МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

«БРАТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра управления в технических системах

УТВЕРЖДАЮ:
Проректор по учебной работе
_____ Е.И. Луковникова
«29» мая 2020 г.

РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ

ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ

Б1.В.ДВ.04.02

НАПРАВЛЕНИЕ ПОДГОТОВКИ

27.03.04 Управление в технических системах

ПРОФИЛЬ ПОДГОТОВКИ

Управление и информатика в технических системах

Программа академического бакалавриата

Квалификация (степень) выпускника: бакалавр

Программа составлена в соответствии с федеральным государственным образовательным стандартом высшего образования по направлению подготовки 27.03.04 Управление в технических системах от 20.10.2015 г № 1171 и учебным планом ФГБОУ ВО «БрГУ» от 03.02.2020 г № 46 для очной формы обучения, заочно - ускоренной формы обучения для набора 2020 года

СОДЕРЖАНИЕ ПРОГРАММЫ **Стр.**

1. ПЕРЕЧЕНЬ ПЛАНИРУЕМЫХ РЕЗУЛЬТАТОВ ОБУЧЕНИЯ ПО ДИСЦИПЛИНЕ, СООТНЕСЕННЫХ С ПЛАНИРУЕМЫМИ РЕЗУЛЬТАТАМИ ОСВОЕНИЯ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

Вид деятельности выпускника

Дисциплина охватывает круг вопросов, относящихся к проектно-конструкторскому виду профессиональной деятельности выпускника в соответствии с компетенцией и видами деятельности, указанными в учебном плане.

Цель дисциплины

Изложение базовых принципов эффективной разработки и использования локальных, корпоративных и глобальных сетей для решения практических задач.

Задачи дисциплины

Подготовить обучающихся к самостоятельной работе по решению практических задач, связанных с созданием сетевых программных комплексов в операционной среде Windows.

Код компетенции	Содержание компетенций	Перечень планируемых результатов обучения по дисциплине
1	2	3
ОПК-9	способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности	**знать:** - основные требования информационной безопасности; **уметь:** - использовать навыки работы с компьютером; **владеть:** – методами информационных технологий.
ПК-6	способность производить расчёты и проетирование отдельных блоков и устройств систем автоматизации и управления и выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления	**знать:** - возможности использования современных языков и технологий программирования для разработки сетевых приложений; **уметь:** - выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления; **владеть:** - навыками расчётов и проетирования отдельных блоков и устройств систем автоматизации и управления.

2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

Дисциплина Б1.В.ДВ.04.02 «Программирование сетевых приложений» относится к вариативной части, дисциплина по выбору.

Дисциплина «Программирование сетевых приложений» базируется на знаниях, полученных при изучении таких учебных дисциплин, как Б1.В.07 Информатика, Б1.В.15 Структуры и алгоритмы обработки данных, Б1.В.18 Технологии программирования, Б1.Б.14 Программирование и основы алгоритмизации.

Дисциплина «Программирование сетевых приложений» представляет основу для изучения дисциплин: Б1.В.ДВ.11.01 Проектирование автоматизированных систем, Б2.В.03(П) Производственная (преддипломная) практика, Б3.Б.01 Защита выпускной квалификационной работы, включая подготовку к процедуре защиты и процедуру защиты.

3. РАСПРЕДЕЛЕНИЕ ОБЪЕМА ДИСЦИПЛИНЫ

3.1. Распределение объема дисциплины по формам обучения

Форма обучения	*Курс*	*Семестр*	*Трудоемкость дисциплины в часах*						*Курсовая работа (проект), контрольная работа, реферат, РГР*	*Вид промежуточной аттестации*
			Всего часов (с экз.)	*Аудиторных часов*	*Лекции*	*Лабораторные работы*	*Практические занятия*	*Самостоятельная работа*		
1	2	3	4	5	6	7	8	9	10	11
Очная	4	7	144	51	17	34	-	93	-	экзамен
Заочная	4	-	144	9	4	5	-	135	-	экзамен
Заочная (ускоренное обучение)	-	-	-	-	-	-	-	-	-	экзамен
Очно-заочная	-	-	-	-	-	-	-	-	-	-

3.2. Распределение объема дисциплины по видам учебных занятий и трудоемкости

Вид учебных занятий	*Трудоемкость (час.)*	*в т.ч. в интерактивной, активной, инновациионной формах, (час.)*	*Распределение по семестрам, час*
			7
1	2	3	4
I. Контактная работа обучающихся с преподавателем (всего)	51	10	51
Лекции (Лк)	17	6	17
Лабораторные работы (ЛР)	34	4	34
Групповые (индивидуальные) консультации	+	-	+
II.Самостоятельная работа обучающихся (СР)	66	-	66
Подготовка к лабораторным работам	57	-	57
Подготовка к экзамену в течении семестра	9	-	9
III. Промежуточная аттестация экзамен	27	-	27
Общая трудоемкость дисциплины …..... час.	144	-	144
зач. ед.	4	-	4

4. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

4.1. Распределение разделов дисциплины по видам учебных занятий

- для очной формы обучения:

№ раз-дела и темы	*Наименование раздела и тема дисциплины*	*Трудоем-кость, (час.)*	*Виды учебных занятий, включая самостоятельную работу обучающихся и трудоемкость; (час.)*		
			учебные занятия		*самостоятельная работа обучающихся*
			лекции	*лабораторные работы*	
1	2	3	4	5	6
1.	**Введение в программирование сетевых приложений на языке Java**	**15**	**2**	**4**	**9**
1.1.	История созданияJava	2,5	0,5	-	2
1.2.	Программирование на Java: достоинства, недостатки, особенности и состав	3,5	0,5	-	3
1.3.	Жизненный цикл программы на Java	9	1	4	4
2.	**Основы объектно-ориентированного программирования**	**14**	**2**	**4**	**8**
2.1.	Методология процедурно-ориентированного программирования	3,5	0,5	1	2
2.2.	Методология объектно-ориентированного программирования	3,5	0,5	1	2
2.3.	Характеристики объектов и классов	3,5	0,5	1	2
2.4.	Достоинства и недостатки объектно-ориентированного программирования	3,5	0,5	1	2
3.	**Обзор основных конструкций языка Java**	**16**	**2**	**6**	**8**
3.1.	Конструкции языка	1,4	0,4	-	1
3.2.	Виды лексем	2,9	0,4	1,5	1
3.3.	Дополнение. Работа с операторами	3,9	0,4	1,5	2
3.4.	Переменные и базовые типы данных	3,9	0,4	1,5	2
3.5.	Ссылочные типы	3,9	0,4	1,5	2
4.	**Имена и пакеты**	**15**	**2**	**4**	**9**
4.1.	Имена и элементы языка	4,5	0,5	2	2
4.2.	Пакеты	5,5	0,5	2	3
4.3.	Область видимости имён	2,5	0,5	-	2
4.4.	Соглашения по именованию	2,5	0,5	-	2
5.	**Классы и приведение типов**	**14**	**2**	**4**	**8**
5.1.	Модификаторы доступа	7	1	2	4
5.2.	Объявление классов	7	1	2	4
6.	**Объектная модель в Java**	**16**	**3**	**4**	**9**
6.1.	Статические элементы	3,5	0,5	1	2

6.2.	Ключевые слова this и super	4	1	1	2
6.3.	Ключевое слово abstract	3,5	0,5	1	2
6.4.	Интерфейсы	5	1	1	3
7.	**Массивы данных**	**13**	**2**	**4**	**7**
7.1.	Массивы как тип данных в Java	4,5	0,5	2	2
7.2.	Преобразование типов для массивов	3,5	0,5	1	2
7.3.	Клонирование	5	1	1	3
8.	**Операторы и структура кода. Исключения**	**14**	**2**	**4**	**8**
8.1.	Управление ходом программы	3,5	0,5	1	2
8.2.	Операторы разветвления	3,5	0,5	1	2
8.3.	Управление циклами	3,5	0,5	1	2
8.4.	Операторы прерывания и изменения хода выполнения программы	3,5	0,5	1	2
	ИТОГО	**117**	**17**	**34**	**66**

- для заочной формы обучения:

№ раз-дела и темы	***Наименование раздела и тема дисциплины***	***Трудоем-кость, (час.)***	***Виды учебных занятий, включая самостоятельную работу обучающихся и трудоемкость; (час.)***		
			учебные занятия		***самостоятельная работа обучающихся***
			лекции	***лабораторные работы***	
1	2	3	4	5	6
1.	**Введение в программирование сетевых приложений на языке Java**	**17**	**0,5**	**0,5**	**16**
2.	**Основы объектно-ориентированного программирования**	**16**	**0,5**	**0,5**	**15**
3.	**Обзор основных конструкций языка Java**	**17,5**	**0,5**	**1**	**16**
4.	**Имена и пакеты**	**17**	**0,5**	**0,5**	**16**
5.	**Классы и приведение типов**	**17**	**0,5**	**0,5**	**16**
6.	**Объектная модель в Java**	**17,5**	**0,5**	**1**	**16**
7.	**Массивы данных**	**17**	**0,5**	**0,5**	**16**
8.	**Операторы и структура кода. Исключения**	**16**	**0,5**	**0,5**	**15**
	ИТОГО	**135**	**4**	**5**	**126**

4.2. Содержание дисциплины, структурированное по разделам и темам

Раздел 1. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVA

Тема 1.1. 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.

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 - (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 необязательно 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.

Тема 1.2. Программирование на Java: достоинства, недостатки, особенности и состав

Java является объектно-ориентированным языком программирования, разработанным фирмой Sun Microsystems (сокращенно, Sun).

Основные достоинства языка:

- наибольшая среди всех языков программирования степень переносимости программ;
- мощные стандартные библиотеки;
- встроенная поддержка работы в сетях (как локальных, так и Internet/Intranet).

Основные недостатки:

- низкое, в сравнении с другими языками, быстродействие, повышенные требования к объему оперативной памяти (ОП);

- большой объем стандартных библиотек и технологий создает сложности в изучении языка;

- постоянное развитие языка вызывает наличие как устаревших, так и новых средств, имеющих одно и то же функциональное назначение.

Основные особенности:

- Java является полностью объектно-ориентированным языком. Например, C++ тоже является объектно-ориентированным, но в нем есть возможность писать программы не в объектно-ориентированном стиле, а в Java так нельзя;

- реализован с использованием интерпретации P-кода (байт-кода). Т.е. программа сначала транслируется в машинонезависимый P-код, а потом интерпретируется некоторой программой-интерпретатором (виртуальная Java-машина, JVM).

Версии Java

Есть несколько версий Java (1.0, 1.1, 2.0). Последняя версия Java 2.0. Это версии языка. Существуют также версии стандартных средств разработки Java-программ от Sun. Sun выпускает новые версии этих стандартных средств раз или два в год. Ранее они назывались JDK (Java Development Kit), в последних версиях название изменено на SDK (Software Development Kit) Официальное название текущей версии «Java (TM) SDK, Standart Edition, Version 1.3.0». Предыдущая версия имеет номер 1.2.2. Как и текущая версия она соответствует версии языка Java 2.0.

SDK - это базовая среда разработки программ на Java. Она является невизуальной и имеет бесплатную лицензию на использование. Есть и визуальные среды разработки (JBuilder, Semantec Cafe, VisualJ и др.).

Новые версии языка и версии SDK являются расширением прежних. Это сделано для того, чтобы не возникала необходимость переписывать существующие программы. Так программа, написанная на Java 1.0 или Java 1.1, будет работать и под SDK 1.3. Правда, при компиляции некоторых таких программ могут выдаваться предупреждающие сообщения типа «Deprecated ...». Это означает, что в программе использованы возможности (классы, методы), объявленные в новой версии устаревшими (deprecated).

Аплеты

Аплеты являются одной из важных особенностей Java. Java позволяет строить как обычные приложения так и аплеты.

Аплет - это небольшая программа, выполняемая браузером (например, на Internet Explorer или Netscape Navigator). Аплет встраивается специальным образом в web-страничку. При подкачке такой странички браузером он выполняется виртуальной Java-машиной самого браузера. Аплеты расширяют возможности формирования web-страниц.

Тема 1.3 Жизненный цикл программы на Java

Под жизненным циклом мы будем понимать процесс, необходимый для создания работающего приложения. Для программ на Java он отличается от жизненного цикла программ на других языках программирования. Типичная картина жизненного цикла для большинства языков программирования выглядит примерно так.

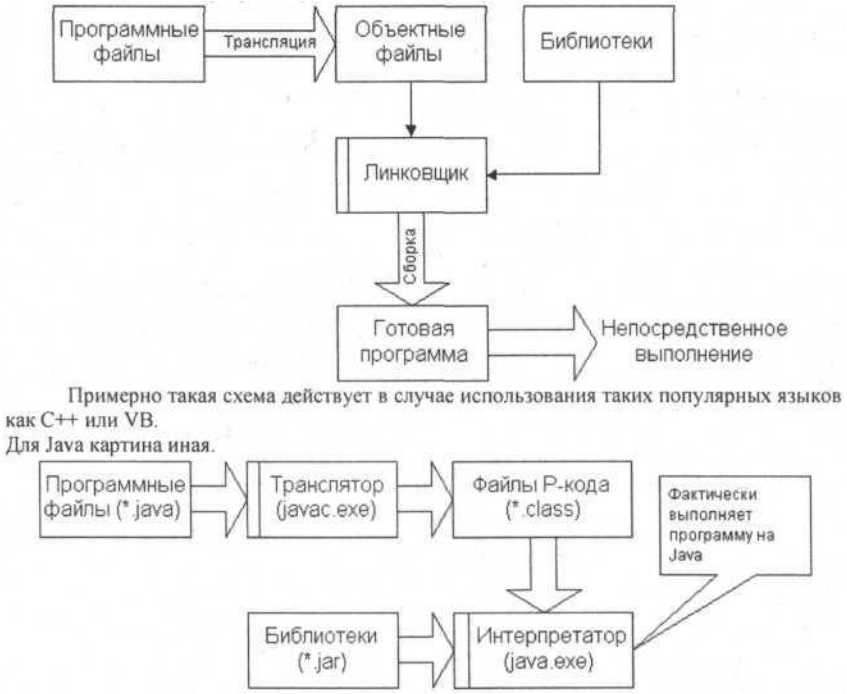


Рис. 1.1. Жизненный цикл языков программирования

Из рисунка видно, что исходная Java-программа должна быть в файле с расширением Java. Программа транслируется в байт-код компилятором javac.exe. Оттранслированная в байт-код программа имеет расширение class. Для запуска программы нужно вызвать интерпретатор java.exe, указав в параметрах вызова, какую программу ему следует выполнять. Кроме того, ему нужно указать, какие библиотеки нужно использовать при выполнении программы. Библиотеки размещены в файлах с расширением jar (в предыдущих версиях SDK использовались файлы *.zip и некоторые библиотеки все еще в таких файлах).

