ItemListener и ItemEvent

Это событие могут генерировать такие классы, как `Checkbox`, `Choice`, `List`. Слушатель имеет один метод `itemStateChanged`, который сигнализирует об изменении состояния элементов.

AdjustmentListener и AdjustmentEvent

Это событие генерируется компонентом `ScrollBar`. Слушатель имеет один метод `adjustmentValueChanged`, сигнализирующий об изменении состояния полосы прокрутки.

WindowListener и WindowEvent

Это событие сигнализирует об изменении состояния окна (класс `Window` и его наследники).

Рассмотрим особо один из методов слушателя – `windowClosing`. Этот метод вызывается, когда пользователь предпринимает попытку закрыть окно, например, нажимая на соответствующую кнопку в заголовке окна. Мы видели из примеров ранее, что в Java окна при этом не закрываются. Дело в том, что AWT лишь посылает `WindowEvent` в ответ на такое действие, а инициировать закрытие окна должен программист:

```
public class WindowClosingAdapter
      extends WindowAdapter {
   public void windowClosing(WindowEvent e)
   {
     ((Window)e.getSource()).dispose();
   }
}
```

Объявленный адаптер в методе `windowClosing` получает ссылку на окно, от которого пришло событие. Обычно мы пользовались методом `setVisible(false)`, чтобы сделать компонент невидимым. Но поскольку `Window` автоматически порождает окно операционной системы, существует специальный метод `dispose`, который освобождает все системные ресурсы, связанные с этим окном.

Когда окно будет закрыто, у слушателя вызывается еще один метод – `windowClosed`.

ComponentListener и ComponentEvent

Это событие отражает изменение основных параметров компонента – положение, размер, свойство `visible`.

ContainerListener и ContainerEvent

Это событие позволяет отслеживать изменение списка содержащихся в этом контейнере компонент.

С развитием Java в AWT появляются и другие события, например, позволяющие поддерживать колесико мыши. Однако все они работают по точно такой же схеме, а потому их можно легко освоить самостоятельно.

Обработка событий с помощью внутренних классов

Еще в лекции, посвященной объявлению классов, было указано, что в теле класса можно объявлять внутренние классы. До сих пор такая возможность не была востребована в наших примерах, однако обработка событий AWT – как раз удобный случай рассмотреть такие классы на примере анонимных классов.

Предположим, в приложение добавляется кнопка, которой следует добавить слушателя. Зачастую бывает удобно описать логику действий в отдельном методе того же класса. Если вводить слушателя, как делалось

раньше – в отдельном классе, то это сразу порождает ряд неудобств: появляется новый, малосодержательный класс, которому к тому же необходимо передать ссылку на исходный класс и так далее.

Гораздо удобнее поступить следующим образом:

```
Button b = new Button();
b.addActionListener(new ActionListener()
{
  public void actionPerformed(ActionEvent e)
  {
    processButton();
  }
});
```

Рассмотрим подробно, что происходит в этом примере. Сначала создается кнопка, у которой затем вызывается метод `addActionListener`. Обратим внимание на аргумент этого метода. Может сложится впечатление, что производится попытка создать экземпляр интерфейса (`new ActionListener()`), однако это невозможно. Дело меняет фигурная скобка, которая указывает, что порождается экземпляр нового класса, объявление которого последует за этой скобкой. Класс наследуется от `Object` и реализует интерфейс `ActionListener`. Ему необходимо реализовать метод `actionPerformed`, что и делается. Обратите внимание на еще одну важную деталь – в этом методе вызывается `processButton`. Это метод, который мы планировали разместить во внешнем классе. Таким образом, внутренний класс может напрямую обращаться к методам внешнего класса.

Такой класс называется анонимным, он не имеет своего имени. Однако правило, согласно которому компилятор всегда создает `.class` -файл для каждого класса Java, действует и здесь. Если внешний класс называется `Test`, то после компиляции появится файл `Test$1.class`.

Пример приложения, использующего модель событий

В заключение темы, посвященной событиям, рассмотрим пример приложения, которое активно их использует.

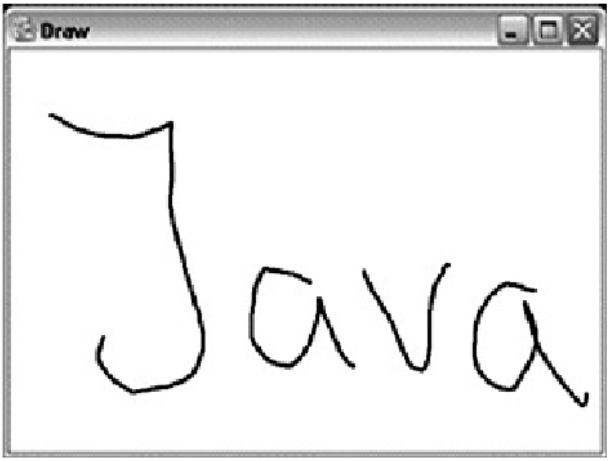
Попробуем написать примитивный графический редактор, который позволяет рисовать с помощью курсора – если перемещать его с нажатой кнопкой мыши, то будет появляться линия. Нажатие пробела очищает поле.

```
import java.awt.*;
import java.awt.event.*;
public class DrawCanvas extends Canvas {
  private int lastX, lastY;
  private int ex, ey;
  private boolean clear=false;

  public DrawCanvas () {
    super();
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        lastX = e.getX();
        lastY = e.getY();
      }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        ex=e.getX();
        ey=e.getY();
        repaint();
      }
    });

    addKeyListener(new KeyAdapter() {
      public void keyTyped(KeyEvent e) {
        if (e.getKeyChar()==' ') {
          clear = true;
          repaint();
        }
      }
```

```
      });
    }

    public void update(Graphics g) {
      if (clear) {
        g.clearRect(0, 0, getWidth(), getHeight());
        clear = false;
      } else {
        g.drawLine(lastX, lastY, ex, ey);
        lastX=ex;
        lastY=ey;
      }
    }
    public static void main(String s[]) {
      final Frame f = new Frame("Draw");
      f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          f.dispose();
        }
      });
      f.setSize(400, 300);

      final Canvas c = new DrawCanvas();
      f.add(c);

      f.setVisible(true);
    }
  }
```

Класс `DrawCanvas` и является тем полем, на котором можно рисовать. В его конструкторе инициализируются все необходимые слушатели. В случае прихода события инициализируется перерисовка (метод `repaint`), логика которой описана в `update`. Запускаемый метод `main` инициализирует `frame`, не забывая про `windowClosing`.

В результате можно что-нибудь нарисовать:



Апплеты

Перейдем к рассмотрению апплетов (`applets`) – Java-приложений, которые исполняются в браузере как часть HTML-страницы. Это означает, что такие приложения всегда визуальные. Действительно, класс `Applet` является наследником AWT-компонента `Panel`. Сам класс находится в пакете `java.applet`.

Жизненный цикл апплета

Важным вопросом для понимания работы апплетов является их жизненный цикл. Он описывается четырьмя методами.

init

Этот метод вызывается браузером при конструировании апплета. Зачастую все инициализирующие действия описываются здесь, а не в конструкторе. Это может быть, например, создание AWT-компонент, запуск потоков исполнения, установление сетевых соединений и т.д.

start

Этот метод вызывается после инициализации апплета. Он нужен по следующей причине. Апплет может содержать какие-то динамические части, например, анимацию или бегущую строку. Если пользователь воспользуется какой-нибудь ссылкой и уйдет со страницы с апплетом, браузер не станет его уничтожать – ведь пользователь может вернуться (нажав в браузере кнопку `Back`), и он будет ожидать, что апплет сохранит свое состояние. Значит, апплет может оказаться в неактивном состоянии, когда лучше приостановить динамические процессы для экономии системных ресурсов.

Метод `start` сигнализирует о переходе в активное состояние.

stop

Этот метод всегда вызывается после метода `start` и сигнализирует о переходе в пассивное состояние.

destroy

По завершении работы апплет необходимо корректно удалить, чтобы он имел возможность освободить занимаемые ресурсы. Для этого браузер вызывает метод `destroy`.

В остальном апплет является полноценным AWT-компонентом и в методе `init` может добавить другие компоненты для создания пользовательского интерфейса, или даже открыть новый фрейм. Единственное, но существенное ограничение – это условие безопасности. Ведь код апплета скачивается по сети, а значит, может содержать в себе опасные действия. Поэтому браузер запускает виртуальную машину с ограничениями – апплетам запрещено обращаться к файловой структуре, запрещено устанавливать сетевые соединения с кем-либо, кроме сервера, откуда они были загружены, все вновь открываемые окна помечаются предупреждением. Более того, пользователь может так настроить свой браузер, что вовсе запретит исполнение Java. Можно, напротив, позволить апплетам то же, что и локальным приложениям.

Есть и еще одно ограничение – версия Java, поддерживаемая браузером. Как говорилось в первой лекции, самый популярный на данный момент браузер – MS Internet Explorer – остановился на поддержке лишь Java 1.1, и то не в полном объеме. В некоторых случаях можно воспользоваться дополнительным продуктом Sun – Java Plug-in, который позволяет установить на браузер JVM любой версии.

Продолжим рассмотрение апплетов.

HTML-тег

Раз апплет является частью HTML -страницы, значит, необходимо каким-то образом указать, где именно он располагается. Для этого служит специальный тег <applet>. Синтаксис тега <APPLET> в настоящее время таков:

```
<APPLET
  CODE = appletFile
  WIDTH = pixels
  HEIGHT = pixels
  [ARCHIVE = jarFiles]
  [CODEBASE = codebaseURL]
  [ALT = alternateText]
  [NAME = appletInstanceName]
  [ALIGN = alignment]
  [VSPACE = pixels]
  [HSPACE = pixels]
>
[HTML-текст, отображаемый при отсутствии
поддержки Java]
</APPLET>
```

- CODE = appletClassFile ; CODE – обязательный атрибут, задающий имя файла, в котором содержится описание класса апплета. Имя файла задается относительно codebase, то есть либо от текущего каталога, либо от каталога, указанного в атрибуте

CODEBASE.

- WIDTH = pixels
- HEIGHT = pixels ; WIDTH и HEIGHT - обязательные атрибуты, задающие размер области апплета на HTML -странице.
- ARCHIVE = jarFiles ; Этот необязательный атрибут задает список jar -файлов (разделяется запятыми), которые предварительно загружаются в Web -браузер. В них могут содержаться классы, изображения, звук и любые другие ресурсы, необходимые апплету. Архивирование наиболее необходимо именно апплетам, так как их код и ресурсы передаются через сеть.
- CODEBASE = codebaseURL ; CODEBASE – необязательный атрибут, задающий базовый URL кода апплета; является 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. С помощью этого атрибута можно на странице одного сайта разместить апплет, находящийся на другом сайте.
- 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 присвоения имени данному экземпляру апплета. Имена апплетам нужны для того, чтобы другие апплеты на этой же странице могли находить их и общаться с ними, а также для обращений из Java Script.
- ALIGN = alignment
- VSPACE = pixels
- HSPACE = pixels ; Эти три необязательных атрибута предназначены для того же, что и в теге IMG. ALIGN задает стиль выравнивания апплета, возможные значения: LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM, ABSBOTTOM.

Следующие два задают ширину свободного пространства в пикселах сверху и снизу апплета (`VSPACE`), а также слева и справа от него (`HSPACE`).

Приведем пример простейшего апплета:

```
import java.applet.*;
import java.awt.*;
public class HelloApplet extends Applet {
   public void init() {
      add(new Label("Hello"));
   }
}
```

HTML-тег для него:

```
<applet code=HelloApplet.class
      width=200 height =50>
</applet>
```

Передача параметров

Существует очень полезная возможность передавать из HTML параметры в апплет. Таким образом, можно настроить программу без необходимости менять ее исходный код.

В HTML параметры указываются следующим образом:

```
<applet code=HelloApplet.class
      width=200 height =50>
<param name="text" value="Hello!!!">
</applet>
```

В апплете значение параметров считывается таким образом:

```
import java.applet.*;
import java.awt.*;

public class HelloApplet extends Applet {
```

```
    public void init() {
      String text = getParameter("text");
      add(new Label(text));
    }
  }
```

Теперь выводимый текст можно настраивать из HTML.

Интерфейс AppletContext

Доступ к этому интерфейсу из апплета предоставляется методом `getAppletContext`. С его помощью апплет может взаимодействовать со страницей, откуда он был загружен, и с браузером. Так, именно в этом интерфейсе определен метод `getApplet`, с помощью которого можно обратиться по имени к другому апплету, находящемуся на той же странице.

Метод `showStatus` меняет текст поля статуса в окне браузера.

Метод `showDocument` позволяет загрузить новую страницу в браузер.

Менеджеры компоновки

При размещении компонент в контейнере поначалу всегда кажется удобным задавать их размер и положение явно с помощью метода `setBounds`.

Однако дальше становятся очевидны недостатки такого подхода. Например, удобно предоставить пользователю возможность изменять размер фрейма, а это означает необходимость перестраивать компоненты. Развитие приложения также может привести к добавлению или удалению компонента, после чего придется пересчитывать координаты оставшихся элементов.

Есть проблемы и другого характера. Мы указывали, что `JVM` выбирает шрифты из имеющихся в системе. Поэтому под разными платформами они могут оказаться разного размера или наклона. В результате приложение, красиво смотрящееся на машине разработчика, может

"поплыть" у клиента. Даже под одной платформой пользователь может сменить системные настройки, вследствие чего внешний вид приложения может измениться не в лучшую сторону.

Все эти соображения наводят на мысль, что было бы полезно каким-то образом автоматизировать расположение компонентов. Именно для этой цели служат менеджеры компоновки. Их задача – вычислить и установить размер и местоположение компонентов в контейнере. Вообще говоря, они могут использовать следующий набор параметров для своих вычислений:

- размер контейнера;
- начальное положение и размер компонента;
- его порядковый номер в наборе компонентов;
- специальный параметр-ограничитель (constraint), который может быть установлен при добавлении компонента.

В AWT каждый контейнер обладает менеджером компоновки. Если он равен `null`, то используются явные параметры компонентов. Настоящие же классы менеджеров должны реализовывать интерфейс `LayoutManager`. Этот интерфейс принимает в качестве constraints строку (`String`). Со временем это было признано недостаточно гибким (фирмы стали разрабатывать и предлагать свои менеджеры, обладающие самой разной функциональностью). Поэтому был добавлен новый интерфейс – `LayoutManager2`, принимающий в качестве ограничителя `constraints`.

Рассмотрим работу нескольких наиболее распространенных менеджеров компоновки. Но перед этим отметим общий для них всех факт. Дело в том, что не всегда вся область контейнера подходит для размещения в ней компонент. Например, фрейм имеет рамку и полосу заголовка. В результате его полезная площадь меньше. Поэтому все менеджеры компоновки начинают с обращения к методу `getInsets` класса `Container`. Этот метод возвращает значение типа `Insets`. Это класс, который имеет четыре открытых поля – `top`, `right`, `bottom`, `left`, значения которых описывают отступы со всех четырех сторон, которые необходимо сделать, чтобы получить область, доступную для расположения компонент.

Класс FlowLayout

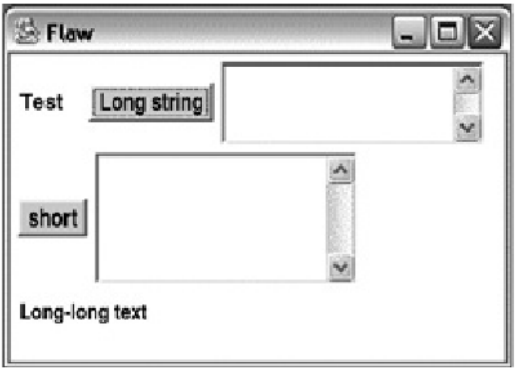
Этот менеджер является стандартным для `Panel`. Он не меняет размер компонент, а только располагает их один за другим в линию, как буквы в строке. Когда заканчивается первая "строка", он переходит на следующую, и так далее, пока либо не закончится область контейнера, либо не будут расположены все компоненты.

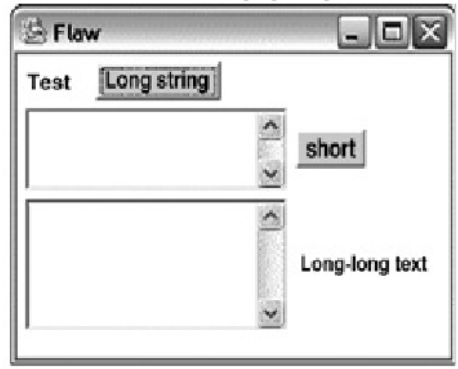
В качестве параметров конструктору можно передать значение выравнивания по горизонтали (определены константы `LEFT`, `RIGHT`, `CENTER` – значение по умолчанию), а также величину необходимых отступов между компонентами по вертикали (`vgap`) и горизонтали (`hgap`). Их значение по умолчанию – 5 пикселов.

Рассмотрим пример:

```
final Frame f = new Frame("Flaw");
f.setSize(400, 300);
f.setLayout(new FlowLayout(FlowLayout.LEFT));
f.add(new Label("Test"));
f.add(new Button("Long string"));
f.add(new TextArea(2, 20));
f.add(new Button("short"));
f.add(new TextArea(4, 20));
f.add(new Label("Long-long text"));
f.setVisible(true);
```

Если теперь менять размер этого фрейма, то можно видеть, как перераспределяются компоненты:





Класс BorderLayout

Этот менеджер является стандартным для контейнера `Window` и его наследников `Frame` и `Dialog`.

`BorderLayout` использует ограничитель. При добавлении компонента необходимо указать одну из 5 констант, определенных в этом классе: `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER` (используется по умолчанию). Первыми располагаются северный и южный компонент. Их высота не изменяется, а ширина становится равной ширине контейнера. Северный компонент помещается на самый верх контейнера, южный – вниз. Затем располагаются восточный и западный компоненты. Их ширина не меняется, а высота становится равной высоте контейнера за вычетом места, которое заняли первые две компоненты. Наконец, все оставшееся место занимает центральная компонента.

Рассмотрим пример:

```
final Frame f = new Frame("Border");
f.setSize(200, 150);
f.add(new Button("North"),
    BorderLayout.NORTH);
f.add(new Button("South"),
    BorderLayout.SOUTH);
f.add(new Button("West"),
    BorderLayout.WEST);
```

```
f.add(new Button("East"),
      BorderLayout.EAST);
f.add(new Button("Center"),
      BorderLayout.CENTER);
f.setVisible(true);
```

Вот как выглядит такой фрейм:

И в этом менеджере есть параметры `hgap` и `vgap` (по умолчанию их значение равно нулю).

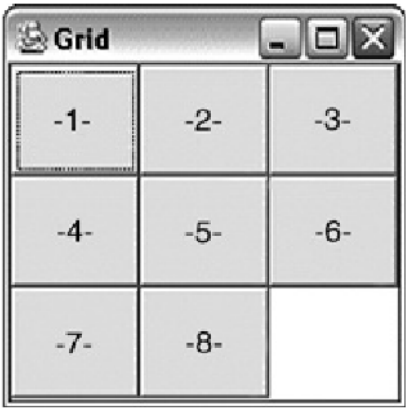
Класс GridLayout

Этот менеджер поступает следующим образом – он разделяет весь контейнер на одинаковые прямоугольные сектора (отсюда и его название – решетка). Далее последовательно каждый компонент полностью занимает свой сектор (таким образом, они все становятся одинакового размера).

В конструкторе указывается количество строк и столбцов для разбиения:

```
final Frame f = new Frame("Grid");
f.setSize(200, 200);
f.setLayout(new GridLayout(3, 3));
for (int i=0; i<8; i++) {
  f.add(new Button("-"+(i+1)+"-"));
}
f.setVisible(true);
```

Вот как выглядит такой фрейм:



И в этом менеджере есть параметры `hgap` и `vgap` (по умолчанию их значение равно нулю).

Класс CardLayout

Этот менеджер ведет себя подобно колоде карт. В один момент виден лишь один компонент, и он занимает всю область контейнера. Программист может управлять тем, какой именно компонент показывается пользователю.

Заключение

Библиотека AWT имеет множество классов и внутренних механизмов. Новые версии Java добавляют новые возможности и пересматривают старые. Тем не менее, основные концепции были подробно рассмотрены в этой лекции и на их основе можно построить полнофункциональный графический интерфейс пользователя (`GUI`).

Стандартные компоненты AWT иерархически упорядочены в дерево наследования с классом `Component` в вершине. Важным его наследником является класс `Container`, который может хранить набор компонентов. Прямые наследники `Component` составляют набор управляющих элементов ("контролов", от англ. controls), а наследники `Container` – набор контейнеров для группировки и расположения компонентов. Для упрощения размещения отдельных элементов пользовательского интерфейса применяются менеджеры

компоновки (Layout managers).

Особое место в AWT занимает процедура отрисовки компонентов, которая может инициироваться как операционной системой, так и программой. Специальные классы служат для задания таких атрибутов, как цвет, шрифт и т.д.

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`.

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.

Апплеты – небольшие программы, предназначенные для работы в браузерах как небольшие части `HTML` -страниц. Класс `java.applet.Applet` является наследником `Panel`, а потому обладает всеми свойствами AWT-компонент. Были представлены этапы жизненного цикла апплета, отличного от цикла обычного приложения, которое запускается методом `main`. Для размещения апплета на `HTML` - странице необходимо использовать специальный тег `<applet>`. Кроме этого, можно указывать специальные параметры, чтобы апплет настраивался без перекомпиляции кода.

Потоки выполнения. Синхронизация

В этой лекции завершается описание ключевых особенностей Java. Последняя тема раскрывает особенности создания многопоточных приложений - такая возможность присутствует в языке, начиная с самых первых версий. Первый вопрос - как на много- и, самое интересное, однопроцессорных машинах выполняется несколько потоков одновременно и для чего они нужны в программе. Затем описываются классы, необходимые для создания, запуска и управления потоками в Java. При одновременной работе с данными из нескольких мест возникает проблема синхронного доступа, блокировок и, как следствие, взаимных блокировок. Изучаются все механизмы, предусмотренные в языке для корректной организации такой логики работы.

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 лекции сначала рассматриваются преимущества такого подхода, способы реализации и возможные недостатки.

Затем описываются базовые классы Java, которые позволяют запускать потоки исполнения и управлять ими. 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 необходимыми 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`.

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 небольшие интервалы. Перед началом каждого интервала принимается решение, какой именно поток выполнения будет отрабатываться на протяжении этого кванта времени. За счет 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 запущенного 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 бытует 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 завершил 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 станет обслуживать несколько клиентов одновременно. Можно легко представить себе пример из сферы обслуживания, когда имеется 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 необходимо рассчитывать и перерисовывать изменяющееся состояние игрового поля.

Понятно, что в случае отсутствия поддержки многопоточности для реализации подобных приложений потребовалось бы реализовывать квантование 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 куда более громоздко, чем независимое описание действий для каждого потока.

Следующее преимущество проистекает из того, что компьютер состоит не только из одного или нескольких процессоров. Вычислительное устройство – лишь один из ресурсов, необходимых для выполнения

задач. Всегда есть оперативная память, дисковая подсистема, сетевые подключения, периферия и т.д. Предположим, пользователю требуется распечатать большой документ и скачать большой файл из сети. Очевидно, что обе задачи требуют совсем незначительного участия процессора, а основные необходимые ресурсы, которые будут задействованы на пределе возможностей, у них разные – сетевое подключение и принтер. Значит, если выполнять задачи одновременно, то 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, два часа, то вполне вероятно, что и при одновременном исполнении потребуется не более тех же двух 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 то 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, два 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 ошибку. Получается, что процессор выполняет большее количество вычислений, но при этом приносит гораздо меньше пользы.

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 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 'а 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.

Базовые классы для работы с потоками

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 выполняться в новом потоке исполнения. Чтобы запустить его, необходимо создать экземпляр класса-наследника и вызвать унаследованный метод `start()`, который сообщает виртуальной машине, что требуется запустить новый поток исполнения и начать выполнять в нем 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.

Интерфейс Runnable

Описанный подход имеет один недостаток. Поскольку в Java множественное наследование отсутствует, требование наследоваться от Thread может привести к конфликту. Если еще раз посмотреть на приведенный выше пример, станет понятно, 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()`, реализующим необходимую функциональность, были объединены в одном экземпляре класса `MyThread`, то теперь они разделены. Какой из двух подходов удобней, решается 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()`.

Работа с приоритетами

Рассмотрим, как в Java можно назначать потокам приоритеты. Для этого в классе `Thread` существуют методы `getPriority()` и `setPriority()`, а также объявлены три константы:

MIN_PRIORITY
MAX_PRIORITY
NORM_PRIORITY

Из названия понятно, что их значения описывают минимальное, максимальное 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
```

Мы видим, 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 приоритетами, 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 запускаемых потоков. 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 мы видим, что чем выше приоритет, тем быстрее отрабатывает поток. Тем не менее, весьма показательно, что поток с минимальным приоритетом (`Thread 0`) все же 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 500000 вычислений, а не 50000, и выводить сообщение каждое 1000-е вычисление, а не 10000-е), то можно будет наглядно увидеть, что все три потока имеют возможность выполнять свои действия одновременно, просто более высокий приоритет позволяет выполнять их чаще.

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
  // находиться 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 достижении 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) {}
      }
    }
  }
```

Пример 12.1.

В этом примере происходит следующее. Потоки `ThreadTest` имеют некоторое стартовое значение, передаваемое 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 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,
```

Пример 12.2.

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 активизировать метод, который приостановил свою работу. Например, если поток занят выполнением метода `sleep()`, то есть бездействует на протяжении указанного периода времени, 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 затормозит работу остальных потоков.

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 невозможно предугадать, 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 потокам время отработать
      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);
  }
}
```

Пример 12.3.

