Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
Амурский государственный университет
(ФГБОУ ВО «АмГУ»)

Факультет математики и информатики
Кафедра информационной безопасности

JAVA-технологии

Сборник учебно-методических материалов

для направления подготовки 01.03.02 Прикладная математика и информатика

Благовещенск
Амурский государственный университет
2023

Печатается по решению
редакционно-издательского совета
факультета математики и информатики
Амурского государственного
университета

Составитель: Фомин Д.В.

Java-технологии: сборник учебно-методических материалов для направления подготовки 01.03.02 «Прикладная математика и информатика»/ сост. Д. В. Фомин, Амур. гос. ун-т. – Благовещенск: АмГУ, 2023. – 172 с.

© Амурский государственный университет, 2023

© Кафедра информационной безопасности, 2023

© Фомин Д.В., составление

КРАТКОЕ ИЗЛОЖЕНИЕ ЛЕКЦИОННОГО МАТЕРИАЛА

На языке программирования Java пишут программные продукты в следующих областях:

- приложения для Андроид – Java практически единственный язык для них;
- десктопные приложения;
- промышленные программы;
- банковские программы;
- научные программы;
- программы для работы с большими данными;
- веб-приложения, веб-сервера, сервера приложений;
- встроенные системы – от маленьких чипов до специальных компьютеров;
- корпоративный софт.

Чаще всего вы встретите Java в веб-разработке и в приложениях для Android, но и в остальных сферах она тоже очень популярна. Рассмотрим некоторые из этих областей более подробно.

Разработка Java началась в 1990 году, первая официальная версия – Java 1.0, – была выпущена только 21 января 1996 года. Вторая версия была выпущена 19 февраля 1997 года.

Java широко известна как объектно-ориентированный язык, легкий в изучении и позволяющий создавать программы, которые могут исполняться на любой платформе без каких-либо доработок (свойство кроссплатформенности). Также стоит отметить, 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.

Было выпущено пять основных версий языка, начиная с 1.0 в 1995 году и заканчивая 1.4 в феврале 2002 года (21 сентября 2017 вышла 9я версия). Следующая версия 1.5 выпущена в 2004 году. Все версии и документацию к ним всегда можно было бесплатно получить на официальном web-сайте Java. 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.

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 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.

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 впоследствии, как известно, самым распространенным браузером стал 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". На момент создания курса 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 мо-

нополиста, 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 С;
- 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 момент создания курса 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 про-

граммистов, 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.

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.

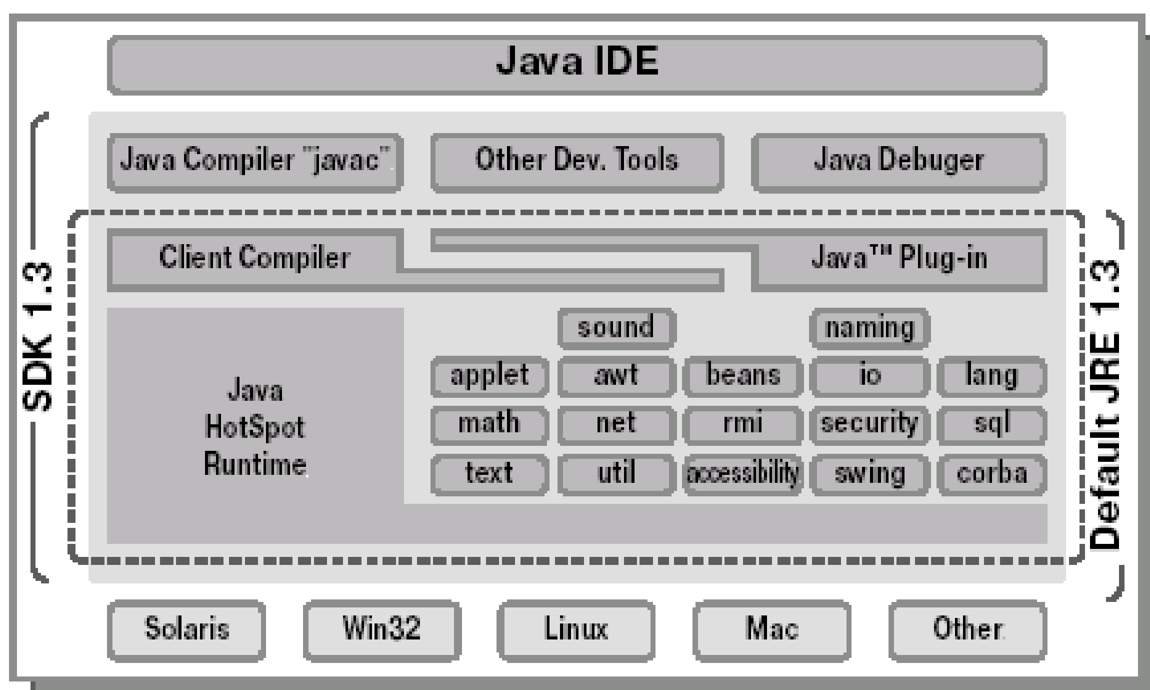
Новая платформа может работать в двух режимах - клиентском и серверном. Режимы различались настройками и другими оптимизирующими алгоритмами. По умолчанию работа идет в клиентском режиме.

