ДЖОРУПБЕКОВ С.

ЯЗЫК ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Учебное пособие

Рекомендовано Учебно-методическим советом
Кыргызского национального университета
имени Ж. Баласагына

Бишкек – 2009

Рецензенты:

к.ф.м.-н., доцент Кыргызского-Турецкого университета Манас
Султанов Р. К.,
и.о. профессора, заведующий кафедрой прикладной информатики Института информационных и коммуникационных технологий Кыргызского национального университета им. Ж. Баласагына
Бердимуратов А.И.

Джорупбеков С.

Язык объектно-ориентированного программирования: (Учебное пособие). – Б.: 2009. – 122 с.

Учебное пособие подготовлено по программе международного проекта для магистров с двойной компетенцией: «Информатика и социальные науки».

Универсальный язык программирования Java можно считать вехой, отметившей начало эры программирования для Интернет. Разработанный специально для создания приложений для Интернет и использующий принцип «Написан один раз, но может запускаться везде», Java положил начало новой парадигме программирования. Java внес настолько фундаментальные понятия в принятые нормы программирования, что историю компьютерных языков программирования можно разделить на два этапа: до и после Java. Программисты эпохи, предшествующей Java, создавали программы и запускали их на отдельном компьютере. Программисты, использующие Java, создают программы для компьютеров, распределенных по сети и работающих в сетевом окружении. Java – язык программирования мирового значения для разработки компьютерных приложений промышленного уровня, работающих на самых разных устройствах, начиная от сотового телефона и кончая серверами крупнейших предприятий.

Рекомендовано к печати Учебно-методическим советом Кыргызского национального университета им. Ж. Баласагына

СОДЕРЖАНИЕ

Введение

Хотя спорно путь развития современного общества, но тем не менее ясно, что прогресс невозможен без широкого использования компьютеров, компьютерных информационных технологий и информационных систем в деятельности современного человека. Изучение и реализация сложных систем и процессов в деятельности человека невозможно без применения информационно-коммуникационных компьютерных технологий, связанных с программированием.

Современный язык программирования Java является мощным объектно-ориентированным языком, который достаточно легко может быть использован как начинающими, так и опытными программистами при построении сложных информационных систем. Java представляет средства процедурного и объектно-ориентированного программирования. Считается что, парадигма основанного на объектах программирования (с классами, инкапсуляцией и объектами) и объектно-ориентированного программирования (с наследованием и полиморфизмом) имеет ключевое значение при разработке элегантных, устойчивых и легко сопровождаемых программных систем.

Хотя разработчики Java ориентировались на создание языка для Интернет, Java является языком не только для Интернет. Это по-настоящему универсальный язык с широкими возможностями, разработанный для современного сетевого мира. Java можно использовать для решения любых задач программирования, хотя в нем и сделаны акценты на решении задач программирования для сетей. Java – это лучший язык для преподавания студентам первого года обучения технологий работы с графикой, изображениями, анимацией, звуком, видео, базами данных и технологий программирования приложений для сетевых, многопотоковых и распределенных вычислений.

Java, кроме решения любых задач программирования, используется для создания веб-страниц с динамическим и интерактивным содержанием, масштабируемых корпоративных приложений, для расширения функциональности веб-серверов, создания приложений для бытовых приборов, таких как сотовые телефоны, пейджеры, персональные цифровые помощники. Возможности Java полностью отвечают современным требованиям компаний и организаций к обработке информации.

Учебное пособие знакомит с основой языка Java, необходимых для написания простых программ и для освоения более сложных его средств.

Сатылган Джорупбеков – доцент
Кыргызского национального университета им. Ж. Баласагына

Язык объектно-ориентированного программирования

План лекций и лабораторных занятий

Лекции – 32 ч.

Лабораторные занятия – 32 ч

№	Тема лекции	Содержание	Лек-ции	Лаб. зан.
1	Объектно-ориентированное программирование и язык Java	Основные парадигмы программирования. Принципы объектно-ориентированного программирования. Язык Java, Системы программирования на Java. Этапы разработки, выполнения Java приложения	2 Лек 1	2 Lab1
2	Лексемы Java	Пробельные символы Идентификаторы Константы, ключевые слова, знаки операций, разделители и комментарии. Структура и виды Java программы,	2 Лек 2	2 Lab2
3	Типы данных. Переменные. Оператор присваивания. Выражения.	Переменные Примитивные типы Целочисленные типы Дробные типы Булевский тип. Операции. Оператор присваивания. Область действия и время жизни переменных. Преобразование и приведение типов	2 Лек 3	2 Lab3
4	Введение в апплеты Java	Начальные понятия апплета. Составление простого апплета, компиляция и выполнение апплета.	2 Лек 4	2 Lab4
5	Управление последовательностью действий	Управление ходом программы Нормальное и прерванное выполнение операторов Пустой оператор Метки Оператор if Оператор switch Цикл while Цикл do, цикл for Операторы break и continue Именованные блоки	2 Лек 5	2 Lab5
6	Массивы и строки	Одномерные массивы Многомерные массивы Строки	2 Лек 6	2 Lab6
7	Программные модули Java: методы	Методы. Определение метода Пример	2 Лек 7	2 Lab7
8	Программные модули Java: классы	Классы. Определение класса Объединение класса и программы. Объявление объектов Оператор new	2 Лек 8	2 Lab8

		Назначение ссылочных переменных объекта Добавление метода к классу Возврат значений Добавление метода с параметрами		
9	Конструктор и его использование	Определение конструктора Параметризованные конструкторы Ключевое слово this Скрытие переменной экземпляра Об определении классов и методов	2 Лек 9	2 Lab9
10	Наследование	Управление доступом к элементам класса Основы наследования Статические элементы Использование ключевого слова super Особенности взаимодействия подкласса и суперкласса	2 Лек 10	2 Lab10
11 - 12	Полиморфизм.	Полиморфизм Применение переопределения методов. Связывание Использование абстрактных классов Перегрузка методов	4 Лек 11-12	2 Lab11 -12
13	Пакеты и библиотека классов Java	Определение пакета Пример простого пакета Создание повторно используемого класса Защита доступа	2 Лек 13	2 Lab13
14	Интерфейсы	Определение интерфейса Реализация интерфейсов Реализации доступа через интерфейсные ссылки Частичные реализации	2 Лек 14	2 Lab14
15	Ошибки при работе программы. Исключения	Причины возникновения ошибок Обработка исключительных ситуаций Конструкция try-catch-finally Использование оператора throw Проверяемые и непроверяемые исключения Создание пользовательских классов исключений Генерации тсключений на примере вычисления факториалов	2 Лек 15	2 Lab15
16	Компоненты Java	Основы компонентов Построение оконного приложения с графическим интерфейсом пользователя	2 Лек 16	2 Lab16

Лабораторные занятия

№	ТЕМА
Lab1	Знакомство с Java программой
Lab2	Составление Java-программы
Lab3	Использование диалогового окна сообщений для ввода-вывода
Lab4	Простой апплет – вывод текстовой строки
Lab5	Повторение, управляемое счетчиком
Lab6	Использование операций break, continue
Lab7	Создание и использование пользовательского метода
Lab8	Создание класса и его использование
Lab9	Создание и использование конструктора
Lab10	Использование наследования для расширения класса
Lab11	Использование абстрактного класса для построения полиморфного метода
Lab12	Перегрузка метода и конструктора
Lab13	Создание и использование пакета
Lab14	Использование интерфейса для организации полиморфизма
Lab15	Обработка исключительных ситуаций
Lab16	Обработка событий

1. ЛЕКЦИИ

Лекция 1. Объектно-ориентированное программирование и язык Java

Основные парадигмы программирования

Как вы знаете, все компьютерные программы состоят из двух элементов: *кода описания действий алгоритма* и *данных. Соответственно с*уществуют две основные парадигмы (основополагаюших подхода), которые управляют конструированием программ. Первый подход называет программу *моделью, которая ориентирована на процесс, действия*. При этом подходе программу определяют последовательности *операторов* (действий) ее кода, т.е. в языках программирования с такой моделью акцент делается на действия. В этих языках данные существуют для поддержки действий, необходимых программе. Данные в этих языках «негибки». Существует всего несколько встроенных типов данных, и создание программистом своих собственных новых типов данных представляет определенные трудности. Процедурные языки, такие как С, Паскаль, успешно эксплуатируют такую модель. Однако, как показала практика программирования, при таком подходе возникают проблемы, когда возрастает размер и сложность создаваемых программ.

Компьютерная программа должна эмулировать реальный мир, т.е. имитировать функционирования всей или части одной реальной системы. Реальные системы состоят из объектов. Человек живет в мире объектов. Любой объект описывается с помощью двух наборов свойств: *атрибутов* и *поведения*. *Атрибут* – это характеристика объекта. Например, у человека есть имя, рост, вес и много других характеристик или у игральной карты есть масть, значение, лицевая сторона. *Поведение* – это действие, которое объект может осуществить. Человек может сидеть, стоять, идти и это, не считая тысячи других вариантов поведения, а с картой можно выполнять операции: установить значение, перевернута ли лицевой строной вверх, перевернуть.

С программисткой точки зрения атрибуты объекта это данные (простые или составные), а поведения это действия (алгоритм). Практика программирования сложных задач показала, что необходима идея жесткого объединения данных и действий, производимых над этими данными, в единое целое, которое называется классом. Так что класс – это набор методов (подпрограмм-функций) и данных, которые объединены в одной сущности как одно целое. Он предоставляет некоторое обслуживание или выполняет некоторую работу и представляет собой пакет функций. На основе класса в программе строятся программные объекты в памяти компьютера, имитирующие реальные объекты. Это приводит к тому, что мы обращаемся к элементам пространства задачи и их представлениям в пространстве решения как к «объектам». Тогда программа может адаптироваться к трудностям задачи, создавая новые типы объектов по мере необходимости. Так возникла парадигма объектно-ориентированного программирования (ООП), которая позволяет описывать проблему в ее выражениях, а не в терминах компьютера, как было в предыдущих парадигмах программирования.

Основные идеи объектно-ориентированного подхода опираются на следующие положения:

- Программа представляет собой модель некоторого реального процесса, части реального мира;
- Модель реального мира или его части может быть описана как совокупность взаимодействующих между собой объектов;
- Объект описывается набором параметров, значения которых определяют состояние объекта, и набором операций, которые может выполняться с объектом;
- Взаимодействие между объектами осуществляется посылкой специальных сообщений от одного объекта к другому;
- Объекты, описанные одним и тем же набором параметров и способные выполнять один и тот же набор действий, представляет собой класс однотипных объектов.

Опыт программирования показывает, что любой методический подход в технологии программирования не должен применяться слепо с игнорированием других подходов. Это относится и к ООП. Существует ряд типовых проблем, для которых его полезность наиболее очевидна, к таким проблемам относятся, в частности, задачи имитационного моделирования, программирование диалогов с пользователем. Существуют и задачи, в которых применение объектного подхода ни к чему, кроме излишних затрат труда, не приведет.

Какая бы парадигма программирования не применялась текст программы кодируется по правилам структурированного кодирования структурного программирования.

Принципы ООП

Все языки объектно-ориентированного программирования обеспечивают механизмы, которые помогают вам реализовать объектно-ориентированную модель. К ним относятся *абстракция, инкапсуляция, наследование* и *полиморфизм.* Эти механизмы абстрагированы из реальных ситуаций, где человек использует их в своей деятельности.

Абстракция

Существенным элементом, объектно-ориентированного программирования является *абстракция*. Как известно, человек управляет сложностью через абстракцию. Люди обладают удивительной способностью абстрагирования. Например, люди не представляют себе автомобиль как набор десятков тысяч индивидуальных частей (деталей). В их воображении автомобиль — хорошо определенный *объект (абстрактный)* со своим собственным уникальным *поведением и характеристикой.* Люди познают объекты путем изучения их атрибутов и наблюдения за их поведением. Эта абстракция позволяет людям использовать автомобиль, не задумываясь над сложностью частей, из которых он состоит. Игнорируя подробности работы двигателя, трансмиссионной и тормозной системы, они могут свободно пользоваться объектом в целом. Все человеческое знание построено на абстракции. Аналогично все языки программирования построены на абстракции. Программисты рассматривают объекты двумя способами: как абстрактные объекты и как реальные объекты. Считайте, что абстрактный объект – это описание реального объекта минус подробности. Абстрактный объект используется как модель для реального объекта. Такой подход позволяет нам уделять внимание важным для нас подробностям и игнорировать те подробности, которые нам не интересны с точки зрения постановки проблемы.