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 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 10000, 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 длина непредсказуема и оценить их последствия невозможно.

Для более глубокого понимания принципов многопоточной работы в Java рассмотрим организацию памяти в виртуальной машине для нескольких потоков.

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, то их значения не могут быть доступны другим потокам, поэтому они не представляют интереса.

Для каждого потока создается его собственная рабочая память (working memory), в которую перед использованием копируются значения всех переменных.

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 дальнейшего хранения;
- `write` – сохраняет в основном хранилище значение переменной, переданной командой store.

Подчеркнем, что перечисленные команды не являются методами каких-либо классов, они недоступны 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 текущего значения и присвоения нового. Кроме того, должны осуществляться действия по передаче значений в основное хранилище и из него. 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 эти изменения могут быть переданы в основное хранилище в обратном порядке.

Поток создается с чистой рабочей памятью и должен перед использованием загрузить все необходимые переменные из основного хранилища. Любая переменная сначала создается в основном хранилище и лишь затем копируется в рабочую память потоков, которые будут ее применять.

Таким образом, потоки никогда не взаимодействуют друг с другом напрямую, только через главное хранилище.

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 с 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 должна проводиться 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.

Следует 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 выполняются 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.

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 данным объектом нельзя пользоваться, что его поля и методы становятся недоступными,– это не так. Единственное действие, которое становится невозможным,– установка этой же блокировки другим потоком, 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. При завершении исполнения (как успешном, так и в случае ошибок) 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 методы `static synchronized`. 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();
  }
}
```

Пример 12.4.

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. Очевидно, что в результате оба потока "замрут", не смогут продолжить свое выполнение. Первый поток будет ждать освобождения 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 "OK"), 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 всегда выполнялась бы 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 им объекты будут оставаться недоступными для блокировок со стороны других потоков, а это потенциальный `deadlock`. Такие ситуации крайне сложно выявить путем тестирования и отладки, поэтому вопросам синхронизации надо уделять много времени на этапе проектирования.

Методы wait(), notify(), notifyAll() класса Object

Наконец, перейдем к рассмотрению трех методов класса `Object`, завершая 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 этого же объекта метод `notifyAll()`, который пробуждает все потоки из `wait-set`. Метод `notify()` пробуждает один случайно выбранный поток из данного 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 обращения 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()`, 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();
    }
}
```

Пример 12.5.

Результатом работы программы будет:

```
Thread 3 after notifyAll()
Thread 1 after wait()
Thread 2 after wait()
```

Пример 12.6.

Рассмотрим, что произошло. 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, это не помешало и потоку `2` зайти в `synchronized` -блок, а затем 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 того, 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.

Заключение

В этой лекции были рассмотрены принципы построения многопоточного приложения. В начале разбирались достоинства и недостатки такой архитектуры – как правило ОС не выделяет отдельный процессор под каждый процесс, а значит применяется процедура time slicing. Было выделено три признака, указывающие на целесообразность запуска нескольких потоков в рамках программы.

Основу работы с потоками в Java составляют интерфейс `Runnable` и класс `Thread`. С их помощью можно запускать и останавливать потоки, менять их свойства, среди которых основные: приоритет и свойство daemon. Главная проблема, возникающая в таких программах - одновременный доступ нескольких потоков к одним и тем же данным, в первую очередь -– к полям объектов. Для понимания, как в Java решается эта задача, был сделан краткий обзор по организации памяти в JVM, работы с переменными и блокировками. Блокировки, несмотря на название, сами по себе не ограничивают доступ к переменной. Программист использует их через ключевое слово `synchronized`, которое может быть указано в сигнатуре метода или в начале блока. В результате выполнение не будет продолжено, пока блокировка не освободится.

Новый механизм порождает новую проблему - взаимные блокировки (deadlock), к которой программист всегда должен быть готов, тем более, что Java не имеет встроенных средств для определения такой ситуации. В лекции разбирался пример, как организовать работу программы без "зависания" ожидающих потоков.

В завершение рассматривались специализированные методы базового класса `Object`, которые также позволяют управлять

последовательностью работы потоков.

Пакет java.lang

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 рассматривавшиеся ранее,– для работы с исключительными ситуациями и потоками исполнения.

Введение

В состав пакета `java.lang` входят классы, составляющие основу для всех других, и поэтому он является наиболее важным из всех, входящих в Java API. Поскольку без него не может обойтись ни один класс, каждый модуль компиляции содержит неявное импортирование этого пакета (`import java.lang.*;`).

Перечислим классы, составляющие основу пакета.

`Object` – является корневым в иерархии классов.

`Class` – экземпляры этого класса являются описаниями объектных типов в памяти JVM.

`String` – представляет собой символьную строку, содержит средства работы с нею.

`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 набор базовых 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` – позволяет запускать и работать с потоками выполнения в Java. `Runnable` – может использоваться в сочетании с классом `Thread` для описания потоков выполнения.

`ThreadGroup` – позволяет объединять потоки в группу и производить действия сразу над всеми потоками в ней. Существуют ограничения по безопасности на манипуляции с потоками из других групп.

`System` – содержит полезные поля и методы для работы системного уровня.

`Runtime` – позволяет приложению взаимодействовать с окружением, в котором оно запущено.

`Process` – представляет интерфейс к внешней программе, запущенной при помощи `Runtime`.

`ClassLoader` – отвечает за загрузку описания классов в память JVM.

`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;

`Comparable` – позволяет упорядочивать (сортировать, сравнивать) объекты каждого класса, реализующего этот интерфейс.

Object

Класс `Object` является базовым для всех остальных классов. Он

определяет методы, которые поддерживаются любым классом в Java.

Метод `public final native Class getClass()` возвращает объект типа Class, соответствующий классу объекта. Этот метод уже рассматривался в лекции 4.

Метод `public boolean equals(Object obj)` определяет, являются ли объекты одинаковыми. Если оператор `==` проверяет равенство по ссылке (указывают на один и тот же объект), то метод `equals()` – равенство по значению (состояния объектов одинаковы). Поскольку класс `Object` не содержит полей, реализация в нем этого метода такова, что значение `true` будет возвращено только в случае равенства по ссылке, то есть:

```
public boolean equals(Object obj) {
        return (this == obj);
}
```

В классах-наследниках этот метод при необходимости может быть переопределен, чтобы поддержать расширенное состояние объекта (например, если добавилось поле, характеризующее состояние). Рассмотрим сравнение объектов-оберток целых чисел (класс `Integer`). Оно должно по всей логике возвращать значение `true`, если равны значения `int` чисел, которые обернуты, даже если это два различных объекта.

Метод `equals()` может быть переопределен любым способом (например, всегда возвращать `false`, или, наоборот, `true`) – компилятор, конечно же, не будет проводить анализ реализации и давать рекомендации. Однако существуют соглашения, которые необходимо соблюдать, чтобы программа имела предсказуемое поведение, в том числе и с точки зрения других программистов:

- рефлексивность: для любой объектной ссылки `x`, отличной от `null`, вызов `x.equals(x)` возвращает `true` ;
- симметричность: для любых объектных ссылок `x` и `y`, вызов `x.equals(y)` возвращает `true` только в том случае, если вызов `y.equals(x)` возвращает `true` ;
- транзитивность: для любых объектных ссылок `x`, `y` и `z`, если

`x.equals(y)` возвращает true и `y.equals(z)` возвращает `true`, то вызов `x.equals(z)` должен вернуть `true` ;

- 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`);
- 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));
  }
}
```

Пример 13.1.

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
```

Пример 13.2.

В этом примере метод `equals()` у класса `Rectangle` был переопределен таким образом, чтобы прямоугольники были равны, если их можно наложить друг на друга (геометрическое равенство).

Большинство стандартных классов переопределяет этот метод, строго следуя всем соглашениям.

Метод `public int hashCode()` возвращает хеш-код (*`hash code`*) для объекта. Хеш-код – это целое число, которое сопоставляется с данным объектом. Оно позволяет организовать хранение набора объектов с возможностью быстрой выборки (стандартная реализация такого механизма присутствует в Java и будет описана в следующей лекции).

Для этого метода также принят ряд соглашений, которым стоит следовать при переопределении:

- если два объекта идентичны, то есть вызов метода `equals(Object)` возвращает `true`, то вызов метода `hashCode()` у каждого из этих двух объектов должен возвращать одно и то же значение;
- во время одного запуска программы для одного объекта при вызове метода `hashCode()` должно возвращаться одно и то же значение, если между этими вызовами не были затронуты данные, используемые для проверки объектов на идентичность в методе `equals()`. Это число не обязательно должно быть одним и тем же при повторном запуске той же программы, даже если все

данные будут идентичны.

В классе `Object` этот метод реализован на уровне JVM. Сама виртуальная машина генерирует хеш-код, основываясь на расположении объекта в памяти. Это позволяет для различных объектов (неравенство по ссылке) получать различные хеш-коды.

В силу первого соглашения при переопределении метода `equals()` необходимо переопределить также метод `hashCode()`. При этом нужно стремиться, во-первых, к тому, чтобы метод возвращал значение как можно быстрее, иначе основная цель – быстрая выборка – не будет достигнута. Во-вторых, желательно для различных объектов, то есть когда метод `equals(Object)` возвращает `false`, генерировать различные хеш-коды. В этом случае хеш-таблицы будут работать особенно эффективно. Однако, понятно, что это не всегда возможно. Диапазон значений `int` – 2^{32}, а количество различных строк, или двумерных точек, с координатами типа `int` – заведомо больше.

Большинство стандартных классов переопределяет этот метод, строго следуя всем соглашениям.

Метод `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 шестнадцатеричном формате.

В классах-наследниках этот метод может быть переопределен для получения более наглядного описания 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 )
```

Большинство стандартных классов переопределяет этот метод. Экземпляры класса `String` возвращают ссылку на самих себя (`this`).

Метод `wait()`, `notify()`, `notifyAll()` используются для поддержки многопоточности и были подробно рассмотрены в лекции 12. Они определены с атрибутом `final` и не могут быть переопределены в классах-наследниках.

Метод `protected void finalize() throws Throwable` вызывается Java-машиной перед тем, как garbage collector (сборщик мусора) освободит память, занимаемую объектом. Этот метод уже подробно рассматривался в лекции 4.

Метод `protected native Object clone() throws`

`CloneNotSupportedException` создает копию объекта. Механизм клонирования подробно рассматривался в лекции 9.

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 описания классов из `class` -файлов. Получить экземпляр `Class` для конкретного класса можно с помощью метода `forName()`:

`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 `ClassLoader`, который применялся для загрузки вызывающего класса. При этом класс будет инициализирован, только если значение `initialize` равно `true` и класс не был инициализирован ранее.

Зачастую 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).

`public Object newInstance()` – создает и возвращает объект класса, который представляется данным экземпляром `Class`. Создание будет происходить с использованием конструктора без параметров. Если такового в классе нет, будет брошено исключение `InstantiationException`. Это же исключение будет брошено, если объект `Class` соответствует абстрактному классу, интерфейсу, или какая-то другая причина помешала созданию нового объекта.

Каждому методу, полю, конструктору класса также соответствуют объекты, список которых можно получить вызовом соответствующих методов объекта `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 – этот механизм называется reflection . Необходимые классы содержатся в пакете `java.lang.reflection`.

Рассмотрим пример использования этой технологии:

```
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 class 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);
          } catch(InstantiationException e) {
            System.out.println("Exception: " + e);
          }
        }
      }
    }
```

Пример 13.3.

Если запустить эту программу, на экран будет выведено следующее:

```
look for class for: demo.lang.Automobile
creating vehicle...
create vehicle: class demo.lang.Automobile
Automobile go!
look for class for: demo.lang.Truck
creating vehicle...
Exception: java.lang.InstantiationException
look for class for: demo.lang.Tank
Class not found: java.lang.ClassNotFoundException: demo.lang.Tank
```

Пример 13.4.

В этом примере делается попытка создать с помощью `reflection` три объекта. Имена классов, от которых они должны быть порождены, записаны в массив `vehicleNames`. Объект класса `Automobile` был успешно создан, причем, дальнейшая работа с ним велась через интерфейс `Vehicle`. Класс `Truck` был найден, но при попытке создания объекта было брошено, а затем обработано исключение `java.lang.InstantiationException`, поскольку конструктор без параметров отсутствует. Класс `java.lang.Tank` определен не был и поэтому при попытке получить соответствующий ему объект `Class` было выброшено исключение `java.lang.ClassNotFoundException`.

Классы-обертки

Во многих случаях предпочтительней работать именно с объектами, а не с примитивными типами. Так, например, при использовании коллекций просто необходимо значения примитивных типов 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 (если необходим объект, хранящий другое значение, его нужно создать заново), 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`, ссылающееся на объект `Class`, соответствующий примитивному оборачиваемому типу.

Также классы-обертки содержат статические методы для обеспечения удобного манипулирования соответствующими примитивными типами, например, преобразование к строковому виду.

В таблице 13.1 приведены примитивные типы и соответствующие им классы-обертки.

Таблица 13.1. 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 объявлены методы, возвращающие числовое значение во всех числовых форматах Java (`byte`, `short`, `int`, `long`, `float` и `double`).

Все классы-обертки реализуют интерфейс `Comparable`. Все классы-обертки числовых типов имеют метод `equals(Object)`, сравнивающий примитивные значения объектов.

Рассмотрим более подробно некоторые из классов-оберток.

Integer

Наиболее часто используемые статические методы:

- `public static int parseInt(String s)` – преобразует строку, представляющую десятичную запись целого числа, в `int` ;
- `public static int parseInt(String s, int radix)` – преобразует строку, представляющую запись целого числа в системе счисления *`radix`*, в `int`.

Оба 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)
```

Данные методы выполняют аналогичную работу, только результат представляют в виде объекта-обертки.

Существует также два конструктора для создания экземпляров класса `Integer`:

- `Integer(String s)` – конструктор, принимающий в качестве параметра строку, представляющую числовое значение.
- `Integer(int i)` – конструктор, принимающий числовое значение.

`public 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:

- `public static String toOctalString(int i)` – восьмеричное;
- `public static String toBinaryString(int i)` – двоичное;
- `public static String toHexString(int i)` – шестнадцатеричное.

Имеются также две статические константы:

- `Integer.MIN_VALUE` – минимальное `int` значение;
- `Integer.MAX_VALUE` – максимальное `int` значение.

Аналогичные константы, описывающие границы соответствующих типов, определены и для всех остальных классов-оберток числовых примитивных типов.

`public int intValue()` возвращает значение примитивного типа для данного объекта `Integer`. Классы-обертки остальных примитивных целочисленных типов – `Byte`, `Short`, `Long` – содержат аналогичные методы и константы (определенные для соответствующих типов: `byte`, `short`, `long`).

Рассмотрим 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. 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 (они подробно разбирались в лекции 4):

- `NEGATIVE_INFINITY` – отрицательная бесконечность;
- `POSITIVE_INFINITY` – положительная бесконечность;
- `NaN` – нечисловое значение.

Кроме того, другой смысл имеет значение `MIN_VALUE` – вместо наименьшего значения оно представляет минимальное положительное (строго > 0) значение, которое может быть представлено этим примитивным типом.

Кроме классов-оберток для примитивных числовых типов, таковые определены и для остальных примитивных типов Java.

Character

Реализует интерфейсы `Comparable` и `Serializable`.

Из конструкторов имеет только один, принимающий `char` в качестве параметра.

Кроме стандартных методов `equals()`, `hashCode()`, `toString()`, содержит только два нестатических метода:

- `public char charValue()` – возвращает обернутое значение Character;
- `public int compareTo(Character anotherCharacter)` – сравнивает обернутые значения char как числа, то есть возвращает значение `return this.value – anotherCharacter.value`.

Также 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, но все они просты и логика их работы понятна из названия. Большинство из них - это методы, принимающие `char` и проверяющие всевозможные свойства. Например:

```
public static boolean isDigit(char c)
// проверяет, является ли char цифрой.
```

Эти методы возвращают значение истина или ложь, в соответствии с тем, выполнен ли критерий проверки.

Boolean

Представляет класс-обертку для примитивного типа `boolean`.

Реализует интерфейс `java.io.Serializable` и во всем напоминает аналогичные классы-обертки.

Для получения примитивного значения используется метод `booleanValue()`.

Void

Этот класс-обертка, в отличие 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 также являются классами-утилитами, т.е.

предоставляют набор статических методов для работы с примитивными типами;
- классы-обертки являются неизменяемыми.

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 основание натурального логарифма. Часто этот класс еще называют классом-утилитой (Utility class). Так как все методы класса статические, нет необходимости создавать экземпляр данного класса, потому он и не имеет открытого конструктора. Нельзя также и наследоваться от этого класса, так как он объявлен с модификатором `final`.

Итак, константы определены следующим образом:

- `public static final double Math.PI` – задает число π ("пи");
- `public static final double Math.E` – основание натурального логарифма.

В таблице 13.2 приведены все методы класса и дано их краткое описание.

Таблица 13.2. Методы класса Math и их краткое описание.

Возвращаемое значение	Имя метода и параметры	Описание
…	`abs(… a)`	абсолютное значение (модуль) для типов `double`, `float`, `int`, `long`
`double`	`acos(double a)`	арккосинус
`double`	`asin(double a)`	арксинус
`double`	`atan(double a)`	арктангенс
`double`	`ceil(double a)`	наименьшее целое число, большее `a`

`double`	`floor(double a)`	наименьшее целое число, меньшее `a`
`double`	`IEEEremainder (double a, double b)`	остаток по стандарту IEEE 754 (подробно рассматривался в лекции 3)
`double`	`sin(double a)`	синус (здесь и далее: аргумент должен быть в радианах)
`double`	`cos(double a)`	косинус
`double`	`tan(double a)`	тангенс
`double`	`exp(double a)`	`e` в степени `a`
`double`	`log(double a)`	натуральный логарифм `a`
…	`max(… a, … b)`	большее из двух чисел (для типов `double`, `float`, `long`, `int`)
…	`min(… a, … b)`	меньшее из двух чисел (для типов `double`, `float`, `long`, `int`)
`double`	`pow(double a, double b)`	`a` в степени `b`
`double`	`random()`	случайное число от 0.0 до 1.0
`double`	`rint(double a)`	значение `int`, ближайшее к `a`
…	`round(… a)`	значение `long` для `double` (`int` для `float`), ближайшее к `a`
`double`	`sqrt(double a)`	квадратный корень числа `a`
`double`	`toDegrees(double a)`	преобразование из радианов в градусы
`double`	`toRadians(double a)`	преобразование из градусов в радианы

Строки

String

Этот класс используется в Java для представления строк. Он обладает свойством неизменяемости. После того как создан экземпляр этого класса, его содержимое уже не может быть модифицировано.

Существует много способов создать объект `String`. Наиболее простой, если содержимое строки известно на этапе компиляции, – 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 же деле различие есть, хотя в большинстве случаев оно несущественно. 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
```

Почему результат изменился? Дело в том, что создание нового объекта – это одна из самых трудоемких процедур в Java. Поэтому компилятор стремится уменьшить их количество, если это не приводит к непредсказуемому поведению программы.

В примере объявляются две переменные, которые инициализируются одинаковым значением. Поскольку класс `String` неизменяемый, их значения всегда будут одинаковыми. Это позволяет компилятору завести скрытую вспомогательную текстовую переменную, которая будет хранить такое значение, а все остальные переменные будут ссылаться на него же, а не порождать новые объекты. В результате в первом варианте программы создается лишь один объект `String`. Для большинства операций это несущественная разница. Исключение составляют действия, которые привязаны к конкретному объекту, а не к его значению. Например, методы `wait/notify`.

Во втором варианте указано динамическое обращение к конструктору. В этом случае компилятор уже не имеет возможности заниматься оптимизацией и JVM во время исполнения программы действительно

создаст второй объект с точно таким же значением. Что мы и видим по результату выполнения примера.

В Java для строк определен оператор `+`. При использовании этого оператора производится конкатенация строк. В классе `String` также определен метод:

```
public String concat(String s);
```

Он возвращает новый объект-строку, дополненный справа строкой `s`.

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 (объект `String`, на который ссылается переменная `s`), можно изменять. В действительности это не так. В результате выполнения методов `trim` (отсечение пробелов в начале и конце строки) и `concat` создаются новые объекты-строки и ссылка `s` начинает указывать на новый объект-строку. Таким образом, меняется значение ссылки, объекты же неизменяемы.

Как уже отмечалось, строка состоит из двухбайтных Unicode-символов. Однако во многих случаях требуется работать со строкой как с набором байт (ввод/вывод, работа с базой данных и т.д.). Преобразование строки

в последовательность байтов производится следующими 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 указанной кодировке `encoding`.

Для выполнения обратной операции (преобразования байтов в строку) необходимо сконструировать новый объект-строку с помощью следующих методов:

- `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.

StringBuffer

Этот класс используется для создания и модификации строковых выражений, которые после можно превратить в `String`. Он реализован на основе массива `char[]`, что позволяет, в отличие от `String`, модифицировать его значение после создания объекта.

Рассмотрим наиболее часто используемые конструкторы класса `StringBuffer`:

- `StringBuffer()` – создает пустой `StringBuffer` ;
- `StringBuffer(String s)` – буфер заполняется указанным значением `s` ;
- `StringBuffer(int capacity)` – создает экземпляр класса `StringBuffer` с указанным размером (длина `char[]`). Задание размера не означает, что нельзя будет оперировать строками с большей длиной, чем указано в конструкторе. На самом деле этим гарантируется, что при работе со строками меньшей длины дополнительное выделение памяти не потребуется.

Разница между `String` и `StringBuffer` может быть продемонстрирована на следующем примере:

```
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 в буфер;
- `public StringBuffer insert(int offset, String str)` – вставка строки, начиная с позиции `offset` (пропустив `offset` символов).

Стоит обратить внимание, что оба метода имеют варианты, принимающие в качестве параметров различные примитивные типы Java вместо `String`. При использовании этих методов аргумент предварительно приводится к строке (с помощью `String.valueOf()`).

Еще один важный момент, связанный с этими методами, – они возвращают сам объект, у которого вызываются. Благодаря этому, возможно их использование в цепочке. Например:

```
public static void main(String[] args) {
StringBuffer sb = new StringBuffer("abc");
String str = sb.append("e").insert(4,
          "f").insert(3,"d").toString();
System.out.println(str);
}
```

В результате на экран будет выведено:

```
abcdef
```

При передаче экземпляра класса `StringBuffer` в качестве параметра метода следует помнить, что этот класс изменяемый:

```
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
```

Поскольку все объекты передаются по ссылке, в методе `doTest`, при выполнении операций с `theSB`, будет модифицирован объект, на который ссылается `sb`.

Системные классы

Следующие классы, которые будут рассмотрены, обеспечивают взаимодействие с внутренними механизмами JVM и средой исполнения приложения:

- `ClassLoader` – загрузчик классов; отвечает за загрузку описания классов в память JVM;
- `SecurityManager` – менеджер безопасности; содержит различные методы проверки допустимости запрашиваемой операции;
- `System` – содержит набор полезных статических полей и методов;
- `Runtime` – позволяет приложению взаимодействовать со средой исполнения;
- `Process` – представляет интерфейс для взаимодействия с внешней программой, запущенной при помощи `Runtime`.

ClassLoader

Это абстрактный класс, ответственный за загрузку типов. По имени класса или интерфейса он находит и загружает в память данные, которые составляют определение типа. Обычно для этого используется простое правило: название типа преобразуется в название `class` - файла, из которого и считывается вся необходимая информация.

Каждый объект `Class` содержит ссылку на объект `ClassLoader`, с помощью которого он был загружен.

Для добавления альтернативного способа загрузки классов можно реализовать свой загрузчик, унаследовав его от `ClassLoader`. Например, описание класса может загружаться через сетевое соединение. Метод `defineClass()` преобразует массив байт в экземпляр класса `Class`. С помощью метода `newInstance()` могут быть получены экземпляры такого класса. В результате загруженный класс становится полноценной частью исполняемого Java-приложения.

Для иллюстрации 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 наследник загрузчика классов должен определить и реализовать методы `findClass()` и `loadClassData()` для загрузки описания класса. Когда описание получено, массив байт передается в метод `defineClass()` для создания экземпляра `Class`. 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 воспользоваться методом `loadClass()`:

```
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 соответствующих прав для создания экземпляров этого класса (что проверяется менеджером безопасности), будет брошено исключение `IllegalAccessException`.

SecurityManager – менеджер безопасности

С помощью методов этого класса приложения перед выполнением потенциально опасных операций проверяют, является ли операция допустимой в данном контексте.

Класс `SecurityManager` содержит много методов с именами, начинающимися с приставки `check` ("проверить"). Эти методы вызываются из стандартных классов библиотек Java перед тем, как в них будут выполнены потенциально опасные операции. Типичный вызов выглядит примерно следующим образом:

```
SecurityManager security =
      System.getSecurityManager();
if(security != null){
  security.checkX(…);
}
```

где `X` – название потенциально опасной операции: `Access`, `Read`, `Write`, `Connect`, `Delete`, `Exec`, `Listen` и т.д.

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, поскольку в системе `SecurityManager` отсутствует. Предполагается, что запускаемому локально приложению можно полностью доверять. Если же приложение может быть опасно (например, его код был загружен из сети, как это происходит в случае апплетов), то менеджер безопасности выставляется и его уже нельзя убрать или заменить (попытки вызовут `SecurityException`). Он контролирует работу с локальной файловой системой, сетевыми соединениями, потоками исполнения и т.д.

System

Класс `System` содержит набор полезных статических методов и полей. Экземпляр этого класса не может быть создан или получен.

Пожалуй, наиболее широко используемой возможностью, предоставляемой `System`, является стандартный вывод, доступный через переменную `System.out`. Ее тип – `PrintStream` (потоки данных будут подробно рассматриваться в лекции 15). Стандартный вывод можно перенаправить в другой поток (файл, массив байт и т.д., главное, чтобы 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
```

Аналогично могут быть перенаправлены стандартный ввод `System.in` – вызовом `System.setIn(InputStream)` и поток вывода сообщений об ошибках `System.err` – вызовом `System.setErr(PrintStream)` (по умолчанию все потоки – `in`, `out`, `err` – работают с консолью приложения).