Структура пакета SDK

Пакеты Java SDK являются бесплатными. Текущая версия пакета может быть получена по адресу http://iava.sun.com/i2se/.

Загруженная с этого адреса инсталляция пакета SDK будет, скорее всего, в виде одного большого exe-файла - файла инсталляции. Для установки SDK этот файл нужно запустить. При этом будет запрошен каталог для установки. Лучше выбрать тот каталог, который предлагается в инсталляции по умолчанию.

В выбранном Вами при инсталляции каталоге будет построена структура подкаталогов пакета SDK. Кратко рассмотрим эту структуру и выясним содержимое и назначение всех подкаталогов.

bin\ - каталог инструментария разработчика

demo\ - каталог с примерами

include\ - для взаимодействия с программами на C, C++

include-old\ - аналогично, но предыдущая версия

jre\ - каталог инструментария пользователя (то, что поставляется конечному пользователю при установке (deployment) готового приложения)

jre\bin\ - Java-машина(ы) (JVM)

jre\lib\ - библиотеки Java для конечных пользователей + ряд настроечных файлов

lib\ - библиотеки Java для разработчиков

Каталог bin\ содержит ряд программ, которые необходимы в процессе разработки Java-приложений. В частности, в нем расположен транслятор Java - javac.exe.

Каталог demo\ содержит ряд примеров, демонстрирующих те или иные возможности Java. Это, как всегда, полезно и интересно.

Каталоги include\ и include-old\ мы сейчас рассматривать не будем. Они используются при совместном использовании Java и C++.

Каталог jre\ нужен для поставки конечному пользователю разработанного приложения. Он состоит из двух подкаталогов jre\bin\ и jre\lib\. Подкаталог jre\bin\, в частности, содержит

виртуальную машину Java (JVM) - java.exe, которая нужна для запуска готовых приложений. Этот же файл (java.exe) можно найти и в подкаталоге bin\. Вы можете вызывать JVM как из bin\, так и из jre\bin\. Лично я предпочитаю jre\bin\.

Подкаталог jre\lib\ содержит библиотеки, необходимые для выполнения готовых приложений, а также ряд настроечных файлов. Основная системная библиотека находится в файле rt.jar (очевидно, от Run Time). Ее необходимо подключать при трансляции и выполнении любой Java-программы. Вторая по важности библиотека il8n.jar. Если Вам необходимо, чтобы программа поддерживала русскоязычную кодировку, то придется подключить и эту библиотеку.

Раздел 2. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Лекция проводится в интерактивной форме с текущим контролем (2 час.)

Тема 2.1. Методология процедурно-ориентированного программирования

Появление первых электронных вычислительных машин, или компьютеров, ознаменовало новый этап в развитии техники вычислений. Казалось, достаточно разработать последовательность элементарных действий, каждое из которых можно преобразовать в понятные компьютеру инструкции, и любая вычислительная задача будет решена. Эта идея оказалась настолько жизнеспособной, что долгое время доминировала над всем процессом разработки программ. Появились специализированные языки программирования, созданные для разработки программ, предназначенных для решения вычислительных задач. Примерами таких языков могут служить FOCAL (FOrmula CALculator) и FORTRAN (FORmula TRANslator).

Основой такой методологии разработки программ являлась процедурная, или алгоритмическая, организация структуры программного кода. Это было настолько естественно для решения вычислительных задач, что целесообразность такого подхода ни у кого не вызывала сомнений. Исходным в данной методологии было понятие алгоритма. Алгоритм - это способ решения вычислительных и других задач, точно описывающий определенную последовательность действий, которые необходимо выполнить для достижения заданной цели. Примерами алгоритмов являются хорошо известные правила нахождения корней квадратного уравнения или системы линейных уравнений.

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

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

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

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

Со временем вычислительные задачи становились все сложнее, а значит, и решающие их программы увеличивались в размерах. Их разработка превратилась в серьезную проблему. Когда программа становится все больше, ее приходится разделять на все более мелкие фрагменты. Основой для такого разбиения как раз и стала процедурная декомпозиция, при которой отдельные части программы, или модули, представляли собой совокупность процедур для решения одной или нескольких задач. Одна из основных особенностей процедурного программирования заключается в том, что оно позволило создавать библиотеки подпрограмм (процедур), которые можно было бы использовать повторно в различных проектах или в рамках одного проекта. При процедурном подходе для визуального представления алгоритма выполнения программы применяется так называемая блок-схема. Соответствующая система графических обозначений была зафиксирована в ГОСТ 19.701-90. Пример блок-схемы изображен на рисунке (рис. 2.1).

Рис. 2.1. Пример блок-схемы.

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

Дальнейшее увеличение программных систем способствовало формированию новой точки зрения на процесс разработки программ и написания программных кодов, которая получила название методологии структурного программирования. Ее основой является

процедурная декомпозиция предметной области решаемой задачи и организация отдельных модулей в виде совокупности процедур. В рамках этой методологии получило развитие нисходящее проектирование программ, или проектирование «сверху вниз». Пик популярности идей структурного программирования приходится на конец 70-х - начало 80-х годов.

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

Тема 2.2. Методология объектно-ориентированного программирования

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

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

Позже появились персональные компьютеры, которые имели гораздо меньшую стоимость и были значительно компактнее. Это позволило широко использовать их в малом и среднем бизнесе. Основными задачами в этой области являются обработка данных и манипулирование ими, поэтому вычислительные и расчетно-алгоритмические задачи с появлением персональных компьютеров отошли на второй план. Как показала практика, традиционные методы процедурного программирования не способны справиться ни с нарастающей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Ею стало объектно-ориентированное программирование (ООП).

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

Последний термин нуждается в пояснении. Прагматика определяется целью разработки программной системы, например, обслуживание клиентов банка, управление работой аэропорта, обслуживание чемпионата мира по футболу и т.п. В формулировке цели участвуют предметы и понятия реального мира, имеющие отношение к создаваемой системе (см. рисунок 2.2). При объектно-ориентированном подходе эти предметы и понятия заменяются моделями, т.е. определенными формальными конструкциями.

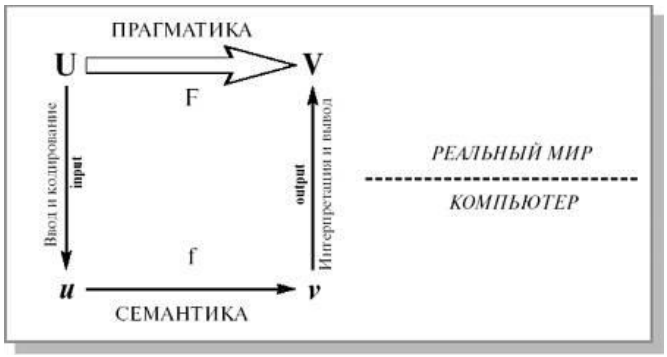


Рис. 2.2. Семантика (смысл программы с точки зрения выполняющего ее компьютера) и прагматика (смысл программы с точки зрения ее пользователей).

Модель содержит не все признаки и свойства представляемого ею предмета или понятия, а только те, которые существенны для разрабатываемой программной системы. Таким образом, модель «беднее», а следовательно, проще представляемого ею предмета или понятия.

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

Объектно-ориентированный подход обладает такими преимуществами, как:

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

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

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

Тема 2.3. Характеристики объектов и классов

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

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

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

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

Гради Буч дает следующее определение объекта:

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

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

Состояние

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

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

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

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

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

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

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

Поведение

Для каждого объекта существует определенный набор действий, которые с ним можно произвести. Например, возможные действия с некоторым файлом операционной системы ПК:

- создать;
- открыть;
- читать из файла;
- писать в файл;
- закрыть;
- удалить.

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

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

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

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

Уникальность

Уникальность - это то, что отличает объект от других объектов. Например, у вас может быть несколько одинаковых монет. Даже если абсолютно все их свойства (атрибуты)

одинаковы (год выпуска, номинал и т.д.) и при этом вы можете использовать их независимо друг от друга, они по-прежнему остаются разными монетами.

В машинном представлении под параметром уникальности объекта чаще всего понимается адрес размещения объекта в памяти.

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

Наиболее распространенной ошибкой является понимание уникальности как имени ссылки на объект. Это неверно, т.к. на один объект может указывать несколько ссылок, и ссылки могут менять свои значения (ссылаться на другие объекты).

Итак, уникальность (identity) - свойство объекта; то, что отличает его от других объектов.

Классы

Все монеты из предыдущего примера принадлежат одному и тому же классу объектов (именно с этим связана их одинаковость). Номинальная стоимость монеты, металл, из которого она изготовлена, форма - это атрибуты класса. Совокупность атрибутов и их значений характеризует объект. Наряду с термином «атрибут» часто используют термины «свойство» и «поле», которые в объектно-ориентированном программировании являются синонимами.

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

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

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

В соответствии с UML (Unified Modelling Language - унифицированный язык моделирования), класс имеет следующее графическое представление.

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

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

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

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

Инкапсуляция

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

Внутри объекта данные и методы могут обладать различной степенью открытости (или доступности). Степени доступности, принятые в языке Java, подробно будут рассмотрены в лекции 6. Они позволяют более тонко управлять свойством инкапсуляции.

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

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

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

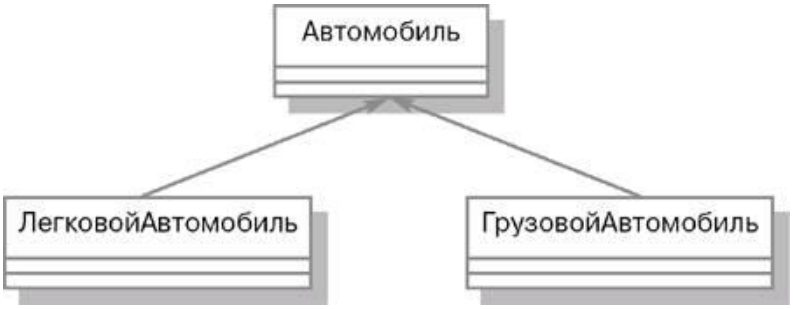
Во-вторых, не составит труда изменить способ хранения данных. Если информация станет храниться не в памяти, а в долговременном хранилище, таком как файловая система или база данных, потребуется изменить лишь ряд методов одного класса, а не вводить эту функциональность во все части системы.

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

Наследование

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

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



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

Использование наследования способствует уменьшению количества кода, созданного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода.

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

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

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

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

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

Полиморфизм

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

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

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

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

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

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

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

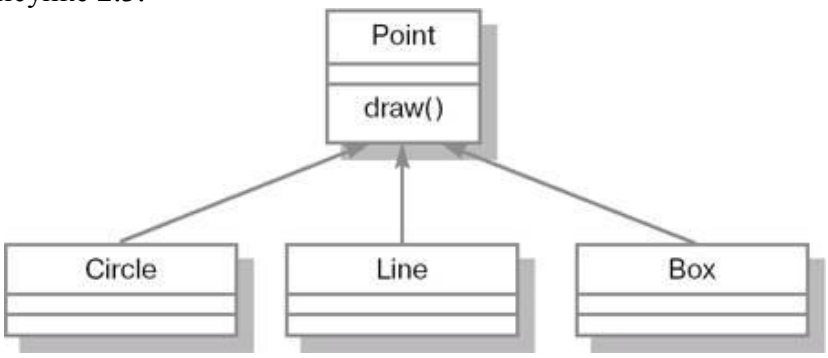


Рис. 2.3.

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

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

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

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

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

нарисуется круг, а не точка.

В заключение приведем формальное определение полиморфизма.

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

Тема 2.4. Достоинства и недостатки ООП

Достоинства ООП

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

Классы позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что позволяет абстрагироваться от деталей реализации.

Данные и операции над ними образуют определенную сущность, и они не разносятся по всей программе, как нередко бывает в случае процедурного программирования, а описываются вместе. Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.

Инкапсуляция позволяет привнести свойство модульности, что облегчает распараллеливание выполнения задачи между несколькими исполнителями и обновление версий отдельных компонентов.

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

Полиморфизм оказывается полезным преимущественно в следующих ситуациях.

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

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

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

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

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

Сокращается время на разработку, которое может быть отдано другим задачам.

Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.

Когда некий компонент используется сразу несколькими клиентами, улучшения, вносимые в его код, одновременно оказывают положительное влияние и на множество работающих с ним программ.

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

Недостатки ООП

Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим

каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод. Для абстрактных методов, которые пусты, в документации должно говориться о том, для каких целей предполагается использовать переопределяемый метод.

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

Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными, зато их намного больше. В коротких методах легче разобраться, но они неудобны тем, что код для обработки сообщения иногда «размазан» по многим маленьким методам.

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

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

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

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

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

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

3. Излишняя универсальность. Неэффективность также может означать, что в программе реализованы избыточные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, они становятся мертвым грузом. Это не влияет на время выполнения, но сказывается на размере кода.

Одно из возможных решений - строить базовый класс с минимальным числом методов, а затем уже реализовывать различные расширения этого класса, которые позволят нарастить функциональность. Другой подход - дать компоновщику возможность удалять лишние методы. Такие интеллектуальные компоновщики уже существуют для различных языков и операционных систем.

Но нельзя утверждать, что ООП неэффективно. Если классы используются лишь там, где это действительно необходимо, то потеря эффективности из-за повышенного расхода памяти и меньшей производительности незначительна. Кроме того, надежность программного обеспечения и быстрота его написания часто бывает важнее, чем производительность.