Инкапсуляция

Реальный объект – это целостностная сущность. На практике набор однотипных объектов определяется как один класс. Логически *способ связывания* атрибутов и процедур для формирования целостного объекта есть нечто иное, как *инкапсуляция*. В объектно-ориентированном языке инкапсуляция – это группировка понятий или поведения в класс.

Класс является основой инкапсуляции в объектно-ориентированном языке Java. Она определяет, какие варианты поведения имеет класс, не уточняя, как это поведение реализовано. Класс определяет характеристики и поведение (данные и код) некоторого набора объектов. Инкапсуляция — это механизм, который связывает код вместе с обрабатываемыми им данными и сохраняет их в безопасности как от внешнего влияния, так от ошибочного использования. Можно представить инкапсуляцию *защитную оболочку,* которая предохраняет код и данные от произвольного доступа из других кодов, определенных *вне* этой оболочки. Доступ к коду и данным *внутри* оболочки строго контролируется через хорошо определенный *интерфейс*. ООП инкапсулирует данные (атрибуты, характеристики) и функции (способы поведения) в *объекты*, в так называемом *экземпляре класса*. Инкапсуляция позволяет программисту

реализовать проверку правильности использования атрибутов и процедур, поместив атрибуты и процедуры в класс, а затем в классе определив правила для управления доступом к ним.

Каждый объект заданного класса содержит как характеристики (данные) так и поведение, определяемые кодом (подпрограммой). По этой причине об объекте говорят как об *экземпляре класса.* Таким образом, *класс — это логи*ческая конструкция, а *объект* — это физическая конструкция, построенная компьютером в памяти по описанию класса. Объект – это экземпляр класса. Класс, по существу это тип данных, а объект – это переменная этого типа.

Любой компонент задачи, которую вы хотите решить, может быть взята в качестве программного объекта. Программный объект – это человек, место, вещь, понятие и, возможно, событие. Тогда текст программы на языке Java – это текст совокупности описания классов и использование экземпляров классов – программных объектов (просто объектов). Так как цель класса — инкапсуляция сложности, то существуют в языке специальные механизмы скрытия сложности реализации внутри класса.

Наследование

Наибольшая часть знаний человека становится управляемой только с помощью иерархических (т. е. организованных "сверху вниз") классификаций. Например, классификация по классам в биологии. Наследование есть процесс, с помощью которого один объект приобретает свойства другого объекта. Оно важно потому, что поддерживает концепцию, иерархической классификации. Наследование взаимодействует также и с инкапсуляцией. Если данный класс инкапсулирует некоторые атрибуты, то любой подкласс будет иметь те же атрибуты плюс атрибут, который он добавляет как часть своей специализации. Это ключевая концепция, которая позволяет объектно-ориентированным программам, расти по сложности *линейно,* а не *геометрически. Новый подкласс* наследует все атрибуты всех его предков.

Полиморфизм

На практике человека встречается случай, когда дается одна команда для выполнения целого набора действий: например, «сделать ремонт», но фактическая реализация этого действия зависит от природы ремонтируемого объекта. Программные объекты подобное могут осуществять с помощью механизма полиморфизма. Полиморфизм ("имеющий много форм") — свойство, которое позволяет использовать один интерфейс (одну команду) для общего вида действий. Полиморфизм означает, что метод с одним именем может осуществлять множество вариантов поведения (реализаций).

Итак, каждая Java-программа может быть объектно-ориентированной, т.е. она может строится с помощью механизмов абстракции, инкапсуляции, наследования и полиморфизма.

Язык Java

Java является упрощенной версией языка C++ с некоторыми добавками из других языков программирования. Из него убраны некоторые трудные для понимания и вряд ли кому-то по-настоящему нужные свойства.

Язык Java является объектно-ориентированным языком, и программирование на таком языке называется объектно-ориентированным программированием (ООП). Каждая Java-программа состоит из одного или нескольких классов. В тексте программы на основе этих классов строятся объекты (программные объекты) и выполнение программы может сводиться к взаимодействию таких объектов. Так что, Java-программа – это группа программных объектов, говорящих друг другу, что делать, посредством сообщений. Сообщение это запрос к объекту выполнить какую-то свою вариант поведения (т.е. подпрограмму). Тем не менее язык Java позволяет сочетать объектный подход с другими методологиями.

Язык Java начал разрабатываться в 1991 г. для системного программирования небольших электронных устройств. В 1995 г. это направление сменилось на работу с компьютерами всех размеров. Это изменение произошло благодаря популярности и разнообразию машин, на которых Интернет использовался.

Как известно, много различных типов CPU (Central processor unit, процессор) используются как контроллеры в различных электронных устройствах. Язык Java не зависит от платформы и может использоваться для создания программного обеспечения с целью внедрения в электронные устройства различных потребителей (типа микроволновых печей и дистанционных пультов управления). Java программа может выполняться на разных процессорах в отличающихся средах. Неприятности использования для этого процедурного языка (например, C) заключается в том, что он спроектирован так, чтобы компилироваться только для определенного адресата (платформы). Программы на нем не являются переносимыми - их необходимо каждый раз компилировать заново на каждой новой платформе. Кроме того, такой язык оставляет такие вещи, как размеры и форматы внутренних структур данных, на усмотрение разработчиков конкретной операционной среды - в Java же все они заранее строго определены и неизменны, ибо Java-программа везде выполняется всегда одной специальной программой.

А также, как известно многоплатформная среда Web предъявляет экстраординарные требования к программе, потому что та должна выполниться надежно в самых разнообразных системах.. Java делает, возможным создание кросс-платформных программ, компилируя в промежуточное представление, названное *байт-кодом Java*. Этот, код может интерпретироваться в любой системе Интернета, которая обеспечена *виртуальной Java-машиной* (JVM). JVM – - это интерпретатор байт-кода., это программа , которая должна быть установлена на компьютере, чтобы интерпретировать и выполнить скомпилированный байт-код Java-программы. Перевод программы Java в байт-код делает более простым ее выполнение в широком разнообразии сред Интернета. Единственное условие: JVM должен быть реализован для.каждой компьютерной платформы. Например, современные Web-браузеры снабжены такой JVM. Так как Java-программы работают под управлением JVM, они могут делать только то, что им позволяет JVM. Это не дает Java-программам делать потенциально вредные или опасные вещи пользователям.

Java стал основным языком разработки приложений для Интернет и интранет и программного обеспечения для устройств, которые обмениваются данными по сети. Например, новые бытовые устройства, платежные терминалы, автонавигационные системы или беспроводные устройства – сотовые телефоны, пейджеры и персональные цифровые помощники – могут оказаться связанными по так называемому беспроводному Интернету с помощью основанных на Java протоколов.

Итак, технология Java обеспечивает разработчиков основой для создания решений, при разработке которых не нужно задумываться об операционной системе и аппаратной платформе, на которых эти решения будут функционировать. Платформа Java 2 (Java Development Kit (JDK) 1.2), объявленной фирмой Sun в 1998 года, представляет собой физическую реализацию технологии Java имеет три вида редакции: J2EE – Java Enterprise Edition для серверов, J2SE - Java Standart Edition для персональных компьютеров, J2ME - Java Micro Edition для карманных устройств. Это связано тем, что для проектирования серверных систем, настольных компьютеров и небольших устройств надо использовать совершенно разные подходы, и это совершенно разные задачи. Мы будем знакомиться только с некоторыми возможностями J2SE.

Системы программирования на Java

Решив начать программировать на языке Java, вы первым делом должны установить на свой компьютер *компилятор* и *оболочку времени выполнения* для этого языка. Эти компоненты входят в состав комплекта разработчика Java (Java Development Kit, JDK) - пакета программ, который бесплатно распространяется фирмой Sun Microsystems.

Обычно системы программирования на Java состоит из нескольких частей: *среда разработки, язык, библиотека классов Java* или *программный интерфейс приложений Java (API Java, который кроме классов содержит еще интерфейсы*

Стандартное издание среды разработки программ на Java 2 (J2SDK - Java 2 Software Development Kit) включает *минимальный набор инструментальных средств*, необходимых для разработки программного обеспечения на Java. В таком простом средстве разработки взаимодействие пользователя с программой делается только через *командную строку*. Существуют среды разработки, которые обеспечивают взаимодействие через *графический пользовательский интерфейс*, как, например, Eclipse. J2SDK поставляется с исходными текстами классов API, чтобы разобраться, как эти классы работают, и освоить методы программирования, с которыми они, возможно, еще не знакомы. JVM реализована в J2SDK в интерпретаторе **java,** который транслирует байт-коды Java в машинный язык компьютерной платформы, на которой работает пользователь. Вы можете загрузить последние изменения J2SDK с Веб-сайта компании Sun **java.sun.com/j2se.** Sun Microsystems, Inc. предлагает мощную *интегрированную среду разработки* Java – Forte for Java, Community Edition, - которую вы можете загрузить с сайта **www.sun.com/forte/ffj.** Многие фирмы - поставщики программного обеспечения (в частности, Borland и Symantec) разрабатывают свои оболочки для программирования на Java.

Этапы разработки и выполнения Java-приложения

Для объектно-ориентированного программирования характерно следующие этапные работы:

- *Осуществление объектно-ориентированного анализа условий и требований задачи*. Делается моделирование в терминах задачи, т.е. формализованное описание задачи в ее пространстве, а не в терминах компьютера. Делается постановка задачи в виде совокупности действующих экземпляров классов - объектов. Анализом задачи формируется необходимый набор суперклассов, подклассов – моделей будущих программных объектов.
- *Осуществление объектно-ориентированного проектирования*. Проектирование программной системы на основе модели объектно-ориентированного анализа. Проектирование программы, как программной системы, в виде взаимодействующих программных объектов (экземпляров классов).

Таким образом, ООП – это процесс реализации программ, основанной на представлении программы в виде совокупности объектов некоторых классов. ООП предполагает, что любая подпрограмма (функция или процедура) в программе представляет собой метод объекта некоторого класса, причем класс должен формироваться в программе естественным образом, как только в программе возникает необходимость описания новых физических предметов или их абстрактных понятий (объектов программирования). Классы из предметной области непосредственно отражают понятия, которые использует конечный пользователь для описания своих задач и методов их решения. Каждый новый шаг в разработке алгоритма также должен представлять собой разработку нового класса на основе уже существующих классов, т.е. технология ООП иначе может быть названа как программирование «*от класса к классу*». Текст Java программы представляет собой совокупность текстов классов. Важно осознавать, что разработка программы является пошаговым, последовательным процессом, так же, как, например, обучение людей. Вы можете провести столько анализа и планирования, сколько вам заблагорассудится, но пока вы не приступите к делу, все до конца ясно не будет. У вас будет больше успехов – и больше немедленных результатов, - если вы начнете «выращивать» ваш проект в качестве органичного, эволюционирующего существа, а не сконструируете его сразу, как небоскреб из стеклянного каркаса. ООП позволяет описать проблему в ее выражениях (в виде набора объектов задачи (экземпляров классов), взаимодействующих между собой) и , а не в терминах компьютера, на котором будет исполнено решение.

Мы сейчас рассмотрим этапы работ с Java программой после выполнения этапа анализа условия и требований задачи и этапа проектирования. Программы Java обычно проходят пять стадий обработки, прежде чем они будут выполнены: *редактирование, компиляция, загрузка, проверка байт-кода* и *выполнение.* Мы обсуждаем эти концепции в контексте средства разработки *Java 2 Software Development Kit (J2SDK).*

Первый этап представляет собой *редактирование* файла. Эта задача выполняется при помощи *программы редактирования.* Программист вводит текст программы Java, используя простой редактор, а затем вносит необходимые исправления, если они потребуются. Когда программист решает, что редактируемый файл следует сохранить, программа сохраняется на устройстве внешней памяти, например жестком диске. Программный файл Java имеет расширение *.java.* В UNIX и Linux системах широко используются два редактора — vi и emacs. В Windows 95/98/ME и Windows NT/2000 подойдут самые простые программы редактирования, наподобие команды DOS Edit и редактора Windows *Notepad (Блокнот). Интегрированные среды разработки Java* (IDE), например, Forte (Forte for Java Community Edition), NetBeans, JBuilder компании Borland, Visual *Cafe* от Symantec и VisualAge от IBM имеют встроенные в среду программирования редакторы.