Следующие методы класса `System` позволяют работать с некоторыми параметрами системы:

- `public static void runFinalizersOnExit(boolean value)` – определяет, будет ли производиться вызов метода `finalize()` у всех объектов (у кого еще не вызывался), когда выполнение программы будет окончено (по умолчанию выставлено значение `false`);
- `public static native long currentTimeMillis()` – возвращает текущее время; это время представляется как количество миллисекунд, прошедших с 1 января 1970 года;
- `public static String getProperty(String key)` – возвращает значение свойства с именем `key`.

Чтобы получить все свойства, определенные в системе, можно воспользоваться следующим методом:

- `public static java.util.Properties getProperties()` – возвращает объект `java.util.Properties`, в котором содержатся значения всех

определенных системных свойств.

Метод `arrayCopy(Object source, int srcPos, Object target, int trgPos, int length)` предоставляет возможность быстрого копирования содержимого одного массива в другой. Первый параметр задает исходный массив, второй – номер позиции, начиная с которой брать элементы для копирования. Третий параметр – массив-"получатель", четвертый – номер позиции в нем, начиная с которого будут записываться скопированные элементы. Наконец, последний параметр задает количество элементов, которые надо скопировать. Оба массива должны быть созданы, иметь совместимые типы и достаточную длину, иначе будут сгенерированы соответствующие исключения.

Runtime

Во время исполнения приложению Java сопоставляется экземпляр класса `Runtime`. Этот объект позволяет взаимодействовать с окружением, в котором запущена Java-программа. Получить его можно с помощью статического метода `Runtime.getRuntime()`.

Объект этого класса:

- `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 void runFinalization()` – производит запуск выполнения методов `finalize()` у всех объектов, этого ожидающих;
- `public native long freeMemory()` – возвращает количество свободной памяти, доступной приложению JVM. 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 изменяться даже в течение одного запуска, что зависит от реализации платформы, на которой запущена Java-машина. Также не стоит закладываться на объем памяти, занимаемой одним определенным объектом, – эта величина тоже 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 также 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 `loadLibrary()`, только принимает в качестве параметра именно название файла, а не библиотеки, тем самым позволяя загрузить любой файл с `native` кодом;
- `public Process exec(String command)` – в отдельном процессе запускает команду, представленную переданной строкой. Возвращаемый объект `Process` может быть использован для взаимодействия с этим процессом.

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. Методы класса `Process`:

- `public InputStream getInputStream()` – дает возможность получать поток ввода процесса;
- `getErrorStream()`, `getOutputStream()` – методы, аналогичные `getInputStream()`, но получающие, соответственно, стандартные потоки сообщений об ошибках и вывода;
- `public void destroy()` – уничтожает процесс; все подпроцессы, запущенные из него, также 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 полагаться ни на какой из них, так как различные Java-машины могут демонстрировать различное поведение.

Потоки исполнения

Многопоточная архитектура в Java была подробно рассмотрена в лекции 12. Остановимся более подробно на методах применяемых классов.

Runnable

`Runnable` – это интерфейс, содержащий один-единственный метод без параметров: `run()`.

Thread

Объекты этого класса представляют возможность запускать и управлять потоками исполнения.

Итак, для управления потоками в классе `Thread` предусмотрены следующие методы:

- `public void start()` – производит запуск нового потока;
- `public final void join()` – если поток `A` вызывает этот метод у объекта `Thread`, представляющего поток `B` (`threadB.join()`), то выполнение потока `A` приостанавливается до тех пор, пока не закончит выполнение поток `B` ;
- `public static void yield()` – поток, из которого вызван этот метод, временно приостанавливается, чтобы дать возможность выполняться другим потокам;
- `public static void sleep(long millis)` – поток, из которого вызван этот метод, перейдет в состояние "сна" на указанное количество миллисекунд, после чего сможет продолжить выполнение. При этом нужно учесть, что через время `millis` миллисекунд этому потоку может быть выделено процессорное время, а может, ему придется и подождать немного дольше. Можно сказать, что поток продолжит выполнение не раньше, чем через время `millis` миллисекунд.

Существует еще несколько методов, которые объявлены deprecated и рекомендуется их избегать. 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`. Этот класс унаследован от `Error`. Если ошибка не будет обработана в программе и, соответственно, произойдет прекращение работы потока, сообщение о ненормальном завершении выведено не будет, так как такое завершение рассматривается как нормальное. Если же в программе эта ошибка обрабатывается (например, для проведения каких-то дополнительных действий перед закрытием потока), то очень важно позаботиться о том, чтобы эта же ошибка была брошена дальше, чтобы поток действительно закончил свое выполнение. Класс `ThreadDeath` специально унаследован от `Error`, а не от `Exception`, так как очень часто используется перехват всех исключений класса `Exception`, что не позволит корректно остановить поток.

Также `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 также 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. Также можно узнать, в каком состоянии сейчас находится поток: вызовом методов `isAlive()` – выполняется ли еще, `isInterrupted()` – прерван ли.

ThreadGroup

Для того, чтобы отдельный поток не мог 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. Группа потоков может содержать другие группы, что позволяет организовать все потоки и группы в иерархическое дерево, в котором каждый объект `ThreadGroup`, за исключением корневого, имеет родителя.

Класс `ThreadGroup` обладает методами для изменения свойств всех входящих в него потоков, таких, как приоритет, `daemon` и т.д. Метод `list()` позволяет получить список потоков.

Исключения

Подробно механизм использования исключений описан в лекции 10. 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 наследоваться от него. Класс `Throwable`, а также наиболее значимые его наследники – классы `Error`, `Exception`, `RuntimeException`, – содержатся именно в пакете `java.lang`.

Заключение

В этой лекции мы рассказали 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) – служат для представления

примитивных значений в виде объектов, так как многие классы работают именно с объектами;

- `Math` – класс, предоставляющий набор статических методов, реализующих базовые 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.

Пакет java.util

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 существенно переработаны в ходе создания версии Java2. Также рассматриваются классы для работы с датой, для генерации случайных чисел, обеспечения поддержки многих национальных языков в приложении и др.

Работа с датами и временем

Класс Date

Класс `Date` изначально предоставлял набор функций для работы с датой - для получения текущего года, месяца и т.д. Однако сейчас все перечисленные методы не рекомендованы к использованию и практически всю функциональность для этого предоставляет класс `Calendar`.

Существует несколько конструкторов класса `Date`, однако рекомендовано к использованию два:

Date() и Date(long date)

Второй конструктор принимает в качестве параметра значение типа `long`, указывающее на количество миллисекунд, прошедших с 1 января 1970 г., 00:00:00 по Гринвичу. Первый конструктор создает экземпляр, соответствующий текущему моменту. Фактически это эквивалентно второму варианту `new Date(System.currentTimeMillis())`. Можно уже после создания экземпляра класса `Date` использовать метод `setTime(long time)` для того, чтобы задать нужное время.

Для сравнения дат служат методы `after(Date date)` и `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()` возвращает строковое описание даты. Однако для более понятного и удобного преобразования даты в текст рекомендуется пользоваться классом `SimpleDateFormat`, определенным в пакете `java.text`.

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 Григорианского календаря - `GregorianCalendar`. Экземпляр этого класса получается путем вызова статического метода `getInstance()`, который возвращает экземпляр класса `GregorianCalendar`. Подклассы класса `Calendar` должны интерпретировать объект `Date` по-разному. В будущем предполагается реализовать также 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 некоторые параметры упущены, то для них будут использованы значения по умолчанию для начала отсчета, т.е.

YEAR = 1970, MONTH = JANUARY, DATE = 1 и т.д.

Для считывания и установки различных "частей" даты используются методы `get(int field)`, `set(int field, int value)`, `add(int field, int amount)`, `roll(int field, int amount)`, переменная типа `int` с именем `field` 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.

Метод set(int field,int value).

Как уже говорилось, 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 `getTimeInMillis()`. Таким образом, последовательная установка нескольких полей не вызовет ненужных 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 бы внутреннее представление даты изменялось 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:
    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()));

  }
}
```

Пример 14.1.

Результатом будет:

```
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
```

Пример 14.2.

Как мы видим, в данном примере при изменении месяца день месяца остался неизменным и было унаследовано его предыдущее значение. Но поскольку в сентябре 30 дней, дата автоматически была переведена на 1 октября, и когда было бы установлено 30 число, оно относилось бы уже к октябрю. В следующем примере считывание даты не производится, соответственно, ее вычисление не выполняется 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:

    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.getT
  }
}
```

Пример 14.3.

Результатом будет:

```
Initialy set date: 2002 August 31 23:03:51
Date with day and month changed: 2002 September 30 23:03:51
```

Пример 14.4.

Метод 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, причем, после изменения младшее по отношению к изменяемому полю принимает некорректное значение, то оно изменяется на то, которое максимально близко к "старому".

```
public class Test {

  public Test() {
  }
  public static void main(String[] args) {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mn
    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()));
  }
}
```

Пример 14.5.

Результатом будет:

```
Current date: 2002 August 31 19:30:00
Current date: 2002 August 31 19:31:15
Current date: 2002 September 30 19:31:15
```

Пример 14.6.

Метод 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:mn
    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()));
  }
}
```

Пример 14.7.

Результатом будет:

```
Current date: 2002 August 31 19:30:00
Rule 1: 2002 August 31 19:30:15
Rule 2: 2002 September 30 19:30:15
```

Пример 14.8.

Как видно из результатов работы приведенного выше кода, действие правила 1 изменилось по сравнению с методом `add`, а правило 2 действует так же.

Класс TimeZone

Класс `TimeZone` предназначен для совместного использования с классами `Calendar` и `DateFormat`. Класс абстрактный, поэтому от него порождать объекты нельзя. Вместо этого определен статический метод `getDefault()`, который возвращает экземпляр наследника `TimeZone` с настройками, взятыми из операционной системы, под управлением которой работает JVM. Для того, чтобы указать произвольные параметры, можно воспользоваться статическим методом `getTimeZone(String ID)`, в качестве параметра которому передается наименование конкретного временного пояса, для которого необходимо получить объект `TimeZone`. Набор полей, определяющих возможный набор параметров для `getTimeZone`, нигде явно не описывается. Вместо этого определен статический метод `String[] getAvailableIds()`, который возвращает возможные значения для параметра `getTimeZone`. Так можно 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.

```
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() +
    // Display 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 ) + ")");
    }
    // Display 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);
    }
  }
}
```

Пример 14.9.

Результатом будет:

```
Current TimeZone Moscow Standard TimeEurope/Moscow
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)
```

Пример 14.10.

Класс SimpleTimeZone

Класс `SimpleTimeZone`, как потомок `TimeZone`, реализует его абстрактные методы и предназначен для применения в настройках, использующих Григорианский календарь. В большинстве случаев нет необходимости создавать экземпляр данного класса с помощью конструктора. Вместо этого лучше использовать статические методы, которые возвращают тип `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).

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, чтобы установить день недели после конкретного числа, специфицируется отрицательное значение дня недели. Например, чтобы указать первый понедельник после 23 февраля, используется вот такой набор: `startDayOfWeek=-MONDAY, startMonth=FEBRUARY, startDay=23`
- для того, чтобы указать последний день недели перед каким-либо числом, указывается отрицательное значение этого числа и отрицательное значение дня недели. Например, для того, чтобы указать последнюю субботу перед 23 февраля, необходимо задать такой набор параметров: `startDayOfWeek=-SATURDAY, startMonth=FEBRUARY, startDay=-23;`
- все вышеперечисленное относится также 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;
    }
}
```

Пример 14.11.

Результатом будет:

```
java.util.SimpleTimeZone[id=Europe/Moscow,offset=10800000,dstSavings=360
startYear=0,startMode=2,startMonth=2,startDay=-1,startDayOfWeek=1,startTi
endMode=2,endMonth=9,endDay=-1,endDayOfWeek=1,endTime=10800000,
```

Пример 14.12.

Интерфейс Observer и класс Observable

Интерфейс `Observer` определяет всего один метод, `update (Observable o, Object arg)`, который вызывается, когда обозреваемый объект изменяется.

Класс `Observable` предназначен для поддержки обозреваемого объекта в парадигме MVC (model-view-controller), которая, как и другие проектные решения и шаблоны, описана в специальной литературе. Этот класс должен быть унаследован, если возникает необходимость в том, чтобы отслеживать состояние какого-либо объекта. Обозреваемый объект может иметь несколько обозревателей. Соответственно, они

должны реализовать интерфейс `Observer`.

После того, как в состоянии обозреваемого объекта что-то меняется, 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 выполняется 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");
    }
}
```

Пример 14.13.

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
```

Пример 14.14.

На практике использовать `Observer` не всегда удобно, так как в Java отсутствует множественное наследование и `Observer` необходимо наследовать в самом начале построения иерархии классов. Как вариант, можно предложить определить интерфейс, задающий функциональность, сходную с `Observer`, и реализовать его в подходящем классе.

Коллекции

Зачастую в программе работа идет не с одним объектом, а с целой группой более или менее однотипных экземпляров (например, автопарк организации). Проще всего сделать это с помощью массивов. Однако, несмотря на то, что это достаточно эффективное решение для многих случаев, оно имеет некоторые ограничения. Так, обращаться к элементу массива можно только по его номеру (индексу). Также необходимо заранее задать длину массива и больше ее не менять.

Массивы существовали в Java изначально. Кроме того, было определено два класса для организации более эффективной работы с наборами объектов: `Hashtable` и `Vector`. В JDK 1.2 набор классов, поддерживающих работу с коллекциями, был существенно расширен.

Существует несколько различных типов классов-коллекций. Все они разрабатывались, по возможности, в соответствии с единой логикой и определенными интерфейсами и там, где это возможно, работа с ними унифицирована. Однако все коллекции отличаются внутренними механизмами хранения, скоростью доступа к элементам, потребляемой памятью и другими деталями. Например, в некоторых коллекциях объекты (также называемые элементами коллекций), могут быть упорядочены, в некоторых - нет. В некоторых типах коллекций допускается дублирование ссылок на объект, в некоторых - нет. Далее мы рассмотрим каждый из классов-коллекций.

Классы, обеспечивающие манипулирование коллекциями объектов, объявлены в пакете `java.util`.

Интерфейсы

Интерфейс Collection

Данный интерфейс является корнем всей иерархии классов-коллекций. Он определяет базовую функциональность любой коллекции - набор методов, которые позволяют добавлять, удалять, выбирать элементы коллекции. Классы, которые реализуют интерфейс `Collection`, могут содержать дубликаты и пустые (`null`) значения.

`AbstractCollection`, как абстрактный класс, служит основой для

создания конкретных классов коллекций и содержит реализацию некоторых методов, определенных в интерфейсе `Collection`.

Интерфейс Set

Классы, которые реализуют этот интерфейс, не допускают наличия дубликатов. В коллекции этого типа разрешено наличие только одной ссылки типа `null`. Интерфейс `Set` расширяет интерфейс `Collection`, таким образом, любой класс, имплементирующий `Set`, реализует все методы, определенные в `Collection`. Любой объект, добавляемый в `Set`, должен реализовать метод `equals`, чтобы его можно было сравнить с другими.

`AbstractSet`, являясь абстрактным классом, представляет собой основу для реализации различных вариантов интерфейса `Set`.

Интерфейс List

Классы, реализующие этот интерфейс, содержат упорядоченную последовательность объектов (объекты хранятся в том порядке, в котором они были добавлены). В JDK 1.2 был переделан класс `Vector`, так, что он теперь реализует интерфейс `List`. Интерфейс `List` расширяет интерфейс `Collection`, и любой класс, имплементирующий `List`, реализует все методы, определенные в `Collection`, и в то же время вводятся новые методы, которые позволяют добавлять и удалять элементы из списка. `List` также обеспечивает `ListIterator`, который позволяет перемещаться как вперед, так и назад по элементам списка.

`AbstractList`, как абстрактный класс, представляет собой основу для реализации различных вариантов интерфейса `List`.

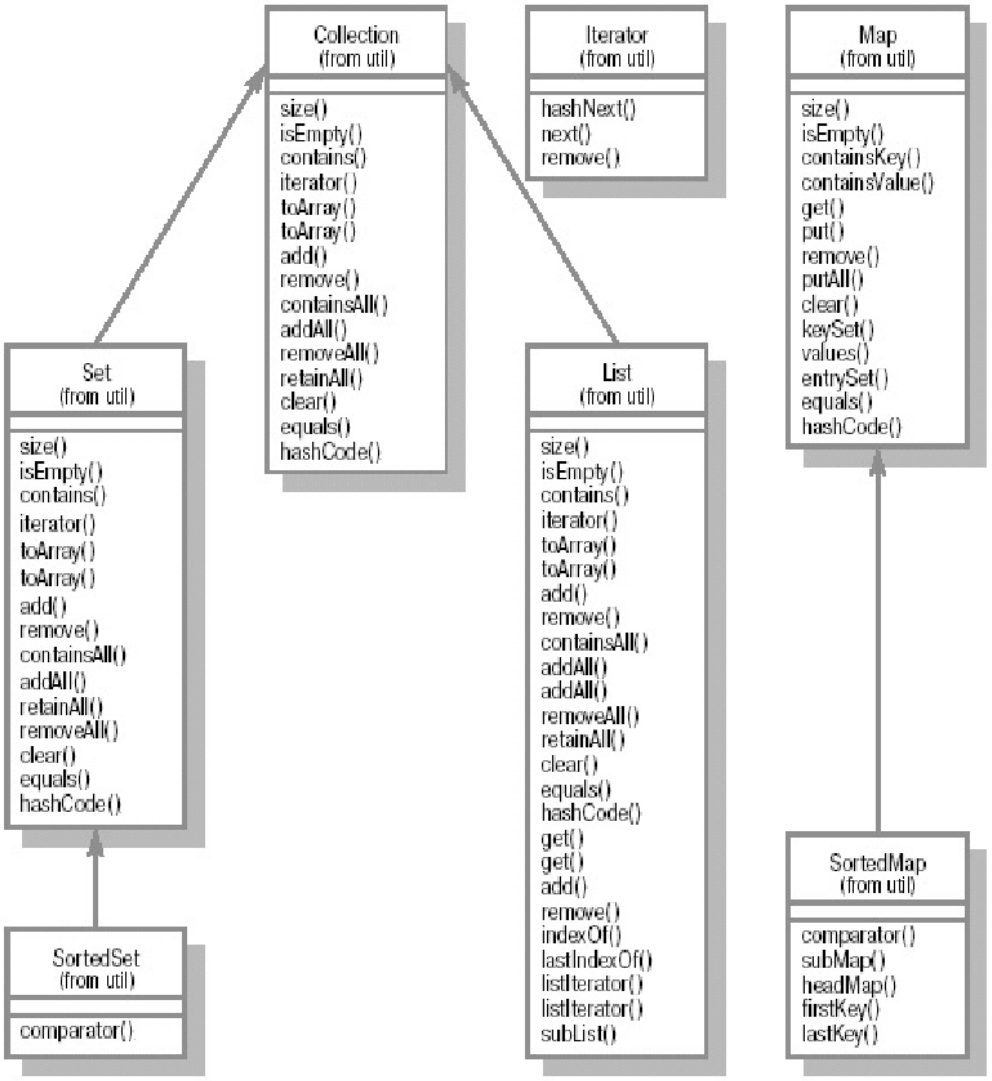


Рис. 14.1. Основные типы для работы с коллекциями.

Интерфейс Map

Классы, которые реализуют этот интерфейс, хранят 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`, будучи абстрактным классом, представляет собой основу для реализации различных вариантов интерфейса `Map`.

Интерфейс SortedSet

Этот интерфейс расширяет `Set`, требуя, чтобы содержимое набора было упорядочено. Такие коллекции могут содержать объекты, которые реализуют интерфейс `Comparable`, либо могут сравниваться с использованием внешнего `Comparator`.

Интерфейс SortedMap

Этот интерфейс расширяет `Map`, требуя, чтобы содержимое коллекции было упорядочено по значениям ключей.

Интерфейс Iterator

В Java 1 для перебора элементов коллекции использовался интерфейс `Enumeration`. В Java 2 для этих целей должны применяться 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` был создан.

Таким образом, 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

Абстрактные классы, используемые при работе с коллекциями

`java.util.AbstractCollection` - данный класс реализует все методы, определенные в интерфейсе `Collection`, за исключением `iterator` и `size`, так что для того, чтобы создать немодифицируемую 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`).

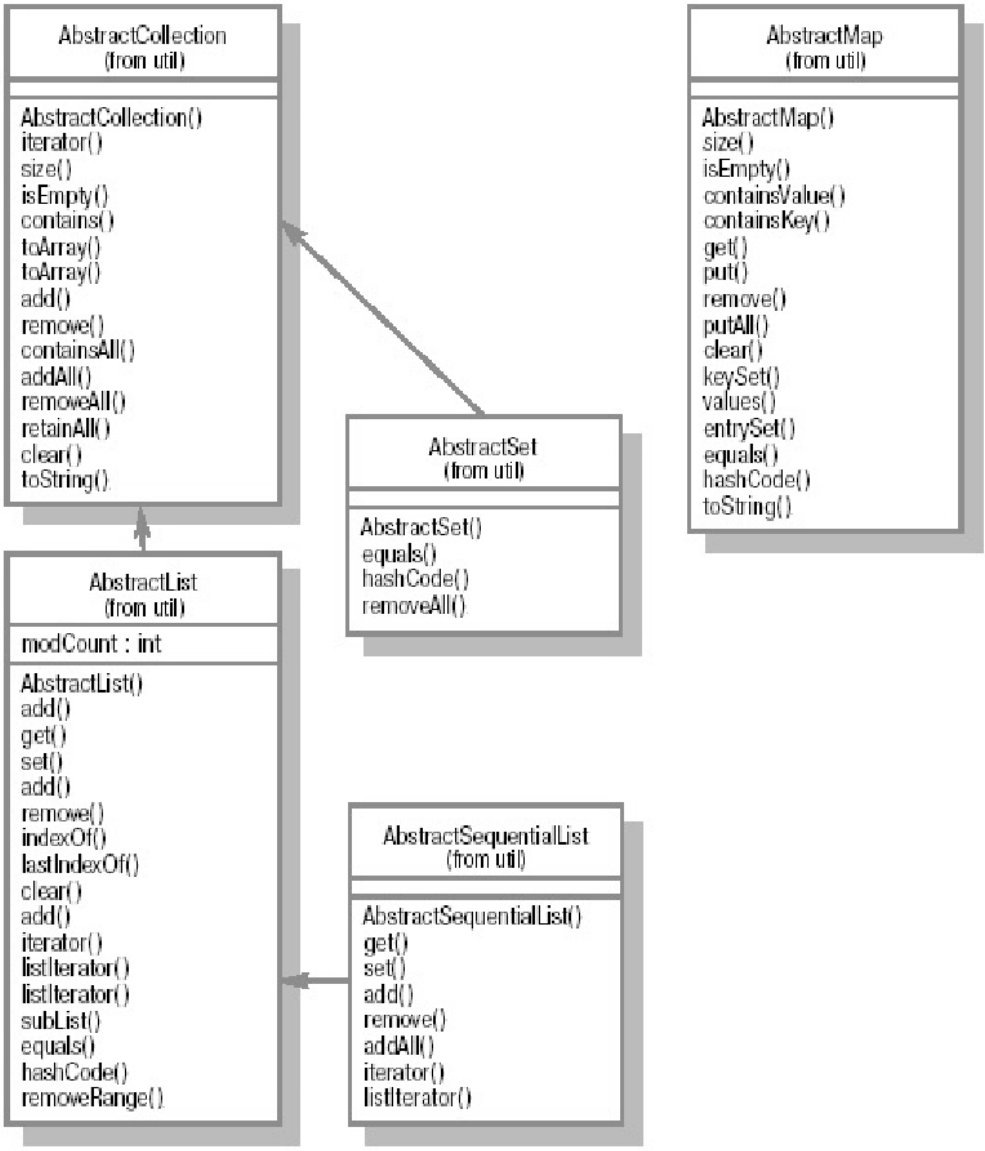


Рис. 14.2. Базовые абстрактные классы.

Необходимо также определить два конструктора: без аргументов и с аргументом `Collection`. Первый должен создавать пустую коллекцию, второй - коллекцию на основе существующей. Данный класс расширяется классами `AbstractList` и `AbstractSet`.

`java.util.AbstractList` - этот класс расширяет

`AbstractCollection` и реализует интерфейс `List`. Для создания немодифицируемого списка необходимо имплементировать методы `public Object get(int index)` и `public int size()`. Для реализации модифицируемого списка необходимо также реализовать метод `public void set(int index,Object element)` (в противном случае при его вызове будет возбуждено исключение `UnsupportedOperationException`).

В отличие от `AbstractCollection`, в этом случае нет необходимости реализовывать метод `iterator`, так как он уже реализован поверх методов доступа к элементам списка `get, set, add, remove`.

`java.util.AbstractSet` - данный 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.AbstractMap` - этот класс расширяет основную функциональность, определенную в интерфейсе `Map`. Для реализации немодифицируемого 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 и удаления элементов из набора. Для реализации модифицируемого класса `Map` необходимо также переопределить метод put и добавить в итератор, возвращаемый `entrySet().iterator()`, поддержку метода `remove`.