Заключение

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

Тема 3. ОБЗОР ОСНОВНЫХ КОНСТРУКЦИЙ ЯЗЫКА JAVA

Раздел 3.1. Конструкции языка

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

Как известно, Unicode представляет символы кодом из 2 байт, описывая, таким образом, 65535 символов. Это позволяет поддерживать практически все распространенные языки мира. Первые 128 символов совпадают с набором ASCII. Однако понятно, что требуется некоторое специальное обозначение, чтобы иметь возможность задавать в программе любой символ Unicode, ведь никакая клавиатура не позволяет вводить более 65 тысяч различных знаков. Эта конструкция представляет символ Unicode, используя только символы ASCII.

Например, если в программу нужно вставить знак с кодом 6917, необходимо его представить в шестнадцатеричном формате (1B05) и записать:

\u1B05,

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

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

Итак, используя простейшую кодировку ASCII, можно ввести произвольную последовательность символов Unicode. Далее будет показано, что Unicode используется не для всех лексем, а только для тех, для которых важна поддержка многих языков, а именно:

комментарии, идентификаторы, символьные и строковые литералы. Для записи остальных лексем вполне достаточно ASCII -символов.

Анализ программы

Компилятор, анализируя программу, сразу разделяет ее на:

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

Пробелы

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

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

можно записать и в таком виде:

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

В обоих случаях компилятор сгенерирует абсолютно одинаковый код. Единственное соображение, которым должен руководствоваться разработчик, – легкость чтения и дальнейшей поддержки 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 = 2Math.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.

Тема 3.2. Виды лексем

Ниже перечислены все виды лексем в Java:

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

Рассмотрим их по отдельности.

Идентификаторы

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

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

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

В этой лекции уже применялись следующие идентификаторы:

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

Также допустимыми являются идентификаторы:

Computer, COLOR_RED, _, aVeryLongNameOfTheMethod

Ключевые слова

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

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

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

Значение всех ключевых слов будет рассматриваться в следующих лекциях.

Литералы

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

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

Рассмотрим их по отдельности.

Целочисленные литералы

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

0
0X0
0x0

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

0xaBcDeF, 0xCafe, 0xDEC

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

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

Соответственно, допустимые литералы в восьмеричной записи должны быть от 00 до 017777777777 ($=2^{31}-1$), с унарным оператором - допустимо также -020000000000 ($= -2^{31}$).

Аналогично для шестнадцатеричного формата – от 0x0 до 0x7fffffff ($=2^{31}-1$), а также -0x80000000 ($= -2^{31}$).

Тип long имеет длину 64 бита, а значит, позволяет хранить значения от -2^{63} до $2^{63}-1$. Чтобы ввести такой литерал, необходимо в конце поставить латинскую букву L или l, тогда все значение будет трактоваться как long. Аналогично можно выписать максимальные допустимые значения для них:

9223372036854775807L
07777777777777777777L
0x7fffffffffffffffL
//наибольшие отрицательные значения:
-9223372036854775808L
-010000000000000000000L
-0x8000000000000000L

Другие примеры целочисленных литералов типа long:

0L, 123l, 0xC0B0L

Дробные литералы

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

Примеры:

3.14
2.
.5
7e10
3.1E-20

Таким образом, дробный литерал состоит из следующих составных частей:

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

Целая и дробная части записываются десятичными цифрами, а указатель типа (аналог указателя L или l для целочисленных литералов типа long) имеет два возможных значения – латинская ASCII -буква D (для типа double) или F (для типа float) в произвольном регистре. Они будут подробно рассмотрены ниже.

Необходимыми частями являются:

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

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

1.
.1
1e1
1f

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

// float-литералы:
1f, 3.14F, 0f, 1e+5F
// double-литералы:
0., 3.14d, 1e-4, 31.34E45D

В Java дробные числа 32-битного типа float и 64-битного типа double хранятся в памяти в бинарном виде в формате, стандартизированном спецификацией IEEE 754 (полное

название – IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE, New York)). В этой спецификации описаны не только конечные дробные величины, но и еще несколько особых значений, а именно:

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

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

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

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

Кроме того, для дробных величин становится важным еще одно предельное значение – минимальное положительное ненулевое значение:

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

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

```
// ошибка, выражение должно быть округлено до 0
0.000000000000000000000000000000000000000001f
// ошибки нет, компилятор сам округляет до 1
1.000000000000000000000000000000000000000001f
```

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

Тема 3.3. Дополнение. Работа с операторами

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

Операторы присваивания и сравнения

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

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

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

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

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

Условие «не равно» записывается как !=. Например:

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

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

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

Арифметические операции

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

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

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

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

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

Попытка получить остаток от деления на 0 приводит к ошибке.

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

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

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

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

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

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

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

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

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

Логические операторы

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

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

Например:

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

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

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

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

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

Тема 3.4. Переменные и базовые 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:

1. целочисленные:
    а) byte;
    б) short;
    в) int;
    г) long;
    д) char (также является целочисленным типом);
2. дробные
    е) float;
    ж) double;
    з) булевые.
3. 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 базовых характеристики:

- имя;
- тип;
- значение.

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

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

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

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

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

Резюмируем: объявление переменных и возможная инициализация при объявлении описываются следующим образом. Сначала указывается тип переменной, затем ее имя и, если необходимо, инициализатор, который может быть константой или выражением, вычисляемым во время компиляции или исполнения программы. В частности, можно пользоваться уже объявленными переменными. Далее можно поставить запятую и объявить новую переменную точно такого же типа.

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

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

Простейший пример объявления final-переменной:

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

Тема 3.5. Ссылочные типы

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

Объекты и правила работы с ними

Объект (object) – это экземпляр некоторого класса, или экземпляр массива. Массивы будут подробно рассматриваться в соответствующей лекции. Класс – это описание объектов

одинаковой структуры, и если в программе такой класс используется, то описание присутствует в единственном экземпляре. Объектов этого класса может не быть вовсе, а может быть создано сколь угодно много.

Объекты всегда создаются с использованием ключевого слова new, причем одно слово new порождает строго один объект (или вовсе ни одного, если происходит ошибка). После ключевого слова указывается имя класса, от которого мы собираемся породить объект. Создание объекта всегда происходит через вызов одного из конструкторов класса (их может быть несколько), поэтому в заключение ставятся скобки, в которых перечислены значения аргументов, передаваемых выбранному конструктору. В приведенных выше примерах, когда создавались объекты типа Point, выражение new Point (3,5) означало обращение к конструктору класса Point, у которого есть два аргумента типа int. Кстати, обязательное объявление такого конструктора в упрощенном объявлении класса отсутствовало. Объявление классов рассматривается в следующих лекциях, однако приведем правильное определение Point:

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

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

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

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

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

Рассмотрим пример:

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

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

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

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

```
Point p = null;
try
{
  // в следующей строке, используя лишь
  // текстовое имя класса Point, порождается
  // объект без применения ключевого слова new
p=(Point)Class.forName("Point").newInstance();
}
catch (Exception e)
{
  // обработка ошибок
   System.out.println(e);
}
```

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

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

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

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

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

Над ссылочными значениями можно производить следующие операции:

- обращение к полям и методам объекта;
- оператор instanceof (возвращает булево значение);
- операции сравнения == и != (возвращают булево значение);
- оператор приведения типов;
- оператор с условием ?:;
- оператор конкатенации со строкой +.

Обращение к полям и методам объекта можно назвать основной операцией над ссылочными величинами. Осуществляется она с помощью символа . (точка). Примеры ее применения будут приводиться.

Используя оператор instanceof, можно узнать, от какого класса произошел объект. Этот оператор имеет два аргумента. Слева указывается ссылка на объект, а справа – имя типа, на совместимость с которым проверяется объект. Например:

```
Parent p = new Child();
// проверяем переменную p типа Parent
// на совместимость с типом Child
print(p instanceof Child);
```

Результатом будет true. Таким образом, оператор instanceof опирается не на тип ссылки, а на свойства объекта, на который она ссылается. Но этот оператор возвращает истинное значение не только для того типа, от которого был порожден объект. Добавим к уже объявленным классам еще один:

```
// Объявляем новый класс и наследуем
// его от класса Child
class ChildOfChild extends Child { }
```

Теперь заведем переменную нового типа:

```
Parent p = new ChildOfChild();
print(p instanceof Child);
```

В первой строке объявляется переменная типа Parent, которая инициализируется ссылкой на объект, порожденный от ChildOfChild. Во второй строке оператор instanceof анализирует совместимость ссылки типа Parent с классом Child, причем задействованный объект не порожден ни от первого, ни от второго класса. Тем не менее, оператор вернет true, поскольку класс, от которого этот объект порожден, наследуется от Child.

Добавим еще один класс:

```
class Child2 extends Parent {}
```

И снова объявим переменную типа Parent:

```
Parent p=new Child();
print(p instanceof Child);
print(p instanceof Child2);
```

Переменная p имеет тип Parent, а значит, может ссылаться на объекты типа Child или Child2. Оператор instanceof помогает разобраться в ситуации:

```
true
false
```

Для ссылки, равной null, оператор instanceof всегда вернет значение false.

С изучением свойств объектной модели Java мы будем возвращаться к алгоритму работы оператора instanceof.

Раздел 4. ИМЕНА И ПАКЕТЫ

Тема 4.1. Имена и элементы языка

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:

1. пакеты;
2. классы;
3. интерфейсы;
4. элементы (member) ссылочных типов:
    а) поля;
    б) методы;
    в) внутренние классы и интерфейсы;
5. аргументы:
    г) методов;
    д) конструкторов;
    е) обработчиков ошибок;
6. 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.

Простые и составные имена. Элементы

Имена бывают простыми (simple), состоящими из одного идентификатора (они определяются во время объявления) и составными (qualified), состоящими из

последовательности идентификаторов, разделенных точкой. Для пояснения этих терминов необходимо рассмотреть еще одно понятие.

У пакетов и ссылочных типов (классов, интерфейсов, массивов) есть элементы (members). Доступ к элементам осуществляется с помощью выражения, состоящего из имен, например, пакета и класса, разделенных точкой.

Далее классы и интерфейсы будут называться объединяющим термином тип (type).

Элементами пакета являются содержащиеся в нем классы и интерфейсы, а также вложенные пакеты. Чтобы получить составное имя пакета, необходимо к полному имени пакета, в котором он располагается, добавить точку, а затем его собственное простое имя. Например, составное имя основного пакета языка Java – java.lang (то есть простое имя этого пакета lang, и он находится в объемлющем пакете java). Внутри него есть вложенный пакет, предназначенный для типов технологии reflection, которая упоминалась в предыдущих главах. Простое название пакета reflect, а значит, составное – java.lang.reflect.

Простое имя классов и интерфейсов дается при объявлении, например, Object, String, Point. Чтобы получить составное имя таких типов, надо к составному имени пакета, в котором находится тип, через точку добавить простое имя типа. Например, java.lang.Object, java.lang.reflect.Method или com.myfirm.MainClass. Смысл последнего выражения таков: сначала идет обращение к пакету com, затем к его элементу – вложенному пакету myfirm, а затем к элементу пакета myfirm – классу MainClass. Здесь com.myfirm – составное имя пакета, где лежит класс MainClass, а MainClass – простое имя. Составляем их и разделяем точкой – получается полное имя класса com.myfirm.MainClass.

Для ссылочных типов элементами являются поля и методы, а также внутренние типы (классы и интерфейсы). Элементы могут быть как непосредственно объявлены в классе, так и получены по наследству от родительских классов и интерфейсов, если таковые имеются. Простое имя элементов также дается при инициализации. Например, toString(), PI, InnerClass. Составное имя получается путем объединения простого или составного имени типа, или переменной объектного типа с именем элемента. Например, ref.toString(), java.lang.Math.PI, OuterClass.InnerClass. Другие обращения к элементам ссылочных типов уже неоднократно применялись в предыдущих главах.

Имена и идентификаторы

Теперь, когда мы рассмотрели простые и составные имена, уточним разницу между идентификатором (напомним, что это вид лексемы) и именем. Понятно, что простое имя состоит из одного идентификатора, а составное - из нескольких. Однако не всякий идентификатор входит в состав имени.

Во-первых, в выражении объявления (declaration) идентификатор еще не является именем. Другими словами, он становится именем после первого появления в коде в месте объявления.

Во-вторых, существует возможность обращаться к полям и методам объектного типа не через имя типа или объектной переменной, а через ссылку на объект, полученную в результате выполнения выражения. Пример такого вызова:

country.getCity().getStreet();

В данном примере getStreet является не именем, а идентификатором, так как соответствующий метод вызывается у объекта, полученного в результате вызова метода getCity(). Причем country.getCity как раз является составным именем метода.

Наконец, идентификаторы также используются для названий меток (label). Эта конструкция рассматривается позже, однако приведем пример, показывающий, что пространства имен и названий меток полностью разделены.

```
num:
  for (int num = 2; num <= 100; num++)
  {
       int n = (int)Math.sqrt(num)+1;
       while (--n != 1)
          {
              if (num%n==0)
                 {
```

```
                    continue num;
                }
            }
        System.out.print(num+" ");
    }
```

Результатом будут простые числа меньше 100:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Мы видим, что здесь применяются одноименные переменная и метка num, причем последняя используется для выхода из внутреннего цикла while на внешний for.

Очевидно, что удобнее использовать простое имя, а не составное, т.к. оно короче и его легче запомнить. Однако понятно, что если в системе есть очень много классов со множеством переменных, можно столкнуться с ситуацией, когда в разных классах есть одноименные переменные или методы. Для решения этой и других подобных проблем вводится новое понятие – область видимости.

Область видимости (введение)

Чтобы не заставлять программистов, совместно работающих над различными классами одной системы, координировать имена, которые они дают различным конструкциям языка, у каждого имени есть область видимости (scope). Если обращение, например, к полю, идет из части кода, попадающей в область видимости его имени, то можно пользоваться простым именем, если нет – необходимо применять составное.

Например:

```
class Point
{
  int x,y;
  int getX()
  {
    return x; // простое имя
  }
}
class Test
{
  void main()
  {
    Point p = new Point();
    p.x=3; // составное имя
  }
}
```

Видно, что к полю x изнутри класса можно обращаться по простому имени. К нему же из другого класса можно обратиться только по составному имени. Оно составляется из имени переменной, ссылающейся на объект, и имени поля.