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).

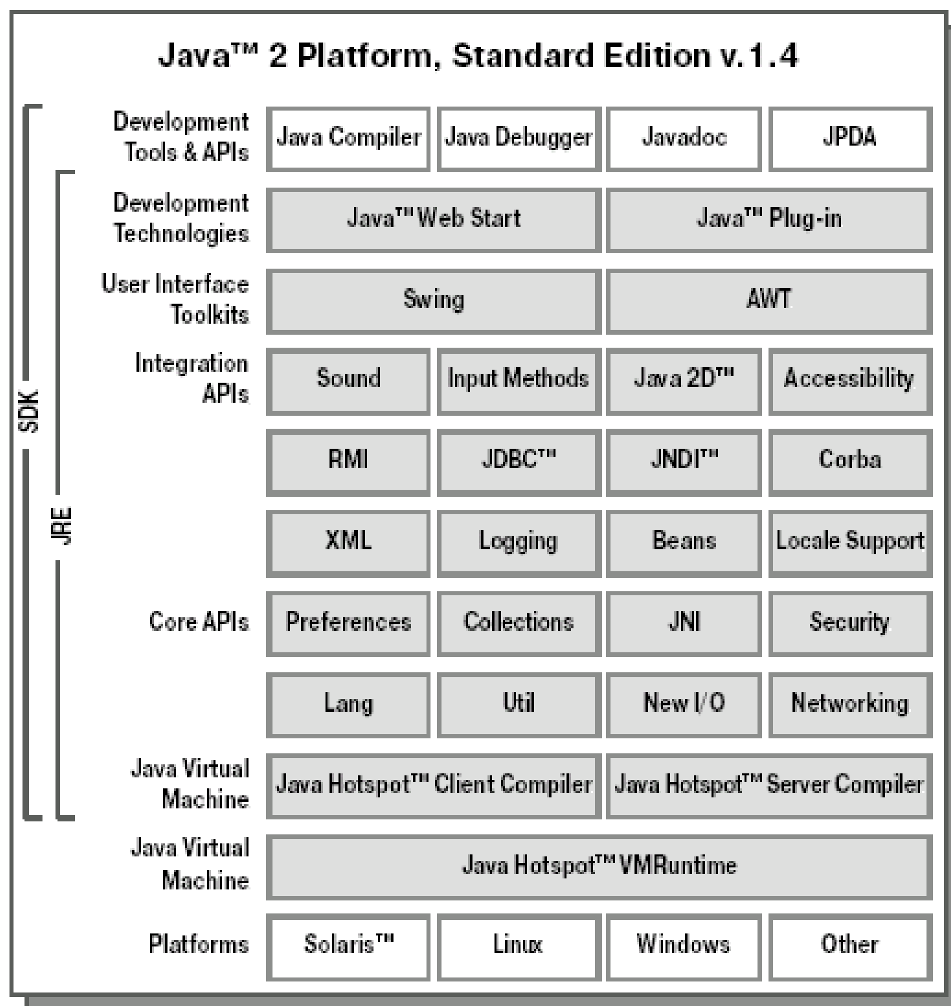
Наконец, последняя на данный момент версия J2SE 1.4 вышла в феврале 2002 года. Она была разработана для более полной поддержки web-сервисов (web services). Поэтому основные изменения коснулись работы с XML (Extensible Markup Language). Другое революционное добавление - выражение assert, позволяющее в 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.

Лексика и синтаксис

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.