`java.util.AbstractSequentialList` - 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)`, `add(int index, Object element)` и `remove(int index)`. Для того, чтобы реализовать данный класс, необходимо переопределить методы `listIterator` и `size`. Причем, если реализуется немодифицируемый список, для

итератора достаточно реализовать методы `hasNext`, `next`, `hasPrevious`, `previous` и `index`. Для модифицируемого списка необходимо дополнительно реализовать метод `set`, а для списков переменной длины еще и `add`, и `remove`.

Конкретные классы коллекций

`java.util.ArrayList` - этот класс расширяет `AbstractList` и весьма похож на класс `Vector`. Он также динамически расширяется, как `Vector`, однако его методы не являются синхронизированными, вследствие чего операции с ним выполняются быстрее. Для того, чтобы 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("First 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());
    }
  }
}
```

Пример 14.15.

Результатом будет:

```
First element
Second element
Third element

Firts element
Second element
Insertion
Third element
```

Пример 14.16.

`java.util.LinkedList` - представляет собой реализацию интерфейса `List`. Он реализует все методы интерфейса `List`, помимо этого добавляются еще новые методы, 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());
        }
    }
}
```

Пример 14.17.

Результатом будет:

```
Element3
Element2
Element1
Element4

Element3
Element2
Element4

Element3
Element2
```

Пример 14.18.

Классы `LinkedList` и `ArrayList` имеют схожую функциональность. Однако с точки зрения производительности они отличаются. Так, в `ArrayList` заметно быстрей (примерно на порядок) осуществляются операции прохода по всему списку (итерации) и получения данных. `LinkedList` почти на порядок быстрее выполняет операции удаления и добавления новых элементов.

`java.util.Hashtable` - расширяет абстрактный класс `Dictionary`. В JDK 1.2 класс `Hashtable` также реализует интерфейс `Map`. `Hashtable` предназначен для хранения объектов в виде пар ключ/значение. Из самого названия следует, что `Hashtable` использует алгоритм хэширования для увеличения скорости доступа к данным. Для того, чтобы выяснить принципы работы данного алгоритма, рассмотрим несколько примеров.

Предположим, имеется массив строк, содержащий названия городов. Для того, чтобы найти элемент массива, содержащий название города, в общем случае требуется просмотреть весь массив, а если необходимо найти все элементы массива, то для поиска каждого, в среднем, потребуется просматривать половину массива. Такой подход может оказаться приемлемым только для небольших массивов.

Как уже отмечалось ранее, для того, чтобы увеличить скорость поиска, используется алгоритм хэширования. Каждый объект в Java унаследован от `Object`. Как уже отмечалось ранее, `hash` определено как целое число, которое уникально идентифицирует экземпляр класса `Object` и, соответственно, все экземпляры классов, унаследованных от `Object`. Это число возвращает метод `hashCode()`. Именно оно используется при сохранении ключа в `Hashtable` следующим образом: разделив длину массива, предназначенного для хранения ключей, на код, получаем 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, и если этот индекс совпадает с уже имеющимся, то создается список ключей, на который указывает элемент массива ключей. Таким образом, при обратном 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. В частности, размер массива. Если массив окажется слишком мал, то связанные списки будут слишком длинными и скорость поиска станет существенно снижаться, так как просмотр элементов списка будет такой же, как в обычном массиве. Чтобы этого избежать, 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 окажется слишком мал, то он будет быстро заполняться и будет производиться 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, придется обрабатывать 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 также конструктор без параметров, который использует значения по умолчанию 101 для размера массива (в последней версии значение уменьшено до 11) 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 смысл задействовать коллекции, применяющие алгоритм хэширования.

`java.util.HashMap` - этот класс расширяет `AbstractMap` и весьма похож на класс `Hashtable`. `HashMap` предназначен для хранения пар объектов ключ/значение. Как для ключей, так и для элементов допускаются значения типа `null`. Порядок хранения элементов в этой коллекции не совпадает с порядком их добавления. Порядок элементов в коллекции также 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 делать размер коллекций чрезмерно большим, если достаточно часто придется осуществлять итерацию по элементам.

Методы `HashMap` не являются синхронизированными. Для того, чтобы обеспечить нормальную работу в многопоточном варианте, следует использовать либо внешнюю синхронизацию потоков, либо синхронизированный вариант коллекции.

```
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 = "";
    // Итератор для перебора всех ключей в Map
    it = hm.keySet().iterator();
    while(it.hasNext()){
      key = (String)it.next();
      System.out.println( "For key = " + key + " value = " +
```

```
            ((TestObject)hm.get(key)).getText());
        }
    }
}
```

Пример 14.19.

Результатом будет:

```
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
```

Пример 14.20.

`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 применение `null` значений для ключей. При использовании дубликатов ключей ссылка на объект, сохраненный с таким же ключом, будет утеряна. Например:

```
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"));
    }
}
```

Результатом будет:

```
String1
String2
```

Класс Collections

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 синхронизированных 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`.

```
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());
      }
    }
  }
```

Пример 14.21.

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 лекции 15). Как правило, это текстовый файл, в котором хранятся параметры. Параметры - это строки, которые представляют собой 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 до тех пор, 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, отличный от пробела, считается началом ключа. Первый встретившийся пробел, двоеточие или знак равенства 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, также 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, чтобы 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` определен еще метод `list(PrintWriter out)`, который практически идентичен `save`. Отличается лишь заголовок, который изменить нельзя. Кроме того, строки усекаются по ширине. Поэтому данный 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 = Value1 \n");
    sw.write(" Key2 : Value2 \r\n");
    sw.write(" Key3 Value3 \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);
  }
}
```

Пример 14.22.

Результатом будет:

```
-- listing properties --
Key3=Value3
Key2=Value2
Key1=Value1

-- listing properties --
Key4=Added Value4
Key3=Value3
Key2=Value2
Key1=Modified Value1
```

Пример 14.23.

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 определенного пользователем алгоритма сравнивает объекты, переданные в качестве параметров. Метод compare должен вернуть:

-1 если obj1 < obj2
0 если obj1 = obj2
1 если obj1 > obj2

Класс Arrays

Класс `Arrays` обеспечивает набор методов для выполнения операций над массивами, таких, как поиск, сортировка, сравнение. В `Arrays` также 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.

```
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]);
    }
  }
}
```

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 returnDelims)`.

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());
      }
    }
  }
```

Результатом будет:

```
word1
word2
word3
word4
```

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 изменяться. 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);
  }
}
```

Результатом будет:

```
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`.

Класс Random

Класс `Random` используется для получения последовательности псевдослучайных чисел. В качестве "зерна" применяется 48-битовое число. Если для инициализации `Random` задействовать одно и то же число, будет получена та же самая последовательность псевдослучайных чисел.

В классе `Random` определено также 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, и `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();
  }
}
```

Пример 14.24.

Результатом будет:

```
-1193959466 -1139614796 837415749 -1220615319 -1429538713 118249
-1139614796 837415749 -1220615319 -1429538713 118249332 -9515892
-98;
```

Пример 14.25.

Локализация

Класс Locale

Класс `Locale` предназначен для отображения определенного региона. Под регионом принято понимать не только географическое положение, но также языковую и культурную среду. Например, помимо того, что указывается страна Швейцария, можно указать также 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. Список поддерживаемых стран и языков можно получить и с помощью вызова статических методов `Locale.getISOLanguages()` `Locale.getISOCountries()`, соответственно. Во втором варианте конструктора указан также строковый параметр `variant`, в котором кодируется информация о платформе. Если здесь необходимо указать дополнительные параметры, то их требуется 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 `Locale` 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());
  }
}
```

Пример 14.26.

Результатом будет:

```
US United States USA
en English eng
```

```
RU Russia RUS
ru Russian rus
WINDOWS WINDOWS
```

Пример 14.27.

Класс ResourceBundle

Абстрактный класс `ResourceBundle` предназначен для хранения объектов, специфичных для локали. Например, когда необходимо получить набор строк, зависящих от локали, используют `ResourceBundle`.

Применение `ResourceBundle` настоятельно рекомендуется, если предполагается использовать программу в многоязыковой среде. С помощью этого класса легко манипулировать наборами ресурсов, зависящих от локалей, их можно 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 дополнительный 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);
```

На основе указанного базового имени (первый параметр), указанной

локали (второй параметр) и локали по умолчанию (задается настройками ОС или JVM) генерируется список возможных имен ресурса. Причем, указанная локаль имеет более высокий приоритет, чем локаль по умолчанию. Если обозначить составляющие указанной локали (язык, страна, вариант) как 1, а локали по умолчанию - 2, то список примет следующий вид:

```
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"
```

Пример 14.28.

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. Таким образом, если нужный класс не будет найден, то к требуемому имени класса будет добавлено расширение ".properties" и будет предпринята 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 (там, где вызывается getResourceBundle), то есть не быть private и т.д.

Всегда должен создаваться базовый класс без суффиксов, т.е. если вы создаете 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()
protected Object handleGetObject(String key)
```

Первый метод должен возвращать список всех ключей, которые определены в ResourceBundle, 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 вариант");
    }
    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",L
    System.out.println(rb.getString("TestKey"));
    rb = ResourceBundle.getBundle("experiment.MyResource", new Locale("ru","
      System.out.println(rb.getString("TestKey"));
    }
  }
```

Пример 14.29.

Результатом будет:

```
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 extends ResourceBundle {
    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"));
        Behavior be = (Behavior)rb.getObject("Behavior");
        System.out.println(be.getBehavior());
        System.out.println(be.getCapital());
        rb = ResourceBundle.getBundle("lecture.MyResourceBundle", new Locale("
        System.out.println((String)rb.getObject("Bundle description"));
        Behavior be = (Behavior)rb.getObject("Behavior");
        System.out.println(be.getBehavior());
        System.out.println(be.getCapital());
    }
```

Пример 14.30.

Результатом будет:

```
Русский набор ресурсов
Русский вариант поведения
Москва
English resource bundle
English behavior
London
```

Пример 14.31.

Классы ListResourceBundle и PropertiesResourceBundle

У класса `ResourceBundle` определено два прямых потомка `ListResourceBundle` и `PropertiesResourceBundle`. `PropertiesResourceBundle` хранит набор ресурсов в файле, который представляет собой набор строк.

Алгоритм конструирования объекта, содержащего набор ресурсов, был описан в предыдущем параграфе. Во всех случаях, когда в качестве последнего элемента используется `.properties`, например, `baseclass + "_" + language1 + "_" + country1 + ".properties"`, речь идет о создании `ResourceBundle` из файла с наименованием `baseclass + "_" + language1 + "_" +`

country1 и расширением properties. Обычно класс ResourceBundle помещают в пакет resources, а файл свойств - в каталог resources. Тогда для того, чтобы инстанциировать нужный класс, необходимо указать полный путь к этому классу (файлу):

```
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;
- 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",
        Vector v = (Vector)rb.getObject("VectorKey");
        Iterator it = v.iterator();
        while(it.hasNext()) {
          System.out.println(it.next());
        }
      }
    }
```

Пример 14.32.

Результатом будет:

```
Element 1
Element 2
Element 3
```

Создание ресурсов для локалей, отличных от локали по умолчанию, осуществляется так же, как было показано для `ResourceBundle`.

Заключение

В этой лекции были рассмотрены вспомогательные классы пакета `java.util`. Как можно было заметить, они относятся к самым разным задачам, а потому редкая программа обходится без использования хотя бы одного класса этого пакета.

Напомним кратко все основные классы и их особенности:

- Для работы с датой и временем должны использоваться классы `Date`, `Calendar`. Класс `Calendar` абстрактный, существует конкретная реализация этого класса `GregorianCalendar`.
- Интерфейс `Observer` и класс `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 (управляются

либо самими данными, либо внешними алгоритмами).

- Наборы (`Set`) не допускают дублированных элементов.
- Карты (`Maps`) используют уникальные ключи для поиска содержимого.
- Применение массивов делает добавление, удаление и увеличение количества элементов затруднительным.
- Использование связанных списков (`LinkedList`) обеспечивает хорошую производительность при вставке, удалении элементов, но снижает скорость индексированного доступа к ним.
- Использование деревьев (`Tree`) облегчает вставку, удаление и увеличение размера хранилища, снижает скорость индексированного доступа, но увеличивает скорость поиска.
- Применение хэширования облегчает вставку, удаление и увеличение размера хранилища, снижает скорость индексированного доступа, но увеличивает скорость поиска. Однако хэширование требует наличия уникальных ключей для запоминания 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`.
- Манипулировать ресурсами, которые различаются в зависимости от локализации, удобно с помощью классов `ResourceBundle`, `ListResourceBundle`, `PropertiesResourceBundle`. Собственно локаль задается с помощью класса `Locale`.

Пакет java.io

Эта лекция описывает реализованные в Java возможности передачи информации, что является важной функцией для большинства программных систем. Сюда входит работа с файлами, 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 потоках символов (Reader и Writer). В библиотеке java.io содержатся все эти классы и их многочисленные наследники, предоставляющие полезные возможности. Отдельно рассматривается механизм сериализации объектов и работа с файлами.

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, можно 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 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 данными, 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 более 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, способным получать или предоставлять информацию. Соответственно, потоки делятся на входящие – читающие данные и выходящие – передающие (записывающие) данные. Введение концепции stream позволяет отделить основную логику программы, обменивающейся информацией с любыми устройствами одинаковым образом, 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 одним байтом, находится в диапазоне от 0 до 2^8-1=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 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 преобразуется в нужный формат.

На рисунке 15.1 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`.

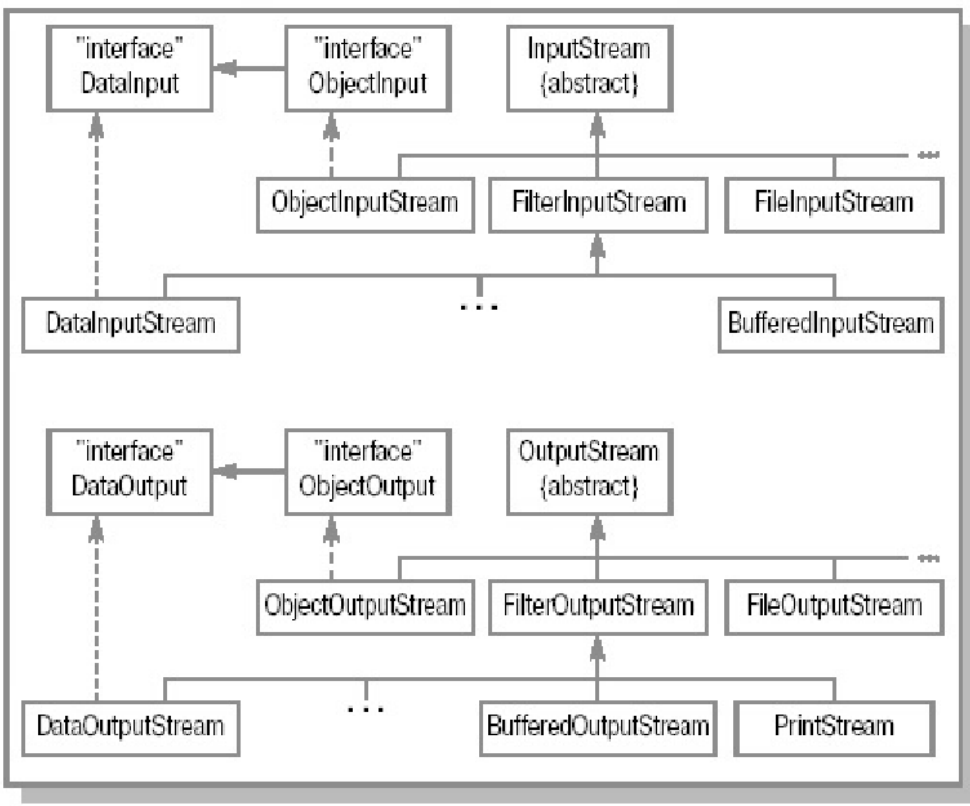


Рис. 15.1. Иерархия классов ввода/вывода.

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, которые наследуются 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 том случае, если считывание произошло успешно, 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.

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, будет брошено исключение `java.io.IOException`. Этот класс наследуется от `Exception`, т.е. его всегда необходимо обрабатывать явно. Дело в том, что каналы передачи информации, будь то `Internet` 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 указанных 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 данные в потоке 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".

Чтобы узнать, сколько байт в потоке готово к считыванию, применяется метод `available()`. Этот метод возвращает значение типа `int`, которое показывает, сколько байт в потоке готово к считыванию. При этом не стоит путать 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 ничего не возвращают (`void`).

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 бросает `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 плюс длина будет больше длины массива, – во всех этих случаях кидается исключение `IndexOutOfBoundsException`.

Реализация потока может быть такой, что данные записываются не сразу, а хранятся некоторое время в памяти. Например, мы хотим записать в файл какие-то данные, которые 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 внутренний механизм для буферизации (временного хранения перед отправкой) данных. Чтобы 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 (вызовом метода `flush`), после чего поток закрывается и освобождаются все связанные с ним системные ресурсы. 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 распространенные, будут рассмотрены далее.

Классы-реализации потоков данных

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`. Обратите внимание, в данном примере второе считанное значение равно 255, а не -1, как можно было бы ожидать. Чтобы 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 `int`, получаемое приставкой 24-х нулей, равно `255` (в десятичной системе). Однако если явно привести его к `byte`, получим исходное значение.

Аналогично, для записи байт в массив применяется класс `ByteArrayOutputStream`. Этот класс использует внутри себя объект `byte[]`, куда записывает данные, передаваемые при вызове методов `write()`. Чтобы 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`.

Использовать классы `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. Эти классы хороши тем, что позволяют сразу просмотреть результат и не нужно создавать ни файл, ни сетевое соединение, ни что-либо еще.

Классы FileInputStream и FileOutputStream

Класс `FileInputStream` используется для чтения данных из файла. Конструктор такого класса в качестве параметра принимает название файла, из которого будет производиться считывание. При указании строки имени файла нужно учитывать, что она будет напрямую передана операционной системе, поэтому формат имени файла и пути к нему может различаться на разных платформах. Если при вызове этого конструктора передать строку, указывающую на несуществующий файл или каталог, то будет брошено `java.io.FileNotFoundException`. Если же 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 имени файла, также можно указать, будут ли данные дописываться в конец файла, либо файл будет перезаписан. Если указанный файл не существует, то сразу после создания `FileOutputStream` он будет создан. При вызовах методов `write()` передаваемые значения будут записываться в этот файл. По окончании работы необходимо вызвать метод `close()`, чтобы 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");
    for (int i=0;i<count;i++)
      System.out.print(bytesReaded[i]+",");
    System.out.println();
    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());
  }
```

Пример 15.1.

Результатом работы программы будет:

```
Файл открыт для записи
Записано: 3 байт
Выходной поток закрыт
Файл открыт для чтения
```

```
Готово к считыванию: 3 байт
Считано: 3 байт
1,2,3,
Входной поток закрыт
```

Пример 15.2.

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 байт, которое может быть на данный момент считано без блокирования. Тот факт, что, скорее всего, это число и будет длиной файла, является всего лишь частным случаем работы на некоторых платформах.

В приведенном примере для наглядности закрытие потоков производилось сразу же после окончания их использования в основном блоке. Однако лучше закрывать потоки в `finally` блоке.

```
...
} finally {
try{inFile.close();}catch(IOException e){};
}
```

Такой подход гарантирует, что поток будет закрыт и будут освобождены все связанные с ним системные ресурсы.

PipedInputStream и PipedOutputStream

Классы `PipedInputStream` и `PipedOutputStream` характеризуются тем, что их объекты всегда используются в паре – к одному объекту `PipedInputStream` привязывается (подключается) один объект `PipedOutputStream`. Они могут быть полезны, если в программе необходимо организовать обмен данными между модулями (например, между потоками выполнения).

Эти классы применяются следующим образом: создается по объекту `PipedInputStream` и `PipedOutputStream`, после чего они

могут быть соединены между собой. Один объект `PipedOutputStream` может быть соединен с ровно одним объектом `PipedInputStream`, и наоборот. Затем в объект `PipedOutputStream` записываются данные, после чего они могут быть считаны именно в подключенном объекте `PipedInputStream`. Такое соединение можно обеспечить либо вызовом метода `connect()` с передачей соответствующего объекта `PipedI/OStream` (будем так кратко обозначать пару классов, в данном случае `PipedInputStream` и `PipedOutputStream`), либо передать этот объект еще при вызове конструктора.

Использование связки `PipedInputStream` и `PipedOutputStream` показано в следующем примере:

```
try {
  int countRead = 0;
  byte[] toRead = new byte[100];
  PipedInputStream pipeIn = new PipedInputStream();
  PipedOutputStream pipeOut = new PipedOutputStream(pipeIn);

  // Считывать 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) {
  System.out.println ("Impossible IOException occur: ");
```

```
    e.printStackTrace();
  }
```

Пример 15.3.

Данный пример носит чисто демонстративный характер (в результате его работы массив `toRead` будет заполнен случайными числами). Более явно выгода от использования `PipedI/OStream` в основном проявляется при разработке многопоточного приложения. Если в программе запускается несколько потоков исполнения, организовать передачу данных между ними удобно с помощью этих классов. Для этого нужно создать связанные объекты `PipedI/OStream`, после чего передать ссылки на них в соответствующие потоки. Поток выполнения, в котором производится чтение данных, 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);
    }
    // обработка полученных данных из readedBytes
    // …
  } catch(IOException e) {

  /* IOException будет брошено, когда поток inStream, либо
  связанный с ним PipedOutputStream, уже закрыт, и при этом
  производится попытка считывания из inStream */

  System.out.println("работа с потоком inStream завершена");
  }
```

Пример 15.4.

Если с объектом `inStream` одновременно могут работать несколько потоков выполнения, то необходимо использовать блок

`synchronized` (как и сделано в примере), который гарантирует, что в период между вызовами `inStream.available()` и `inStream.read(…)` ни в каком другом потоке выполнения не будет производиться считывание из `inStream`. Поэтому вызов `inStream.read(readedBytes)` не приведет к блокировке и все данные, готовые к считыванию, будут считаны.

StringBufferInputStream

Иногда бывает удобно работать с текстовой строкой `String` как с потоком байт. Для этого можно воспользоваться классом `StringBufferInputStream`. При создании объекта этого класса необходимо передать конструктору объект `String`. Данные, возвращаемые методом `read()`, будут считываться именно из этой строки. При этом символы будут преобразовываться в байты с потерей точности – старший байт отбрасывается (напомним, что символ char состоит из двух байт).

SequenceInputStream

Класс `SequenceInputStream` объединяет поток данных из других двух и более входных потоков. Данные будут вычитываться последовательно – сначала все данные из первого потока в списке, затем из второго, и так далее. Конец потока `SequenceInputStream` будет достигнут только тогда, когда будет достигнут конец потока, последнего в списке.

В этом классе имеется два конструктора – принимающий два потока и принимающий `Enumeration` (в котором, конечно, должны быть только экземпляры `InputStream` и его наследников). Когда вызывается метод `read()`, `SequenceInputStream` пытается считать байт из текущего входного потока. Если в нем больше данных нет (считанное из него значение равно `-1`), у него вызывается метод `close()` и следующий входной поток становится текущим. Так продолжается до тех пор, пока не будут получены все данные из последнего потока. Если при считывании обнаруживается, что больше

входных потоков нет, `SequenceInputStream` возвращает `-1`. Вызов метода `close()` у `SequenceInputStream` закрывает 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){};
}
```

Пример 15.5.

В результате выполнения этого примера в файл `file3.txt` будет записано содержимое файлов `file1.txt` и `file2.txt` – сначала полностью `file1.txt`, потом `file2.txt`. Закрытие потоков производится в блоке `finally`. Поскольку при вызове метода `close()` может возникнуть `IOException`, необходим `try-catch` блок. Причем, каждый вызов метода `close()` взят в отдельный `try-catch` блок - для того, чтобы возникшее исключение при закрытии одного потока не помешало закрытию другого. При этом нет необходимости закрывать потоки `inFile1` и `inFile2` – они будут автоматически закрыты при использовании в sequenceStream - либо

когда в них закончатся данные, либо при вызове у `sequenceStream` метода `close()`.

Объект `SequenceInputStream` можно было создать и другим способом: сначала получить объект `Enumeration`, содержащий все потоки, и передать его 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 en = vector.elements();
sequenceStream = new SequenceInputStream(en);
```

Пример 15.6.

Если заменить в предыдущем примере инициализацию `sequenceStream` на приведенную здесь, то в файл `file3.txt`, кроме содержимого файлов `file1.txt` и `file2.txt`, будут записаны еще три строки – одна в начале файла, одна между содержимым файлов `file1.txt` и `file2.txt` и еще одна в конце `file3.txt`.

Классы FilterInputStream и FilterOutputStream и их наследники

Задачи, возникающие при вводе/выводе весьма разнообразны - это может быть считывание байтов из файлов, объектов из файлов, объектов из массивов, буферизованное считывание строк из массивов и т.д. В такой ситуации решение с использованием простого наследования приводит к возникновению слишком большого числа подклассов. Более эффективно применение надстроек (в ООП этот шаблон 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, поэтому может быть, в свою очередь, дополнительно надстроена.

В `java.io` интерфейс для таких надстроек ввода/вывода предоставляют классы `FilterInputStream` (для входных потоков) и `FilterOutputStream` (для выходных потоков). Эти классы унаследованы от основных базовых классов ввода/вывода – `InputStream` и `OutputStream`, соответственно. Конструктор `FilterInputStream` принимает в качестве параметра объект `InputStream` и имеет модификатор доступа `protected`.

Классы `FilterI/OStream` являются базовыми для надстроек и определяют общий интерфейс для надстраиваемых объектов. Потоки-надстройки не являются источниками данных. Они лишь модифицируют (расширяют) работу надстраиваемого потока.

BufferedInputStream и BufferedOutputStream

На практике при считывании с внешних устройств ввод данных почти всегда необходимо буферизировать. Для буферизации данных служат классы `BufferedInputStream` и `BufferedOutputStream`.

`BufferedInputStream` содержит массив байт, который служит буфером для считываемых данных. То есть когда байты из потока считываются либо пропускаются (метод `skip()`), сначала заполняется буферный массив, причем, из надстраиваемого потока загружается сразу много байт, чтобы не требовалось обращаться к нему при каждой операции `read` или `skip`. Также класс `BufferedInputStream` добавляет поддержку методов `mark()` и `reset()`. Эти методы определены еще в классе `InputStream`, но там их реализация по умолчанию бросает исключение `IOException`. Метод `mark()` запоминает точку во входном потоке, а вызов метода `reset()` приводит к тому, что все байты, полученные после последнего вызова `mark()`, будут считываться повторно, прежде, чем новые байты начнут поступать из надстроенного входного потока.

`BufferedOutputStream` предоставляет возможность производить

многократную запись небольших блоков данных без обращения к устройству вывода при записи каждого из них. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и, соответственно, запись в него, произойдет, когда буфер заполнится. Инициировать передачу содержимого буфера на устройство вывода можно и явным образом, вызвав метод `flush()`. Так же буфер освобождается перед закрытием потока. При этом будет закрыт и надстраиваемый поток (так же поступает `BufferedInputStream`).

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 time: " + 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 time: " + (time) + " millisec");

   // Теперь применим буферизацию
```

```
    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 time: " + (time) + " millisec");
  } catch (IOException e) {
    System.out.println("IOException: " + e.toString());
    e.printStackTrace();
  }
```

Пример 15.7.