Теперь необходимо рассмотреть области видимости для всех элементов языка. Однако прежде выясним, что такое пакеты, как и для чего они используются.

Тема 4.2. 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.

Тема 4.3. 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.

Тема 4.4. 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.

Заключение

В этой главе был рассмотрен механизм именования элементов языка. Для того, чтобы различные части большой системы не зависели друг от друга, вводится понятие «область видимости имени», вне которой необходимо использовать не простое, а составное имя. Затем были изучены элементы (members), которые могут быть у пакетов и ссылочных типов. Также рассматривалась связь терминов «идентификатор» (из темы «Лексика») и имя.

Затем были рассмотрены пакеты, которые используются в Java для создания физической и логической структуры классов, а также для более точного разграничения области видимости. Пакет содержит вложенные пакеты и типы (классы и интерфейсы). Вопрос о платформенной поддержке пакетов привел к рассмотрению модулей компиляции как текстовых файлов, поскольку именно в виде файлов и каталогов, как правило, хранятся и распространяются Java-приложения. Тогда же впервые был рассмотрен вопрос разграничения доступа, так как доступ к модулям компиляции определяется именно платформенной поддержкой, а точнее – операционной системой.

Модуль компиляции состоит из трех основных частей – объявление пакета, импорт-выражения и объявления верхнего уровня. Важную роль играет безымянный пакет, или пакет по умолчанию, хотя он и не рекомендуется для применения при создании больших систем. Были изучены детали применения двух видов импорт-выражений – импорт класса и импорт пакета. Наконец, было начато рассмотрение объявлений верхнего уровня (эта тема будет продолжена в главе, описывающей объявление классов). Пакеты, как и другие элементы языка, имеют определенные соглашения по именованию, призванные облегчить понимание кода и уменьшить возможность возникновения ошибок и двусмысленных ситуаций в программе.

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

Раздел 5. КЛАССЫ И ПРИВЕДЕНИЕ ТИПОВ

Лекция проводится в интерактивной форме с разбором конкретных ситуаций (2 час.)

Тема 5.1. 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 модификаторо вдоступа. Итак, пусть в классе Human есть поле age целочисленного типа, и чтобы все желающие могли пользоваться этим полем, оно объявляется public.

```
public class Human
{
  public int age;
}
```

Проходит время, и если в группу программистов, работающих над системой, входят десятки разработчиков, логично предположить, что все, или многие, из них начнут использовать это поле.

Может получиться так, что целочисленного типа данных будет уже недостаточно и захочется сменить тип поля на дробный. Однако если просто изменить int на double, вскоре все разработчики, которые пользовались классом Human и его полем age, обнаружат, что в их коде появились ошибки, потому что поле вдруг стало дробным, и в строках, подобных этим:

```
Human h = getHuman();
  // получаем ссылкуint i=h.age;   // ошибка!!
```

будет возникать ошибка из-за попытки провести неявным образом сужение примитивного типа.

Получается, что подобное изменение (в общем, небольшое и локальное) потребует модификации многих и многих классов. Поэтому внесение его окажется недопустимым, неоправданным с точки зрения количества усилий, которые необходимо затратить. То есть, объявив один раз поле или метод как public, можно оказаться в ситуации, когда малейшие изменения (имени, типа, характеристик, правил использования) в дальнейшем станут невозможны.

Напротив, если бы поле было объявлено как private, а для чтения и изменения его значения были бы введены дополнительные методы, ситуация поменялась бы в корне:

```
public class Human
{
  private int age;
  // метод, возвращающий значение age
  public int getAge()
  {
    return age;
  }   // метод, устанавливающий значение age
   public void setAge(int a)
  {
    age=a;
  }
}
```

В этом случае с данным классом могло бы работать множество программистов и могло быть создано большое количество классов, использующих тип Human, но модификатор

private дает гарантию, что никто напрямую этим полем не пользуется и изменение его типа было бы совсем несложной операцией, связанной с изменением только в одном классе.

Получение величины возраста выглядело бы следующим образом:

```
Human h = getHuman();
int i=h.getAge();
  // обращение через метод
```

Рассмотрим, как выглядит процесс смены типа поля age:

```
public class Human
{
  // поле получает новый тип double
  private /*int*/ double age;
  // старые методы работают с округлением
   // значения
  public int getAge()
  {
    return (int)Math.round(age);
  }
  public void setAge(int a)
  {
    age=a;
  }   // добавляются новые методы для работы
      // с типом double
  public double getExactAge()
  {
    return age;
  }
  public void setExactAge(double a)
  {
    age=a;
  }
}
```

Видно, что старые методы, которые, возможно, уже применяются во многих местах, остались без изменения. Точнее, остался без изменений их внешний формат, а внутренняя реализация усложнилась. Но такая перемена не потребует никаких модификаций остальных классов системы. Пример использования

```
Human h = getHuman();
int i=h.getAge();   // корректно
```

остается верным, переменная i получает корректное целое значение. Однако изменения вводились для того, чтобы можно было работать с дробными величинами. Для этого были добавлены новые методы и во всех местах, где требуется точное значение возраста, необходимо обращаться к ним:

```
Human h = getHuman();
double d=h.getExactAge();      // точное значение возраста
```

Итак, в класс была добавлена новая возможность, не потребовавшая никаких изменений кода.

За счет чего была достигнута такая гибкость? Необходимо выделить свойства объекта, которые потребуются будущим пользователям этого класса, и сделать их доступными (в данном случае, public).Те же элементы класса, что содержат детали внутренней реализации логики класса, желательно скрывать, чтобы не образовались нежелательные зависимости, которые могут сдерживать развитие системы.

Этот пример одновременно иллюстрирует и другое теоретическое правило написания объектов, а именно: в большинстве случаев доступ к полям лучше реализовывать через специальные методы (accessors) для чтения(getters) и записи (setters). То есть само поле рассматривается как деталь внутренней реализации. Действительно, если рассматривать внешний интерфейс объекта как целиком состоящий из допустимых действий, то

доступными элементами должны быть только методы, реализующие эти действия. Один из случаев, в котором такой подход приносит необходимую гибкость, уже рассмотрен.

Есть и другие соображения. Например, вернемся к вопросу о корректном использовании объекта и установке верных значений полей. Как следствие, правильное разграничение доступа позволяет ввести механизмы проверки входных значений:

```
public void setAge(int a)
{
  if (a>=0)
  {
    age=a;
  }
}
```

В этом примере поле age никогда не примет некорректное отрицательное значение. (Недостатком приведенного примера является то, что в случае неправильных входных данных они просто игнорируются, нет никаких сообщений, позволяющих узнать, что изменения поля возраста на самом деле не произошло; для полноценной реализации метода необходимо освоить работу с ошибками в Java.)

Бывают и более существенные изменения логики класса. Например, данные можно начать хранить не в полях класса, а в более надежном хранилище, например, файловой системе или базе данных. В этом случае методы-аксессоры опять изменят свою реализацию и начнут обращаться к persistent storage (постоянное хранилище, например, БД) для чтения/записи значений. Если доступа к полям класса не было, а открытыми были только методы для работы с их значениями, то можно изменить код этих методов, а наружние типы, которые использовали данный класс, совершенно не изменятся, логика их работы останется той же.

Подведем итоги. Функциональность класса необходимо разделять на открытый интерфейс, описывающий действия, которые будут использовать внешние типы, и на внутреннюю реализацию, которая применяется только внутри самого класса. Внешний интерфейс в дальнейшем модифицировать невозможно, или очень сложно, для больших систем, поэтому его требуется продумывать особенно тщательно. Детали внутренней реализации могут быть изменены на любом этапе, если они не меняют логику работы всего класса. Благодаря такому подходу реализуется одна из базовых характеристик объектной модели – инкапсуляция, и обеспечивается важное преимущество технологии ООП – модульность.

Таким образом, модификаторы доступа вводятся не для защиты типа от внешнего пользователя, а, напротив, для защиты, или избавления, пользователя от излишних зависимостей от деталей внутренней реализации. Что же касается неправильного применения класса, то его создателям нужно стремиться к тому, чтобы класс был прост в применении, тогда таких проблем не возникнет, ведь программист не станет намеренно писать код, который порождает ошибки в его программе.

Конечно, такое разбиение на внешний интерфейс и внутреннюю реализацию не всегда очевидно, часто условно. Для облегчения задачи технических дизайнеров классов в Java введено не два (public и private), а четыре уровня доступа. Рассмотрим их и весь механизм разграничения доступа в Java более подробно.

Разграничение доступа в Java

Уровень доступа элемента языка является статическим свойством, задается на уровне кода и всегда проверяется во время компиляции. Попытка обратиться к закрытому элементу напрямую вызовет ошибку.

В Java модификаторы доступа указываются для:

- типов (классов и интерфейсов) объявления верхнего уровня;
- элементов ссылочных типов (полей, методов, внутренних типов);
- конструкторов классов.

Как следствие, массив также может быть недоступен в том случае, если недоступен тип, на основе которого он объявлен.

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

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

Первые два из них уже были рассмотрены. Последний уровень (доступ по умолчанию) упоминался в прошлой лекции – он допускает обращения из того же пакета, где объявлен и сам этот класс. По этой причине пакеты в Java являются не просто набором типов, а более структурированной единицей, так как типы внутри одного пакета могут больше взаимодействовать друг с другом, чем с типами из других пакетов.

Наконец, protected дает доступ наследникам класса. Понятно, что наследникам может потребоваться доступ к некоторым элементам родителя, с которыми не приходится иметь дело внешним классам.

Однако описанная структура не позволяет упорядочить модификаторы доступа так, чтобы каждый следующий строго расширял предыдущий. Модификатор protected может быть указан для наследника из другого пакета, а доступ по умолчанию допускает обращения из классов-ненаследников, если они находятся в том же пакете. По этой причине возможности protected были расширены таким образом, что он включает в себя доступ внутри пакета. Итак, модификаторы доступа упорядочиваются следующим образом (от менее открытых – к более открытым):

- private(none);
- default;
- protected;
- public.

Эта последовательность будет использована далее при изучении деталей наследования классов.

Теперь рассмотрим, какие модификаторы доступа возможны для различных элементов языка.

- пакеты доступны всегда, поэтому у них нет модификаторов доступа (можно сказать, что все они public, то есть любой существующий в системе пакет может использоваться из любой точки программы);
- типы (классы и интерфейсы) верхнего уровня объявления. При их объявлении существует всего две возможности: указать модификатор public или не указывать его. Если доступ к типу является public, то это означает, что он доступен из любой точки кода. Если же он не public, то уровень доступа назначается по умолчанию: тип доступен только внутри того пакета, где он объявлен;
- массив имеет тот же уровень доступа, что и тип, на основе которого он объявлен (естественно, все примитивные типы являются полностью доступными);
- элементы и конструкторы объектных типов. Обладают всеми четырьмя возможными значениями уровня доступа. Все элементы интерфейсов являются public.

Для типов объявления верхнего уровня нет необходимости во всех четырех уровнях доступа. Private-типы образовывали бы закрытую мини-программу, никто не мог бы их использовать. Типы, доступные только для наследников, также не были признаны полезными.

Разграничения доступа сказываются не только на обращении к элементам объектных типов или пакетов (через составное имя или прямое обращение), но также при вызове конструкторов, наследовании, приведении типов. Импортировать недоступные типы запрещается.

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

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

```
    return radius;
  }
}
```

Значение поля radius недоступно снаружи класса, однако открытый метод getRadius() корректно возвращает его.

Рассмотрим следующие два модуля компиляции:

```
package first;
// Некоторый класс Parent
public class Parent {}
```

```
package first;
// Класс Child наследуется от класса Parent,
// но имеет ограничение доступа по умолчанию
class Child extends Parent {}
public class Provider
{
  public Parent getValue()
  {
    return new Child();
  }
}
```

К методу getValue() класса Provider можно обратиться и из другого пакета, не только из пакета first, поскольку метод объявлен как public. Данный метод возвращает экземпляр класса Child, который недоступен из других пакетов. Однако следующий вызов является корректным:

```
package second;
import first.*;
public class Test
{
  public static void main(String s[])
  {
    Provider pr = new Provider();
    Parent p = pr.getValue();
    System.out.println(p.getClass().getName());
    // (Child)p - приведет к ошибке компиляции!
  }
}
```

Результатом будет:

```
first.Child
```

То есть на самом деле в классе Test работа идет с экземпляром недоступного класса Child, что возможно, поскольку обращение к нему делается через открытый класс Parent. Попытка же выполнить явное приведение вызовет ошибку. Да, тип объекта «угадан» верно, но доступ к закрытому типу всегда запрещен.

Следующий пример:

```
public class Point
{
  private int x, y;
  public boolean equals(Object o)
  {
    if (o instanceof Point)
      {
        Point p = (Point)o;
        return p.x==x && p.y==y;
      }
    return false;
  }
```

}

В этом примере объявляется класс Point с двумя полями, описывающими координаты точки. Обратите внимание, что поля полностью закрыты – private. Далее попытаемся переопределить стандартный метод equals() таким образом, чтобы для аргументов, являющихся экземплярами класса Point, или его наследников (логика работы оператора instanceof), в случае равенства координат возвращалось истинное значение. Обратите внимание на строку, где делается сравнение координат,– для этого приходится обращаться к private-полям другого объекта!

Тем не менее, такое действие корректно, поскольку private допускает обращения из любой точки класса, независимо от того, к какому именно объекту оно производится.

Другие примеры разграничения доступа в Java будут рассматриваться по ходу курса.

Тема 5.2. Объявление классов

Рассмотрим базовые возможности объявления классов.

Объявление класса состоит из заголовка и тела класса.

Заголовок класса

Вначале указываются модификаторы класса. Модификаторы доступа для класса уже обсуждались. Допустимым является public, либо его отсутствие – доступ по умолчанию.

Класс может быть объявлен как final. В этом случае не допускается создание наследников такого класса. На своей ветке наследования он является последним. Класс String и классы-обертки, например, представляют собой final -классы.

После списка модификаторов указывается ключевое слово class, а затем имя класса – корректный Java-идентификатор. Таким образом, кратчайшим объявлением класса может быть такой модуль компиляции:

```
class A {}
```

Фигурные скобки обозначают тело класса, но о нем позже.