Установив значения переменных окружения, вы можете приступать к программированию на языке Java. Например, если вы поместили файлы дистрибутива в каталог C:\JAVA, вы должны установить переменную CLASSPATH, напечатав следующую команду в строке приглашения DOS: `C: SET CLASSPATH=.;C:\JAVA\LIB`

Эту команду имеет смысл поместить в файл AUTOEXEC.BAT. Кроме того, вам, вероятно, покажется удобным добавить каталог с исполняемыми файлами JDK в путь поиска, задаваемый командой PATH. Если вы установили JDK в каталог C:\JAVA, то все исполняемые файлы будут помещены в каталог C:\JAVA\BIN, который и нужно будет добавить к списку каталогов команды PATH в файле AUTOEXEC.BAT.

Предположим, что пользователь подготовил следующий исходный текст простой Java-программы на Блокноте и сохранил в виде файла:

```
/* Это простая Java-программа. Она состоит только из одного класса. При сохранении назовите
этот файл "Example.java". Файл исходного кода должен получить имя, совпадающее с именем класса,
в котором находиться метод с именем main   */

class Example {
  // Программа начинает выполняться с вызова метода main().
  public static void main (String args [])  {
   System.out.println{"Это простая Java-программа.");
  }
 }
```

На втором этапе делается *компиляция* исходного файла. Чтобы откомпилировать программу **Example.java**, вы должны открыть окно командной строки (выполнить команду cmd), переходя к каталогу, где сохранена программа и ввести команду

javac Example.java

Указанная выше команда javac переводит исходную Java-программу, сохраненную в Example.java, в байт-код Java. Если программа успешно откомпилируется, то компилятор создаст файл по имени **Example.class**. Этот файл содержит байт-коды. Если в исходном коде было несколько классов, то получаемый в процессе трансляции код для каждого класса записывается в отдельном выходном файле, с именем совпадающем с именем класса, и расширением **class**.

Java относится к частично компилируемым языкам. В отличие от "просто компилируемых" языков, компилятор Java не создает окончательно скомпилированный файл, готовый к запуску на компьютере. Вместо этого он создает файл, который может исполнять специальная система -

оболочка времени выполнения (JRE –java runtime environment). Эта оболочка содержит средства проверки кода, обеспечивающие надежность и защищенность программ, загрузчик классоа, который динамически загружает классы в процессе выполнения и *виртуальную машину (JVM),* которая выполняет последовательность байт-кодов, взаимодействуя с конкретной операционной системой. Особенность технологии Java - программа компилируется сразу в машинные команды, но не команды какого-то конкретного процессора, а в команды *JVM*. Другая особенность – все стандартные функции, вызываемые в программе, подключаются к ней только на этапе выполнения, а не включаются в байт-коды. Это означает, что вы можете написать и скомпилировать Java-программу на одной платформе, затем перенести ее на другую платформу и сразу же запустить без повторной компиляции. Байтовые коды представляют собой не что иное, как инструкции для оболочки времени выполнения. Программы, написанные на других языках, как правило, привязаны к конкретной операционной платформе, а программы на Java - нет.

На третьем этапе программа *Загрузчик классов* переносит байт-коды в оперативную память. Загрузчик классов может загружать файлы .class двух типов `Java`-программ — *приложений* и *апплетов.*

Приложениями обычно называют программы, находящиеся и выполняющиеся на локальном компьютере пользователя; речь о программах типа текстовый процессор, электронная таблица, графический редактор или просто программа пользователя.

Апплетами принято называть небольшие программы, которые обычно располагаются на удаленном компътере, с которым пользователи соединяются с помощью браузера. Апплеты загружаются с удаленного компьютера в браузер клиента, выполняются в браузере и удаляются после завершения своей работы.

На четвертом этапе программа *Верификатор байт-кодов* проверяет корректность всех байт-кодов и отсутствие нарушений требований Java к безопасности.

На пятом этапе программа *Интерпретатор* (*JVM)* читает байт-коды, транслирует их в понятный компьютеру язык и выполняет (интерпретирует) и, возможно, сохраняет данные, полученные при выполнении программы. Язык Java – интерпретирующий язык. Java-программы исполняются в виртуальной машине, размещенной внутри компьютера, на котором запущена программа. Java-программа не имеет никакого контакта с настоящим, физическим компьютером. Такой подход приводит к следующим особенностям Java:

Во-первых, как уже отмечалось выше, Java-программы не зависят от компьютерной платформы, на которой они исполняются. Когда вы напишете и скомпилируете Java-программу, она будет работать без изменений на любой платформе, где есть виртуальная машина. Другими словами, Java-программа всегда пишется только для единственной платформы – для виртуальной машины.

Во-вторых, виртуальная машина решает, что Java-программе позволено, а что делать нельзя. Поскольку Java-программы запускаются виртуальной машиной, ее разработчики и решают, что можно, а чего нельзя позволять делать программе. Виртуальная машина играет роль бастиона на пути между Java-программой и компьютером, на котором та выполняется. Java-программа никогда не сможет получить прямой доступ к устройствам ввода-вывода, файловой системе и даже памяти. Вместо Java-программы все это делает виртуальная машина.

Когда загружается и запускается апплет, виртуальная машина полностью запрещает ему доступ к файловой системе. Виртуальная машина может дать только косвенный доступ к избранным системным ресурсам - вот почему мы доверяем апплетам и знаем, что они не способны уничтожать файлы или распространять вирусы.

В третьих, оболочки времени выполнения Java (*JRE*) позволяет программе собираться по кусочкам прямо в процессе выполнения. Это практично, поскольку наиболее важные части программы можно постоянно хранить в памяти, а менее важные - загружать по мере необходимости. Java-программы умеют делать это, пользуясь механизмом "*динамического*

связывания" (dynamic binding). Если все ваши программы загружаются с жесткого диска быстрого компьютера, это свойство не так уж важно. Все меняется, как только вы начинаете загружать программу из Интернет. Здесь вступает в силу ограниченная скорость сетевого соединения. В этом случае Java-программа способна сперва загрузить часть, необходимую для начала работы, запуститься, а уж затем постепенно подгрузить оставшуюся часть. Как мы увидим ниже, динамическое связывание, кроме всего прочего, облегчает сопровождение Java-программ.

Платформой в программировании понимается совокупность аппаратных и (или) программных окружений, в котором работают приложения. Технология Java – это более чем язык, это платформа. Язык Java необычен тем, что программа одновременно и компилируется, и интерпретируется. Исходный текст программы компилируется в промежуточный байт-код, который является платформно-независимым и выполняется интерпретатором платформы Java. Если многие платформы являются аппаратно-программной, то платформа Java является чисто программной и состоит из двух частей: *JVM и Java API*. JVM и Java API изолируют Java-программу от аппаратного обеспечения компьютера и от проблем совместимости. Итак, если обычно компьютерные программы исполняются на конкретной аппаратно-программной платформе, то Java приложения исполняются в рамках платформно-независимой исполняющей среде.

Чтобы выполнить приложение Java по имени Example, т.е. чтобы запустить JVM для выполнения Example нужно выполнить команду

```
java Example
```

Эта команда запускает интерпретатор Java. Интерпретатор автоматически ищет файлы с расширением **.class** в текущем каталоге. После запуска указанной команды Example.class загружается в память, а интерпретатор вызывает метод **main** для начала выполнения программы.

Лекция 2. Лексемы языка Java

Java-программа это набор пробельных символов, идентификаторов, констант, ключевых слов, знаков операций, разделителей и комментариев. Можно сказать, они являются лексемами языка Java. Из таких лексем строятся предложения текста программы: *объявления* и *операторы*. Компилятор, рассматривая текст программы, просто как последовательность символов, последовательно выделяет и распознает в ней вышеназванные лексемы.

Пробельные символы

Java является *языком свободной формы*. Это означает, что, вам не нужно следовать каким бы то ни было специальным правилам расположения текста. Вы можете переносить текст на новую строку, где может стоять пробел. К пробельным символам относятся пробел (space), символ табуляции (tab) или символ новой строки. Пробельные символы в тексте программы отделяют одну лексему от другой.

Идентификаторы

Идентификаторы используются в качестве имен классов, методов и переменных. Идентификатор может быть любой последовательностью букв верхнего и нижнего регистров, чисел или символов подчеркивания и знака коммерческого $ (а также символы-буквы национальных алфавитов). Он не должен начинаться с цифры, чтобы не вступать в конфликт числовой константой и не должен содержать пробелы. Напомним также, что язык Java чувствителен к регистру, так что VALUE есть идентификатор, отличающийся от value.

Константы

Постоянные значения в Java создаются с использованием их *литерального* представления. Вот несколько констант:

100 98.6 'X' "This is a test"

Первая константа специфицирует целое число, следующая — числовое значение с плавающей точкой, третья — символ и последняя — строку. Константу можно использовать везде, где допустимо значение ее типа.

Целочисленные литералы

Любое полное числовое значение — целый литерал. Например, 3 и 42. Это все десятичные значения, т. е. числа с основанием 10. Имеются два других основания, которые можно использовать в целых литералах — восьмеричное (octal), с основанием 8, и шестнадцатеричное (hexadecimal), с основанием 16. Восьмеричные значения обозначены в Java ведущим нулем. Обычные десятичные числа не могут иметь ведущий нуль. Таким образом, казалось бы, правильное значение 09 даст ошибку компилятора, так как 9 — вне восьмеричного диапазона от 0 до 7. Более обычное основание для чисел, используемых программистами — шестнадцатеричное, которое четко согласуется с размерами слов по модулю 8, такими как 8, 16, 32 и 64 бита. Шестнадцатеричную константу обозначают с. ведущими нулями (Ох или ОХ). Для представления шестнадцатеричного значения используются цифры от 0 до 9 и буквы латинского алфавита от А до F или от а до f (для значений от 10 до 15).

Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения с дробным компонентом. Они могут быть выражены или в стандартной или научной (экспоненциальной) форме. Стандартная форма состоит из числового компонента за которым следует десятичная точка, а далее — дробная компонента. Например, 2.0, 3.14159 и 0.6667 представляют правильные числа с плавающей точкой в стандартной системе обозначений. Научная форма представления использует стандартные обозначения т- число с плавающей запятой плюс суффикс, который определяет степень 10, на которую должно быть умножено число. Экспонента обозначается буквой Е или е, за которой следует положительное или отрицательное десятичное число. Например: 6.022E23, 314159E-05 и 2e+100.

Булевы литералы

Имеются только два логических значения — true и false. значения true и false не конвертируются в какое-либо числовое представление. Литерал true в Java не равняется 1, а литерал false не равно 0. В Java они могут быть назначены только логическим переменным или использоваться в выражениях с булевскими операциями.

Символьные литералы

Все Java-программы кодируются с использованием Unicode, и все строки и одиночные символы, используемые в программах, хранятся в памяти в виде 16-битовых кодов. Это обусловлено тем, чтобы его могли с равным успехом использовать программисты, работающие не только на разных компьютерных платформах, но и живущие в разных странах и говорящие на разных языках.

Символы Java являются 16-разрядными значениями, которые можно преобразовывать в целые числа, можно манипулировать целочисленными операциями, такими как сложение и вычитание. Литеральный символ представляется внутри пары одиночных кавычек. Все видимые символы ASCII могут быть непосредственно введены внутри кавычек, например: 'я', ‘z’ и '@'. Для символов, которые невозможно ввести непосредственно существуют несколько escape-последовательностей, позволяющих ввести нужный символ, например, '\n'—для символа

newline. Имеется также механизм для прямого ввода значения символа в восьмеричном или шестнадцатеричном представлении. Для восьмеричной' формы используют обратный слэш (\), за которым следует число из трех цифр, Например, '\141' – символ 'a'. Для шестнадцатеричного представления нужно ввести обратный слэш с символом u (\u), затем четыре шестнадцатеричных цифры. Например, ' \uOO61' вводит символ 'a' набора ISO-LATIN-l, потому что старший байт нулевой. Чтобы задать константу-символ в программе, вы можете использовать как обычный символ, так и escape- последовательность для прямого указания кода в Unicode

Строковые литералы

Строковые литералы в Java определяются так же, как в большинстве других языков — включением последовательности символов между парой двойных кавычек. Примеры строковых литералов:

```
"Hello World"
"two\nlines"
"V'This is in quotesV"
```

Escape-последовательности и восьмеричные/шестнадцатеричные системы обозначений, которые были определены для символьных литералов, работают аналогичным образом и внутри строковых литералов. Обратите внимание на одну важную деталь: строковые литералы должны начинаться и заканчиваться на той же строке. Никакой escape-последовательности продолжения строки не существует, как в других языках.