Результатом могут быть, например, такие значения:

```
Writing time: 359 millisec
Direct read time: 6546 millisec
Buffered read time: 250 millisec
```

Пример 15.8.

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 в файл производилась с буферизацией, однако ее влияние на скорость записи нетрудно проверить, убрав из программы строку, создающую `BufferedOutputStream`.

Классы `BufferedI/OStream` добавляют только внутреннюю логику обработки запросов, но не добавляют никаких новых методов. Следующие два фильтра предоставляют некоторые дополнительные возможности для работы с потоками.

LineNumberInputStream

Класс `LineNumberInputStream` во время чтения данных производит подсчет, сколько строк было считано из потока. Номер строки, на которой в данный момент происходит чтение, можно узнать путем вызова метода `getLineNumber()`. Также 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 исходящего потока отсутствует. `LineNumberInputStream`, начиная с версии 1.1, объявлен *`deprecated`*, то есть использовать его не рекомендуется. Его заменил класс `LineNumberReader` (рассматривается ниже), принцип работы которого точно такой же.

PushbackInputStream

Этот фильтр позволяет вернуть во входной поток считанные из него данные. Такое действие производится вызовом метода `unread()`. Понятно, что обеспечивается подобная функциональность за счет наличия в классе специального буфера – массива байт, который хранит считанную информацию. Если будет произведен откат (вызван метод `unread`), то во время следующего считывания эти данные будут выдаваться еще раз как только полученные. При создании объекта можно указать размер буфера.

PrintStream

Этот класс используется для конвертации и записи строк в байтовый поток. В нем определен метод `print(…)`, принимающий в качестве аргумента различные примитивные типы Java, а также тип `Object`. При вызове передаваемые данные будут сначала преобразованы в строку вызовом метода `String.valueOf()`, после чего записаны в поток. Если возникает исключение, оно обрабатывается внутри метода print и дальше не бросается (узнать, произошла ли ошибка, можно с

помощью метода `checkError()`). При записи символов в виде байт используется кодировка, принятая по умолчанию в операционной системе (есть возможность задать ее явно при запуске JVM).

Этот класс также является *deprecated*, поскольку работа с кодировками требует особого подхода (зачастую у двухбайтовых символов Java старший байт просто отбрасывается). Поэтому в версии Java 1.1 появился дополнительный набор классов, основывающийся на типах `Reader` и `Writer`. Они будут рассмотрены позже. В частности, вместо `PrintStream` теперь рекомендуется применять `PrintWriter`. Однако старый класс продолжает активно использоваться, поскольку статические поля `out` и `err` класса `System` имеют именно это тип.

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 их реализации – классы-фильтры `DataInputStream` и `DataOutputStream`. Их место в иерархии классов ввода/вывода можно увидеть на рис.15.1.

Интерфейсы `DataInput` и `DataOutput` определяют, а классы `DataInputStream` и `DataOutputStream`, соответственно, реализуют методы считывания и записи значений всех примитивных типов. При этом происходит конвертация этих данных в набор `byte` и обратно. Чтение необходимо организовать так, чтобы данные запрашивались в виде тех же типов, в той же последовательности, как и производилась запись. Если записать, например, `int` и `long`, а потом считывать их как `short`, чтение будет выполнено корректно, без исключительных ситуаций, но числа будут получены совсем другие.

Это наглядно показано в следующем примере:

```
try {
   ByteArrayOutputStream out = new ByteArrayOutputStream();
   DataOutputStream outData = new DataOutputStream(out);
```

```
    outData.writeByte(128);
    // этот метод принимает аргумент int, но записывает
    // лишь младший байт
    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("Чтение в правильной последовательности: ");
    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("Чтение в измененной последовательности:");
    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();
  }
```

Пример 15.9.

Результат выполнения программы:

```
Чтение в правильной последовательности:

readByte: -128
readInt: 128
readLong: 128
readDouble: 128.0
```

Чтение в измененной последовательности:

```
readInt: -2147483648
readDouble: -0.0
readLong: -9205252085229027328
```

Итак, значение любого примитивного типа может быть передано и считано из потока данных.

Сериализация объектов (serialization)

Для объектов процесс преобразования в последовательность байт и обратно организован несколько сложнее – объекты имеют различную структуру, хранят ссылки на другие объекты и т.д. Поэтому такая процедура получила специальное название - сериализация (serialization), обратное действие, – то есть воссоздание объекта из последовательности байт – десериализация.

Поскольку сериализованный объект – это последовательность байт, которую можно легко сохранить в файл, передать по сети и т.д., то и объект затем можно восстановить на любой машине, вне зависимости от того, где проводилась сериализация. Разумеется, Java позволяет не задумываться при этом о таких факторах, как, например, используемая операционная система на машине-отправителе и получателе. Такая гибкость обусловила широкое применение сериализации при создании распределенных приложений, в том числе и корпоративных (enterprise) систем.

Стандартная сериализация

Для представления объектов в виде последовательности байт определены унаследованные от `DataInput` и `DataOutput` интерфейсы `ObjectInput` и `ObjectOutput`, соответственно. В java.io имеются реализации этих интерфейсов – классы `ObjectInputStream` и `ObjectOutputStream`.

Эти классы используют стандартный механизм сериализации, который предлагает JVM. Для того, чтобы объект мог быть сериализован, класс,

от которого он порожден, должен реализовывать интерфейс `java.io.Serializable`. В этом интерфейсе не определен ни один метод. Он нужен лишь для указания, что объекты класса могут участвовать в сериализации. При попытке сериализовать объект, не имеющий такого интерфейса, будет брошен `java.io.NotSerializableException`.

Чтобы начать сериализацию объекта, нужен выходной поток `OutputStream`, в который и будет записываться сгенерированная последовательность байт. Этот поток передается в конструктор `ObjectOutputStream`. Затем вызовом метода `writeObject()` объект сериализуется и записывается в выходной поток. Например:

```
ByteArrayOutputStream os =
  new ByteArrayOutputStream();
Object objSave = new Integer(1);
ObjectOutputStream oos =
  new ObjectOutputStream(os);
oos.writeObject(objSave);
```

Чтобы увидеть, во что превратился объект `objSave`, можно просмотреть содержимое массива:

```
byte[] bArray = os.toByteArray();
```

А чтобы восстановить объект, его нужно десериализовать из этого массива:

```
ByteArrayInputStream is =
  new ByteArrayInputStream(bArray);
ObjectInputStream ois =
  new ObjectInputStream(is);
Object objRead = ois.readObject();
```

Теперь можно убедиться, что восстановленный объект идентичен исходному:

```
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 мы видим, восстановленный объект не совпадает с исходным (что очевидно – ведь восстановление могло происходить и на другой машине), но равен сериализованному по значению.

Как обычно, для упрощения в примере была опущена обработка ошибок. Однако, сериализация (десериализация) объектов довольно сложная процедура, поэтому возникающие сложности не всегда очевидны. Рассмотрим основные исключения, которые может генерировать метод `readObject()` класса `ObjectInputStream`.

Предположим, объект некоторого класса `TestClass` был сериализован и передан по сети на другую машину для восстановления. Может случиться так, что у считывающей JVM на локальном диске не окажется описания этого класса (файл `TestClass.class`). Поскольку стандартный механизм сериализации записывает в поток байт лишь состояние объекта, для успешной десериализации необходимо наличие описание класса. В результате будет брошено исключение `ClassNotFoundException`.

Причина появления `java.io.StreamCorruptedException` вполне очевидна из названия – неправильный формат входного потока. Предположим, происходит попытка считать сериализованный объект из файла. Если этот файл испорчен (для эксперимента можно открыть его в текстовом редакторе и исправить несколько символов), то стандартная процедура десериализации даст сбой. Эта же ошибка возникнет, если считать некоторое количество байт (с помощью метода `read`) непосредственно из надстраиваемого потока `InputStream`. В таком случае `ObjectInputStream` снова обнаружит сбой в формате данных и будет брошено исключение

`java.io.StreamCorruptedException`.

Поскольку `ObjectOutput` наследуется от `DataOutput`, `ObjectOutputStream` может быть использован для последовательной записи нескольких значений как объектных, так и примитивных типов в произвольной последовательности. Если при считывании будет вызван метод `readObject`, а в исходном потоке следующим на очереди записано значение примитивного типа, будет брошено исключение `java.io.OptionalDataException`. Очевидно, что для корректного восстановления данных из потока их нужно считывать именно в том порядке, в каком были записаны.

Восстановление состояния

Итак, сериализация объекта заключается в сохранении и восстановлении состояния объекта. В Java в большинстве случаев состояние описывается значениями полей объекта. Причем, что важно, не только тех полей, которые были явно объявлены в классе, от которого порожден объект, но и унаследованных полей.

Предположим, мы бы попытались своими силами реализовать стандартный механизм сериализации. Нам передается выходной поток, в который нужно записать состояние нашего объекта. С помощью `DataOutput` интерфейса можно легко сохранить значения всех доступных полей (будем для простоты считать, что они все примитивного типа). Однако в большинстве случаев в родительских классах могут быть объявлены недоступные нам поля (например, `private`). Тем не менее, такие поля, как правило, играют важную роль в определении состояния объекта, так как они могут влиять на результат работы унаследованных методов. Как же сохранить их значения?

С другой стороны, не меньшей проблемой является восстановление объекта. Как говорилось раньше, объект может быть создан только вызовом его конструктора. У класса, от которого порожден десериализуемый объект, может быть несколько конструкторов, причем, некоторые из них, или все, могут иметь аргументы. Какой из них вызвать? Какие значения передать в качестве аргументов?

После создания объекта необходимо установить считанные значения его полей. Однако многие классы имеют специальные `set` -методы для этой цели. В таких методах могут происходить проверки, могут меняться значения вспомогательных полей. Пользоваться ли этими методами? Если их несколько, то как выбрать правильный и какие параметры ему передать? Снова возникает проблема работы с недоступными полями, полученными по наследству. Как же в стандартном механизме сериализации решены все эти вопросы?

Во-первых, рассмотрим подробнее работу с интерфейсом `Serializable`. Заметим, что класс `Object` не реализует этот интерфейс. Таким образом, существует два варианта – либо сериализуемый класс наследуется от `Serializable` -класса, либо нет. Первый вариант довольно прост. Если родительский класс уже реализовал интерфейс `Serializable`, то наследникам это свойство передается автоматически, то есть все объекты, порожденные от такого класса, или любого его наследника, могут быть сериализованы.

Если же наш класс впервые реализует `Serializable` в своей ветке наследования, то его суперкласс должен отвечать специальному требованию – у него должен быть доступный конструктор без параметров. Именно с помощью этого конструктора будет создан десериализуемый объект и будут проинициализированы все поля, унаследованные от классов, не наследующих `Serializable`.

Рассмотрим пример:

```
// Родительский класс, не реализующий Serializable
public class Parent {
   public String firstName;
   private String lastName;
   public Parent(){
      System.out.println("Create Parent");
      firstName="old_first";
      lastName="old_last";
   }
   public void changeNames() {
      firstName="new_first";
      lastName="new_last";
```

```
    }
    public String toString() {
      return super.toString()+",first="+firstName+",last="+lastName;
    }
  }
  // Класс Child, впервые реализовавший Serializable
  public class Child extends Parent implements Serializable {
    private int age;
    public Child(int age) {
      System.out.println("Create Child");
      this.age=age;
    }
    public String toString() {
      return super.toString()+",age="+age;
    }
  }
  // Наследник Serializable-класса
  public class Child2 extends Child {
    private int size;
    public Child2(int age, int size) {
      super(age);
      System.out.println("Create Child2");
      this.size=size;
    }
    public String toString() {
      return super.toString()+",size="+size;
    }
  }
  // Запускаемый класс для теста
  public class Test {
    public static void main(String[] arg) {
      try {
        FileOutputStream fos=new FileOutputStream("output.bin");
        ObjectOutputStream oos=new ObjectOutputStream(fos);
        Child c=new Child(2);
        c.changeNames();
        System.out.println(c);
        oos.writeObject(c);
        oos.writeObject(new Child2(3, 4));
```

```
            oos.close();
            System.out.println("Read objects:");
            FileInputStream fis=new FileInputStream("output.bin");
            ObjectInputStream ois=new ObjectInputStream(fis);
            System.out.println(ois.readObject());
            System.out.println(ois.readObject());
            ois.close();
        } catch (Exception e) { // упрощенная обработка для краткости
            e.printStackTrace();
        }
    }
}
```

Пример 15.10.

В этом примере объявлено 3 класса. Класс `Parent` не реализует `Serializable` и, следовательно, не может быть сериализован. В нем объявлено 2 поля, которые при создании получают значения, содержащие слово `"old"` ("старый"). Кроме этого, объявлен метод, позволяющий модифицировать эти поля. Он выставляет им значения, содержащие слово `"new"` ("новый'). Также переопределен метод `toString()`, чтобы дать возможность узнать значения этих полей.

Поскольку класс `Parent` имеет доступный конструктор по умолчанию, его наследник может реализовать интерфейс `Serializable`. Обратите внимание, что у самого класса `Child` такого конструктора уже нет. Также объявлено поле и модифицирован метод `toString()`.

Наконец, класс `Child2` наследуется от `Child`, а потому автоматически является допустимым для сериализации. Аналогично, имеет новое поле, значение которого отображает `toString()`.

Запускаемый класс `Test` сериализует в файл `output.bin` два объекта. Обратите внимание, что у первого из них предварительно вызывается метод `changeNames()`, который модифицирует значения полей, унаследованных от класса `Parent`.

Результат выполнения примера:

```
Create Parent
Create Child
Child@ad3ba4,first=new_first,last=new_last,age=2
Create Parent
Create Child
Create Child2
Read objects:
Create Parent
Child@723d7c,first=old_first,last=old_last,age=2
Create Parent
Child2@22c95b,first=old_first,last=old_last,age=3,size=4
```

Пример 15.11.

Во всех конструкторах вставлена строка, выводящая сообщение на консоль. Так можно отследить, какие конструкторы вызываются во время десериализации. Видно, что для объектов, порожденных от `Serializable` -классов, конструкторы не вызываются вовсе. Идет обращение лишь к конструктору без параметров не- `Serializable` -суперкласса.

Сравним значения полей первого объекта и его копии, полученной десериализацией. Поля, унаследованные от не- `Serializable` -класса (`firstName`, `lastName`), не восстановились. Они имеют значения, полученные в конструкторе `Parent` без параметров. Поля, объявленные в `Serializable` -классе, свои значения сохранили. Это верно и для второго объекта – собственные поля `Child2` и унаследованные от `Child` имеют точно такие же значения, что и до сериализации. Их значения были записаны, а потом считаны и напрямую установлены из потока данных.

Иногда в классе есть поля, которые не должны участвовать в сериализации. Тому может быть несколько причин. Например, это поле малосущественно (временная переменная) и сохранять его нет необходимости. Если сериализованный объект передается по сети, то исключение такого поля из сериализации позволяет уменьшить нагрузку на сеть и ускорить работу приложения.

Некоторые поля хранят значения, которые не будут иметь смысла при

пересылке объекта на другую машину, или при воссоздании его спустя какое-то время. Например, сетевое соединение, или подключение к базе данных, в таких случаях нужно устанавливать заново.

Затем, в объекте может храниться конфиденциальная информация, например, пароль. Если такое поле будет сериализовано и передано по сети, его значение может быть перехвачено и прочитано, или даже подменено.

Для исключения поля объекта из сериализации его необходимо объявить с модификатором *transient*. Например, следующий класс:

```
class Account implements
      java.io.Serializable {
   private String name;
   private String login;
   private transient String password;
   /* объявление других элементов класса
   ...
   */
}
```

У такого класса поле `password` в сериализации участвовать не будет и при восстановлении оно получит значение по умолчанию (в данном случае `null`).

Особого внимания требуют статические поля. Поскольку они принадлежат классу, а не объекту, они не участвуют в сериализации. При восстановлении объект будет работать с таким значением `static` -поля, которое уже установлено для его класса в этой JVM.

Граф сериализации

До этого мы рассматривали объекты, которые имеют поля лишь примитивных типов. Если же сериализуемый объект ссылается на другие объекты, их также необходимо сохранить (записать в поток байт), а при десериализации – восстановить. Эти объекты, в свою очередь, также могут ссылаться на следующие объекты. При этом важно, что

если несколько ссылок указывают на один и тот же объект, то этот объект должен быть сериализован лишь однажды, а при восстановлении все ссылки должны вновь указывать на него одного. Например, сериализуемый объект `A` ссылается на объекты `B` и `C`, каждый из которых, в свою очередь, ссылается на один и тот же объект `D`. После десериализации не должно возникать ситуации, когда `B` ссылается на `D1`, а `C` – на `D2`, где `D1` и `D2` – равные, но все же различные объекты.

Для организации такого процесса стандартный механизм сериализации строит граф, включающий в себя все участвующие объекты и ссылки между ними. Если очередная ссылка указывает на некоторый объект, сначала проверяется – нет ли такого объекта в графе. Если есть – объект второй раз не сериализуется. Если нет – новый объект добавляется в граф.

При построении графа может встретиться объект, порожденный от класса, не реализующего интерфейс `Serializable`. В этом случае сериализация прерывается, генерируется исключение `java.io.NotSerializableException`.

Рассмотрим пример:

```
import java.io.*;
class Point implements 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 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{
        // записываем объекты в файл
        FileOutputStream os = new FileOutputStream(fileName);
        ObjectOutputStream oos = new ObjectOutputStream(os);
        oos.writeObject(line1);
        oos.writeObject(line2);
        // меняем состояние line1 и записываем его еще раз
        line1.setIndex(3);
        //oos.reset();
        oos.writeObject(line1);
        // закрываем потоки
```

```
        // достаточно закрыть только поток-надстройку
        oos.close();
        // считываем объекты
        System.out.println("Read objects:");
        FileInputStream is = new FileInputStream(fileName);
        ObjectInputStream ois = new ObjectInputStream(is);
        for (int i=0; i<3; i++) { // Считываем 3 объекта
          Line line = (Line)ois.readObject();
          line.printInfo();
        } ois.close();
      } catch(ClassNotFoundException e) {
        e.printStackTrace();
      } catch(IOException e) {
        e.printStackTrace();
      }
    }
}
```

Пример 15.12.

В этой программе работа идет с классом `Line` (линия), который имеет 2 поля типа `Point` (линия описывается двумя точками). Запускаемый класс `Main` создает два объекта класса `Line`, причем, одна из точек у них общая. Кроме этого, линия имеет номер (поле `index`). Созданные линии (номера 1 и 2) записываются в поток, после чего одна из них получает новый номер (3) и вновь сериализуется.

Выполнение этой программы приведет к выводу на экран примерно следующего:

```
Constructing line: 1
Constructing line: 2
line 1 = Line@7d39
line 2 = Line@4ec
Read objects:
Line: 1
  Object reference: Line@331e
  from point (1.0,1.0) reference=Point@36bb
  to point (2.0,2.0) reference=Point@386e
```

```
Line: 2
  Object reference: Line@6706
  from point (2.0,2.0) reference=Point@386e
  to point (3.0,3.0) reference=Point@68ae
Line: 1
  Object reference: Line@331e
  from point (1.0,1.0) reference=Point@36bb
  to point (2.0,2.0) reference=Point@386e
```

Пример 15.13.

Из примера видно, что после восстановления у линий сохраняется общая точка, описываемая одним и тем же объектом (хеш-код `386e`).

Третий записанный объект идентичен первому, причем, совпадают даже объектные ссылки. Несмотря на то, что при записи третьего объекта значение `index` было изменено на 3, в десериализованном объекте оно осталось равным `1`. Так произошло потому, что объект, описывающий первую линию, уже был задействован в сериализации и, встретившись во второй раз, повторно записан не был.

Чтобы указать, что сеанс сериализации завершен, и получить возможность передавать измененные объекты, у `ObjectOutputStream` нужно вызвать метод `reset()`. В рассматриваемом примере для этого достаточно убрать комментарий в строке

```
//oos.reset();
```

Если теперь запустить программу, то можно увидеть, что третий объект получит номер 3.

```
Constructing line: 1
Constructing line: 2
line 1 = Line@ea2dfe
line 2 = Line@7182c1
Read objects:
Line: 1
  Object reference: Line@a981ca
  from point (1.0,1.0) reference=Point@1503a3
```

```
    to point (2.0,2.0) reference=Point@a1c887
  Line: 2
    Object reference: Line@743399
    from point (2.0,2.0) reference=Point@a1c887
    to point (3.0,3.0) reference=Point@e7b241
  Line: 3
    Object reference: Line@67d940
    from point (1.0,1.0) reference=Point@e83912
    to point (2.0,2.0) reference=Point@fae3c6
```

Пример 15.14.

Однако это будет уже новый объект, ссылка на который отличается от первой считанной линии. Более того, обе точки будут также описываться новыми объектами. То есть в новом сеансе все объекты были записаны, а затем восстановлены заново.

Расширение стандартной сериализации

Некоторым сложно организованным классам требуется особый подход для сериализации. Для расширения стандартного механизма можно объявить в классе два метода с точно такой сигнатурой:

```
  private void writeObject(
              java.io.ObjectOutputStream out)
    throws IOException;
  private void readObject(
              java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException;
```

Если в классе объявлены такие методы, то при сериализации объекта для записи его состояния будет вызван `writeObject`, который должен сгенерировать последовательность байт и записать ее в поток `out`, полученный в качестве аргумента. При этом можно вызвать стандартный механизм записи объекта путем вызова метода

```
  out.defaultWriteObject();
```

Этот метод запишет все не- *`transient`* и не- `static` поля в поток

данных.

В свою очередь, при десериализации метод `readObject` должен считать данные из потока `in` (также полученного в качестве аргумента) и восстановить значения полей класса. При реализации этого метода можно обратиться к стандартному механизму с помощью метода:

```
in.defaultReadObject();
```

Этот метод считывает описание объекта из потока и присваивает значения соответствующих полей в текущем объекте.

Если же процедура сериализации в корне отличается от стандартной, то для таких классов предназначен альтернативный интерфейс `java.io.Externalizable`.

При использовании этого интерфейса в поток автоматически записывается только идентификация класса. Сохранить и восстановить всю информацию о состоянии экземпляра должен сам класс. Для этого в нем должны быть объявлены методы `writeExternal()` и `readExternal()` интерфейса `Externalizable`. Эти методы должны обеспечить сохранение состояния, описываемого полями самого класса и его суперкласса.

При восстановлении `Externalizable` -объекта экземпляр создается путем вызова конструктора без аргументов, после чего вызывается метод `readExternal`.

Метод `writeExternal` имеет сигнатуру:

```
void writeExternal(ObjectOutput out)
  throws IOException;
```

Для сохранения состояния вызываются методы `ObjectOutput`, с помощью которых можно записать как примитивные, так и объектные значения. Для корректной работы в соответствующем методе

```
void readExternal(ObjectInput in)
  throws IOException,ClassNotFoundException;
```

эти значения должны быть считаны в том же самом порядке.

Классы Reader и Writer и их наследники

Рассмотренные классы – наследники `InputStream` и `OutputStream` – работают с байтовыми данными. Если с их помощью записывать или считывать текст, то сначала необходимо сопоставить каждому символу его числовой код. Такое соответствие называется кодировкой.

Известно, что Java использует кодировку `Unicode`, в которой символы представляются двухбайтовым кодом. Байтовые потоки зачастую работают с текстом упрощенно – они просто отбрасывают старший байт каждого символа. В реальных же приложениях могут использовать различные кодировки (даже для русского языка их существует несколько). Поэтому в версии Java 1.1 появился дополнительный набор классов, основывающийся на типах `Reader` и `Writer`. Их иерархия представлена на рис. 15.2.

Эта иерархия очень схожа с аналогичной для байтовых потоков `InputStream` и `OutputStream`. Главное отличие между ними – `Reader` и `Writer` работают с потоком символов (`char`). Только чтение массива символов в `Reader` описывается методом `read(char[])`, а запись в `Writer` – `write(char[])`.

В таблице 15.1 приведены соответствия классов для байтовых и символьных потоков.

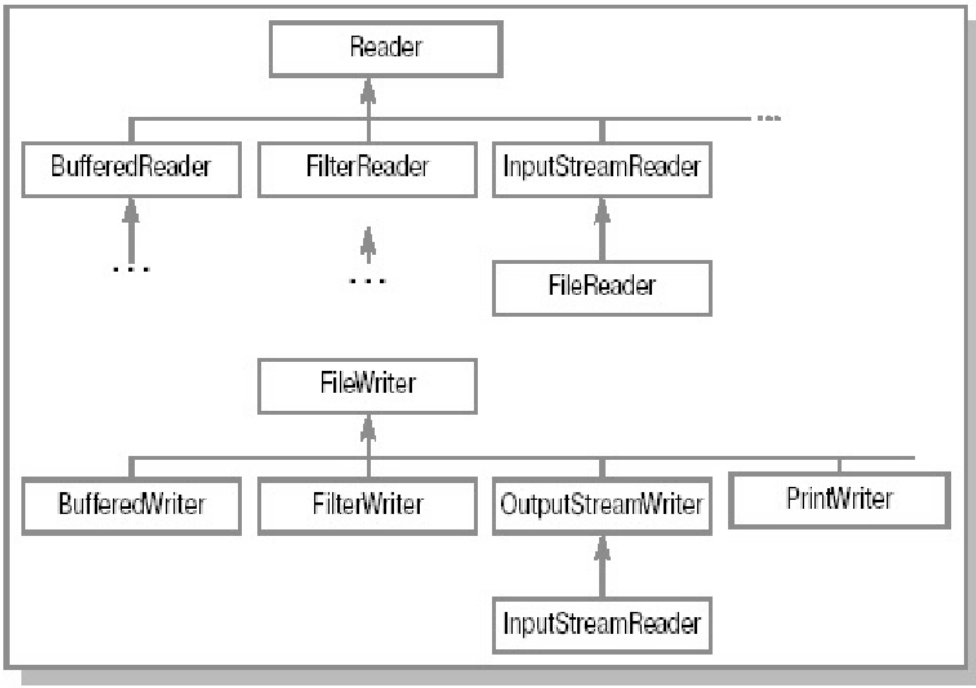


Рис. 15.2. Иерархия классов Reader и Writer.

Таблица 15.1. Соответствие классов для байтовых и символьных потоков.

Байтовый поток	Символьный поток
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 и предсказуемы.

Например, конечно же, отсутствует преобразование в символьное представление примитивных типов Java и объектов (`DataInput/Output`, `ObjectInput/Output`). Добавлены классы-мосты, преобразующие символьные потоки в байтовые: `InputStreamReader` и `OutputStreamWriter`. Именно на их основе реализованы `FileReader` и `FileWriter`. Метод `available()` класса `InputStream` в классе `Reader` отсутствует, он заменен методом `ready()`, возвращающим 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";

//Строка, которая будет записана в файл
String data = "Some data to be written and read.\n";
try{
   FileWriter fw = new FileWriter(fileName);
   BufferedWriter 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();

    // Считываем результат
    FileReader fr = new FileReader(fileName);
    BufferedReader 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();
  }
```

Пример 15.15.