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.
```

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.

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 С, 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 10000000, 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.

Типы данных

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 инициализировать до первого использования 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 ни-

каких 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.

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
111111111111111111L

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 секунд 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
println(y1);
println(y2);
```

```
println(y3);
println(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.

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 одного родителя. Таким образом, мы можем проследить цепочку наследования от 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.).

Приведение типов

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().getRoom()));
```

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

Имена и иерархия элементов

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

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

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 поля 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.

Классы

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 = new Human(); // 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 = new Human();
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 = new Human();
```

```
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 = new Human();
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.

Объявление классов

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 А 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 пример 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 при-

водить 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)
```

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);
```

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.

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.

Массивы

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.

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:

null ;

значения точно такого же типа, что и тип переменной;

все значения типа массив, основанный на типе, приводимом к базовому типу исходного массива.

Все эти утверждения непосредственно следуют из рассмотренных выше особенностей приведения типов массивов.

Еще раз напомним про исключительный класс Object. Переменные такого типа могут ссылаться на любые объекты, порожденные как от классов, так и от массивов.

Сведем все эти утверждения в таблицу.

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.

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).

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 С, 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).

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 path
  specified)
 at java.io.FileInputStream.open(Native Method)
 at java.io.FileInputStream.<init>(FileInputStream.java:64)
 at experiment.Test.doFileInput(Test.java:38)
 at experiment.Test.main(Test.java:21)
Exception hash code 6129586
```

Несложно заметить, что, хотя последовательность вызовов одна и та же, в вызываемом и вызывающем методах обрабатываются разные объекты исключений.

Многопоточность

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 "ОК"), занятости ОС (из-за нехватки физической оперативной памяти пришлось воспользоваться виртуальной), значений приоритетов потоков и так далее.

В Java нет никаких средств распознавания или предотвращения ситуаций deadlock. Также нет способа перед вызовом синхронизированного метода узнать, заблокирован ли уже объект другим потоком. Программист сам должен строить работу программы таким образом, чтобы неразрешимые блокировки не возникали. Например, в рассмотренном примере достаточно было организовать блокировки объектов в одном порядке (всегда сначала первый, затем второй) – и программа 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.

- 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().

Методы 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.

ЛИТЕРАТУРА

1. Соколова, В. В. Вычислительная техника и информационные технологии. Разработка мобильных приложений : учебное пособие для вузов / В. В. Соколова. – Москва : Издательство Юрайт, 2022. – 175 с. – (Высшее образование). – ISBN 978-5-9916-6525-4. – Текст : электронный // Образовательная платформа Юрайт [сайт]. – URL: https://urait.ru/bcode/490305 (дата обращения: 16.03.2022).

2. Мухаметзянов, Р. Р. Основы программирования на Java : учебное пособие / Р. Р. Мухаметзянов. – Набережные Челны : Набережночелнинский государственный педагогический университет, 2017. – 114 c. – Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. – URL: https://www.iprbookshop.ru/66812.html (дата обращения: 10.06.2021). – Режим доступа: для авторизир. пользователей. - DOI: https://doi.org/10.23682/66812

3. Гуськова, О. И. Объектно ориентированное программирование в Java : учебное пособие / О. И. Гуськова. – Москва : Московский педагогический государственный университет, 2018. – 240 c. – ISBN 978-5-4263-0648-6. – Текст : электронный // Электронно-библиотечная система IPR BOOKS : [сайт]. – URL: https://www.iprbookshop.ru/97750.html (дата обращения: 10.06.2021). – Режим доступа: для авторизир. пользователей.

4. Вязовик Н. Программирование на Java [Электронный ресурс]. – URL: https://intuit.ru/studies/courses/16/16/info (дата обращения: 15.05.2023).

ОГЛАВЛЕНИЕ

КРАТКОЕ ИЗЛОЖЕНИЕ ЛЕКЦИОННОГО МАТЕРИАЛА .... /3

История создания Java...4
Лексика и синтаксис 25
Типы данных 36

Переменные 36

Имена и иерархия элементов 69
Классы 84
Массивы 121
Управление ходом программы 131

Управление циклами 136
Операторы break и continue..139
Именованные блоки 140
Ошибки при работе программы. Исключения (Exceptions) 143

Многопоточность..152

Базовые классы для работы с потоками 154
Синхронизация...160
Потоки исполнения..170

Литература...172
Оглавление 173