Итак, в Java все строки и одиночные символы увеличиваются в размерах в два раза. Конечно, на первый взгляд это может показаться недостатком языка. Однако вспомните, что при разработке Java главной целью было вовсе не экономное расходование памяти, а эффективное программирование для Интернет и возможность создания переносимых программ. Кодировка Unicode позволяет специфицировать множество самых экзотических непечатаемых символов и букв иностранных алфавитов. То, что эти главные задачи успешно решены, позволяет мириться с несколько неэффективным расходом памяти для символьных значений.

Комментарии

В Java определены несколько типов комментариев.

1. Однострочный комментарий вида

```
// таков текст однострочного комментария
```

2. Многострочный комментарий вида

```
/*  таков текст многострочного комментария  */
```

Третий тип называется *документационный комментарий* (documentation comment). Этот тип комментария используется для производства HTML-файла, который документирует вашу программу. Документационный комментарий начинается с последовательности символов /** и заканчивается последовательностью */. Комментарий может стоять там, где может стоять знак пробела.

Разделители

В Java кроме пробельного символа существуют несколько символов, которые используются как разделители лексем. Чаще всего встречается точка с запятой (;). Он используется для завершения оператора. Этот символ в Java является неотъемлемой частью оператора. Разделителям еще относятся следующие знаки: круглые скобки, фигурные скобки, квадратные скобки, точка с запятой, запятая и точка. Роль разделителей также играют знаки операций. Все подобные знаки могут играть роль разделителя, поскольку за каждым знаком закреплена определенная функция, однозначно распазнаваемой компилятором и интерпретуемой JVM.

Ключевые слова языка Java

В языке Java используются несколько десятков, так называемых, ключевых слов, взятых из естественного языка. Смыслы ключевых слов одинаковы для человека, компилятора и

используются они для построения предложений текста Java-программы.. Ключевые слова нельзя использовать в качестве имен переменных, классов или методов

Структура Java-программы.

Из таких лексических единиц складываются предложения текста языка. Предложения Java называются *объявлениями* или *операторами*. А из таких предложений строятся модули (фрагменты) программы, называемые *классами*. Для записи класса используется ключевое слово **class**. *Класс* – это набор методов и данных, которые объединены в одной сущности. Он предоставляет *некоторое обслуживание* или выполняет *некоторую работу* и представляет собой *набор функций*. Из *текстов классов* составляется *текст Java программы.*

На практике принято из коды разработанных классов составлять *пакеты*, а из пакет образовывать *библиотеки классов*. Вы можете при разработке программы на Java использовать богатые коллекции существующих классов, представленных в *библиотеках классов Java*. Таким образом, при изучении Java, необходимо освоить две вещи. *Первое, это познакомиться с самим языком Java, чтобы вы могли создавать свои собственные классы, а второе, это научиться использовать огромное количество разработанных классов из библиотек классов Java.* Библиотеки классов поставляются поставщиками компиляторов, но многие из этих библиотек классов предлагаются и независимыми поставщиками программных продуктов. Кроме того, много библиотек классов доступны в Интернете, как свободно распространяемое программное обеспечение или условно-бесплатное программное обеспечение.

Типы Java программ

Программу, которая выполняется с использованием интерпретатора **java (**JVM), принято называть *приложением*. Один из классов *Java-приложения* содержит *метод* (подпрограмму) с названием **main**, с которого всегда начинается выполнение приложения.

Другой тип программы Java представлен *апплетом*. *Апплеты* – это небольшие Java-программы, которые доступны на Internet-сервере, транспортируются по Internet, автоматически устанавливаются и выполняются как часть Web-документа. В апплете метод **main** не используется, а используются другие методы.

Введение в объектно-ориентированный анализ и проектирование

При программировании сложных задач необходимо прежде всего детально проанализировать решаемую задачу и разработать проект, который бы соответствовал требованиям решаемой задачи. Сейчас при проектировании такой системы рекомендуется использовать *объектно-ориентированный подход*. Тогда такой процесс принято называть *объектно-ориентированным анализом и проектированием* (ООАП).

Небольшие задачи, подобные тем, что мы рассматриваем в этом курсе, не требуют применения методов ООАП. Для них достаточно написать псевдокод после небольшого анализа, а затем перенести его на Java. Псевдокод – это текстовое описание поведения, в котором используется комбинация слов естественного языка и синтаксис языка программирования. Но по мере того, как увеличивается сложность проблемы и увеличивается число людей, решающих ее, наступает черед применения методов ООАП. Большие программные системы, например, обслуживающие работу банкоматов, или системы управления воздушным движением, могут состоять из миллионов строк кода. Поэтому задача эффективного анализа и проектирования таких сложных систем имеет очень большое значение.

ООАП – это обобщенный термин для идей, лежащих в основе подхода, который используется для анализа проблемы и выработки метода ее решения. Опыт программирования задач показывает, что какой бы простой проблема не казалась, время, потраченное на анализ и проектирование, может существенно сэкономить время, которое вы неизбежно потратите на

этапе реализации, если плохо спланируете программную систему. Существует много методов ООАП, однако широкое использование для представления результатов проекта ООАП получил графический язык UML (Unified Modeling Language – унифицированный язык моделирования). UML – это универсальный язык моделирования. Он представляет собой набор диаграмм и стандартов текстового описания для определения процессов и классов, задействованных в процессах. Так что для овладения технологией ООАП необходимо знать язык UML.

Лекция 3. Типы данных, переменные, оператор присваивания и выражения.

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.

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

- *Целочисленные:* byte, short, int, long, char (также является целочисленным типом)
- *Дробные:* float, double
- *Булевский:* boolean

Это те типы, о которых компилятор знает все, что ему нужно знать без каких-либо предварительных объявлений или спецификаций. Любой элемент, принадлежащий к типу, определенному пользователем, может быть разложен на составляющие примитивных типов.

Вторую группу составляют *объектные*, или *ссылочные* (от английского reference), типы данных.

Переменные

Переменные используются в программе для хранения данных, ибо в программировании понятия «переменная» и «ячейка памяти» отождествляются. Любая переменная имеет характеристики:

имя; тип; значение**.**

Имя уникально идентифицирует переменную и позволяет обращаться к ней в программе. Имя переменной используется в программе для ссылки на содержимое соответствующего места в памяти. *Имя переменной* – это имя ячейки памяти, состоящей из определенного числа байтов. Количество байтов определяется согласно *типу* переменной. Работа с переменной всегда начинается с ее *объявления* (declaration).

О*бъявление* 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;
```

Память в Java с точки зрения программиста представляется *как некое виртуальное пространство*, в котором существуют объекты. И доступ в таком пространстве осуществляется не по физическому адресу, а лишь через ссылки на объекты. Ссылка возвращается при создании объекта и далее может быть сохранена в переменной, передана в качестве аргумента и т.д. Допускается наличие нескольких ссылок на один объект. Возможна и противоположная ситуация – когда на какой-то объект не существует ни одной ссылки. Такой объект уже недоступен программе и является "мусором", то есть без толку занимает аппаратные ресурсы. Для их освобождения не требуется никаких усилий. В состав любой виртуальной машины обязательно входит автоматический сборщик мусора garbage collector – фоновый процесс, который как раз и занимается уничтожением ненужных объектов.

Если примитивные типы в языке определены, то ссылочные – нет. Названия ссылочных типов определяются пользователями.

Примитивные типы

Можно сказать, *типы данных* – это *ключевое слово*, которое сообщает компьютеру, данные какого типа вы хотите хранить в памяти. Тип данных сообщает компьютеру, какое количество памяти следует зарезервировать и как обрабатывать данные после того, как они будут сохранены в некоторой области памяти. Тип данных также сообщает компьютеру вид данных, которые будут храниться в этом месте в памяти. Это важно, так как компьютеры манипулируют разными типами данных по-разному.

В языке Java размеры и форматы простых типов данных заранее строго определены и неизменны, ибо Java-программа во всех платформах запускается и выполняется только в одной среде JVM. В то время как в других языках программирования они могут определяться по-разному в разных операционных средах.

В Java новые, пользовательские типы данных определяются с помощью *классов* (class), *интерфейсов* (interface). Поэтому ссылочные типы (т.е. ссылки на объекты) отличаются от примитивных тем, что они не определены в самом языке Java, и поэтому количество памяти, которое требуется для переменных этих типов, заранее знать невозможно. Память для переменных ссылочного типа должна выделяться *во время выполнения программы*. Когда мы выделяем память для переменной ссылочного типа с помощью оператора **new**, то мы тем самым реализуем этот ссылочный тип. Массив также определяется с помощью **new.** Таким образом, каждая переменная ссылочного типа является реализаций или экземпляром типа.

Целочисленные типы

Целочисленные типы – это byte, short, int, long, также к ним относят и char. Первые четыре типа имеют длину 1, 2, 4 и 8 байт соответственно, длина char – 2 байта, это непосредственно следует из того, что все символы Java описываются стандартом Unicode. Длины типов приведены только для оценки областей значения. Как уже говорилось, память в Java представляется виртуальной и вычислить, сколько физических ресурсов займет та или иная переменная, так прямолинейно не получится. 4 основных типа являются знаковыми. char добавлен к целочисленным типам данных, так как с точки зрения JVM *символ* и *его код* – понятия взаимооднозначные. Конечно, код символа всегда положительный, поэтому char – единственный беззнаковый тип. Инициализировать его можно как символьным, так и целочисленным литералом. Во всем остальном char – полноценный числовой тип данных, который может участвовать, например, в арифметических действиях, операциях сравнения и т.п. В таблице 3.1 сведены данные по всем разобранным типам:

Таблица 3.1. Целочисленные типы данных.

Название типа	Длина (байты)	Область значений

byte	1	-128 .. 127
short	2	-32.768 .. 32.767
int	4	-2.147.483.648 .. 2.147.483.647
long	8	-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807 (примерно 10^{19})
char	2	'\u0000' .. '\uffff', или 0 .. 65.535

Обратите внимание, что int вмещает примерно 2 миллиарда, а потому подходит во многих случаях, когда не требуются сверхбольшие числа. Чтобы представить себе размеры типа long, укажем, что именно он используется в Java для отсчета времени. Как и во многих языках, время отсчитывается от 1 января 1970 года в миллисекундах. Так вот, вместимость long позволяет отсчитывать время на протяжении миллионов веков(!), причем как в будущее, так и в прошлое.

Почему были выделены именно эти два типа, int и long? Дело в то м что *целочисленные литералы* (константы) имеют тип int по умолчанию, или тип long, если стоит буква L или l. Именно поэтому корректным литералом считается только такое число, которое укладывается в 4 или 8 байт, соответственно. Иначе компилятор сочтет это ошибкой. Таким образом, следующие литералы являются корректными:

```
1   -2147483648     2147483648L     0L     111111111111111111L
```

Над целочисленными аргументами можно производить следующие операции:

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

Операторы сравнения вполне очевидны и отдельно мы их рассматривать не будем. Их результат всегда *булевского* типа (true или false).

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

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

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

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

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

Результатом такого примера будет:

```
-194313216
```

Возвращаясь к инвертированию числа -2147483648, мы видим, что математический результат равен в точности $+2^{31}$, или, в двоичном формате, 1000 0000 0000 0000 0000 0000 0000 0000 (единица и 31 ноль). Но тип int рассматривает первую единицу как знаковый бит, и результат получается равным -2147483648.

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

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

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

Если хотя бы один аргумент операции имеет тип long, то все аргументы приводятся к этому типу и результат операции также будет типа long. Вычисление будет произведено с точностью в 64 бита, а более старшие биты, если таковые появляются в результате, отбрасываются. Если же аргументов типа long нет, то вычисление производится с точностью в 32 бита, и все аргументы преобразуются в int (это относится к byte, short, char). Результат также имеет тип int. Все биты старше 32-го игнорируются. Никакого способа узнать, произошло ли переполнение, нет.

Отметим еще несколько примеров, которые не столь очевидны и могут создать проблемы при написании программ. Во-первых, подчеркнем, что результатом операции с целочисленными аргументами всегда является целое число. А значит, в следующем примере

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

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

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

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

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

```
604800000          -1702967296
```

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

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

Получаем правильный результат:

```
2592000000
```

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

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

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

Хотя для программиста и очевидно, что переменная b должна получить значение 1, что легко укладывается в тип byte, однако компилятор не может вычислять значение переменной x при обработке второй строки, он знает лишь, что ее тип – int. А вот менее очевидный пример:

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

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

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

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

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

Даже унарный оператор "-" проводит вычисления с точностью не меньше 32 бит. Хотя во всех случаях инициализация переменных приводилась только для примера, а предметом рассмотрения были числовые операции, укажем корректный способ преобразовать тип числового значения:

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

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

Первое из них – операторы инкрементации и декрементации. Их действие заключается в прибавлении или вычитании единицы из значения переменной, после чего результат сохраняется в этой переменной и значение всей операции равно значению переменной (до или после изменения, в зависимости от того, является оператор префиксным или постфиксным). А значит, и тип значения совпадает с типом переменной. (На самом деле, вычисления все равно производятся с точностью минимум 32 бита, однако при присвоении переменной результата его тип понижается.)

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

В результате получаем:

```
5  6  6  5
```

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

```
byte x=-128;
print(-x);
byte y=127;
print(++y);
```

Результатом будет:

```
128   -128
```

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

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

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

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

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

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

Результатом будет:

```
x=1
```

Обратите внимание на следующий пример:

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

Его результатом будет:

```
3text    text12
```

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

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

Переменная i получит значение 9.

Рассмотрим следующий пример:

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

Его результатом будет:

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

В первом случае в метод print было передано значение типа char, поэтому отобразился символ. Во втором случае был передан результат сложения, то есть число, и именно число появилось на экране. Далее при сложении со строкой тип char был преобразован в текст в виде символа. Наконец в последней строке произошло сложение трех чисел: 'c' (код 99), '=' (код 61) и переменной c (т.е. код 'A' - 65).

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

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

Дробные типы

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

Таблица 3.2. Дробные типы данных.

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

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

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

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

Над дробными аргументами можно производить следующие операции:

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

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

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

Такая свобода связана с наличием специальных значений дробного типа. Они определяются спецификацией IEEE 754:

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

Если получаемое значение слишком велико по модулю (*overflow*), то результатом будет бесконечность соответствующего знака.

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

В результате получаем:

```
Infinity
-Infinity
```

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

```
print(1e-40f/1e10f);   // underflow для float
print(-1e-300/1e100);  // underflow для double
float f=1e-6f;
print(f);
f+=0.002f;
```

```
print(f);
f+=3;
print(f);
f+=4000;
print(f);
```

Результатом будет:

```
0.0    -0.0    1.0E-6     0.002001     3.002001    4003.002
```

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

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

Еще раз рассмотрим простой пример:

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

Результатом будет:

```
0       0.5
```

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

Булевский тип

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

Над булевскими аргументами можно производить следующие операции:

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

В операторе с условием ? : первым аргументом может быть только значение типа boolean. Также допускается, чтобы второй и третий аргументы одновременно имели булевский тип. Операция конкатенации со строкой превращает булевскую величину в текст "true" или "false" в зависимости от значения. Только булевские выражения допускаются для управления потоком вычислений, например, в качестве критерия условного перехода if. Никакое число не может быть интерпретировано как булевское выражение. Если предполагается, что ненулевое значение эквивалентно истине (по правилам языка C), то необходимо записать `x.`

Примитивные типы данных Java переносимы на все компьютерные платформы, ибо они в них обрабатываются с помощью JVM. Все базовые типы по умолчанию инициализируются. К вещественным типам применимы все арифметические операции и операции сравнения. Потому целые и вещественные значения можно смешивать в операциях. Хотя тип char занимает 2 байта, в арифметическом выражении он участвует как тип int. Целые константы хранятся в формате int, а действительные константы хранятся в формате double. Итак, в Java есть переменные двух видов: переменные примитивного типа и переменные ссылочного типа.

Операции

Язык Java обеспечен богатым набором операций, большинство которых можно разделить на четыре группы: арифметические, поразрядные, отношений и логические. Знак операции представляет собой специальный символ и предназначен для выполнения какого-то действия над одной, двумя или более переменными (операндами).

Арифметические операции

Это операции сложения (+), вычитания (-), умножения (*), деление (/, например, 6/4=1 для целых или =1.5 для вещественных), остаток от деления по модулю (%, например, 42%10=2 для целых или 42.3%10=2.3 для вещественных), инкремент (++), присваивание со сложением (+=), присваивание с вычитанием (-=), присваивание с умножением (*=), присваивание с делением (/=), декремент (--).

Поразрядные операции

Это – отрицание (~), поразрядное И (&), поразрядное ИЛИ (|), поразрядное исключающее ИЛИ (^), сдвиг вправо (>>), сдвиг вправо с заполнением старшего бита нулем (>>>), сдвиг влево (<<), присваивание с поразрядным И (&=), присваивание с поразрядным ИЛИ (|=), присваивание с поразрядным исключающим ИЛИ (^=), присваивание со сдвигом вправо (>>=), присваивание со сдвигом вправо и заполнением старшего бита нулем (>>>=), присваивание со сдвигом влево (<<=).

Операции отношений

Это – равно (= =), неравно (!=), больше чем (>), меньше чем (<) , больше чем или равно (>=), меньше чем или равно (<=).

Логические операции

Это - логическое И (&), логическое ИЛИ (|), логическое исключающее ИЛИ (^), укороченное ИЛИ (||), укороченное И (&&), логическое отрицание (!), присваивание с И (&=), присваивание с ИЛИ (|=), присваивание с исключающим ИЛИ (^=), не равно (!=), троичная условная операция (?:).

Каждый знак операции имеет свой приоритет выполнения. Конструкция, построенная из знаков операций и операндов в языке программирования называется *выражением*. Вычисление выражения определяется набором правил приоритета.

Оператор присваивания

Вычисления значения выражения и сохранение значения выражения в памяти компьютера в языке программирования задается оператором присваивания, который имеет следующий синтаксис:

```
переменная = выражение;
```

Здесь тип переменной должен быть совместим с типом выражения. В Java допускается следующий вариант оператора присваивания:

```
int x, y, z;
x = y = z = 108;
```

Этот фрагмент устанавливает в переменные x, y, z значение 108.

Область действия и время жизни переменных.

Java позволяет объявлять переменные в пределах любого блока. *Блок* начинается с открывающей фигурной скобки и оканчивается закрывающей фигурной скобкой. Блок определяет область видимости (действия) имен переменных. Таким образом, каждый раз, когда запускается новый блок, создается новая область видимости: Область видимости определяет ту

часть программы, где имена декларированных (в данном блоке) объектов являются "видимыми" из других частей программы. Эта область определяет также "время (продолжительность) жизни" этих объектов.

Большинство машинных языков определяет две общих категории областей действия — *глобальную* и *локальную*. Однако эти традиционные области действия не совсем согласуются со строгой, объектно-ориентированной моделью Java. Хотя можно показать, что относится к глобальной области действия, но это скорее исключение, а не правило. В Java-программах можно выделить две основных области действия: одна определяется классом, а другая — методом. Однако даже такое различие несколько искусственно. Так как область действия класса (class scope) имеет несколько уникальных свойств и атрибутов, которые не применяются к области действия, определяемой методом, эта дискуссия имеет некоторый смысл. Пока мы будем рассматривать только область действия, определенную в пределах метода. Область действия, определенная методом, начинается с его открывающей фигурной скобки. Однако, если метод имеет параметры, они также включаются в его область действия.

Общее правило: переменные, объявленные внутри области действия, невидимы (т. е. недоступны) коду, который определен вне этой области, Таким образом, когда вы объявляете переменную в пределах области действия, вы локализуете эту переменную и защищаете ее от неправомочного доступа и\ или модификации. Действительно, правила области видимости обеспечивают основу для инкапсуляции. Области видимости могут быть вложены. Например, каждый раз, когда вы создаете блок кода, образуется новая, вложенная область действия. Когда это происходит, внешняя область действия включает внутреннюю. Это означает, что объекты, объявленные во внешней области, будут видимы и во внутренней. Однако, обратное — не верно. Объекты, объявленные во внутренней области действия, не будут видимы во внешней.

Внутри блока переменные можно объявлять в любой точке, но область их действия начинается только после того, как они объявлены. Таким образом, если вы определяете переменную в начале метода, она становится доступной всему коду в пределах этого метода. Наоборот, если вы объявляете переменную в конце блока, это совершенно бесполезно. При выполнении программы переменные создаются во время входа в их область действия и разрушаются при выходе из этой области. Это означает, что переменные не будет сохранять свое значение при выходе из области действия. Таким образом, время жизни переменной ограничено ее областью действия. Если объявление переменной включает инициализатор, то эта переменная будет повторно инициализироваться каждый раз при входе в блок, в котором она объявлена.

Преобразование и приведение типов

Если два типа совместимы, то Java (интерпретатор) выполнит преобразование автоматически. Например, всегда возможно назначить int–значение long–переменной, Когда один тип данных назначается переменной другого типа, будет иметь место автоматическое преобразование типов, если выполняются два следующих условия:

- Два типа совместимы;
- Целевой тип больше чем исходный.

Когда эти два условия выполняются, имеет место расширяющее преобразование. Например, тип int всегда достаточно большой, чтобы содержать все допустимые byte-значения. Для расширяющих преобразований числовые типы, включая целый и с плавающей тонкой, являются совместимыми друг с другом. Однако числовые типы не совместимы с char или boolean. Типы char и boolean не совместимы также и друг с другом. Java выполняет

автоматическое преобразование типов также и при сохранении литеральной целочисленной константы в переменных типа byte, short или long.

Преобразование несовместимых типов

Хотя автоматическое преобразование типов полезно, оно не удовлетворяет всем потребностям. Например, как быть, если вы захотите назначить **int**-значение byte-переменной? Это преобразование не будет выполнено автоматически, потому что тип byte меньше, чем int. Данный вид преобразования иногда называется *сужающим преобразованием* (narrowing conversion), т. к. при этом уменьшается количество бит, отведенных для хранения данных. Понятно, что при сужающем преобразовании вы можете потерять часть информации, содержащейся в числе, - либо изменив его значение, либо (в случае числа с плавающей точкой) уменьшив разрядность и тем самым точность представления.

Чтобы создавать преобразование между двумя несовместимыми типами, вы должны использовать *приведение* типов. *Приведение* (cast) — это и есть явное преобразование типов. Оно имеет общий формат:

(target-type) value

Здесь ***target-type*** определяет желаемый тип, к которому следует преобразовать указанное **value.** Например, следующий фрагмент приводит int к byte. Если целое значение больше, чем диапазон byte-типа (256), то оно будет редуцировано по модулю этого диапазона (до остатка от целочисленного деления этого значения наг 256).

```
int a;
byte b;
//….
b = (byte) a;
```

Если вы абсолютно уверены, что значение приводимого целого типа `int` будет всегда попадать в диапазон типа `byte`, то смело прибегайте к приведению типа; в противном случае вам нужно будет предусмотреть какие-то дополнительные действия (например, выдачу предупреждения пользователю о возможной потере данных). Язык Java заставляет вас расписаться в том, что вы отдаете себе отчет, что происходит при таком преобразовании: при любых сужающих преобразованиях вы должны прибегать к явному приведению типа.

Другой тип преобразований — *усечение* (trancation), произойдет, когда значение с плавающей точкой назначается целому типу. Как вы знаете, целые числа не имеют дробных частей. Таким образом, когда значение с плавающей точкой назначается целому типу, дробная часть теряется. Например, если значение 1.23 назначается целой переменной, результирующее значение будет просто 1, а 0.23 будет усечено. Если числовое значение слишком велико, чтобы вписаться в целевой целый тип, то оно будет редуцировано по модулю диапазона целевого типа.

Лекция 4. Введение в апплеты Java. Простой апплет: вывод текстовой строки

Апплеты – это небольшие программы Java, которые могут помещаться HTML-документы, т.е. Web-страницы. Внешне апплет выглядит как окно, в котором показываются результат работы программы. Когда браузер загружает Web-страницу, содержащую апплет, апплет загружается в браузер Web и начинает выполняться. Браузер выступает в данном случае в качестве контейнера апплета и испольняющей средой апплета. Комплект разработки J2SDK включает в себя другой контейнер апплета (визуализатор апплета) appletviewer, в котором удобно проводить тестирование апплетов, если вам не доступен Интернет-браузер, прежде чем вы будете размещать их в Web-страницах.

Мы начнем с простого апплета, который выводит сообщение.