Классы-мосты `InputStreamReader` и `OutputStreamWriter` при преобразовании символов также используют некоторую кодировку. Ее можно задать, передав в конструктор в качестве аргумента ее название. Если оно не будет соответствовать никакой из известных кодировок, будет брошено исключение `UnsupportedEncodingException`. Вот некоторые из корректных значений этого аргумента (чувствительного к регистру!) для распространенных кодировок: `"Cp1251"`, `"UTF-8"`, `"8859_1"` и т.д.

Класс StreamTokenizer

Экземпляр `StreamTokenizer` создается поверх существующего объекта, либо `InputStream`, либо `Reader`. Как и `java.util.StringTokenizer`, этот класс позволяет разбивать

данные на лексемы (token), выделяемые из потока по определенным свойствам. Поскольку работа ведется со словами, конструктор, принимающий `InputStream`, объявлен как *`deprecated`* (предлагается оборачивать байтовый поток классом `InputStreamReader` и вызывать второй конструктор). Общий принцип работы такой же, как и у `StringTokenizer`, – задаются параметры разбиения, после чего вызывается метод `nextToken()`, пока не будет достигнут конец потока. Способы задания разбиения у `StreamTokenizer` довольно разнообразны, но просты, и поэтому здесь не рассматриваются.

Работа с файловой системой

Класс File

Если классы потоков осуществляют реальную запись и чтение данных, то класс `File` – это вспомогательный инструмент, призванный обеспечить работу с файлами и каталогами.

Объект класса `File` является абстрактным представлением файла и пути к нему. Он устанавливает только соответствие с ним, при этом для создания объекта неважно, существует ли такой файл на диске. После создания можно выполнить проверку, вызвав метод `exists`, который возвращает значение `true`, если файл существует. Создание или удаление объекта класса `File` никоим образом не отображается на реальных файлах. Для работы с содержимым файла можно получить экземпляры `FileI/OStream`.