Указанный идентификатор становится простым именем класса. Полное составное имя класса строится из полного составного имени пакета, в котором он объявлен (если это не безымянный пакет), и простого имени класса, разделенных точкой. Область видимости класса, где он может быть доступен по своему простому имени, – его пакет.

Далее заголовок может содержать ключевое слово extends, после которого должно быть указано имя (простое или составное) доступного не-final класса. В этом случае объявляемый класс наследуется от указанного класса. Если выражение extends не применяется, то класс наследуется напрямую от Object. Выражение extends Object допускается и игнорируется.

```
class Parent {}
   // = class Parent extends Object {}
final class LastChild extends Parent {}
  // class WrongChild extends LastChild {}
   // ошибка!!
```

Попытка расширить final-класс приведет к ошибке компиляции.

Если в объявлении класса A указано выражение extends B, то класс A называют прямым наследником класса B.

Класс A считается наследником класса B, если:

- A является прямым наследником B;
- существует класс C, который является наследником B, а A является наследником C (это правило применяется рекурсивно).

Таким образом можно проследить цепочки наследования на несколько уровней вверх.

Если компилятор обнаруживает, что класс является своим наследником, возникает ошибка компиляции:

```
// пример вызовет ошибку компиляции
class A extends B {}
class B extends C {}
class C extends A {}
```

// ошибка! Класс A стал своим наследником

Далее в заголовке может быть указано ключевое слово implements, за которым должно следовать перечисление через запятую имен (простых или составных, повторения запрещены) доступных интерфейсов:

```
public final class String implements
                Serializable, Comparable {}
```

В этом случае говорят, что класс реализует перечисленные интерфейсы. Как видно из примера, класс может реализовывать любое количество интерфейсов. Если выражение implements отсутствует, то класс действительно не реализует никаких интерфейсов, здесь значений по умолчанию нет.

Далее следует пара фигурных скобок, которые могут быть пустыми или содержать описание тела класса.

Тело класса

Тело класса может содержать объявление элементов (members) класса:

- полей;
- внутренних типов (классов и интерфейсов);

и остальных допустимых конструкций:

- конструкторов;
- инициализаторов
- статических инициализаторов.

Элементы класса имеют имена и передаются по наследству, не-элементы – нет. Для элементов простые имена указываются при объявлении, составные формируются из имени класса, или имени переменной объектного типа, и простого имени элемента. Областью видимости элементов является все объявление тела класса. Допускается применение любого из всех четырех модификаторов доступа. Напоминаем, что соглашения по именованию классов и их элементов обсуждались в прошлой лекции.

Неэлементы не обладают именами, а потому не могут быть вызваны явно. Их вызывает сама виртуальная машина. Например, конструктор вызывается при создании объекта. По той же причине не-элементы не обладают модификаторами доступа.

Элементами класса являются элементы, описанные в объявлении тела класса и переданные по наследству от класса-родителя (кроме Object – единственного класса, не имеющего родителя) и всех реализуемых интерфейсов при условии достаточного уровня доступа. Таким образом, если класс содержит элементы с доступом по умолчанию, то его наследники из разных пакетов будут обладать разным набором элементов. Классы из того же пакета могут пользоваться полным набором элементов, а из других пакетов – только protected и public. private-элементы по наследству не передаются.

Поля и методы могут иметь одинаковые имена, поскольку обращение к полям всегда записывается без скобок, а к методам – всегда со скобками.

Рассмотрим все эти конструкции более подробно.

Объявление полей

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

Поле может быть объявлено как final, это означает, что оно инициализируется один раз и больше не будет менять своего значения .Простейший способ работы с final-переменными - инициализация при объявлении:

```
final double PI=3.1415;
```

Также допускается инициализация final-полей в конце каждого конструктора класса.

Не обязательно использовать для инициализации константы компиляции, возможно обращение к различным функциям, например:

```
final long creationTime =
        System.currentTimeMillis();
```

Данное поле будет хранить время создания объекта. Существует еще два специальных модификатора - transient и volatile. Они будут рассмотрены в соответствующих лекциях.

После списка модификаторов указывается тип поля. Затем идет перечисление одного или нескольких имен полей с возможными инициализаторами:

```
int a;
int b=3, c=b+5, d;
Point p, p1=null, p2=new Point();
```

Повторяющиеся имена полей запрещены. Указанный идентификатор при объявлении становится простым именем поля. Составное имя формируется из имени класса или имени переменной объектного типа, и простого имени поля. Областью видимости поля является все объявление тела класса.

Запрещается использовать поле в инициализации других полей до его объявления.

```
int y=x;
int x=3;
```

Однако, в остальном поля можно объявлять и ниже их использования:

```
class Point
{
  int getX() {return x;}
  int y=getX();
  int x=3;
}
public static void main (String s[])
{
  Point p=new Point();
  System.out.println(p.x+", "+p.y);
}
```

Результатом будет:

```
3, 0
```

Данный пример корректен, но для понимания его результата необходимо вспомнить, что все поля класса имеют значение по умолчанию:

- для числовых полей примитивных типов – 0;
- для булевского типа – false;
- для ссылочных – null.

Таким образом, при инициализации переменной y был использован результат метода getX(), который вернул значение по умолчанию переменной x, то есть 0. Затем переменная x получила значение 3.

Объявление методов

Объявление метода состоит из заголовка и тела метода. Заголовок состоит из:

- модификаторов (доступа в том числе);
- типа возвращаемого значения или ключевого слова void;
- имени метода;
- списка аргументов в круглых скобках (аргументов может не быть);
- специального throws -выражения.

Заголовок начинается с перечисления модификаторов. Для методов доступен любой из трех возможных модификаторов доступа. Также допускается использование доступа по умолчанию.

Кроме того, существует модификатор final, который говорит о том, что такой метод нельзя переопределять в наследниках. Можно считать, что все методы final-класса, а также все private- методы любого класса, являются final.

Также поддерживается модификатор native. Метод, объявленный с таким модификатором, не имеет реализации на Java. Он должен быть написан на другом языке (C/C++, Fortran и т.д.) и добавлен в систему в виде загружаемой динамической библиотеки (например, DLL для Windows). Существует специальная спецификация JNI (Java Native Interface), описывающая правила создания и использования native-методов.

Такая возможность для Java необходима, поскольку многие компании имеют обширные программные библиотеки, написанные на более старых языках. Их было бы очень трудоемко и неэффективно переписывать на Java, поэтому необходима возможность подключать их в

таком виде, в каком они есть. Безусловно, при этом Java-приложения теряют целый ряд своих преимуществ, таких, как переносимость, безопасность и другие. Поэтому применять JNI следует только в случае крайней необходимости.

Эта спецификация накладывает требования на имена процедур во внешних библиотеках (она составляет их из имени пакета, класса и самого native-метода), а поскольку библиотеки менять, как правило, очень неудобно, часто пишут специальные библиотеки-«обертки», к которым обращаются Java-классы через JNI, а они сами обращаются к целевым модулям.

Наконец, существует еще один специальный модификатор synchronized, который будет рассмотрен в лекции, описывающей потоки выполнения.

После перечисления модификаторов указывается имя (простое или составное) типа возвращаемого значения; это может быть как примитивный, так и объектный тип. Если метод не возвращает никакого значения, указывается ключевое слово void.

Затем определяется имя метода. Указанный идентификатор при объявлении становится простым именем метода. Составное имя формируется из имени класса или имени переменной объектного типа и простого имени метода. Областью видимости метода является все объявление тела класса.

Аргументы метода перечисляются через запятую. Для каждого указывается сначала тип, затем имя параметра. В отличие от объявления переменной здесь запрещается указывать два имени для одного типа:

```
// void calc (double x, y); - ошибка!
void calc (double x, double y);
```

Если аргументы отсутствуют, указываются пустые круглые скобки. Одноименные параметры запрещены. Создание локальных переменных в методе, с именами, совпадающими с именами параметров, запрещено. Для каждого аргумента можно ввести ключевое слово final перед указанием его типа. В этом случае такой параметр не может менять своего значения в теле метода (то есть участвовать в операции присвоения в качестве левого операнда).

```
public void process(int x, final double y)
{
  x=x*x+Math.sqrt(x);
    // y=Math.sin(x); - так писать нельзя,
   // т.к. y - final!
}
```

О том, как происходит изменение значений аргументов метода, рассказано в конце этой лекции.

Важным понятием является сигнатура (signature) метода. Сигнатура определяется именем метода и его аргументами (количеством, типом, порядком следования). Если для полей запрещается совпадение имен, то для методов в классе запрещено создание двух методов с одинаковыми сигнатурами.

Например,

```
class Point
{   void get() {}
  void get(int x) {}
  void get(int x, double y) {}
  void get(double x, int y) {}
}
```

Такой класс объявлен корректно. Следующие пары методов в одном классе друг с другом несовместимы:

```
void get() {}
int get() {}
void get(int x) {}
void get(int y) {}
public int get() {}
private int get() {}
```

В первом случае методы отличаются типом возвращаемого значения, которое, однако, не входит в определение сигнатуры. Стало быть, это два метода с одинаковыми сигнатурами и они не могут одновременно появиться в объявлении тела класса. Можно составить пример, который создал бы неразрешимую проблему для компилятора, если бы был допустим:

```
// пример вызовет ошибку компиляции
class Test
{   int get()
  {
    return 5;
  }
  Point get()
  {
    return new Point(3,5);
  }
  void print(int x)
  {
    System.out.println("it's int! "+x);
  }
  void print(Point p)
  {
    System.out.println("it's Point! "+p.x+
                  ", "+p.y);
  }
  public static void main (String s[])
  {
    Test t = new Test();
    t.print(t.get());   // Двусмысленность!
  }
}
```

В классе определена запрещенная пара методов get() с одинаковыми сигнатурами и различными возвращаемыми значениями. Обратимся к выделенной строке в методе main, где возникает конфликтная ситуация, с которой компилятор не может справиться. Определены два метода print() (у них разные аргументы, а значит, и сигнатуры, то есть это допустимые методы), и чтобы разобраться, какой из них будет вызван, нужно знать точный тип возвращаемого значения метода get(), что невозможно.

На основе этого примера можно понять, как составлено понятие сигнатуры. Действительно, при вызове указывается имя метода и перечисляются его аргументы, причем компилятор всегда может определить их тип. Как раз эти понятия и составляют сигнатуру, и требование ее уникальности позволяет компилятору всегда однозначно определить, какой метод будет вызван.

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

Аналогично, третья пара различается лишь модификаторами доступа, что также недопустимо.

Наконец, завершает заголовок метода throws-выражение. Оно применяется для корректной работы с ошибками в Java и будет подробно рассмотрено в соответствующей лекции.

Пример объявления метода:

```
public final java.awt.Point
    createPositivePoint(int x, int y)
    throws IllegalArgumentException
  {
    return (x>0 && y>0) ?
         new Point(x, y) : null;
```

```
}
```

Далее, после заголовка метода следует тело метода. Оно может быть пустым и тогда записывается одним символом «точка с запятой». Native-методы всегда имеют только пустое тело, поскольку настоящая реализация написана на другом языке.

Обычные же методы имеют непустое тело, которое описывается в фигурных скобках, что показано в многочисленных примерах в этой и других лекциях. Если текущая реализация метода не выполняет никаких действий, тело все равно должно описываться парой пустых фигурных скобок:

```
public void empty() {}
```

Если в заголовке метода указан тип возвращаемого значения, а не void, то в теле метода обязательно должно встречаться return-выражение. При этом компилятор проводит анализ структуры метода, чтобы гарантировать, что при любых операторах ветвления возвращаемое значение будет сгенерировано. Например, следующий пример является некорректным:

```
// пример вызовет ошибку компиляции
public int get()
{
  if (condition)
  {
    return 5;
  }
}
```

Видно, что хотя тело метода содержит return-выражение, однако не при любом развитии событий возвращаемое значение будет сгенерировано. А вот такой пример является верным:

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

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

В методе без возвращаемого значения (указано void) также можно использовать выражение return без каких-либо аргументов. Его можно указать в любом месте метода и в этой точке выполнение метода будет завершено:

```
public void calculate(int x, int y)
{
  if (x<=0 || y<=0)
  {
    return;   // некорректные входные
              // значения, выход из метода
  }
  ...   // основные вычисления
}
```

Выражений return (с параметром или без для методов с/без возвращаемого значения) в теле одного метода может быть сколько угодно. Однако следует помнить, что множество точек выхода в одном методе может заметно усложнить понимание логики его работы.

Объявление конструкторов

Формат объявления конструкторов похож на упрощенное объявление методов. Также выделяют заголовок и тело конструктора. Заголовок состоит, во-первых, из модификаторов доступа (никакие другие модификаторы недопустимы). Во-вторых, указывается имя класса, которое можно расценивать двояко. Можно считать, что имя конструктора совпадает с

именем класса. А можно рассматривать конструктор как безымянный, а имя класса – как тип возвращаемого значения, ведь конструктор может породить только объект класса, в котором он объявлен. Это исключительно дело вкуса, так как на формате объявления никак не сказывается:

```
public class Human
{
  private int age;
  protected Human(int a)
  {
    age=a;
  }
  public Human(String name, Human mother,
            Human father)
  {
    age=0;
  }
}
```

Как видно из примеров, далее следует перечисление входных аргументов по тем же правилам, что и для методов. Завершает заголовок конструктора throws-выражение. 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).