```
// Файл: WelcomeAppLet.java
// Ваш первый апплет Java.
```

```
//  Базовые пакеты Java
import java.awt.Graphics;  //загрузка класса Graphics
//  Пакеты расширений Java
import javax.swing.JApplet;//загрузка класса JApplet
public class WelcomeAppLet extends Japplet
{
  // вывод текстовой строки в апплете
  public void paint( Graphics g ) {
    // вызов унаследованной версии метода paint
    super.paint( g );
    //вывод строки в точке с координатами x=25 и y=25
    g.drawstring(
    "Добро пожаловать в мир программирования!",25,25);
  }   // окончание метода paint
} // конец определения класса WelcomeApplet
```

Java имеется много предопределенных (ранее кем-то составленных) компонентов, называемых классами, которые сгруппированы в Java в пакеты. Строка

```
import java.awt.Graphics;  //загрузка класса Graphics
```

это оператор import, который указывает компилятору имя класса, которое нужно загрузить *(Graphics)* из пакета *java.awt,* и использовать в этом Java-апплете. Класс Graphics используется в апплетах Java для вывода графики типа линий, прямоугольников, овалов и строк символов. Этот класс также позволяет приложениям рисовать изображения. В строке

```
import javax.swing.JApplet;  //загрузка класса JApplet
```

в операторе import выполняется загрузка класса *JApplet* из пакета javax.swing. Когда вы создаете апплет в Java, то обычно импортируете класс JApplet. (*Заменание:* в пакете *java.applet* имеется более ранняя версия класса по имени Applet, который не используется с новыми компонентами графического интерфейса пользователя Java из пакета javax.swing. По этой причине мы используем класс JApplet для создания апплетов).

Как в случае приложений, каждый из апплетов Java, который вы создаете, содержит по крайней мере, одно определение класса. Одна из особенностей определения классов, состоит в том, что программисту редко приходится создавать определения классов «с нуля». Обычно, когда вы определяете класс, вы используете части определений существующих классов.

В Java. используется механизм *наследования* (о котором мы будем говорить позже в теме, посвященной объектно-ориентированному программированию) для создания новых классов на основе существующих определений классов. Со строки

```
public class WelcomeApplet extends JApplet {
```

начинается определение класса WelcomeApplet. В конце строки левая фигурная скобка ({) открывает тело определения класса. Соответствующая правая фигурная скобка (}) в конце текста обозначает конец определения класса. Определение класса начинается с ключевого слова class. WelcomeApplet — это имя класса. Ключевое слово *extends* указывает на то, что класс WelcomeApplet наследует существующий код другого класса. Класс, наследником которого является WelcomeApplet (т.е. JApplet), указывается справа от ключевого слова *extends.* В этих отношениях наследования класс JApplet называется *суперклассом* или *базовым классом,* а WelcomeApplet называется *подклассом* или *производным классом.* Использование наследования здесь приводит к тому, что класс WelcomeApplet будет иметь атрибуты (данные) и *поведение* (методы) класса JApplet, а кроме того, может иметь новые особенности, которые мы добавим в определение нашего класса WelcomeApplet (в частности, способность выводить строку Добро пожаловать в мир программирования! в окне апплета).

Класс JApplet обеспечивает возможности построения пустого окна апплета на экране, а наша программа просто дописывает указанную строку текста в этой заготовке. Так что программистам не нужно еще раз «изобретать велосипед», а именно заново писать программу построения пустого окна апплета. Указав всего лишь одно ключевое слово extends, вы можете

воспользоваться механизмом наследования и быстро создавать на основе класса JApplet новые апплеты.

Механизм наследования удобен; программист может не знать все детали устройства класса JApplet или любого другого суперкласса, от которого наследует новый класс. Программист должен знать только, что класс JApplet определяет возможности, необходимые для создания минимального апплета. Однако чтобы эффективно использовать возможности класса, программист должен освоить все возможности суперкласса. Для этого следует изучать документацию API Java и возможности тех классов, на основе которых вы создаете свои собственные подклассы. Это поможет вам избежать необходимости повторно «изобретать велосипед», переопределяя те возможности, которые уже заложены в суперклассе.

Классы используются как «шаблоны» или «проекты» *создания объектов,* которые могут использоваться в программе. Объект или *экземпляр класса* находится в памяти компьютера и содержит в себе информацию, используемую программой. Термин объект обычно подразумевает, что с ним связаны определенные атрибуты (данные) и способы поведения (методы). Методы объекта, используя атрибуты, обеспечивают функциональность, необходимую *клиенту объекта* (т.е. коду программы, которая вызывает методы объекта).

Когда контейнер апплета (appletviewer или браузер, в котором выполняется апплет) загружает наш класс WelcomeApplet, контейнер апплета создает объект класса (экземпляр класса) WelcomeApplet, который реализует атрибуты апплета и его поведение. (*Замечание:* термины объект и экземпляр являются взаимозаменяемы) Контейнеры апплетов могут создавать только объекты классов, которые являются открытыми (public) и являются расширениями класса JApplet. Таким образом, контейнеры апплетов требуют, чтобы определения классов апплетов начинались с ключевого слова public. В противном случае контейнер апплета не сможет загрузить и выполнить апплет. Когда вы сохраняете в файл класс, объявленный как public, имя файла должно совпадать с именем класса и иметь расширение имени файла **.java**. Для нашего примера файл апплета должен иметь имя WelcomeApplet.java. Пожалуйста, обратите внимание, что имена класса и файла должны быть идентичны, вплоть до соответствия регистра символов в именах.

В строке

```
public  void  paint( Graphics g )
```

начинается определение *метода* апплета *paint* — одного из трех методов (типов поведения*),* которые вызывает контейнер апплета, когда контейнер начинает выполнять апплет. По порядку, это три следующих метода — *init, start* и *paint.* Ваш класс апплета получает «бесплатные» версии каждого из этих методов от класса JApplet, когда вы указываете ключевые слова extends JApplet в первой строке определения вашего класса. Если вы не определяете свои версии этих методов в вашем апплете, то контейнер апплета вызывает версии методов, унаследованные от класса JApplet. Унаследованные версии методов init и start имеют пустые тела (т.е. они не содержат никаких операторов, так что они не выполняют никакой задачи), и унаследованная версия метода paint, также, ничего не выводит в окно апплета. Итак, оболочка времени выполнения сама вызывает требуемые методы апплета во время работы программы.

Итак, у апплетов нет метода main() – точки входа, с которой бы эта программа начинала выполнение. Вместо этого для написания апплета необходимо создать подкласс класса Japplet и заместить (переопределить) несколько стандартных методов. В нужное время, при возникновении определенных ситуаций, браузер или визуализатор апплетов вызывает эти опредленные вами методы . Апплет не может управлять вызвавшим его потоком исполнения, он просто отсылает ответ браузеру или визуализатору апплетов, когда он его об этом попросит. По этой причине созданные вами методы должны немедленно выполнять необходимые действия и возвращать управление. Им не разрешается входить в длительные циклы. Для выполнения повторяющихся действий, таких как воспроизведение анимации, апплет должен создать

собственный поток исполнения, которым он полностью управляет. Задача написания апплетов сводиться, таким образом, к переопределению наследованных методов: init, paint, mouseDown(), print() и др.

Чтобы наш апплет мог что-нибудь вывести на экран, нужно в классе WelcomeApplet *заместить (заменить* или *переопределить)* заданную по умолчанию версию метода paint, поместив в теле метода paint операторы, которые выводят на экран текст сообщения. Когда контейнер апплета даст команду апплету «изобразить себя на экране», вызвав метод paint, вместо пустого экрана мы увидим наше сообщение: Добро пожаловать в мир программирования!.

В круглых скобках, следующим за методом paint, определяется *список параметров* метода. В списке параметров указываются те данные, которые метод получает и использует для выполнения своей задачи. Обычно эти данные определяет программист и передает их методу в *вызове метода* (когда происходит *инициирование метода* или *отправка сообщения).* Например, раньше мы передавали методу showMessageDialog класса_JOptionPane следующие данные — отображаемое сообщение и тип диалогового окна. *Однако при написании апплетов, программист не вызывает метод paint явным образом (аналогичное делается при создании графического пользовательского интерфейса(см лекцию 16))*. Эту задачу, вызов метода paint, берет на себя контейнер апплета (JVM), который сообщает апплету (приложению), что он должен вывести изображение, и контейнер апплета (JVM) передает методу paint данные, которые требуются ему для выполнения задачи, а именно объект класса Graphics, на который ссылается g. (Класс Graphics управляет *графическим контекстом*, который автоматически формируется при создании графического компонента, т.е. объекта класса Component (см. лек. 16). *В контексте размещается область рисования и вывода текста и изображений*. При запуске апплета также создается его графический контекст. Поскольку контекст сильно зависит от конкретной графической платформы, создание объекта Graphics поручается JVM или контейнеру апплета). В задачу контейнера апплета входит создание объекта Graphics и ссылки g на этот объект. Метод paint использует объект Graphics для вывода графики в окне апплета. Ключевое слово public в начале строки требуется для того, чтобы контейнер апплета смог вызвать метод paint. По этой причине все определения методов должны начинаться с ключевого слова public.

В строке

```
super.paint( g );
```

вызывается версия метода paint, унаследованная от суперкласса JApplet. Этот оператор должен быть первым оператором метода paint в каждом апплете. Хотя простые апплеты могут работать без этого оператора, отсутствие этого оператора может привести к ошибкам в более сложных в, которых используются компоненты графического интерфейса пользователя Мы, включили этот оператор сейчас, чтобы у вас вырабатывалась привычка к использованию этого оператора, что поможет вам сэкономить время и избежать ошибок, когда вы позже перейдете к составлению более сложных апплетов.

В строке

```
g.drawstring(
  "Добро пожаловать в мир программирования!",25,25);
```

используется оператор, который заставляет компьютер выполнить действие (или задачу), а именно, вывести в апплете символы строки Добро пожаловать в.мир программирования!. В этом операторе.используется метод drawString, определенный в классе Graphics (в этом .классе определяются все графические возможноности строк символов и геометрических фигур типа прямоугольников, овалов и линий). В операторе метод dtawString вызывается с использованием объекта.g класса Graphics (указанного в списке параметров метода paint) и операции точки (.), за которой следует имя метода drawString. За именем метода следует список параметров, заключенный в круглые скобки, которые необходимы методу drawString, чтобы он смог выполнить свою задачу. Первый параметр метода drawString это объект String, — строка, которая будет выводиться в апплет. Последующие;два параметра в списке — числа 25 и 25 – координаты x, y *в* апплете *(или позиция)* левого верхнего угла прямоугольника, в котором должна быть выведена строка.

Для того чтобы отобразить апплет, нам нужен ссылающийся на него HTML-файл (см. Lab4). С помощью такого HTML-файл, содержащего ссылку на апплет, вы можете просмотреть апплет в визуализаторе апплетов или браузере.

Лекция 5. Управление последовательностью действий

Управление ходом программы

Управление потоком вычислений является фундаментальной основой всего языка программирования. В данной лекции будут рассмотрены основные языковые конструкции управления последовательностью действий и способы их применения. 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.

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).

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.

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) оператор или блок 1
else оператор 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 оператор или блок 1, в противном случае - оператор 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(выражение) {
   case value1:
      последовательность операторов
      break;
   case value2:
      последовательность операторов
      break;
    ……
   case valuen:
      последовательность операторов
      break;
   default:
       последовательность операторов
   }
```

Причем, фраза default не является обязательной.

Здесь выражение должно иметь тип byte, short, int, char. Каждое value должно иметь тип, совместимый с типом выражения. Каждое значение case должно быть уникальной константой (а не переменной). Дублирование значений case недопустимо.

При выполнении оператора switch производится последовательное сравнение значения выражения с константами, указанными после case, и в случае со падения выполняется последовательность операторов следующего за этим условием. Если ни одна из case–констант не соответствует значению выражения, то выполняется оператор default. Если согласующихся case нет, и default не присутствует, то никаких дальнейших действий не выполняется. Оператор break

используется, чтобы закончить последовательность операторов. Когда встречается оператор break, выполнение передается к первой строке кода, которая следует за полным оператором switch. Он создает эффект досрочного выхода из switch.

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

После оператора case должен следовать литерал, который может быть интерпретирован как 32-битовое целое значение. Здесь не могут применяться выражения и переменные, если они не являются final static.

Рассмотрим пример:

```
int x = 2;
switch(x) {
   case 1:
   case 2:
      System.out.println("Равно 1 или 2");
      break;
   case 3:
   case 4:
      System.out.println("Равно 3 или 4");
      break;
   default:
      System.out.println(
       "Значение не определено");
}
```

В данном случае на консоль будет выведен результат "Равно 1 или 2". Если же убрать операторы break, то будут выведены все три строки.

Вот такая конструкция вызовет ошибку времени компиляции.

```
int x = 5;
switch(x) {
   case y:   // только константы!
      ...
      break;
}
```

В операторе switch не может быть двух case с одинаковыми значениями.

Т.е. конструкция

```
switch(x) {
   case 1:
   System.out.println("One");
   break;
   case 1:
   System.out.println("Two");
   break;
   case 3:
   System.out.println("Tree or other value");
 }
```

недопустима.