Объект `File` может указывать на каталог (узнать это можно путем вызова метода `isDirectory`). Метод `list` возвращает список имен (массив `String`) содержащихся в нем файлов (если объект `File` не указывает на каталог – 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");
  }
}
```

Пример 15.16.

При выполнении этой программы на экран будут выведены названия (в каноническом виде) всех файлов, с расширением `.java`, содержащихся

в текущем каталоге и всех его подкаталогах.

Для определения того, что файл имеет расширение `.java`, использовался интерфейс `FileFilter` с реализацией в виде внутреннего класса `NameFilter`. Интерфейс `FileFilter` определяет только один метод `accept`, возвращающий значение, определяющее, 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 объект файла к проверке, а объект `File`, указывающий на каталог, 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.

Также класс `File` предоставляет возможность получения некоторой информации о файле.

- Методы `canRead` и `canWrite` – возвращается `boolean` значение, можно ли будет приложению производить чтение и изменение содержимого из файла, соответственно.
- `getName` – возвращает строку – имя файла (или каталога).
- `getParent`, `getParentName` – возвращают каталог, где файл находится в виде объекта и строки названия `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 каталог либо файл, соответственно.
- `isHidden` – возвращает `boolean` значение, указывает ли объект на скрытый файл.
- `lastModified` – дата последнего изменения.
- `length` – длина файла в байтах.

Также можно изменить некоторые свойства файла – методы `setReadOnly`, `setLastModified`, назначение которых очевидно из названия. Если нужно создать файл на диске, это позволяют сделать методы `createNewFile`, `mkDir`, `mkDirs`. Соответственно, `createNewFile` создает пустой файл (если таковой еще не существует), `mkDir` создает каталог, если для него все родительские уже существуют, а `mkDirs` создаст каталог вместе со всеми необходимыми родительскими.

Файл можно и удалить – 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) отменить запрос уже невозможно.

Таким образом, класс `File` дает возможность достаточно полного управления файловой системой.

Класс RandomAccessFile

Этот класс реализует сразу два интерфейса – `DataInput` и `DataOutput` – следовательно, может производить запись и чтение всех примитивных типов Java. Эти операции, как следует из названия, производятся с файлом. При этом их можно производить поочередно, произвольным образом перемещаясь по файлу с помощью вызова метода `seek(long)` (переводит на указанную позицию в файле). 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 строку либо `"r"` (только чтение), либо `"rw"` (чтение и запись). Попытка открыть несуществующий файл только на чтение приведет к исключению `FileNotFoundException`. При открытии на чтение и запись он будет незамедлительно создан (или же будет 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`.

Заключение

В данной лекции вы познакомились с таким важным понятием, как потоки данных (stream). Потоки являются очень эффективным способом решения задач, связанных с передачей и получением данных, независимо от особенностей используемых устройств ввода/вывода. Как вы теперь знаете, именно в пакете `java.io` содержатся стандартные классы, решающие задачи обмена данными в самых различных форматах.

Были описаны базовые классы байтовых потоков `InputStream` и `OutputStream`, а также символьных потоков `Reader` и `Writer`. Все классы потоков явным или неявным образом наследуются от них. Краткий обзор показал, для чего предназначен каждый класс, как с ним работать, какие классы не рекомендованы к использованию. Изучено, как передавать в потоки значения примитивных типов Java. Особое внимание было уделено операциям с объектами, для которых существует специальный механизм сериализации.

Наконец, были описаны классы для работы с файловой системой – `File` и `RandomAccessFile`.

Введение в сетевые протоколы

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 значения необходимы, поскольку библиотека java.net, по сути, является интерфейсом для работы с этими протоколами. Рассматриваются классы для соединений через высокоуровневые протоколы, протоколы TCP и UDP.

Основы модели OSI

В течение последних нескольких десятилетий размеры и количество сетей значительно выросли. В 80-х годах существовало 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 сетей (например, различный тип адресации делал эти попытки практически безнадежными).

Эта проблема была рассмотрена Всемирной организацией по стандартизации (International Organization for Standardization, ISO) и было принято решение разработать модель сети, которая могла бы помочь разработчикам и производителям сетевого оборудования и программного обеспечения действовать сообща. В результате в 1984 г. была создана модель OSI – модель взаимодействия открытых систем (Open Systems Interconnected). Она состоит из семи уровней, на которые разделяется задача организации сетевого взаимодействия. Схематично они представлены в таблице 16.1.

Таблица 16.1. Уровни модели OSI.

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 3	Сетевой уровень	Пакет (packet)
Layer 2	Уровень передачи данных	Фрейм (frame)
Layer 1	Физический уровень	Бит (bit)

Хотя сегодня существуют разнообразные модели сетей, большинство разработчиков придерживается именно этой общепризнанной схемы.

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 пересылаемой информации (например, это может быть информация о файле, который необходимо передать, или операции, которую нужно 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 (хотя уровень 6 может считывать служебную информацию уровня 7). Протокол представительского уровня выполняет необходимые действия на основании данных, полученных из заголовка уровня приложений, и добавляет заголовок своего уровня, в котором содержится 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.16.1). При этом служебная информация может добавляться не только в начало сообщения, но и в конец (например, на 3-м уровне, рис.16.2). В итоге получается сообщение, содержащее служебную информацию всех семи уровней.

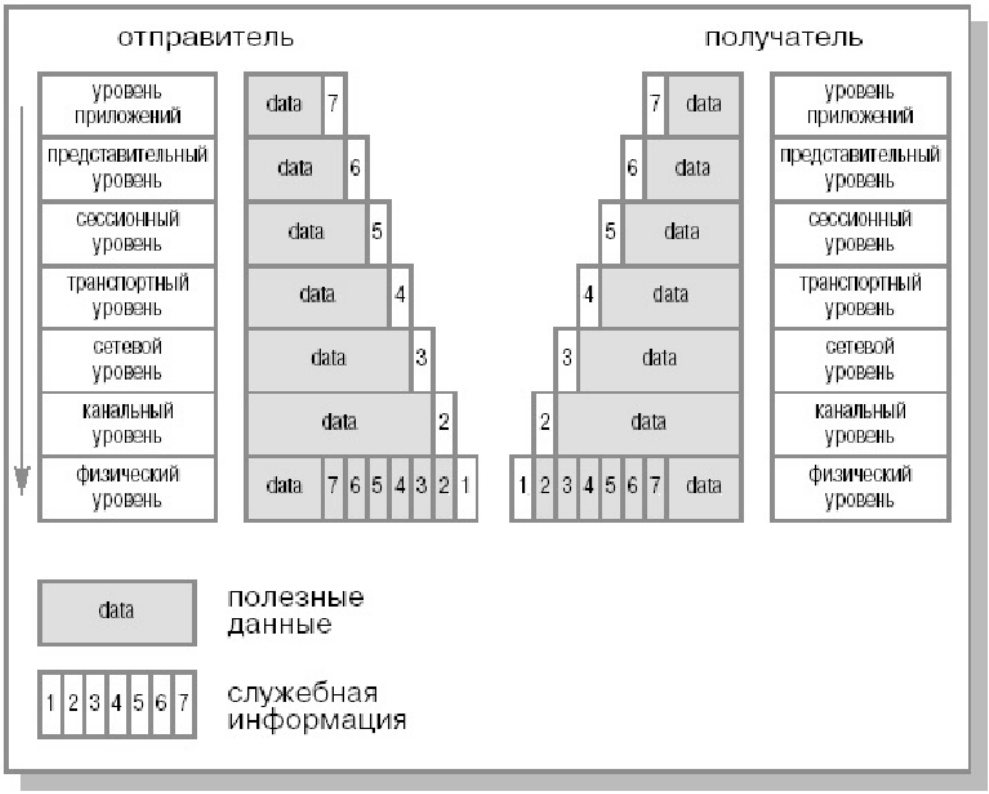


Рис. 16.1. Инкапсуляция и декапсуляция пакета.

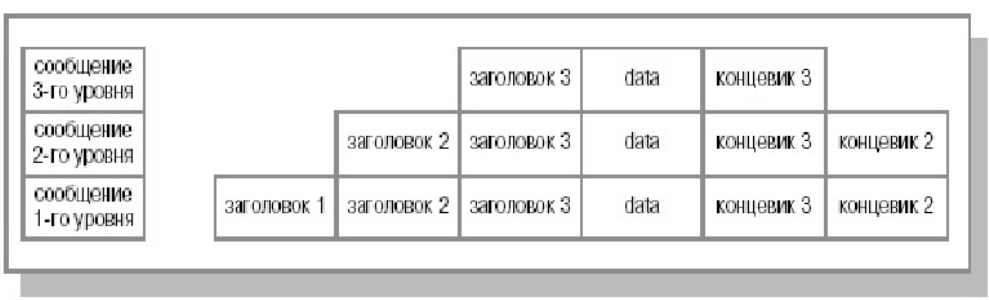


Рис. 16.2. Добавление служебной информации в начало и конец пакета.

Процесс "обертывания" передаваемых данных служебной информацией 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, 0101101010. В простейшем случае на физическом уровне для передачи формируется электрический сигнал, состоящий из 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.16.3). Далее происходит процесс, обратный инкапсуляции,– декапсуляция (decapsulation). На каждом уровне происходит разбор служебной информации. После декапсуляции сообщения на первом уровне (считывания и обработки служебной информации 1-го 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.

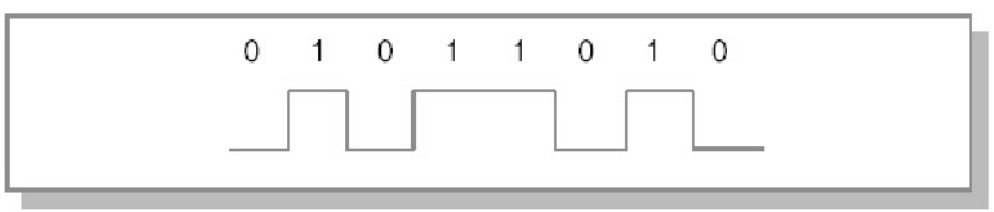


Рис. 16.3. Представление данных в виде электрического импульса.

В качестве примера можно привести обращение браузера к web-серверу. Приложение клиента – браузер – формирует запрос для получения web-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 передается по физической сети (кабелям) в виде электрических импульсов на сервер. На сервере происходит разбор соответствующей системной информации каждого уровня, в результате чего посланный запрос достигает приложения web-

сервера. Там он обрабатывается, после чего клиенту отправляется ответ. Процесс отправки ответа аналогичен отправке запроса – за исключением того, что сообщение посылает сервер, а получает клиент.

Так как каждый уровень модели OSI стандартизирован, потребители могут использовать совместно оборудование и программное обеспечение различных производителей. В результате web-сервер под управлением операционной системы Sun Solaris может передать HTML-страницу пользователю MS Windows.

Разумеется, совместимость можно обеспечить лишь до некоторого уровня. Если одна машина передает данные в виде радиоволн, а другая в виде световых импульсов, то их взаимодействие без использования дополнительного оборудования невозможно. Поэтому было введено понятие сете-независимых и сете-зависимых уровней.

Три нижних уровня – физический, канальный и сетевой – являются сете-зависимыми. Например, смена Ethernet на ATM влечет за собой полную смену протокола физического и канального уровней.

Три верхних уровня – приложений, представительский и сессионный – ориентированы на прикладные задачи и практически не зависят от физической технологии построения сети. Так, переход от Token Ring на Ethernet не требует изменений в перечисленных уровнях.

Транспортный уровень является промежуточным 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 сетевых сообщений.

Вместе с названием сообщение (message) в стандартах ISO для обозначения единицы данных используют термин протокольный блок данных (Protocol Data Unit, PDU). В разных протоколах применяются и другие названия, закрепленные стандартами, или просто традиционные. Например, в семействе протоколов TCP/IP протокол TCP разделяет поток данных на сегменты, протокол UDP работает с датаграммами (или дейтаграммами, от datagram), сам протокол IP использует термин пакеты. Часто так же говорят о кадрах или фреймах.

Для более глубокого понимания принципов работы сети рассмотрим каждый уровень по отдельности.

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, кабели и т.д.) и правила формирования этих сигналов. Рассмотрим по порядку все составляющие этого уровня.

Большая часть сетей строится на кабельной структуре (хотя существуют сети, основанные на передаче информации с помощью, например, радиоволн). Сейчас существуют различные типы кабелей. Наиболее распространенные из них:

- телефонный провод;
- коаксиальный кабель ;
- витая пара ;
- оптоволокно.

Телефонный кабель начал использоваться для передачи данных со времен появления первых компьютеров. Главным преимуществом телефонных линий было наличие уже созданной и развитой инфраструктуры. С ее помощью можно передавать данные между компьютерами, находящимися на разных материках, так же легко, как и вести разговор людям, которые находятся за много тысяч километров друг от друга. На сегодняшний день использование телефонных линий также остается популярным. Пользователи, которых устраивает небольшая скорость передачи данных, могут получить доступ к интернету со своих домашних компьютеров. Основными недостатками использования телефонного кабеля 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 так как 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 еще несколько лет назад, но сегодня это большая редкость. Такой тип кабеля по строению практически идентичен обычному телевизионному коаксиальному кабелю – центральная медная жила отделена слоем изоляции от оплетки. Некоторые отличия есть в электрических характеристиках (в телевизионном кабеле используется кабель с волновым сопротивлением 75 Ом, в сетевом – 50 Ом).

Основными недостатками этого кабеля является низкая скорость передачи данных (до 10 Мбит/с), подверженность воздействиям внешних помех. Кроме того, подключение компьютеров в таких сетях происходит параллельно, а значит, максимальная возможная скорость пропускания делится на всех пользователей. Но, по сравнению с телефонным кабелем, коаксиал позволяет объединять близко расположенные компьютеры с намного лучшим качеством связи и более высокой скоростью передачи данных.

Витая пара (" twisted pair ") – наиболее распространенное средство для передачи данных между компьютерами. В данном типе кабеля используется медный попарно скрученный провод, что позволяет уменьшить количество помех и наводок, как при передаче сигнала по самому кабелю, так и при воздействии внешних помех.

Существует несколько категорий этого кабеля. Перечислим 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 его 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 Cat 3. Основное преимущество витой пары перед телефонными и коаксиальными кабелями – более высокая скорость передачи данных. Также использование Cat 5 в большинстве случаев позволяет, не меняя кабельную структуру, повысить

производительность сети (переходом от 10 к 100 и от 100 к 1000 Мбит/с).

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 применяются более широкие сердечники, которые легче изготовить. В этих кабелях в сердечнике одновременно существует несколько световых лучей, отражающихся от оболочки под разными углами. Угол отражения луча называется модой луча. Оптоволокно обладает следующими преимуществами: устойчивость к электромагнитным помехам, высокие скоростные характеристики на больших расстояниях. Основным недостатком является как дороговизна самого кабеля, так и трудоемкость монтажных работ, так как все работы выполняются на дорогостоящем высокоточном оборудовании.

Физический уровень также отвечает за преобразование сигналов между различными средами передачи данных. Например, при необходимости соединить сегменты сети, построенные на оптоволокне и витой паре, применяют так называемые конверторы (в данном случае они преобразуют световой импульс в электрический).

Для включения компьютера в сеть используется специальное устройство – сетевой адаптер (Network adapter), позволяющий

обмениваться наборами битов, представленными электрическими сигналами. Сетевая карта (так чаще называют сетевой адаптер) обычно имеет шину ISA или PCI для подключения в компьютер и соответствующий разъем для подключения к среде передачи данных (например, для витой пары, коаксиал и т.п.).

Теперь, когда мы знаем, как происходит соединение компьютеров в одну сеть, рассмотрим варианты физической схемы такой сети, или, другими словами, физической топологии (структуры локальной сети).

Топология "шина" (bus) показана на рис. 16.4.

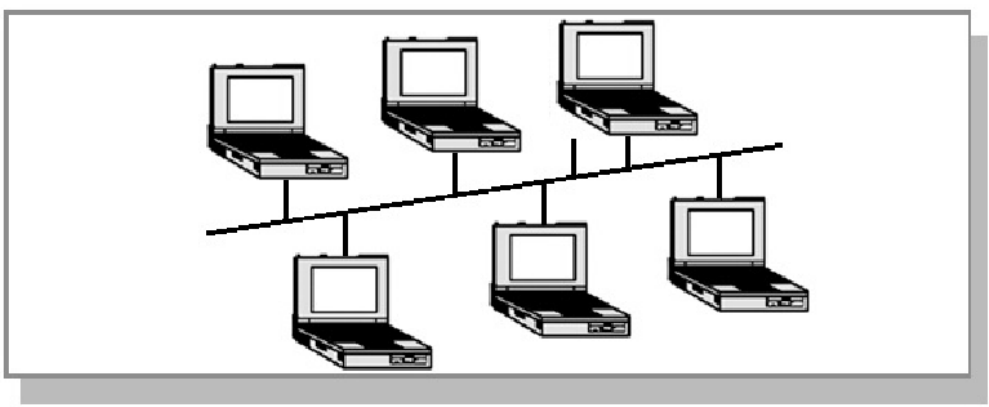


Рис. 16.4. Топология "шина" (bus).

Все компьютеры и сетевые устройства подсоединены к одному проводу и фактически напрямую соединены между собой.

Топология "кольцо" (ring) показана на рис. 16.5.

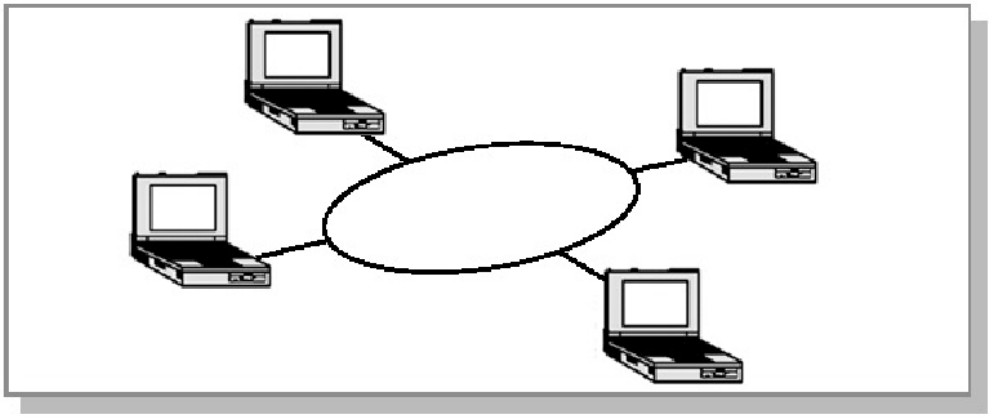


Рис. 16.5. Топология "кольцо" (ring).

Кольцо состоит из сетевых устройств и кабелей между ними, образующих одно замкнутое кольцо.

Топология "звезда" показана на рис. 16.6.

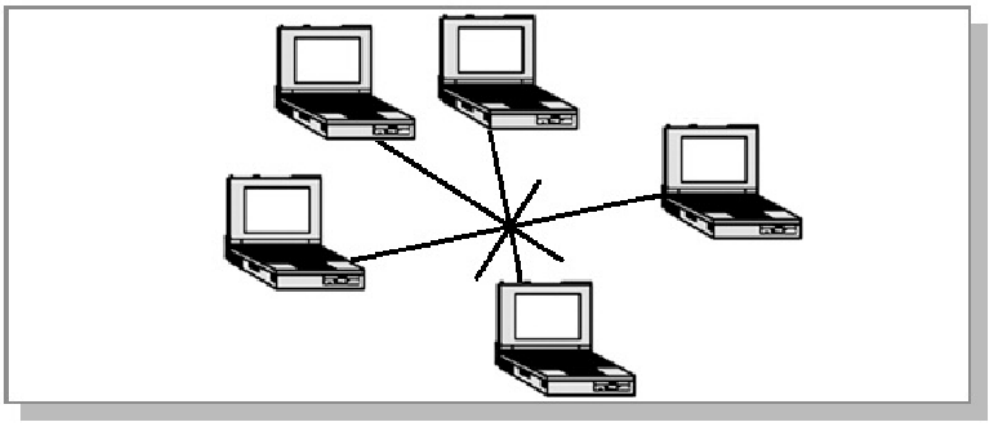


Рис. 16.6. Топология "звезда" (star).

Все компьютеры и сетевые устройства подключены к одному центральному устройству.

Топология "расширенная звезда" (extended star) показана на рис. 16.7.

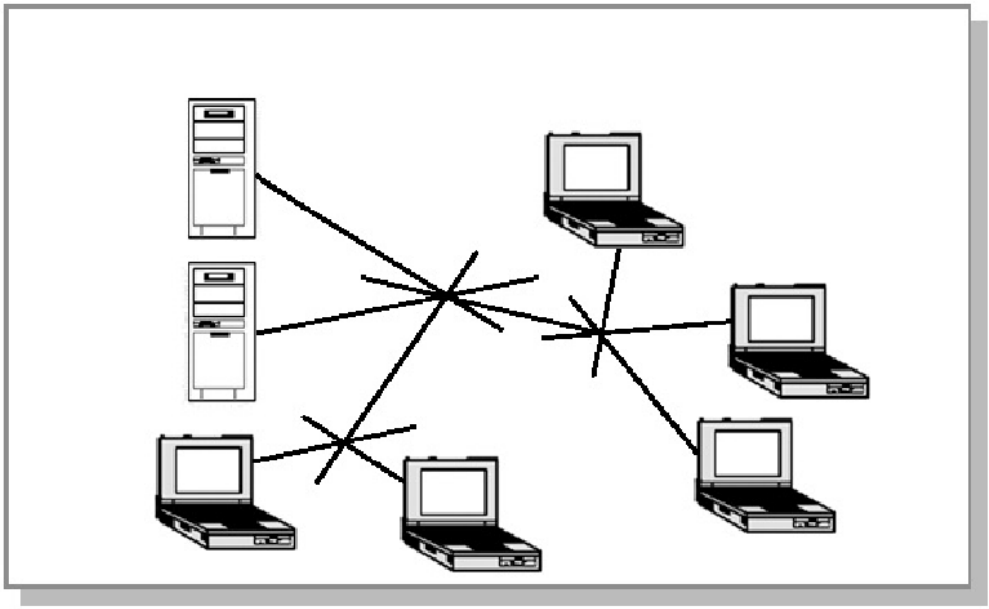


Рис. 16.7. Топология "расширенная звезда"(extended star).

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".

Data layer (layer 2)

Физический уровень пересылает просто набор сигналов – битов. При этом не учитывается, что несколько компьютеров, подключенных к одной среде передачи данных (например, к одному кабелю), могут начать одновременно передавать информацию в виде электрических импульсов, что, очевидно, приведет к смешению сигналов. Поэтому одной из задач Data layer (канальный уровень) является проверка доступности среды передачи. Также этот 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:

- логическая передача данных (Logical Link Control, LLC);
- управление доступом к среде (Media Access Control, MAC).

LLC отвечает за переход со второго уровня на более высокий – третий сетевой уровень.

MAC отвечает за передачу данных на более низкий уровень – Physical layer.

Рассмотрим эти подуровни более подробно.

LLC sublayer

Этот подуровень был создан для обеспечения независимости от существующих технологий. Он обеспечивает обмен данными с сетевым (третьим) уровнем 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 обработки протоколом уровня MAC. Например, это может быть Ethernet, Token Ring, Frame Relay.

MAC sublayer

Этот подуровень обеспечивает доступ к физическому уровню. Для передачи пакетов по сети необходимо организовать идентификацию компьютеров в сети. Для этого у каждого компьютера на канальном уровне определен уникальный адрес, который еще иногда называют физическим адресом, или MAC-адресом.

Он записан в энергонезависимой памяти сетевой карты 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 (Organizational Unique Identifier), организационный уникальный идентификатор. Этот номер выдается каждому производителю сетевого оборудования международной организацией IEEE (Institute of Electrical and Electronic Engineers, Институт инженеров по электротехнике и радиоэлектронике, источник многих стандартов и спецификаций). Последние 3 байта являются идентификационным номером самой сетевой карты. Производитель гарантирует, что все его адаптеры имеют различные номера. Такая

система адресов гарантирует, что в сети не будет двух компьютеров с одинаковыми физическими адресами.

Записываться физический адрес может в разных форматах, например: `00:00:B4:90:4C:8C`, `00-00-B4-90-4C-8C`, `0000.B490.4C8C` – разные производители используют разные стандарты. Рассмотрим, например, адрес `0000.1c12.3456`. Здесь `0000.1c` – идентификатор производителя, а `12.3456` – идентификатор сетевой карты.

Один из самых распространенных протоколов MAC-уровня – протокол Ethernet. В сетях, построенных на его основе, применяется специальный метод для организации доступа к среде передачи данных – CSMA/CD (carrier sense multiple access/collision detect, коллективный доступ с опознаванием несущей и обнаружением коллизий). Предполагается, что основой сети является общая шина (например, коаксиальный кабель), к которой подключены все компьютеры. В результате сообщение, отправленное одной машиной, доставляется всем подключенным сетевым устройствам. CSMA/CD описывает целый комплекс мер, необходимых для предотвращения и корректной обработки коллизий (collision), то есть ситуаций, когда несколько компьютеров одновременно начали передачу данных. Очевидно, что в таком случае никто не сможет получить корректную информацию из сети.

Рассмотрим более подробно процесс передачи данных на Data layer. Пусть один компьютер собирается послать данные другому. Во время процесса инкапсуляции MAC-адрес этой машины и MAC-адрес получателя будут записаны в служебные поля. Сгенерированное сообщение по правилам протокола Ethernet отсылается через общую шину всем машинам, подключенным к этому участку сети.

Каждый компьютер, получивший сообщение, проверяет, кому оно было адресовано. Если MAC-адрес, указанный во фрейме, и MAC-адрес, записанный в сетевом адаптере получателя, 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.

Иногда бывает необходимо послать сообщение, которое должно быть

получено всеми узлами локальной сети. В этом случае в пакете указывается MAC-адрес получателя в виде `FF-FF-FF-FF-FF-FF`. Этот адрес используется для широковещания (broadcast), которое примут все сетевые устройства и передадут на вышестоящий уровень.

Рассмотрим устройства, применяемые для построения сетей в разных топологиях.

Топология шина ("bus") описывает общую среду передачи данных, которая уже рассматривалась для иллюстрации протокола Ethernet. Специальных устройств для построения такой сети не используется (впрочем, конкретные технологии могут предъявлять специфические требования; например, концы коаксиального кабеля должны подключаться к особому устройству – терминатору, но это не влияет на структуру сети).

На топологии кольцо ("ring") основывается протокол Token Ring. Физически сеть представляет собой замкнутое кольцо, в котором каждый компьютер двумя отрезками кабеля соединяется со своими соседями. В отличие от сети, работающей на основе Ethernet, здесь используется более сложная схема. Передача ведется последовательно по кольцу в одном направлении. В сети циркулирует кадр специального формата – маркер (token). Если машина не имеет данных для передачи, она при получении маркера передает его дальше по кольцу. В противном случае она изымает его из обращения, что дает ей доступ к сети, и затем отправляет пакет с адресом получателя, который начинает передаваться по кольцу. Когда он доходит до адресата, тот делает пометку, что пакет получен. Машина-отправитель, получив подтверждение, отправляет соседу новый маркер для обеспечения возможности другим станциям сети передавать данные. Хотя этот алгоритм более сложен, он обеспечивает свойства отказоустойчивости.

При построении сети на основе топологии "звезда" нужно использовать, кроме сетевых карт в компьютере, дополнительное сетевое оборудование в центре, куда подключаются все "лучи звезды". Например, в качестве такого устройства может применяться концентратор (hub). В этом случае каждый компьютер подключается к нему с помощью кабеля " витая пара ". Алгоритм работы концентратора очень прост – получив пакет на один из своих портов, он пересылает

его на все остальные. В результате снова получается общая шина, точнее, – логическая общая шина, поскольку физическая структура сети звездно-образная. Технология Ethernet позволяет снизить количество коллизий с помощью CSMA/CD. Недостатком концентратора является то, что пользователи сети могут "прослушивать" чужой трафик (в том числе перехватить пароль, если он передается в открытом виде). Общая максимальная скорость делится между всеми подключенными пользователями. То есть, если скорость передачи данных составляет 10 Мбит/с, то в среднем на каждого пользователя может приходиться всего 2 Мбит/с.

Более дорогим, но и более производительным решением является использование коммутатора (switch). Коммутатор, в отличие от концентратора, имеет в памяти таблицу, сопоставляющую номера его портов и MAC-адреса подключенных к нему компьютеров. Он анализирует у каждого пересылаемого фрейма адрес отправителя, пытаясь определить, какие машины подключены к каждому из его портов. Таким образом коммутатор заполняет свою таблицу. Далее при прохождении очередного фрейма он проверяет адрес получателя, и если он знает, к какому порту подключена эта машина, он посылает фрейм только на один этот порт. Если адрес получателя коммутатору неизвестен, 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 перегружают своими пакетами другие порты и, соответственно, их пакеты практически невозможно перехватить.

Построенные таким образом сети могут охватывать несколько сотен машин и иметь протяженность в несколько километров. Как правило, такая сеть охватывает одно или несколько зданий одного предприятия, а потому называется локальной сетью (Local area network, LAN).

Network layer (layer 3)

В предыдущей лекции мы рассмотрели второй уровень в модели OSI. Одним из ограничений этого уровня является использование "плоской" одноуровневой модели адресации. При попытке построить большую сеть, применяя для идентификации компьютеров MAC-адреса, мы получим огромное количество broadcast -трафика. Протокол, который

поддерживается третьим уровнем, задействует 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 компьютерной сети также должна позволять устанавливать связь между разрозненными и удаленными сетями.

На сетевом уровне (Network layer) существует несколько протоколов, которые позволяют передавать данные между сетями. Наиболее распространенным из них на сегодняшний день является IP. Его предшественник, протокол IPX, сейчас уже практически не используется в публичных сетях, но его можно найти в частных, закрытых сетях.

Основное устройство, применяемое на 3-м уровне, называется роутером (router), или маршрутизатором. Он соединяет удаленные локальные сети (LAN), образуя глобальную сеть (Wide area network, WAN). Роутер имеет два или более сетевых интерфейса и таким образом подключен сразу к нескольким локальным сетям. Получив пакет с локального устройства или компьютера, принадлежащего к одной из LAN, роутер просматривает заголовок третьего уровня. На основании полученной информации роутер принимает решение, что делать с пакетом. Если получатель пакета находится в той же локальной сети, что и отправитель, роутер игнорирует его, поскольку сообщение, как уже рассматривалось, доставляется средствами более низкоуровневых протоколов (например, Ethernet).

В противном случае пакет нужно передать в одну из других LAN, к которым подключен роутер. Основная задача этого устройства – выбор пути, по которому будет пересылаться сообщение. Поскольку может существовать множество связей между некоторыми двумя сетями

отправителя и получателя, роутер должен выбрать наиболее оптимальный путь. Пересылка пакета от одного узла сети к следующему называется hop (дословно – прыжок, скачок). Выбор очередного узла, которому роутер перешлет сообщение, может зависеть от многих факторов – загрузка сети, наименьший путь до получателя, стоимость трафика по различным маршрутам и т.д.

Новая система адресации, вводимая на сетевом уровне, должна облегчать роутеру определение пути для доставки пакета через глобальные сети. Рассмотрим реализацию наиболее популярного на сегодняшний день протокола IP более подробно.

При прохождении данных с верхних уровней на нижние на сетевом уровне к пакету добавляется служебный заголовок этого уровня. В заголовке IP-пакета содержится необходимая для дальнейшей передачи информация, такая как адреса отправителя и получателя. Понятие IP-адреса очень важно для понимания работы глобальных сетей, поэтому остановимся на нем более подробно.

IP-адрес

IP-адрес представляется 32-битным бинарным числом, которое часто записывают в виде 4 десятичных чисел, от 0 до 255 каждое. Например: `60.13.54.11, 130.154.201.1, 194.11.3.200`. Логически он состоит из двух частей – адреса машины (host) и адреса сети (network). Сетевая часть IP-адреса показывает, к какой сети принадлежит адресат, а хост-часть (host) идентифицирует сетевое устройство в этой сети. Компьютеры с одинаковой сетевой частью находятся в одной локальной сети, а потому могут легко обмениваться данными. Если же у них различные `network-ID`, то, даже находясь в одном физическом сегменте, они обычно не могут "увидеть" друг друга.

Так как IP-адрес состоит из 4-х октетов (так называют эти числа, поскольку $256=2^8$), один, два или три первых октета могут использоваться для определения сетевого адреса, остальные задают host-части. Для удобства выделения адресов пользователям (ведь, как правило, организации требуется их сразу несколько), было введено 5 классов адресов. Их обозначают латинскими буквами от A до E. В

открытых сетях используются первые три из них.

В таблице 16.2 дано примерное разбиение IP-адресов на сетевую (N) и машинную (H) части в зависимости от класса сети.

Таблица 16.2. Примерное разбиение IP-адресов.

	1 октет	2 октет	3 октет	4 октет
Класс A	N	H	H	H
Класс B	N	N	H	H
Класс C	N	N	N	H

Класс A

В классе A для идентификации сети, к которой принадлежит адрес, используется первый октет, причем, первый бит всегда равен 0. Остальные октеты задают адрес хоста. Таким образом, адрес сети класса A может быть в диапазоне `0-126`. 127-й адрес зарезервирован для специального использования – все адреса, начинающиеся со `127`, считаются локальными для сетевого адаптера, то есть всегда отправитель сам является и получателем. Остальные свободные три октета применяются для задания адреса хоста в данной сети. Это означает, что в одной сети может быть использовано до 2^{24} адресов (из них два крайних, то есть `0` и $2^{24}-1$, зарезервированы, они рассматриваются ниже). Стало быть, в каждой из `127` сетей класса A можно адресовать `16,777,214` машин.

Диапазон адресов `10.0.0.0-10.255.255.255` в публичных сетях не используется. Эти адреса специально зарезервированы для применения в локальных сетях и глобальными маршрутизаторами не обрабатываются.

Класс B

В сети класса B первые два октета (причем, первый бит всегда равен 1, второй – 0) используются для определения сети, последние два октета –

для определения адреса хоста. Диапазон адресов сети класса B лежит в пределах от `128.0.x.x` до `191.255.x.x`, что дает `16,384` таких сетей. В каждой из них может быть не более $65,534=2^{16}-2$ адресов (два крайних адреса исключаются).

В этой подсети зарезервированными для локального использования являются следующие адреса: `172.16.0.0-172.31.0.0`.

Класс C

Диапазон сети класса C определяется первыми тремя октетами (первые биты всегда `110`). И в десятичном виде эта сеть может начинаться со `192` по `223`. Для определения адреса хоста используется последний октет. Таким образом, в каждой из `2,097,152` сетей класса C может быть задействовано 2^8 (без двух крайних) или `254` адреса.

Зарезервированными для локального использования являются следующие адреса: `192.168.0.0-192.168.255.255`.

Class D

Этот класс используется для особых задач (multicast-группы). Диапазон адресов – `224.0.0.0-239.255.255.255`.

Class E

Этот класс адресов зарезервирован для применения в будущем. Диапазон адресов – `240.0.0.0-247.255.255.255`.

Два адреса в каждой подсети являются зарезервированными. IP-адрес, в котором вся хост-часть состоит из бинарных нулей, используется для обозначения адреса самой сети. Например, сеть класса A может иметь адрес `112.0.0.0`, а компьютер, подключенный к ней, – адрес `112.2.3.4`. Адрес сети используется роутерами для задания маршрута.

Второй зарезервированный адрес – броадкаст-адрес (broadcast). Этот адрес применяется, когда источник хочет послать данные всем устройствам в локальной сети. Для этого хост-часть заполняется бинарными единицами. Например, для рассмотренной сети `112.0.0.0` это будет адрес `112.255.255.255`, а для сети класса `B` `171.10.0.0` броадкаст-адрес будет выглядеть как `171.10.255.255`. Данные, посланные по адресу `171.10.255.255`, будут получены всеми устройствами в сети `171.10.0.0`.

Подсети. Маска подсети

Введение классов сетей во многом упростило задачу распределения адресов по организациям. Но не всегда имеет смысл использовать, например, целую сеть класса C, если в ней реально будет размещено лишь 10 компьютеров. Для более рационального использования сетей организуют подсети.

Адрес подсети включает в себя сетевую часть от сети класса A, B или C и так называемое поле подсети (subnet field). Для этого значения выделяют дополнительные биты, принадлежащие хост-части (то есть для адреса подсети может быть использовано до 3-х октетов из сети класса A, до 2-х из сети класса B, и 1 для C, соответственно). Таких битов может быть минимально один (таким образом одна сеть разделяется на две подсети), а максимально столько, чтобы для хост-части оставалось еще два бита (иначе подсеть будет состоять лишь из двух служебных адресов - адреса подсети и броадкаст-адреса). Для сетей класса A это дает от 1 до 22 битов, для B – от 1 до 14 битов, для C – от 1 до 6.

Разбиение на подсети уменьшает также размеры броадкаст-доменов, что необходимо, иначе для сети класса A броадкаст-запрос может рассылаться на 16 миллионов компьютеров. И если каждый из них пошлет хотя бы по одному такому запросу, нагрузка на сеть будет чрезмерно большой. Если же компьютер находится в выделенной подсети, то в соседние сети и подсети роутер пересылать броадкаст-запрос не будет, вследствие чего экономится полоса пропускания физических каналов связи.

Для определения длины адреса подсети используется специальное понятие – маска подсети. Это число определяет, какая часть IP-адреса применяется для задания сетевой и подсетевой части. Маску подсети можно определить следующим образом. Запишем IP-адрес в бинарном виде. Все разряды, относящиеся к network- и subnet-части, заменим на 1, все значения, относящиеся к host-части,– на 0. В результате получим маску подсети.

Например, маска подсети для целой сети класса A будет выглядеть как `255.0.0.0`, для сети класса B: `255.255.0.0`, для сети класса C – `255.255.255.0`. Для разделения на подсети, как было сказано выше, нужно некоторые биты хост-части выделить для поля подсети. Например, маска `255.255.255.192` определяет подсеть класса C, для которой количество хостов будет равно `62`.

Протоколы ARP, RARP

Когда формируется пакет для отправления, на сетевом уровне закладывается IP-адрес получателя. Однако для передачи на нижестоящий канальный уровень также нужно знать MAC-адрес. Для определения соответствия IP-адресу MAC-адреса существует ARP-протокол (Address Resolution Protocol, протокол определения адресов). Он работает следующим образом.

Формируется специальный широковещательный (broadcast) запрос. Он рассматривался выше, его особенность в том, что его получают все устройства, подключенные к этой локальной сети. В таком запросе MAC-адрес получателя состоит из одних бинарных единиц, а в поле IP-адреса записывается именно тот адрес, для которого требуется определить MAC-адрес. Когда некий компьютер получает такой запрос, он сравнивает указанный IP-адрес со своим. Если они различаются, сообщение игнорируется. Если они равны, то формируется ответ, в котором по всем правилам указаны IP- и MAC-адреса отправителя, то есть искомой машины.

Для того, чтобы не нагружать широковещательными запросами сеть, ARP-протокол поддерживает специальную ARP-таблицу, которая находится в оперативной памяти и хранит соответствие между IP- и

MAC-адресами. После удачного определения MAC-адреса какого-нибудь узла сети делается соответствующая запись в таблицу, чтобы при следующей отсылке пакета не пришлось снова рассылать broadcast -запросы. Спустя некоторое время запись удаляется. Это позволяет автоматически подстраиваться под изменения в сети, ведь у какого-то узла могли изменить MAC- или IP-адрес. Если отправитель не находит IP-адрес получателя в ARP-таблице, то снова формируется и отправляется ARP-запрос.

Протокол RARP (Reverse ARP – обратный ARP) действует наоборот – он известному MAC-адресу сопоставляет IP-адрес. Это необходимо, например, для работы таких протоколов, как BOOTP (Bootstrap Protocol, протокол автоматической настройки) и DHCP (Dynamic Host Configuration Protocol, протокол динамической конфигурации хостов). Их назначение – облегчить задачи системному администратору. Они позволяют не вводить IP-адрес в каждый компьютер локальной сети, а назначают их сами в автоматическом режиме. При загрузке очередной машины 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 - ???". Если в сети есть DHCP-сервер, он отвечает на RARP-запрос, указывая IP-адрес для этого компьютера (особенно это эффективно при большом количестве компьютеров).

Оба эти протокола работают в рамках лишь локальной сети, поскольку все пакеты, направляемые в другие сети, обрабатываются и маршрутизируются роутером, поэтому знать MAC-адрес не требуется (отправитель указывает MAC-адрес самого роутера).

Transport layer (layer 4)

Рассмотрим протокол 4-го транспортного уровня модели OSI. Семейство TCP/IP включает в себя два таких протокола – TCP и UDP. TCP (Transmission Control Protocol, протокол управления передачей) обеспечивает виртуальные соединения между пользовательскими приложениями и гарантирует точную доставку данных. UDP (User Datagram Protocol, протокол передачи датаграмм пользователя) служит для быстрого обмена специальными сообщениям (датаграммами) без гарантии доставки.

Основные характеристики TCP и UDP показаны в табл. 16.3.

Таблица 16.3. Основные характеристики TCP и UDP.

TCP	UDP
Для работы устанавливает соединение	Работает без соединений
Гарантированная доставка данных	Гарантий доставки нет
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

TCP

TCP/IP представляет собой комбинацию двух уровней, TCP и IP. IP – протокол третьего уровня – обеспечивает наилучшую, но не гарантированную доставку данных через сеть. TCP – протокол четвертого уровня – позволяет эту гарантию обеспечить. Поэтому совместно они могут предоставить большее количество сервисов.

Работа по TCP-протоколу начинается с установления соединения. Два компьютера (один из них инициатор соединения, второй – принимающий) обмениваются специальными пакетами в три этапа. Условно их можно назвать "запрос", "подтверждение" и "подтверждение на подтверждение". Такая процедура необходима, чтобы при получении какого-нибудь старого пакета (например, делается вторая попытка установить соединение) не возникало никаких неоднозначностей.

После успешного установления соединения участники могут начать обмениваться данными. Рассмотрим пример HTTP-сервера, который отправляет HTML-страницу клиенту. Текст может быть слишком длинным, чтобы уместиться в один пакет, поэтому первая задача уровня TCP – разбить сообщение на несколько пакетов, а на стороне

отправителя – собрать их опять в единое целое. Поскольку очередность пакетов несомненно важна, каждый получает порядковый номер.

Следующая задача протокола – обеспечить гарантированную доставку. Делается это с помощью следующей процедуры. Отправитель посылает пакет с номером `n` и начинает ждать. Получатель в случае успешного прихода пакета `n`, отправляет подтверждение о получении ("квитанцию"), в котором также указывает номер `n`. Если отправитель в течение определеннного времени (тайм-аута) не получает подтверждения, он считает пакет `n` потерянным и отсылает его еще раз.

Разумеется, отправителю неэффективно просто ждать, пока получатель получит и обработает каждый пакет по одному. Поэтому процедура усложняется, вводится специальное понятие – "окно" (window). Окно имеет некоторый размер, предположим, 10. Это означает, что передача начинается с отсылки 10 первых пакетов. Получатель может принять их не в том порядке, в каком они были отосланы. Тем не менее, на каждый успешно полученный пакет отсылается подтверждение с указанием номера такого пакета. Если отправитель отослал уже все 10 пакетов, но квитанция о получении пакета 1 так и не пришла, то передача приостанавливается, а по прошествии тайм-аута первый пакет считается потерянным и пересылается еще раз. Если же подтверждения приходят регулярно, то отправляются новые пакеты, но не более 10 единовременно.

Этот алгоритм носит название "скользящего окна". Если представить, что все пакеты выстроены в ряд, то окно "скользит" по нему, определяя, какие пакеты готовы для отсылки. Такой подход обеспечивает гарантированную доставку при максимально возможной скорости передачи данных. Разумеется, протокол TCP работает не столь быстро, ведь часть пропускной способности сети тратится на пересылку квитанций и повторов потерянных пакетов. Однако, большое количество информации требуется доставлять именно таким образом. Понятно, что части, например, текста должны составляться в строгом порядке и без пропусков. Были разработаны специальные механизмы, автоматически регулирующие величины таких характеристик, как тайм-аут и размер окна, для достижения оптимальной производительности.

UDP

В отличие от TCP, UDP не гарантирует доставку данных. UDP не устанавливает виртуального соединения, источник просто шлет специальные сообщения (в UDP они называются датаграммами) получателю. Если данные были доставлены некорректно, или вообще часть пакетов потерялась, UDP не позволяет их восстановить. Запрос на получение данных должен будет выполняться заново.

Казалось бы, недостатков у такого протокола довольно много, что ставит под сомнение его эффективность. Но есть сервисы, где UDP незаменим. Например, при передаче потокового аудио-видео если бы мы использовали TCP, то при потере одного пакета у нас была бы приостановлена трансляция для его повторной передачи. При использовании UDP один потерянный пакет – всего лишь незначительное (наверняка, вообще незаметное пользователю) ухудшение изображения/звука, при этом передача данных не прерывается. Также при использовании UDP не обязательно устанавливать виртуальное соединение, не нужно отсылать квитанции – все это ускоряет работу протокола.

Порты

Как было рассмотрено, для протокола IP достаточно знать IP-адрес, чтобы обработать сообщение. Оба протокола транспортного уровня, TCP и UDP, дополнительно используют порты (port) для взаимодействия с вышестоящими уровнями. Порт описывается числом от `0` до `65535` и позволяет операционной системе распределять пакеты, приходящие на транспортный уровень, между различными прикладными программами. Предположим, пользователь одновременно скачивает файл с FTP-сервера и работает с удаленным сервером базы данных. От обоих этих серверов пользовательская машина будет получать по сети пакеты и необходимо правильно передавать их соответствующим приложениям (FTP-клиенту и БД-клиенту).

Часть портов зарезервирована под стандартные приложения.

Например, для FTP зарезервирован порт `21`, для telnet – `23`, для HTTP – `80`. Далее приведен список распределения портов:

- порты меньше 255 используются для публичных сервисов;
- порты из диапазона 255-1023 назначаются компаниями-разрабочиками для приложений;
- номера свыше 1023 – не регулируемые.

Таким образом, говоря об установленном TCP-соединении, имеют ввиду 4 числа: IP-адрес и порт одной стороны и те же параметры второй стороны. Например, если пользователь со своей машины `194.11.22.33` обратился через браузер к web-серверу `213.180.194.129`, то это означает, что установлено соединение `194.11.22.33:10123-213.180.194.129:80` (номер `10123` выбран произвольно – используется любой незанятый порт).

Используется также термин "сокет" (socket), под которым подразумевается пара "IP-адрес:порт" – адресная "точка" для сетевых обращений.

Session layer (layer 5)

После транспортного уровня пакет поступает на уровень сессий. Когда приложения, запущенные на различных машинах, начинают взаимодействовать через сеть, то между ними происходит множество мини-"переговоров", обменов, диалогов, из которых и состоит сетевая сессия.

Session layer координирует установление и завершение соединений и сессий между приложениями.

Presentation layer (layer 6)

Этот уровень отвечает за представление данных, пересылаемых по сети. Он обеспечивает следующую функциональность: data formatting (presentation, то есть преобразование данных в понятный получателю формат), data encryption (шифрование), data compression (сжатие данных). Presentation layer выполняет одну или все эти функции во время

передачи сообщений между 7-м и 5-м уровнями. Приведем пример использования уровня представлений.

Предположим, хост-получатель использует EBCDIC (кодировка, применяемая на крупных IBM-серверах для передачи символов в виде чисел), а хост- отправитель – ASCII (традиционная кодировка для персональных компьютеров). Presentation layer будет обеспечивать преобразование пересылаемых между этими машинами данных.

Для обеспечения безопасности при передаче частной информации через публичные сети необходимо шифрование данных. Один из распространенных протоколов, используемых для этой цели, – SSL (Secured Sockets Layer) – может быть отнесен к уровню представлений.

Если канал связи обладает низкой пропускной способностью, целесообразно применять компрессию данных. Представительский уровень, используя математические алгоритмы, позволяет уменьшить объем передаваемых данных. Что касается высокоскоростных каналов, то для них использование компрессии может потребовать значительных вычислительных мощностей при больших объемах трафика.

Application layer (layer 7)

Последний уровень – уровень приложений, на котором определяются взаимодействующие стороны, учитывается авторизация пользователя, определяется качество обслуживания (quality of service) и, собственно, обеспечивается выполнение прикладных задач, таких, как обмен файлами, электронными письмами и т.д. Уровень приложения – это не само приложение, хотя зачастую программы выполняют некоторые функции Application layer.

Уже упоминались многие протоколы этого уровня: FTP, HTTP, telnet. Этот список легко продолжить, например, протоколы POP3 и SMTP для получения и отправки электронных писем, или протоколы DNS (Domain Name System, служба имен доменов), обеспечивающие преобразование числовых IP- адресов в текстовые доменные имена и обратно. Хотя Internet с технической точки зрения построен на основе IP-адресации, текстовые имена понятнее и легче запоминаются, а потому гораздо более распространены среди обычных пользователей.

Рассмотрим принцип работы DNS более подробно. Все привыкли обращаться к, например, web-серверам по доменному имени. С другой стороны для установления соединения требуется IP-адрес. Так, при обращении к серверу `www.ru` устанавливается TCP-соединение с хостом `194.87.0.50`.

Поскольку в сети огромное количество серверов, DNS-имена являются иерархическими, иначе с ними было бы очень затруднительно работать. Иерархические части имени записываются через точку. Первый уровень указывается последним. Первоначально существовало 7 трехбуквенных доменов первого уровня:

- `com` – commercial (коммерческие организации);
- `org` – non-profit (некоммерческие организации);
- `net` – network service (организация работы сети);
- `edu` – educational (образование, зачастую – американские университеты);
- `int` – international (международные организации);
- `gov` – government (правительство, организации американского правительства);
- `mil` – military (военные, американские военные организации).

Кроме того, для каждой страны был заведен двухбуквенный домен, например, `ru` - Россия, `su` – СССР, `us` – США, `fr` – Франция и т.д. В последнее время вводятся новые доменные имена верхнего уровня, такие, как `biz` и `info`.

В каждом домене первого уровня может быть множество доменов второго уровня. Так, существует множество сайтов в домене `ru`, или `com`. У домена второго уровня может быть множество доменов третьего уровня и т.д.

Как же определить, какому IP-адресу соответствует доменное имя сервера, к которому обращается пользователь? Для этого существует аналогичная иерархическая система DNS-серверов, каждый из которых отвечает за свой домен. В сетевых настройках компьютера указывается адрес локального DNS-сервера. При запросе к нему сервер сначала проверяет список имен, за которые отвечает он сам, и кеш. Если

искомое имя ему неизвестно, он делает запрос вышестоящему DNS-серверу. Например, при обращении к `intuit.ru` будет сделан запрос к DNS-серверу, отвечающему за домен `ru`.

В свою очередь, сервер `intuit.ru` знает про все имена в своей зоне `intuit.ru`, либо, в случае обращения к домену следующего уровня (например, `node1.host1.intuit.ru`), знает адрес другого сервера (`host1.intuit.ru`), который за него отвечает, и на этот сервер перенаправляет запрос.

Таким образом можно установить IP-адрес для любого зарегистрированного доменного имени.

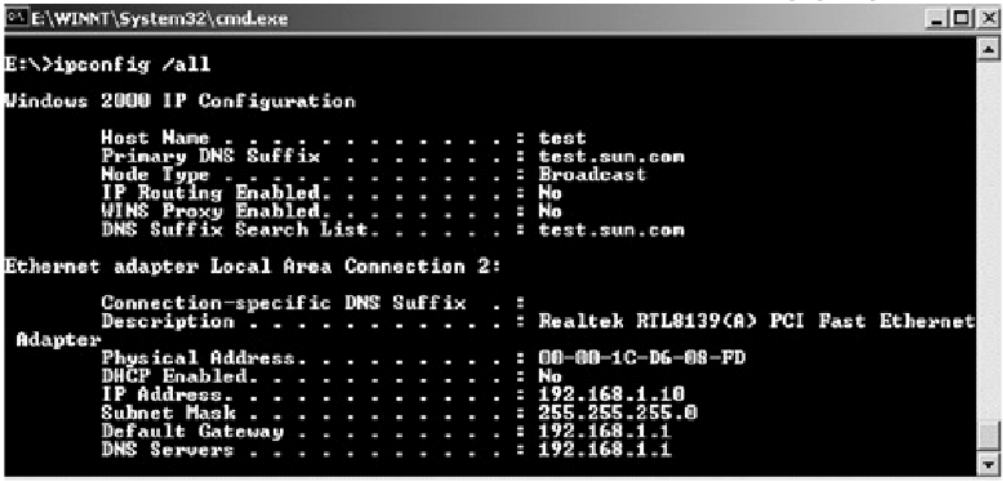
Утилиты для работы с сетью

Рассмотрим основные программы, позволяющие читать и изменять сетевые параметры, диагностировать и выявлять ошибки при работе сети.

В различных ОС существуют свои наборы утилит. Сравним их для двух систем, например, Microsoft Windows NT и Sun Solaris. Какими бы разными ни были эти ОС, в каждой из них реализована модель OSI. Естественно, программная и аппаратная реализация стека этой модели у них различается, но взаимодействие всех уровней осуществляется по установленному стандарту.

IPCONFIG (IFCONFIG)

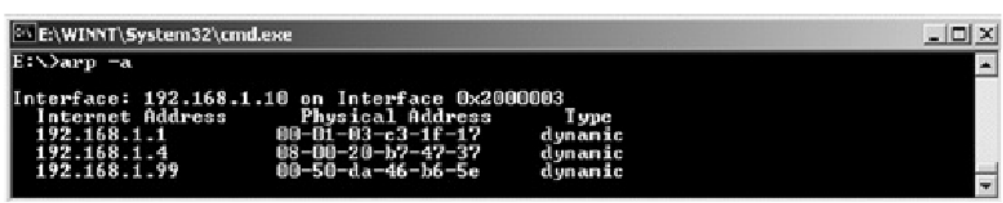
Начнем с утилиты, которая позволяет просматривать, проверять и изменять сетевые настройки. Обычно эти настройки включают в себя информацию 3-го (сетевого) уровня – IP-адрес, маску подсети и т.д. Для работы с ними в ОС Windows можно использовать команду `ipconfig`. Она выдает информацию об IP- адресе, маске подсети (netmask), роутере по умолчанию (default gateway). Задав дополнительный параметр `-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.

ARP

Как уже было сказано ранее, в оперативной памяти компьютера находится ARP-таблица. В ней содержатся MAC-адрес удаленной машины и соответствующий ему IP-адрес. Для просмотра этой таблицы используется команда `arp`. Например, `arp -a` выводит все известные MAC- адреса.



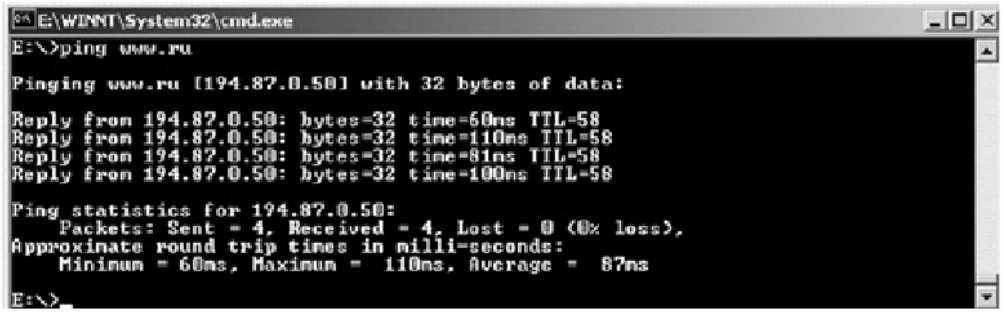
Существует два типа записей в ARP-таблице – статический и динамический. Статическая запись вносится вручную и существует до тех пор, пока вручную же не будет удалена, или компьютер (маршрутизатор) не будет перезагружен.

Динамическая запись появляется при попытке отправить сообщение на IP- адрес, для которого неизвестен MAC-адрес. В этом случае формируется ARP-запрос, который позволяет этот адрес определить, после чего соответствующая динамическая запись добавляется в ARP-таблицу. Храниться там она будет не постоянно. После определенного времени она будет автоматически удалена, если к данному IP-адресу не было обращений. Задержка на получение MAC-адреса составляет порядка нескольких миллисекунд, так что для пользователя это будет практически незаметно, зато появляется возможность отследить изменения в конфигурации сети (в соответствии IP- и MAC-адресов).

Ping

Для выявления различных неполадок в сети существует несколько утилит, которые позволяют определить, на каком уровне модели OSI произошел сбой, или указаны неверные настройки 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 сеть работает корректно, в результате выводится время ожидания прихода ответа от удаленного компьютера и время жизни пакета (TTL, time to live, количество "хопов", после которого пакет был бы отброшен; этот параметр показывает, сколько оставалось допустимых переходов у пакета-ответа).



Протокол ICMP находится на стыке двух уровней – сетевого и транспортного. Основной принцип действия этого протокола – формирование ICMP эхо-запроса (echo-request) и эхо-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 системы, маршрутизацию на машине источника, работоспособность и корректную маршрутизацию на роутерах между источником и получателем, а также работоспособность и правильность маршрутизации получателя.

Таким образом, 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 web-страницу, то проблема, по всей видимости, не в первых трех уровнях модели OSI.

Из примера видно, что по умолчанию размер посылаемого пакета - 32 байта, далее выводится время задержки ответа и TTL. В этом примере показано успешное выполнение команды `ping`. В случаях, когда запросы echo request посылаются, но echo reply не возвращаются, выводится сообщение об истечении времени ожидания ответа.

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, уменьшает значение TTL на единицу. С помощью TTL происходит предотвращение зацикливания пакета в "петлях" маршрутизации, иначе "заблудившиеся" пакеты окончательно перегрузили бы сеть. Однако, при выходе маршрутизатора или линии связи из строя требуется несколько дополнительных переходов для понимания, что данный маршрут потерян и его необходимо обойти. Чтобы предотвратить потерю датаграммы, поле TTL устанавливается на максимальную величину.

Когда маршрутизатор получает IP-датаграмму с TTL, равным 0 или 1, 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 легко 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. Так продолжается до тех пор, пока датаграмма не достигнет хоста назначения. Утилита `traceroute` может посылать в

качестве такой датаграммы UDP-сообщение с номером порта, который заведомо не будет обработан приложением (порт выше 30000), поэтому хост назначения ответит "порт недоступен" (port unreachable). При получении такого ответа делается вывод, что удаленный хост работает корректно. В противном случае максимального значения TTL (по умолчанию 30) не хватило для того, чтобы его достигнуть.

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 7.8
  5 Moscow-BNS045-ATM4-0-3.free.net (147.45.20.37) 12.228 ms 7.041 m
  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
```