Раздел 6. ОБЪЕКТНАЯ МОДЕЛЬ В JAVA

Тема 6.1. 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
  }
}
```

Тема 6.2. 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.

Тема 6.3. 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 неабстрактного класса, который является наследником от Test, и должен был реализовать все абстрактные методы.

По этой же причине можно объявлять переменные типа абстрактный класс. Они могут иметь значение null или ссылаться на объект, порожденный от неабстрактного наследника этого класса.

Тема 6.4. 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.

Раздел 7. МАССИВЫ ДАННЫХ

Тема 7.1. 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
```

И далее появится ошибка времени исполнения, так как индекс превысит максимально возможное для такого массива значение. 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 массив 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
```

Тема 7.2. Преобразование типов для массивов

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

Ранее подробно рассматривались переходы между примитивными и обычными (не являющимися массивами) ссылочными типами. Хотя массивы являются объектными типами, их также будет полезно разделить по базовому типу на две группы – основанные на примитивном или ссылочном типе.

Имейте в виду, что переходы между массивами и примитивными типами являются запрещенными. Преобразования между массивами и другими объектными типами возможны только для класса Object и интерфейсов Cloneable и Serializable. Массив всегда можно привести к этим трем типам, обратный же переход является сужением и должен производиться явным образом по усмотрению разработчика. Таким образом, интерес представляют только переходы между разными типами массивов. Очевидно, что массив, основанный на примитивном типе, принципиально нельзя преобразовать к типу массива, основанному на ссылочном типе, и наоборот.

Пока не будем останавливаться на этом подробно, однако заметим, что преобразования между типами массивов, основанных на различных примитивных типах, невозможны ни при каких условиях.

Для ссылочных же типов такого строгого правила нет. Например, если создать экземпляр массива, основанного на типе Child, то ссылку на него можно привести к типу массива, основанного на типе Parent.

```
Child c[] = new Child[3];
Parent p[] = c;
```

Вообще, существует универсальное правило: массив, основанный на типе A, можно привести к массиву, основанному на типе B, если сам тип A приводится к типу B.

```
// если допустимо такое приведение:
B b = (B) new A();
// то допустимо и приведение массивов:
B b[]=(B[]) new A[3];
```

Применяя это правило рекурсивно, можно преобразовывать многомерные массивы. Например, массив Child[][] можно привести к Parent[][], так как их базовые типы приводимы (Child[] к Parent[]) также на основе этого правила (поскольку базовые типы Child и Parent приводимы в силу правил наследования).

Как обычно, расширения можно проводить неявно (как в предыдущем примере), а сужения – только явным приведением.

Вернемся к массивам, основанным на примитивном типе. Невозможность их участия в преобразованиях типов связана, конечно, с различиями между простыми и ссылочными типами данных. Поскольку элементами объектных массивов являются ссылки, они легко могут участвовать в приведении. Напротив, элементы простых типов действительно хранят числовые или булевские значения. Предположим, такое преобразование осуществимо:

```
// пример вызовет ошибку компиляции
byte b[]={1, 2, 3};
int i[]=b;
```

В таком случае, элементы b[0] и i[0] хранили бы значения разных типов. Стало быть, преобразование потребовало бы копирования с одновременным преобразованием типа всех

элементов исходного массива. В результате был бы создан новый массив, элементы которого равнялись бы по значению элементам исходного массива.

Но преобразование типа не может порождать новые объекты. Такие операции должны выполняться только явным образом с применением ключевого слова new. По этой причине преобразования типов массивов, основанных на примитивных типах, запрещены.

Если же копирование элементов действительно требуется, то нужно сначала создать новый массив, а затем воспользоваться стандартной функцией System.arraycopy(), которая эффективно выполняет копирование элементов одного массива в другой.

Ошибка ArrayStoreException

Преобразование между типами массивов, основанных на ссылочных типах, может стать причиной одной довольно неочевидной ошибки.

Рассмотрим пример:

```
Child c[] = new Child[5];
Parent p[]=c;
p[0]=new Parent();
```

С точки зрения компилятора код совершенно корректен. Преобразование во второй строке допустимо. В третьей строке элементу массива типа Parent присваивается значение того же типа.

Однако при выполнении такой программы возникнет ошибка. Нельзя забывать, что преобразование не меняет объект, изменяется лишь способ доступа к нему. В свою очередь, объект всегда «помнит», от какого типа он был порожден. С учетом этих замечаний становится ясно, что в третьей строке делается попытка добавить в массив Child значение типа Parent, что некорректно.

Действительно, ведь переменная с продолжает ссылаться на этот массив, а значит, следующей строкой может быть такое обращение:

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

где метод onlyChildMethod() определен только в классе Child. Данное обращение совершенно корректно, а значит, недопустима ситуация, когда элемент c[0] ссылается на объект, несовместимый с Child.

Таким образом, несмотря на отсутствие ошибок компиляции, виртуальная машина при выполнении программы всегда осуществляет дополнительную проверку перед присвоением значения элементу массива. Необходимо удостовериться, что реальный массив, существующий на момент исполнения, действительно может хранить присваиваемое значение. Если это условие нарушается, то возникает ошибка, которая называется ArrayStoreException.

Может сложиться впечатление, что разобранная ситуация является надуманной,– зачем преобразовывать массив и тут же задавать для него неверное значение? Однако преобразование при присвоении значений является лишь примером. Рассмотрим объявление метода:

```
public void process(Parent[] p)
{
  if (p!=null && p.length>0)
  {
    p[0]=new Parent();
  }
}
```

Метод выглядит абсолютно корректным, все потенциально ошибочные ситуации проверяются if-выражением. Однако следующий вызов этого метода все равно приводит к ошибке:

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

И это будет как раз ошибка ArrayStoreException.

Переменные типа массив и их значения

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

Как обычно, массивы, основанные на простых и ссылочных типах, мы описываем раздельно.

Переменная типа массив примитивных величин может хранить значения только точно такого же типа, либо null.

Переменная типа «массив ссылочных величин» может хранить следующие значения:

- 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 7.1. – *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

7.3. 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 двухмерный массив, состоящий из двух одномерных. Итого три объекта. Затем, на следующей строке при клонировании будет создан новый двухмерный массив, содержащий ссылки на те же самые одномерные массивы.

Теперь несложно предсказать результат обоих вариантов. В первом случае в исходном массиве меняется ссылка, хранящаяся в первом элементе, что не принесет никаких изменений для клонированного объекта. На консоли появится 1.

Во втором случае модифицируется существующий массив, что скажется на обоих двухмерных массивах. На консоли появится 0.

Обратите внимание, что если из примера убрать условие if-else, так, чтобы отрабатывал первый вариант, а затем второй, то результатом будет опять 1, поскольку в части второго варианта модифицироваться будет уже новый массив, порожденный в части первого варианта.

Таким образом, в Java предоставляется мощный, эффективный и гибкий механизм клонирования, который легко применять и модифицировать под конкретные нужды. Особенное внимание должно уделяться копированию объектных полей, которые по умолчанию копируются только по ссылке.

Заключение

В этой лекции было рассмотрено устройство массивов в Java. Подобно массивам в других языках, они представляют собой набор значений одного типа. Основным свойством массива является длина, которая в Java может равняться нулю. В противном случае, массив обладает элементами в количестве, равном длине, к которым можно обратиться, используя индекс, изменяющийся от 0 до величины длины без единицы. Длина задается при создании массива и у созданного массива не может быть изменена. Однако она не входит в определение типа, а потому одна переменная может ссылаться на массивы одного типа с различной длиной.

Создать массив можно с помощью ключевого слова new, поскольку все массивы, включая определенные на основе примитивных значений, имеют объектный тип. Другой способ – воспользоваться инициализатором и перечислить значения всех элементов. В первом случае элементы принимают значения по умолчанию (0, false, null).

Особым образом в Java устроены многомерные массивы. Они, по сути, являются одномерными, основанными на массивах меньшей размерности. Такой подход позволяет единообразно работать с многомерными массивами. Также он дает возможность создавать не только «прямоугольные» массивы, но и массивы любой конфигурации.

Хотя массив и является ссылочным типом, работа с ним зачастую имеет некоторые особенности. Рассматриваются правила приведения типа массива. Как для любого объектного типа, приведение к Object является расширяющим. Приведение массивов, основанных на ссылочных типах, во многом подчиняется обычным правилам. А вот

примитивные массивы преобразовывать нельзя. С преобразованиями связано и возникновение ошибки ArrayStoreException, причина которой – невозможность точного отслеживания типов в преобразованном массиве для компилятора.

В заключение рассматриваются последние случаи взаимосвязи типа переменной и ее значения.

Наконец, изучается механизм клонирования, существующий с самых первых версий Java и позволяющий создавать точные копии объектов, если их классы позволяют это делать, реализуя интерфейс Cloneable. Поскольку стандартное клонирование порождает только один новый объект, это приводит к особым эффектам при работе с объектными полями классов и массивами.

Раздел 8. ОПЕРАТОРЫ И СТРУКТУРА КОДА. ИСКЛЮЧЕНИЯ

Лекция проводится в интерактивной форме как проблемная (2 час.)

Тема 8.1. 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 С, что облегчает его понимание для программистов, знакомых с этим языком, и вместе с тем имеется ряд отличий, которые будут рассмотрены позднее и на которые следует обратить внимание.

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

Нормальное и прерванное выполнение операторов

Последовательность выполнения операторов может быть непрерывной, а может и прерываться (при возникновении определенных условий). Выполнение оператора может быть прервано, если в потоке вычислений будут обнаружены операторы

break
continue
return

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

Нормальное выполнение оператора может быть прервано также при возникновении исключительных ситуаций, которые тоже будут рассмотрены позднее. Явное возбуждение исключительной ситуации с помощью оператора throw также прерывает нормальное выполнение оператора и передает управление выполнением программы (далее просто управление) в другое место.

Прерывание нормального исполнения всегда вызывается определенной причиной. Приведем список таких причин:

- break (без указания метки);
- break (с указанием метки);
- continue (без указания метки);
- continue (с указанием метки);
- return (с возвратом значения);
- return (без возврата значения);
- throw с указанием объекта Throwable, а также все исключения, вызываемые виртуальной машиной Java.

Выражения могут завершаться нормально и преждевременно (аварийно). В данном случае термин «аварийно» вполне применим, т.к. причиной необычной последовательности выполнения выражения может быть только возникновение исключительной ситуации.

Если в операторе содержится выражение, то в случае его аварийного завершения выполнение оператора тоже будет завершено преждевременно (т.е. нормальный ход выполнения оператора будет нарушен).

В том случае, если в операторе имеется вложенный оператор и его завершение происходит ненормально, то так же ненормально завершается оператор, содержащий вложенный (в некоторых случаях это не так, что будет оговариваться особо).

Блоки и локальные переменные

Блок - это последовательность операторов, объявлений локальных классов или локальных переменных, заключенных в скобки. Область видимости локальных переменных и классов ограничена блоком, в котором они определены.

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

Нельзя объявлять несколько локальных переменных с одинаковыми именами в пределах видимости блока. Приведенный ниже код вызовет ошибку времени компиляции.

```
public class Test
{
  public Test() {   }
  public static void main(String[] args)
  {
    Test t = new Test();
    int x;
    lbl:
   {
      int x = 0;
      System.out.println("x = " + x);
    }
  }
}
```

В то же время не следует забывать, что локальные переменные перекрывают видимость переменных-членов. Так, следующий пример отработает нормально.

```
public class Test
{
  static int x = 5;
  public Test() { }
  public static void main(String[] args)
  {
     Test t = new Test();
     int x = 1;
     System.out.println("x = " + x);
  }
}
```

На консоль будет выведено x = 1.

То же самое правило применимо к параметрам методов.

```
public class Test
{
  static int x;
  public Test() {   }
  public static void main(String[] args)
   {
    Test t = new Test();
    t.test(5);
    System.out.println("Member value x = "
      + x);
```

```
  }
  private void test(int x)
  {
    this.x = x + 5;
    System.out.println("Local value x = "
      + x);
  }
}
```

В результате работы этого примера на консоль будет выведено:

Local value x = 5

Member value x = 10

На следующем примере продемонстрируем, что область видимости локальной переменной ограничена областью видимости блока, или оператора, в пределах которого данная переменная объявлена.

```
public class Test
{
  static int x = 5;
  public Test() {   }
  public static void main(String[] args)
  {
    Test t = new Test();
    {
      int x = 1;
      System.out.println("First block x = "
        + x);
    }
    {
      int x = 2;
      System.out.println("Second block x ="
        + x);
    }
    System.out.print("For cycle x = ");
    for(int x =0; x<5;x++)
    {
      System.out.print(" " + x);
    }
  }
}
```

Данный пример откомпилируется без ошибок и на консоль будет выведен следующий результат:

First block x = 1

Second block x =2

for cycle x = 0 1 2 3 4

Следует помнить, что определение локальной переменной есть исполняемый оператор. Если задана инициализация переменной, то выражение исполняется слева направо и его результат присваивается локальной переменной. Использование неинициализированных локальных переменных запрещено и вызывает ошибку компиляции.

Следующий пример кода

```
public class Test
{
  static int x = 5;
  public Test() {   }
  public static void main(String[] args)
  {
    Test t = new Test();
```

```
    int x;
    int y = 5;
    if( y > 3) x = 1;
    System.out.println(x);
  }
}
```

вызовет ошибку времени компиляции, т.к. возможны условия, при которых переменная x может быть не инициализирована до ее использования (несмотря на то, что в данном случае оператор if(y > 3) и следующее за ним выражение x = 1; будут выполняться всегда).

Пустой оператор

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

Метки

Любой оператор, или блок, может иметь метку. Метку можно указывать в качестве параметра для операторов break и continue. Область видимости метки ограничивается оператором, или блоком, к которому она относится. Так, в следующем примере мы получим ошибку компиляции:

```
public class Test
{
  static int x = 5;
  static {  }
  public Test() {  }
  public static void main(String[] args)
  {
    Test t = new Test();
    int x = 1;
    Lbl1:
    {
      if(x == 0) break Lbl1;
    }
    Lbl2:
    {
      if(x > 0) break Lbl1;
    }
  }
}
```

В случае, если имеется несколько вложенных блоков и операторов, допускается обращение из внутренних блоков к меткам, относящимся к внешним.

Этот пример является вполне корректным:

```
public class Test
{
  static int x = 5;
  static {  }
  public Test() {  }
  public static void main(String[] args)
  {
    Test t = new Test();
    int L2 = 0;
    Test: for(int i = 0; i< 10;i++)
    {
      test: for(int j = 0; j< 10;j++)
      {
        if( i*j > 50) break Test;
      }
```

```
    }
  }
  private void test()
  {     ;   }
}
```

В этом же примере можно увидеть, что метки используют пространство имен, отличное от пространства имен переменных, методов и классов.

Традиционно использование меток не рекомендуется, особенно в объектно-ориентированных языках, поскольку серьезно усложняет понимание порядка выполнения кода, а значит, и его тестирование и отладку. Для Java этот запрет можно считать не столь строгим, поскольку самый опасный оператор goto отсутствует. В некоторых ситуациях (как в рассмотренном примере с вложенными циклами) использование меток вполне оправданно, но, конечно, их применение следует ограничивать лишь самыми необходимыми случаями.

Тема 8.2. Операторы разветвления

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.

Тема 8.3. 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.

Тема 8.4. Операторы прерывания и изменения хода выполнения программы

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 только в циклах while, do, for. Если в потоке вычислений встречается оператор continue, то выполнение текущей последовательности операторов (выражений) должно быть прекращено и управление будет передано на начало блока, содержащего этот оператор.