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

Управление циклами

В языке Java имеется три основных конструкции управления циклами:

- цикл while;
- цикл do;
- цикл for.

Цикл while

Основная форма цикла while может быть представлена так:

while(логическое выражение)
повторяющееся действие или блок;

В данной языковой конструкции повторяющееся действие, 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 действие 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 действие 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){}. Возможно также расширенное использование синтаксиса оператора for(). Предложение и выражение могут состоять из нескольких частей, разделенных запятыми.

```
for(i = 0, j = 0; i<5;i++, j+=2) {
   ...
}
```

Использование такой конструкции вполне правомерно.

Операторы break и continue

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

Оператор continue

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

```
int x = (int)(Math.random()*10);
int arr[10]  = {....}
for(int cnt=0;cnt<10;cnt++) {
   if(arr[cnt] == x) continue;
   ...
}
```

В данном случае, если в массиве arr встретится значение, равное x, то выполнится оператор continue и все операторы до конца блока будут пропущены, а управление будет передано на начало цикла.

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:

```
for(int i=0;i<5;i++) {
   for(int j=0;j<4; j++) {
      ...
      if(j ==3) 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*:

```
outerLoop: for(int i=0;i<5;i++) {
   for(int j=0;j<4; j++){
      ...
      if(j == 3)
        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.

Лекция 6. Массивы и строки.

Массив - это набор однотипных переменных, на которые ссылаются по общему имени. Массивы можно создавать из элементов любого типа, и они могут иметь одно или несколько измерений. К определенному элементу в массиве обращаются по его индексу (номеру). Массивы предлагают удобные средства группировки связанной информации.

Одномерные массивы

Одномерный массив — это, по существу, список однотипных переменных. Чтобы создать массив, сначала следует создать переменную массива. желательного типа. Общий формат объявления одномерного массива:

Тип имяМассива[ ];

Здесь *Тип* объявляет базовый тип массива; *имяМассива* — имя переменной массива. Базовый тип определяет тип данных каждого элемента массива, Например, объявление одномерного массива int-компонентов с именем nedeli_days имеет вид:

```
int nedeli_days[ ] ;
```

Хотя это объявление и устанавливает факт, что nedeli_days является переменной массива, никакой массив в действительности не существует. Фактически, значение nedeli_days установлено в null (пустой указатель). Чтобы связать nedeli_days с фактическим, физическим массивом целых чисел, нужно выделить память для него, используя операцию new, и назначать ее массиву nedeli_days. *new* — это специальная операция, которая распределяет память. Подробнее операция new будет рассмотрена дальше, сейчас же она нужна для выделения памяти под массив. Общий формат new в применении к одномерным массивам имеет вид:

имяМассива = new Тип [size];

где *Тип* — тип распределяемых данных, *size* — число элементов в массиве, *имяМассива* — переменная, которая связана с массивом. Чтобы использовать new для распределения памяти под массив, нужно специфицировать тип и число элементов массива. Элементы в массиве, выделенные операцией new, будут автоматически инициализированы нулями. Следующий пример распределяет память для 7-элементного массива целых чисел и связывает его с переменной nedeli_days.

```
nedeli_days = new int[7];
```

После того как эта инструкция выполнится, nedeli_days будет ссылаться на массив из семи целых чисел. Затем все элементы в массиве будут инициализированы нулями.

Процесс получения массива включает два шага. Во-первых, следует объявить переменную массива желательного типа. Во-вторых, необходимо выделить память, которая будет содержать массив, используя операцию new, и назначать ее переменной массива. Таким образом, в Java все

массивы являются динамически распределяемыми, т.е. память элементам массива делается во время работы программы.

Как только вы выделили память для массива, можно обращаться к определенному элементу в нем, указывая в квадратных скобках индекс. Нумерация элементов массива начинается с нуля. Например, следующий оператор присваивает значение 2 второму элементу массива nedeli_days. month_days[l] =2;

```
// Пример использования одномерного массива
public class Array {
   public static void main ( String args[ ] )  {
       int nedeli_days[ ];
     nedeli_days = new int [7];
     nedeli_days[0] = 1;
     nedeli_days[1] = 2;
    nedeli_days[2] = 3;
    nedeli_days[3] = 4;
    nedeli_days[4] = 5;
    nedeli_days[5] = 6;
    nedeli_days[6] = 7;
       System.out.println( “nedeli_days[2] ” + nedeli_days[2] +
                          “-j den”);
   }
}
```

Возможна комбинация объявления переменной типа массив с выделением памяти непосредственно в объявлении:

```
      int  nedeli_days[ ] = new  int [7];
```

Массивы можно инициализировать во время их объявления. Процесс во многом аналогичен тому, что используется при инициализации простых типов. *Инициализатор массива* — это список разделенных запятыми выражений, окруженный фигурными скобками. Массив будет автоматически создаваться достаточно большим, чтобы содержать столько элементов, сколько вы определяете в инициализаторе массива. Нет необходимости использовать операцию new.

```
// улучшенная версия предыдущей программы
public class Array1 {
   public static void main ( String args[ ] )  {
       int nedeli_days[ ] = { 1, 2, 3, 4, 5, 6, 7 };
       System.out.println( “nedeli_days[2] ” + nedeli_days[2]  +   “-j den”);
   }
}
```

Java делает строгие проверки, чтобы удостовериться, что вы случайно не пробуете сохранять или читать значения вне области хранения массива. Исполнительная система Java тоже делает тщательные проверки, чтобы убеждаться, что все индексы массивов находятся в правильном диапазоне. Если вы попробуете обратиться к элементам вне диапазона массива (индекс меньше нуля или больше, чем длина массива), то вызовете ошибку времени выполнения.

Имеется еще один пример, который использует одномерный массив. Он находит среднее значение набора чисел.

```
// Среднее значение элементов массива
class Average {
public static void main(String args[])   {
  double nums[]={10.1,11.2,12.3,13.4};
  double result =0;
  int  i;
  for(i=0; i<5
      result = result + nums[i];
      System.out.println("Среднее арифметическое равно " + result / 4);
  }
}
```

Задание. Измените текст программы так, чтобы вычисление среднего значения делалось вне оператора вывода.

Многомерные массивы

В Java многомерные массивы — это, фактически, массивы массивов. Они выглядят и действуют подобно регулярным многомерным массивам. Однако имеется пара тонких различий. Чтобы объявить многомерную переменную массива, определите каждый дополнительный индекс, используя другой набор квадратных скобок. Например; следующее утверждение объявляет переменную двумерного массива с именем twoD:

```
int twoD[] [] = new int[4] [5];
```

Оно распределяет память для массива 4x5 и назначает ее переменной twoD. Внутренне эта матрица реализована как массив массивов целых чисел тип int.

Следующая программа нумерует каждый элемент в массиве слева направо, сверху вниз и затем отображает эти значения:

```
//Демонстрирует двумерный массив.
Class TwoDArray {
public static void main(String args[])  {
   int twoD[][] =  new int [4] [5];
   int i, j, k = 0;
   for ( I = 0; I<4; I++)
      for (j=0; j<5; j++) {
           twoD[i][j] = k;
           k++;
      }
      for ( I = 0; I<4; I++)
          for (j=0; j<5; j++) {
              System.out.ptintln(twoD[i][j] + “”);
              System.out.ptintln();
          }
   }
  }
```

Когда вы распределяете память для многомерного массива, нужно специфицировать только первое (крайнее левое) измерение. Остающиеся измерения можно распределять отдельно. Например, в следующем тексте память распределяется только для первого измерения переменной twoD (когда она объявляется). Распределение для второго измерения выполняется вручную:

```
int twoD[][] = new int[4][];                          -
 twoD[0] = new int[5];                                 .
 twoD[1] = new int[5];
 twoD[2] = new int[5];
 twoD[3] = new int[5];
```

Многомерные массивы возможно инициализировать. Для этого просто включают инициализатор каждого измерения в его собственный набор фигурных скобок. Например, так:

```
  double m[ ] [ ] = {
        {1, 2, 3 },
        {2, 3, 4 },
        {3, 4, 5 }
 }
```

Существует альтернативная форма, которая может использоваться для объявления массива:
Например, следующие два объявления эквивалентны:

```
   int a1[] = new int[3];
   int[] a1 = new int[3];
```

Строки

В языке программирования используются предопределенные примитивные типы и типы данных, определяемые пользователем. Строки Java особый тип, который представляет собой гибрид этих двух типов, - это тип String (тип строковых переменных). В основе своей тип String является типом, определенным пользователем, так как он определяется как одноименный класс String, содержащий в себе методы и переменные. Но в то же время этот тип проявляет некоторые свойства примитивного типа, что выражается, в частности, в том, как осуществляется присвоение значений переменным этого типа.

```
String myString="Hello!";
```

Строчный тип Java, называемый *String*, — не просто массив символов (как строки в C/C++). Скорее, *String* определяет объект и полное описание этого требует понимания нескольких объектно-ориентированных свойств. Однако, для того чтобы вы могли использовать простые строки в примерах программ, сделаем следующее краткое введение. Тип *String* используется, чтобы объявлять строковые переменные. Вы можете также объявлять массивы строк, *String*-переменной может быть назначена строковая константа (строка символов в двойных кавычках).

Переменная типа *String* может быть назначена другой переменной типа *String*. Разрешено использовать объект типа *String* как аргумент в методе . Например, рассмотрим следующий фрагмент:

```
String str = "Эта тест";
System.out.println(str);
```

Здесь str — объект типа *String*. Ему назначена строка "Это тест". Она выводится на экран методом println(). Как вы увидете позже, объекты типа *String* имеют много специальных свойств и атрибутов, которые делают их весьма мощными и удобными в использовании.

В Java имеется версия операции + для *конкатенации строк,* которая позволяет объединять объект типа *String* и значение экземпляра другого типа данных (включая тип String). В результате выполнения этой операции получается новый объект типа String (обычно большей длины).

Рассмотрим следующий пример:

```
JOptionPane.showMessageDialog(
              null, "Сумма равна " + sum,      "Ответ:", JOptionPane.PLAIN_MESSAGE );
```

Если мы предположим, что переменная sum содержит целочисленное значение 117, то выражение

"Сумма равна " + sum

будет вычисляться следующим образом:

1. Java обнаруживает, что два операнда операции + — строка "Сумма равна " и целочисленная переменная sum — относятся к различным типам данных, и один из них — это тип String.
2. Java преобразовывает *sum* в тип String.
3. Java добавляет строковое представление переменной *sum* в конец строки "Сумма равна " и в результате будет получена строка "**Сумма равна 117**".

Операцию +, используемую для конкатенации строк, легко перепутать с арифметической операцией +, что может привести к неожиданным результатам. Например, предположим, что целая переменная y содержит значение 5; тогда в результате вычисления выражения "y + 2 = " + y + 2 будет получена строка "y + 2 = 52" а не строка "y + 2 = 7", как можно было бы ожидать. Произошло это потому, что вначале было выполнено объединение значения переменной y и строки "y + 2 = ", в результате которого была получена строка "y + 2 = 5". *Затем эта новая строка была объединена с константой целого типа 2. Чтобы получить правильный результат, в этом примере нужно использовать запись "y + 2 = " + (y + 2).*

Разбор параметров команды командной строки

1-вариант. Следующая программа отображает в окне консоли аргументы командной строки метода **main().** Аргументы представляют массив строк, разделенных пробелами, значения которых присваиваются объектам массива **String[] args**. Объекту **args[0]** присваивается значение первой строки и т.д. Количество аргументов определятся значением **args.length**.

```
/* пример # 2 : вывод аргументов командной строки : OutArgs.java */
public class OutArgs {
   public static void main(String[] args) {
      for (int j = 0; j < args.length; j++)
         System.out.println("Аргумент #" + j + "-> " + args[j]);
   }
}
```

Запуск этого приложения с помощью следующей команды, набранной в командной строке, для выполнения:

```
java OutArgs  2003  argument-2  "Java string"
```

приведет к выводу на консоль следующей информации:

```
Аргумент #0-> 2003
Аргумент #1-> argument-2
Аргумент #2-> Java string
```

Аргументы командной строки могут быть использованы как один из способов ввода строковых данных.