Пример 16.1.

Первая строка содержит имя и IP-адрес хоста назначения, максимальное значение TTL 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 отправляются три датаграммы. Для каждой возвращенной датаграммы определяется 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 пересылку между, например, 5 и 6 узлом, необходимо вычесть из значения времени TTL 6 время TTL 5.

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 она так же, как и в ОС Solaris (`tracert netserv1.chg.ru`). Принципиального различия между утилитами `tracert` и `traceroute` нет. Особенностью `traceroute` является наличие большего количества функций (например, можно указать, начиная с какого TTL выводить информацию).

В случае какой-либо неполадки выводится соответствующее сообщение. Например, при недоступности сети на маршрутизаторе выдается сообщение `!N` (net unreachable):

```
Moscow-BNS045-ATM4-0-3.free.net (147.45.20.37)
947.327 ms !N 996.548 ms !N 995.257 ms
```

Это означает, что `147.45.20.37` – маршрутизатор, начиная с которого, последующий маршрут недоступен. Если недоступен сам хост, то сообщение будет выглядеть так:

```
msu-mipt-atm0.mipt.ru (212.16.1.1)
5.536 ms !H 5.993 ms !H 10.431 ms !H.
```

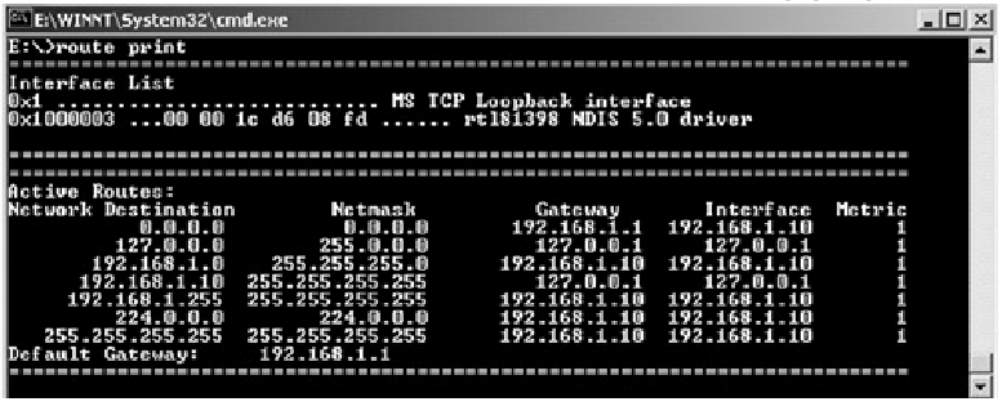
Ошибка `!P` означает недоступность протокола (protocol unreachable).

Route

Для просмотра и редактирования таблицы маршрутов используется утилита `route`. Типичный пример таблицы маршрутизации на персональном компьютере:

Для ОС Windows:

```
route print
```



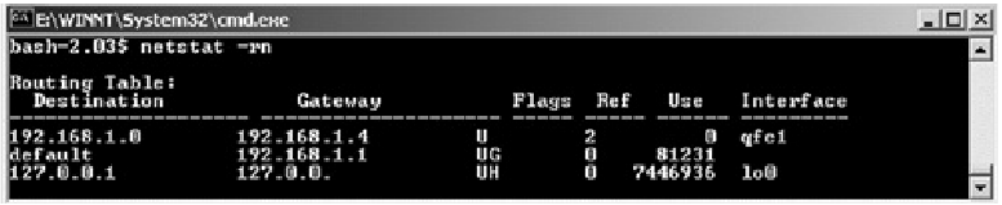
В таблице маршрутизации указывается сеть, маска сети, маршрутизатор, через который доступна эта сеть, интерфейс и метрика маршрута. Из приведенной таблицы видно, что маршрут по умолчанию доступен через маршрутизатор `192.168.1.1`. Сеть `192.168.1.0` с маской `255.255.255.0` является локальной сетью.

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 выполняется 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

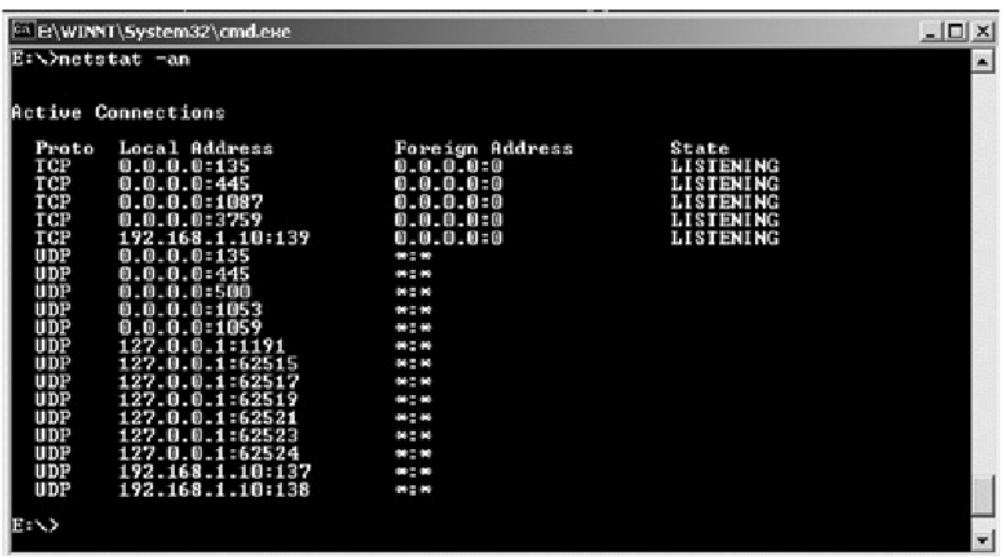
маршрутизации: `route del -net 157.6.`

Netstat

Утилита `netstat` позволяет определить, какие порты открыты и по каким портам происходит передача данных между узлами сети. Например, если запустить web-браузер и открыть для просмотра web-страницу, то, запустив `netstat`, можно увидеть следующую строку:

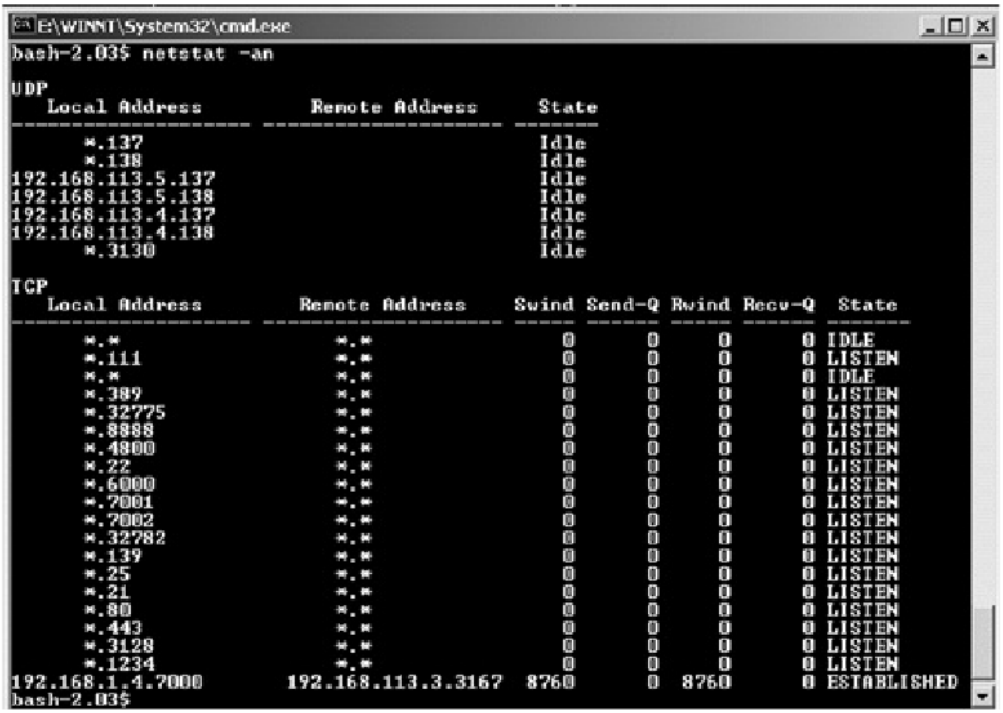
```
TCP mycomp:3687 www.ru:http ESTABLISHED
```

В приведенном примере первое значение – TCP – тип протокола (может быть TCP или UDP), далее следует имя локальной машины и локальный порт, `www.ru:http` – имя удаленного хоста и порта, к которому производится обращение (поскольку использовался порт по умолчанию для протокола HTTP, то отображается не его числовое значение 80, а имя протокола), `ESTABLISHED` – показывает, что TCP-соединение установлено.



В ОС Windows с помощью команды `netstat -an` можно получить список всех открытых портов (параметр `-n` не определяет DNS-имя, а выводит только IP-адрес). Из примера выше 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 применяется утилита `netstat`. Формат вывода практически одинаков.

Пакет java.net

Перейдем к рассмотрению средств Java для работы с сетью.

Классы, работающие с сетевыми протоколами, 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 мы видим, 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 – `getInputStream()` (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-серверы 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 возможность 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 ознакомиться с классом `InetAddress`, который является Internet-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 инициирует 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 использует 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 данного 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 подключения. 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 серверу не "зависать", если никто не пытается начать с ним работать. Тайм-аут 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 готовым 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) {
      }
    }
}
```

Пример 16.2.

Теперь объявим клиента. Эта программа будет запускать несколько потоков, каждый из 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());
      }
    }
```

Пример 16.3.

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 полученные данные (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;
    }
  }
```

Пример 16.4.

Заключение

В данном разделе были рассмотрены теоретические основы сети как одной большой взаимодействующей системы. Были описаны все уровни модели OSI и их функциональные назначения. Также были представлены основные утилиты, используемые для настройки и обнаружения неисправностей в сети. Затем были рассмотрены средства Java для работы с наиболее распространенными сетевыми протоколами. Приведен подробный пример и для более сложного случая – сервер, обслуживающий несколько клиентов одновременно.

Н.А. Вязовик

Разработка приложений в среде Eclipse

В практикуме рассказывается о работе в среде Eclipse, установке необходимых приложений, дается краткое описание интерфейса и демонстрируется процесс разработки простейшего приложения.

Видео

Разработка web-приложений в среде Eclipse

В практикуме рассмотрен пример разработки простейшего web-приложения, включая все тонкости установки и настройки дополнительного программного обеспечения для его работы.

Видео

Список литературы

1. David Bank, The Java Saga, интернет-журнал Wired, декабрь 1995 http://www.wired.com/wired/archive/3.12/java.saga.html

2. Буч Г., Объектно-ориентированное проектирование с примерами применения, М.: Конкорд, 1992

3. Гайсарян С.С., Объектно-ориентированные технологии проектирования прикладных программных систем, http://citforum.ru/programming/oop_rsis/

4. Леоненков А.В., Самоучитель UML, СПб: BHV-С.-Петербург

5. Олифер В.Г., Олифер Н.А., Компьютерные сети, СПб: Питер, 2000

6. Олифер В.Г., Олифер Н.А., Базовые технологии локальных сетей, http://www.citforum.ru/nets/protocols2/

7. Модель OSI, http://www.citforum.ru/nets/switche/osi.shtml 1998

8. Bill Brodgen, Java 2 Exam Cram (Руководство по подготовке к сертификации по Java 2), Second Edition, Coriolis, 2001

9. Страуструп Б., Язык программирования C++, М.: Радио и связь, 1991

10. Booch G., Jacobson I., Rumbaugh J., The Unified Modeling Language, Technical Report, Rational Software Corporation, 1997

11. Robert V. Binder, Testing Object-Oriented Systems: A Status Report, American Programmer, April 1994

12. Официальный сайт Java, URL: http://java.sun.com

13. Официальное представительство Sun в России, URL: http://www.sun.ru

14. Журнал JavaWorld, большое количество полезной информации по Java как для новичков, так и для профессионалов, URL: http://www.javaworld.com

15. История появления Java, URL: http://java.sun.com/features/1998/05/birthday.html , Jon Byous, "Java Technology: An Early History", май 1998

16. Судебное дело Sun против Microsoft, URL: http://java.sun.com/lawsuit/

17. Русскоязычный сайт посвященный объектно-ориентированному анализу и проектированию, URL: http://ooad.asf.ru/

18. Официальный сайт компании Cisco Systems, Inc., URL: http://www.cisco.com/

Содержание

Титульная страница	2
Выходные данные	3
Лекция 1. Что такое Java? История создания	4
Лекция 2. Основы объектно-ориентированного программирования	46
Лекция 3. Лексика языка	74
Лекция 4. Типы данных	106
Лекция 5. Имена. Пакеты	144
Лекция 6. Объявление классов	174
Лекция 7. Преобразование типов	214
Лекция 8. Объектная модель в Java	241
Лекция 9. Массивы	271
Лекция 10. Операторы и структура кода. Исключения	295
Лекция 11. Пакет java.awt	336
Лекция 12. Потоки выполнения. Синхронизация	389
Лекция 13. Пакет java.lang	420
Лекция 14. Пакет java.util	456
Лекция 15. Пакет java.io	505
Лекция 16. Введение в сетевые протоколы	551
Лекция 17. Разработка приложений в среде Eclipse	601
Лекция 18. Разработка web-приложений в среде Eclipse	602
Список литературы	603