```
...int x = (int)(Math.random()*10);
int arr[]  = {....}
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.

4.3. Лабораторные работы

№ п/п	*Номер раздела дисциплины*	*Наименование лабораторной работы*	*Объем (час.)*	*Вид занятия в интерактивной, активной, инновационной формах, (час.)*
1	**2, 5.**	Основные конструкции языка программирования Java	6	-
2	**3, 4.**	Абстрактные классы и интерфейсы	7	-
3	**1, 8.**	Создание приложений с помощью библиотеки Swing	7	-
4	**2, 7.**	Разработка апплетов	7	Решение проблем в группах смешанного состава (4 час.)
5	**4, 6.**	Множественные нити наследования (Multiple threads)	7	-
		ИТОГО	**34**	**4**

4.4. Практические занятия

Учебным планом не предусмотрены.

4.5. Контрольные мероприятия: курсовой проект (курсовая работа), контрольная работа, РГР, реферат

Учебным планом не предусмотрены.

5. МАТРИЦА СООТНЕСЕНИЯ РАЗДЕЛОВ УЧЕБНОЙ ДИСЦИПЛИНЫ К ФОРМИРУЕМЫМ В НИХ КОМПЕТЕНЦИЯМ И ОЦЕНКЕ РЕЗУЛЬТАТОВ ОСВОЕНИЯ ДИСЦИПЛИНЫ

Компетенции / *№, наименование разделов дисциплины*	*Кол-во часов*	*Компетенции*		*Σ комп.*	$t_{ср}$, *час*	*Вид учебных занятий*	*Оценка результатов*
		ОПК	*ПК*				
		9	**6**				
1	2	3	4	12	13	14	15
1. Введение в программирование сетевых приложений на языке Java	15	+	+	2	7,5	Лк, ЛР, СРС	Экзамен
2. Основы объектно-ориентированного программирования	14	+	+	2	7	Лк, ЛР, СРС	Экзамен
3. Обзор основных конструкций языка Java	16	+	+	2	8	Лк, ЛР, СРС	Экзамен
4. Имена и пакеты	15	+	+	2	7,5	Лк, ЛР, СРС	Экзамен
5. Классы и приведение типов	14	+	+	2	7	Лк, ЛР, СРС	Экзамен
6. Объектная модель в Java	16	+	+	2	8	Лк, ЛР, СРС	Экзамен
7. Массивы данных	13	+	+	2	6,5	Лк, ЛР, СРС	Экзамен
8. Операторы и структура кода. Исключения	14	+	+	2	7	Лк, ЛР, СРС	Экзамен
всего часов	**117**	**58,5**	**58,5**	**2**	**58,5**		

6. ПЕРЕЧЕНЬ УЧЕБНО-МЕТОДИЧЕСКОГО ОБЕСПЕЧЕНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

1. Хорстманн, К. JAVA 2 / К. С. Хорстманн, Г. Корнелл. - 7-е изд. - Москва : Вильямс, 2007 - (библиотека профессионала). Т.1 : Основы. - 896 с.
2. Кингсли-Хью, Э. JavaScript 1.5 : учебный курс / Э.Кингсли-Хью; Пер.с англ. - Санкт-Петербург : Питер, 2001. - 266 с. - (Учебный курс).

7. ПЕРЕЧЕНЬ ОСНОВНОЙ И ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ, НЕОБХОДИМОЙ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

№ *(сквозная нумерация)*	***Наименование издания*** *(автор, заглавие, выходные данные)*	***Вид занятия***	***Количество экземпляров в библиотеке,*** **шт.**	***Обеспеченность,*** **(экз./ чел.)**
1	**2**	**3**	**4**	**5**
Основная литература				
1.	Баженова, И.Ю. Язык программирования Java. М.: Диалог-МИФИ, 2008. – 254 с. http://biblioclub.ru/index.php?page=book_red&id=54745&sr=1	Лк	ЭР	1,0
2.	Дуванов, А.А. Web-конструирование. Элективный курс : учебное пособие / А.А. Дуванов. - Санкт-Петербург : БХВ- Петербург, 2006. - 432 с. + 1 эл. опт. диск (CD-ROM)	Лк	10	0,7
3.	Комолова, Н.В. HTML : учебное пособие / Н.В. Комолова. - Санкт-Петербург : БХВ- Петербург, 2007. - 268 с. - (Учебный курс)	Лк	10	0,7
4.	Ноутон, П. Java 2 : учебник / П. Ноутон, Г. Шилдт. - Санкт-Петербург : БХВ- Петербург, 2006. - 1072 с.	ЛР	20	1,0
5.	Хорстманн, К. Java 2 / К. Хортсманн, Г. Корнелл. - 7-е изд. - Москва : Вильямс, 2006 - . Т.2 : Тонкости программирования. - 1168 с. - (Библиотека профессионала).	Лк	7	0,5
Дополнительная литература				
6.	Вайсфельд, М. Объектно-ориентированный подход: Java, Net, C++ : учебник / М. Вайсфельд. - 2-е изд. - Москва : Кудиц-Образ, 2005. - 336 с. - (Библиотека разработчика).	Лк	5	0,3
7.	Еськова, С.В. Основы Web-дизайна и мультимедийных презентаций : методические указания по выполнению лабораторных работ / С. В. Еськова, М. Ю. Вахрушева. - Братск : БрГУ, 2009. - 85 с. - Б. ц.	ЛР	52	1,0
8.	Герман, О.В. Программирование на JAVA и C# для студента : учебное пособие / О.В. Герман. - Санкт-Петербург : БХВ- Петербург, 2005. - 511 с.	ЛР	5	0,3
9.	Иванова, Е.Б. Java 2, Enterprise Edition. Технологии проектирования и разработки : учебное пособие / Е.Б. Иванова, М.М. Вершинин. - Санкт-Петербург : БХВ-Петербург, 2003. - 1088 с. - (Мастер программ).	ЛР	4	0,2
10.	Кулямин, В.В. Технологии программирования. Компонентный подход. М.: Интернет-Университет Информационных Технологий, 2007. – 464 с. http://biblioclub.ru/index.php?page=book_red&id=233311&sr=1	Лк	ЭР	1,0
11.	Кулямин, В. Компонентный подход в программировании. М.: Национальный Открытый	Лк	ЭР	1,0

	Университет «ИНТУИТ», 2016. 591 с. http://biblioclub.ru/index.php?page=book_red&id=429086 &sr=1			
12.	Роганов, Е.А. Основы информатики и программирования: курс. М.: Интернет-Университет Информационных Технологий, 2006. 336 с. http://biblioclub.ru/index.php?page=book_red&id=234651 &sr=1	Лк	ЭР	1,0
13.	Орлов, С.А. Теория и практика языков программирования : учебник для бакалавров и магистров / С.А. Орлов. - Санкт-Петербург : Питер, 2013. - 688 с. - (Учебное пособие. Стандарт третьего поколения).	Лк	5	0,3
14.	Основы WEB-технологий : курс лекций / П.Б. Храмцов, С.А. Брик [и др.]. - М. : ИНТУИТ.РУ, 2003. - 512 с. - (Интернет- Университет Информационных Технологий).	Лк	43	1,0
15.	Фридман, А.Л. Построение Интернет-приложений на языке Java : практический курс / А.Л. Фридман. - Москва : Горячая линия- Телеком, 2002. - 335 с.	Лк	10	0,7
16.	Хорстманн, К. JAVA 2 / К. Хорстман, Г. Корнелл. - 7-е изд. - Москва : Вильямс, 2006. - (Библиотека профессионала). Т.1 : Основы. - 896 с.	Лк	5	0,3

8. ПЕРЕЧЕНЬ РЕСУРСОВ ИНФОРМАЦИОННО - ТЕЛЕКОММУНИКАЦИОННОЙ СЕТИ «ИНТЕРНЕТ» НЕОБХОДИМЫХ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

1.Электронный каталог библиотеки БрГУ http://irbis.brstu.ru/CGI/irbis64r_15/cgiirbis_64.exe?LNG=&C21COM=F&I21DBN=BOOK&P21DBN=BOOK&S21CNR=&Z21ID=.

2. Электронная библиотека БрГУ http://ecat.brstu.ru/catalog .

3. Электронно-библиотечная система «Университетская библиотека online» http://biblioclub.ru .

4. Электронно-библиотечная система «Издательство «Лань» http://e.lanbook.com .

5. Информационная система "Единое окно доступа к образовательным ресурсам" http://window.edu.ru .

6. Научная электронная библиотека eLIBRARY.RU http://elibrary.ru .

7. Университетская информационная система РОССИЯ (УИС РОССИЯ) https://uisrussia.msu.ru/ .

8. Национальная электронная библиотека НЭБ http://xn--90ax2c.xn--p1ai/how-to-search /.

9. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ

9.1. Методические указания для обучающихся по выполнению лабораторных работ

Лабораторная работа №1

Основные конструкции языка программирования Java

Цель работы:

Ознакомление с основными конструкциями языка Java, операциями, литерами, различными типами операторов

Задание (один из возможных вариантов):
1. Ввести с консоли n целых чисел и поместить их в массив. На консоль вывести чётные и нечётные числа.

Порядок выполнения:
1. Проанализировать полученное задание, выделить информационные объекты и действия;
2. Разработать программу с использованием требуемых типов и операторов.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Программный код файла реализации и результаты работы программы.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным во втором и пятом разделах данной дисциплины.

Основная литература

1. Ноутон, П. Java 2 : учебник / П. Ноутон, Г. Шилдт. - Санкт-Петербург : БХВ- Петербург, 2006. - 1072 с.

Дополнительная литература

1. Герман, О.В. Программирование на JAVA и C# для студента : учебное пособие / О.В. Герман. - Санкт-Петербург : БХВ- Петербург, 2005. - 511 с.

Лабораторная работа №2
Абстрактные классы и интерфейсы

Цель работы:
Реализовать абстрактные классы или интерфейсы, а также наследование и полиформизм для классов.

Задание (один из возможных вариантов):
1. Абстрактный класс Книга (Шифр, Автор, Название, Год, Издательство). Подклассы: Справочник и Энциклопедия

Порядок выполнения:
1. Проанализировать полученное задание, выделить информационные объекты и действия;
2. Разработать программу с использованием абстрактных классов или интерфейсов;
3. Использовать при разработке наследование и полиморфизм.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Программный код файла реализации и результаты работы программы.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в третьем и четвёртом разделах данной дисциплины.

Основная литература

1. Ноутон, П. Java 2 : учебник / П. Ноутон, Г. Шилдт. - Санкт-Петербург : БХВ- Петербург, 2006. - 1072 с.

Дополнительная литература

1. Иванова, Е.Б. Java 2, Enterprise Edition. Технологии проектирования и разработки : учебное пособие / Е.Б. Иванова, М.М. Вершинин. - Санкт-Петербург : БХВ- Петербург, 2003. - 1088 с. - (Мастер программ).

Лабораторная работа №3
Создание приложений с помощью библиотеки Swing

Цель работы:

Освоение приёмов применения компонентов библиотеки Swing

Задание (один из возможных вариантов):

1. Поместить в приложение две панели JPanel и кнопку. Первая панель содержит поле ввода и метку «Поле ввода», вторая – поле вывода и метку «Поле вывода». Для размещения в окне двух панелей и кнопки «Скопировать» использовать менеджер размещения BorderLayout.

Порядок выполнения:

1. Проанализировать полученное задание, выделить информационные объекты и действия;
2. Разработать программу с использованием соответствующих элементов (классов) библиотеки Swing.

Форма отчетности:

Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:

1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Программный код файла реализации и окно формы приложения с результатами работы программы.

Рекомендации по выполнению заданий и подготовке к лабораторной работе

Ознакомиться с теоретическим материалом, представленным во первом и восьмом разделах данной дисциплины.

Основная литература

1. Ноутон, П. Java 2 : учебник / П. Ноутон, Г. Шилдт. - Санкт-Петербург : БХВ- Петербург, 2006. - 1072 с.

Дополнительная литература

1. Еськова, С.В. Основы Web-дизайна и мультимедийных презентаций : методические указания по выполнению лабораторных работ / С. В. Еськова, М. Ю. Вахрушева. - Братск : БрГУ, 2009. - 85 с. - Б. ц.

Лабораторная работа №4
Разработка апплетов

Лабораторная работа проводится в интерактивной форме с решением проблем в группах смешанного состава

Цель работы:

Реализовать соответствующий апплет, осуществить его запуск и выполнение.

Задание (один из возможных вариантов):

1. Создать апплет по заданию из третьей лабораторной работы.

Порядок выполнения:
1. Проанализировать полученное задание, выделить информационные объекты и действия;
2. Разработать программу апплета с использованием соответствующих элементов (классов) библиотеки Swing, графических элементов и др.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Программный код файла реализации и результаты работы программы.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным во втором и седьмом разделах данной дисциплины.

Основная литература

1. Ноутон, П. Java 2 : учебник / П. Ноутон, Г. Шилдт. - Санкт-Петербург : БХВ- Петербург, 2006. - 1072 с.

Дополнительная литература

1. Герман, О.В. Программирование на JAVA и C# для студента : учебное пособие / О.В. Герман. - Санкт-Петербург : БХВ- Петербург, 2005. - 511 с.

Лабораторная работа №5
Множественные нити выполнения (Multiple threads)

Цель работы:
Ознакомление с основными принципами многопоточности в языке Java, возможностью написания программ с использованием нитей

Задание (один из возможных вариантов):
1. Организовать сортировку массива методами Шелла, Хоара, пузырька на основе бинарного дерева в разных потоках.

Порядок выполнения:
1. Проанализировать полученное задание, выделить информационные объекты и действия;
2. Разработать программу с использованием нитей.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Программный код файла реализации и результаты работы программы.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в четвёртом и шестом разделах данной дисциплины.

Основная литература

1. Ноутон, П. Java 2 : учебник / П. Ноутон, Г. Шилдт. - Санкт-Петербург : БХВ- Петербург, 2006. - 1072 с.

Дополнительная литература

1. Герман, О.В. Программирование на JAVA и C# для студента : учебное пособие / О.В.

Герман. - Санкт-Петербург : БХВ- Петербург, 2005. - 511 с.

10. ПЕРЕЧЕНЬ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, ИСПОЛЬЗУЕМЫХ ПРИ ОСУЩЕСТВЛЕНИИ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

1. ОС Windows 7 Professional.
2. Microsoft Office 2007 Russian Academic OPEN No Level.
3. Антивирусное программное обеспечение Kaspersky Security.
4. Visual Studio Community.

11. ОПИСАНИЕ МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЙ БАЗЫ, НЕОБХОДИМОЙ ДЛЯ ОСУЩЕСТВЛЕНИЯ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

Вид занятия	***Наименование аудитории***	***Перечень основного оборудования***	***№ ПЗ и ЛР***
1	**3**	**4**	**5**
ЛР	Дисплейный класс	AMD Athlon 64 (5GHz/250Gb/2Gb/DD-RW), 2 ядра	ЛР 1-5
СР	ЧЗ №3	Оборудование 15 - CPU 5000/RAM 2Gb/HDD (Монитор TFT 19 LG 1953S-SF);принтер HP LaserJet P3005	-

Приложение 1

ФОНД ОЦЕНОЧНЫХ СРЕДСТВ ДЛЯ ПРОВЕДЕНИЯ ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

1. Описание фонда оценочных средств (паспорт)

№ компетенции	Элемент компетенции	Раздел	Тема	ФОС
ОПК-9	способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности	**1.** Введение в программирование сетевых приложений на языке Java	1.1. История создания Java	-
			1.2. Программирование на Java: достоинства, недостатки, особенности и состав	Экзаменационный вопрос 1.1
		2. Основы объектно-ориентированного программирования	2.1. Методология процедурно-ориентированного программирования	Экзаменационный вопрос 2.1
			2.2. Методология объектно-ориентированного программирования	Экзаменационный вопрос 2.2
		3. Обзор основных конструкций языка Java	3.1. Конструкции языка Java	Экзаменационный вопрос 3.1
			3.3. Дополнение. Работа с операторами	Экзаменационный вопрос 3.2
			3.5. Ссылочные типы	Экзаменационный вопрос 3.3
		4. Имена и пакеты	4.1. Имена и элементы языка	Экзаменационные вопросы 4.1, 4.2
		5. Классы и приведение типов	5.1. Модификаторы доступа	Экзаменационные вопросы 5.1, 5.2
		6. Объектная модель в Java	6.1. Статические элементы	Экзаменационный вопрос 6.1
			6.2. Ключевые слова this и super	Экзаменационный вопрос 6.2
		7. Массивы данных	7.1. Массивы как тип данных в Java	Экзаменационные вопросы 7.1-7.4
		8. Операторы и структура кода. Исключения	8.1. Управление ходом программы	Экзаменационные вопросы 8.1-8.3
			8.2. Операторы разветвления	Экзаменационные вопросы 8.4, 8.5
ПК-6	способность производить расчёты и проетирование отдельных блоков и устройств	**1.** Введение в программирование сетевых приложений на языке Java	1.3. Жизненный цикл программы на Java	Экзаменационный вопрос 1.2
		2. Основы объектно-ориентированного программирования	2.3. Характеристики объектов и классов	Экзаменационные вопросы 2.3-2.5
			2.4. Достоинства и	Экзаменацион

	систем автоматизации и управления и выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления		недостатки объектно-ориентированного программирования	ные вопросы 2.6, 2.7
		3. Обзор основных конструкций языка Java	3.2. Виды лексем	Экзаменационные вопросы 3.4, 3.5
			3.4. Переменные и базовые типы данных	Экзаменационный вопрос 3.6
		4. Имена и пакеты	4.2. Пакеты	Экзаменационные вопросы 4.3, 4.4
			4.3. Область видимости имён	Экзаменационный вопрос 4.5
			4.4. Соглашения по именованию	Экзаменационный вопрос 4.6
		5. Классы и приведение типов	5.2. Объявление классов	Экзаменационные вопросы 5.3-5.5
		6. Объектная модель в Java	6.3. Ключевое слово abstract	Экзаменационный вопрос 6.3
			6.4. Интерфейсы	Экзаменационные вопросы 6.4-6.6
		7. Массивы данных	7.2. Преобразование типов для массивов	Экзаменационный вопрос 7.5
			7.3. Клонирование	Экзаменационный вопрос 7.6
		8. Операторы и структура кода. Исключения	8.3. Управление циклами	Экзаменационные вопросы 8.6, 8.7
			8.4. Операторы прерывания и изменения хода программы	Экзаменационные вопросы 8.8, 8.9

2. Экзаменационные вопросы

№ п/п	Компетенции		ЭКЗАМЕНАЦИОННЫЕ ВОПРОСЫ	№ и наименование раздела
	Код	Определение		
1	2	3	4	5
1.	ОПК-9	способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности	**1.** Программирование на Java: достоинства, недостатки, особенности и состав	**1.** Введение в программирование сетевых приложений на языке Java
			1. Методология процедурно-ориентированного программирования	**2.** Основы объекно-ориентированного программирования
			2. Методология объектно-ориентированного программирования	
			1. Конструкции языка Java	**3.** Обзор основных конструкций языка Java
			4. Дополнение. Работа с операторами	
			6. Ссылочные типы	
			1. Простые и составные имена. Элементы	**4.** Имена и пакеты
			2. Имена и идентификаторы. Область видимости	
			1. Предназначение модификаторов доступа	**5.** Классы и приведение типов

			2. Разграничение доступа в Java	
			1. Статические элементы	**6.** Объектная модель в Java
			2. Ключевые слова this и super	
			1. Объявления массивов	**7.** Массивы данных
			2. Инициализация массивов	
			3. Многомерные массивы	
			4. Класс массива	
			1. Нормальное и прерванное выполнение операторов	**8.** Операторы и структура кода. Исключения
			2. Блоки и локальные переменные	
			3. Пустой оператор. Метки	
			4. Оператор if	
			5. Оператор switch	
2.	ПК-6	способность производить расчёты и проетирование отдельных блоков и устройств систем автоматизации и управления и выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления	**2.** Жизненный цикл программы на Java	**1.** Введение в программирование сетевых приложений на языке Java
			3. Состояние. Поведение. Уникальность	**2.** Основы объекно-ориентированного программирования
			4. Классы. Инкапсуляция. Наследование	
			5. Полиморфизм	
			6. Достоинства объектно-ориентированного программирования	
			7. Недостатки объекно-ориентированного программирования	
			4. Идентификаторы. Ключевые слова	**3.** Обзор основных конструкций языка Java
			5. Литералы	
			6. Переменные и базовые типы данных	
			3. Элементы пакета. Платформенная поддержка пакетов	**4.** Имена и пакеты
			4. Модуль компиляции	
			5. Область видимости имён	
			6. Соглашения по именованию	
			3. Заголовок класса. Тело класса	**5.** Классы и приведение типов
			4. Объявление полей. Объявление методов	
			5. Объявление конструкторов	
			3. Ключевое слово abstract	**6.** Объектная модель в Java
			4. Объявление интерфейсов	
			5. Реализация интерфейса	
			6. Применение интерфейсов	
			5. Преобразование типов для массивов	**7.** Массивы данных
			6. Клонирование	
			6. Цикл while	**8.** Операторы и структура кода. Исключения
			7. Цикл do. Цикл for	
			8. Оператор continue. Оператор break	
			9. Именованные блоки. Оператор return	

3. Описание показателей и критериев оценивания компетенций

Показатели	Оценка	Критерии
Знать (ОПК-9): - основные требования информационной безопасности; (ПК-6): - возможности использования современных языков и технологий программирования для разработки сетевых приложений; **Уметь** (ОПК-9): - использовать навыки работы с компьютером; (ПК-6): - выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления; **Владеть** (ОПК-9): - методами информационных технологий; (ПК-6): - навыками расчётов и проетирования отдельных блоков и устройств систем автоматизации и управления.	**отлично**	Обучающийся должен во время ответа показать знания: основных направлений развития в области сетевых технологий, состояния и перспективы развития языков программирования и сетевых технологий, возможности использования современных языков и технологий программирования для разработки сетевых приложений, основных терминов, используемых в научно-технической литературе по сетевым технологиям. Обучающийся должен иметь навыки владения: методами информационных технологий, понимания материала и способности высказывания мыслей на научно-техническом языке. Обучающийся во время ответа должен продемонстрировать умения: использовать навыки работы с компьютером, выбирать наиболее перспективные и рациональные способы организации и разработки сетевых приложений, использовать наиболее перспективные клиентские и серверные технологии.
	хорошо	Ответ содержит неточности. Дополнительные вопросы требуется, но обучающийся с ними справляется отлично.
	удовлетворительно	Ответил только на один вопрос, либо слабо ответил на оба вопроса. На дополнительные вопросы отвечает неуверенно.
	неудовлетворительно	На оба вопроса обучающийся отвечает неубедительно. На дополнительные вопросы преподавателя также не может ответить.

4. Методические материалы, определяющие процедуры оценивания знаний, умений, навыков и опыта деятельности

Дисциплина Программирование сетевых приложений направлена на ознакомление с базовыми принципами эффективной разработки и использования локальных, корпоративных и глобальных сетей; на получение теоретических знаний и практических навыков создания сетевых программных комплексов на языке Java для их дальнейшего использования в практической деятельности.

Изучение дисциплины предусматривает:

- лекции;
- лабораторные работы;

– самостоятельную работу;
– экзамен.

В ходе освоения раздела 1 «Введение в программирование сетевых приложений на языке Java» обучающиеся должны уяснить: достоинства и недостатки, особенности программирования и жизненный цикл программы на языке Java.

В ходе освоения раздела 2 «Основы объектно-ориентированного программирования» обучающиеся должны знать: методологию процедурно-ориентированного и объектно-ориентированного программирования, характеристики объектов и классов.

В ходе освоения раздела 3 «Обзор основных конструкций языка Java» обучающиеся должны уяснить конструкции языка Java: лексемы, дополнения, операторы, переменные, базовые и ссылочные типы данных.

В ходе освоения раздела 4 «Имена и пакеты» обучающиеся должны знать: имена и элементы языка Java, элементы пакета, области видимости имён, соглашения по именованию.

В ходе освоения раздела 5 «Классы и приведение типов» обучающиеся должны уяснить: четыре модификатора доступа в Java, объявление классов.

В ходе освоения раздела 6 «Объектная модель в Java» обучающиеся должны знать: статические элементы, ключевые слова: this, super, abstract; объявление, реализацию и применение интерфейсов.

В ходе освоения раздела 7 «Массивы данных» обучающиеся должны уяснить: объявление, инициализацию, класс и преобразование типов для массивов, клонирование.

В ходе освоения раздела 8 «Операторы и структуры кода. Исключения» обучающиеся должны знать: приёмы управления ходом программы, операторы разветвления, прерывания, изменения хода выполнения программы, управления циклами.

В процессе проведения лабораторных работ происходит закрепление знаний, формирование умений и навыков использования ИСР Borland Java Builder 2005 для разработки объектно-ориентированного обеспечения.

При подготовке к экзамену рекомендуется особое внимание уделить следующим вопросам: жизненный цикл программы на Java, конструкции языка Java, объявление классов и массивов.

Работа с литературой является важнейшим элементом в получении знаний по дисциплине. Прежде всего, необходимо воспользоваться списком рекомендуемой литературы. Дополнительные сведения по изучаемым темам можно найти в периодической печати и Интернете.

Предусмотрено проведение аудиторных занятий в интерактивной форме (лекции с текущим контролем, лабораторные работы с разбором конкретных ситуаций) в сочетании с внеаудиторной работой

Приложение 2

АННОТАЦИЯ
рабочей программы дисциплины
Программирование сетевых приложений

1. Цель и задачи дисциплины

Целью изучения дисциплины является изложение базовых принципов эффективной разработки и использования локальных, корпоративных и глобальных сетей для решения практических задач.

Задачей изучения дисциплины является подготовка обучающихся к самостоятельной работе по решению практических задач, связанных с созданием сетевых программных комплексов в операционной среде Windows.

2. Структура дисциплины

2.1 Распределение трудоемкости по отдельным видам учебных занятий, включая самостоятельную работу: Лк – 17 час.; ЛР – 34 час.; СР – 66 час.

Общая трудоемкость дисциплины составляет .144 часа, 4 зачетные единицы.

2.2 Основные разделы дисциплины:

1 – Введение в программирование сетевых приложений на языке Java;
2 – Основы объектно-ориентированного программирования;
3 – Обзор основных конструкций языка Java;
4 – Имена и пакеты;
5 – Классы и приведение типов;
6 – Объектная модель в Java;
7 – Массивы данных;
8 – Операторы и структура кода. Исключения.

3. Планируемые результаты обучения (перечень компетенций)

Процесс изучения дисциплины направлен на формирование следующих компетенций:
ОПК-9 – способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности;
ПК-6 - способность производить расчёты и проетирование отдельных блоков и устройств систем автоматизации и управления и выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления.

4. Вид промежуточной аттестации: экзамен.

Приложение 3

Протокол о дополнениях и изменениях в рабочей программе
на 201___ - 201___ учебный год

1. В рабочую программу по дисциплине вносятся следующие дополнения:

_____
_____

2. В рабочую программу по дисциплине вносятся следующие изменения:

_____
_____
_____

Протокол заседания кафедры №____ от «___» _____ 201 __ __ г.,
(разработчик)

Заведующий кафедрой _____ _____
(подпись) *(Ф.И.О.)*