2-вариант. Вот как выглядит другой вариант метода main, который печатает на выходе все, что передается ему в командной строке.

```
public class PrintCommandLineParameters {
 public final static void main(String S[] ) {
  System.out.println("Hello, Java!");
  System.out.println("Here is what was passed to me:");
  for(int i=0;i.length;i++)
   System.out.println(S[i]);
 }
}
```

Наша программа теперь будет печатать на выходе все переданные ей параметры командной строки. Например, если вы запустите эту программу такой командой:

java PrintCommandLineParameters parameter1 parameter2 parameter3 parameter4

то на выходе вы получите следующее:

```
Hello, Java!
Here is what was passed to me:
parameter1
parameter2
parameter3
parameter4
```

Лекция 7. Программные модули Java: методы.

Методы.

Решение сложной задачи делается по методике, которая следует известному принципу *разделяй и властвуй*. Согласно этому принципу, решаемая задача разбивается на подзадачи. Программы решения подзадач оформляются в виде отдельных программных модулей. Модули в Java называются *методами* (в других языках подпрограммами-функциями) и *классами*. Это позволяет программисту для решения своей задачи использовать кем-то ранее составленные модули. Поэтому программы на Java пишутся на основе *новых методов и классов*, созданных программистом, и предопределенных, упакованных в *пакеты*, методов и классов API Java, часто называемых *библиотеками классов Java*; а также *методов из других библиотек методов и классов.*

В API содержится богатая коллекция классов и методов, используемых для выполнения стандартных математических вычислений, обработки строк, символов, операций ввода и вывода, обработки ошибок и для выполнения многих других полезных действий. Эти модули существенно упрощают задачу программиста, потому что эти модули позволяют решить многие задачи, с которыми встречаются программисты. Методы API Java являются составной частью средства разработки Java 2 Software Development Kit (J2SDK).

Программист может написать методы для выполнения некоторых задач, которые будут использоваться в программе в нескольких местах,. Такие методы иногда называют *методами, определенными программистом*. Операторы, определяющие метод, записываются только один раз. Эти операторы скрыты от других методов.

Формат определения метода имеет вид

```
ТипВозвращаемогоЗначения  имяМетода(список параметров)
{
  объявления и операторы
}
```

Считайте, что метод – это группа операторов, которые осуществляют некоторое поведение и определяются следующим образом:

- Название метода
- Список параметров метода
- Тело метода
- Возвращаемое значение

Имя метода, как и имя переменной в Java, начинается со *строчной буквы*, а если имя состоит из нескольких слов, то каждое следующее слово начинается с заглавной буквы. Название метода должно отражать тип поведения, который осуществляет метод. Например, dropCourse – это хорошее название метода, который удаляет курс из расписания студента.

Список параметров – это список данных, находящиеся вне определения метода, которые необходимы методу для осуществления поведения. Методу они даются в виде списка аргументов в операторе вызова метода. Тело метода расписывается в терминах списка параметров. Список параметров – это список пар: объявлений и имен параметров, разделенные запятыми. В программировании список параметров еще называется списком формальных параметров. Некоторым методам для осуществления поведения не требуется данные извне, другим требуются данные извне.

Тело метода – это часть метода, содержащая операторы, которые выполняются, когда вызывается метод. Тело метода – это блок. Операторы выполняются в теле метода последовательно, начиная с первого оператора.

Методы *выполняются* (т.е. выполняют возложенные на них задачи) путем *вызова метода*. В вызове метода задается имя метода и предоставляется информация (так называемом в виде *аргументов*), которая необходима вызываемому методу для выполнения своей задачи. Аргументами в вызове метода могут быть *константы, переменные* или *выражения*. Аргументы, передаваемые в метод, должны соответствовать типу параметров в объявлении метода, должны следовать в том же порядке и по числу совпадать с числом параметров. Если метод не получает каких-либо значений, список параметров должен быть пустым.

Когда вызов метода завершается, метод либо возвращает результат своего выполнения *вызвавшему методу*, либо просто возвращает управление вызывающему методу. Существует три способа возврата управления в точку, из которой был вызван метод. Если метод не должен возвращать результат, управление возвращается или при достижении правой фигурной скобки, завершающей определение метода, или при выполнении оператора

```
return;
```

Если метод должен возвращать результат, то оператор

```
return выражение;
```

вычисляет значение выражения и возвращает значение выражения оператору, вызвавшему этот метод. В результате выполнения оператора return управление передается оператору, следующему сразу за оператором вызова метода.

В тексте тела метода кроме формальных параметров могут использоваться переменные, объявленные в этом тексте. Переменные, объявленные а определениях метода называются *локальными переменными*. Методы не знают деталей реализации других методов (включая их локальные переменные).

Как мы знаем, аргументы в пользу модульного метода построения программ состоят в том, что процесс разработки программы становится более управляемым, а также появляется возможность повторного использования программного кода – применение существующих методов в качестве строительных блоков при создании новых программ. Если методы хорошо продуманы и описаны, то программы можно создавать из таких стандартных методов, а не писать каждый раз новый код. Еще один аргумент в пользу модульного подхода – он позволяет избежать

повторения кода в программе. Оформление фрагмента кода в виде метода дает возможность выполнять этот код в различных местах программы путем простого вызова метода.

Каждый метод должен быть ограничен выполнением одной корректно поставленной задачи, а имя метода должно выражать суть этой задачи. Такой подход способствует многократному использованию программного кода.

Если вы затрудняетесь в выборе краткого имени, отражающего назначение метода, то, возможно, ваш метод выполняет слишком много различных задач. Лучше будет разбить его на несколько методов меньшего размера.

Определение метода

Программы, которые мы рассматривали ранее в этой книге, состояли из определения класса, содержащего, по крайней мере, определение одного метода, в котором вызываются методы Java API для выполнения решаемой программой задачи. Теперь мы рассмотрим случай, когда программист разрабатывает свои собственные методы. Пока мы рассмотрим апплет, в котором используется метод **square** (вызываемый из метода апплета **init)** для вычисления значений квадратов целых чисел от 1 до 10.

```
// Файл: Squarelntegers. java
// Метод square - разработанный пользователем метод
// Базовые пакеты Java
import java.awt.Container;
// Пакеты расширений Java
import javax.swing.*;
public class Squarelntegers extends JApplet {
// Определение GUI-компонента и расчет квадратов  чисел от 1 до 10
   public void init() {
      // Объект JTextArea будет использоваться для вывода результата
      JTextArea outputArea = new JTextArea();
      // Определение контейнера апплета
      Container container = getContentPane();
      // Связывание outputArea с container
      container.add( outputArea );
       int  result;
       // В переменной сохраняется результат вызова метода square
       String output = “”;
      //  Строковое представление результатов
      // Цикл из  10 итераций
          for (int counter = 1; counter<=10; counter++)  {
           // Расчет квадратного значения переменной  counter
           result = square(counter);                              '
           // Значение  result объединяется со  строкой output
           output += "Квадрат значения " +  counter + " равен " + result + ''\n";
          }    // окончание структуры for
       outputArea.setText(  output  );
       // результаты помещаются в объект JTextArea
    }    // окончание метода init

//  определение  метода  square
public int square( int y )
  {
     return y*y;//возвращается квадратное значение y
   } //окончание метода square

} // конец класса SquareIntegers
```

Если Java-приложение начинает свое выполнение с вызова метода main(), то апплет начинает свое выполнение с того, что контейнер апплета вызывает метод **init** апплета.. В строке

```
JTextArea outputArea = new JTextArea();
```

объявляется ссылка **outputArea** типа **JTextArea,** которая инициализируется ссылкой на новый объект **JTextArea.** Этот объект **JtextArea** будет отображать результаты выполнения программы.

Эта программа первая, в которой мы используем для отображения данных в апплете компонент графического интерфейса пользователя. Занимаемая объектом класса **JApplet** площадь экрана содержит *область вывода содержания (контейнер),* к которой компоненты графического интерфейса пользователя должны иметь доступ, чтобы они могли выводить изображения во время выполнения программы. Область вывода содержания — это объект класса **Container** из *пакета java.awt.* Этот класс импортируется в строке

```
import java.awt.Container;
```

для последующего использования в апплете. В строке

```
Container container = getContentPane();
```

объявляется ссылка **container** на объект **Container,** которой присваивается результат вызова метода ***getContentPane*** — один из многих методов, которые наш класс **SquareInt** наследует от класса **Japplet.** Метод ***getContentPane*** возвращает ссылку на контейнер апплета. Программа использует эту ссылку для подключения компонента графического интерфейса пользователя, в данном случае **JtextArea,** к пользовательскому интерфейсу апплета.

В строке

```
container.add( outputArea );
```

компонент графического интерфейса пользователя **JtextArea,** на который ссылается **outputArea** связывается с апплетом. Когда апплет выполняется, то все компоненты графического интерфейса пользователя, связанные с ним, отображаютс на экране. Метод add класса Container связывает компонент графического интерфейса пользователя с контейнером. Такой компонент занимает всю область контейнера апплета.

В строке

```
int  result;
```

объявляется переменная result типа int в которой будет храниться результат каждого вычисления квадратного значения.

В строке

```
String output = "";
```

Объявляется ссылка output типа String, которая инициализируется ссылкой на пустую строку. Эта строка будет содержать результаты возведения в квадрат значений от 1 до 10.

Далее определяется структура повторения for. На каждой. итерации этого цикла вычисляется квадратное значение текущего значения управляющей переменной x, полученное значение сохраняется в переменной result, и значение переменной result добавляется к текущему значению строки output.

Апплет вызывает свой метод square в операторе

```
result = square( counter );
```

Круглые скобки, следующие за именем метода square, представляют операцию вызова метода, которая имеет высшее старшинство. В этой точке программы создается копия значения counter (параметра вызова метода) и управление программой передается первой строке метода square. Метод square получает копию значения переменной counter через параметр y. Затем в методе square вычисляется значение выражение y*y. В методе square используется оператор *return,* чтобы возвратить (т.е. вернуть обратно) результат вычисления оператору метода init, который вызвал метод square. В методе init возвращаемое значение присваивается переменной result. Далее конкатенируются строка "Квадрат значения ", значение переменной counter, строка "равен", значение result и символ новой строки со строкой output. Этот процесс повторяется на каждой итерации структуры повторения. Затем метод setText назначает строку output в качестве строки, выводимой в outputArea.

Заметим, что мы объявили ссылки output, outputArea и container и переменную result

как локальные переменные метода init, потому что они используются только в методе init. Переменные должны объявляться как переменные экземпляра класса только в том случае, если они будут использоваться более чем в одном методе класса или если программа должна сохранять их значения между вызовами методов класса. Также заметьте, что метод init вызывает метод square непосредственно, без указания имени класса, предшествующего имени метода, и операции точка или имени ссылки и операции точка. Каждый метод класса может вызывать другие методы класса непосредственно. Однако у этого правила имеется исключение. Статические методы класса могут непосредственно вызывать только другие статические методы класса.

Из определения метода square видно, что square имеет целочисленный параметр y; метод square использует это имя для обозначения значения, которое метод получает через параметр. Ключевое слово int, предшествующее имени метода указывает, что square возвращает целочисленный результат. Оператор return в методе square передает результат вычисления y*y обратно, вызывающему методу. Отметим, что определение метода задается между фигурными скобками определения класса SquareIntegers. Все методы должны определяться внутри определения класса

Заметьте, что пример фактически содержит определения двух методов - метода init и square. Помните, что контейнер апплета вызывает метод init, чтобы инициализировать апплет. В этом примере метод init повторно вызывает метод square для выполнения вычислений, а затем выводит результаты в объект JTextArea, который связан с контейнером апплета. Как только апплет появляется на экране, результаты сразу отображаются в JTextArea.

Обратите внимание на синтаксис оператора вызова метода square — мы используем только имя метода, после которого, в круглых скобках, указываются аргументы.

Методы, определенные в классе, могут вызывать другие методы из определения класса, используя этот же синтаксис вызова. (Имеется исключение из этого правила.) В классе доступны методы, определенные в нем, а также унаследованные методы (методы из класса, который текущий класс расширяет, — в данном случае речь идет о классе JApplet). На данный момент нам знакомы три способа вызова метода: непосредственно по имени метода (как это делается в данном примере в вызове square(x)); по ссылке на объект, за которой указывается операция точка (.) и имя метода (например outputArea.setText(output)); по имени класса, за которым указывается имя метода (например Integer.parseInt(ToConvert)). Последний вариант вызова используется только для вызова статических методов класса.

Определение метода внутри другого метода является синтаксической ошибкой.

Объявление параметров метода, имеющих одинаковый тип, как float x, y вместо float x, float y, вызывает ошибку компиляции, так как типы надо указывать для каждого параметра в списке.

Метод, требующий большого количества параметров, возможно, выполняет слишком много задач. Попробуйте разделить такой метод на небольшие методы, которые выполняют различные задачи. Заголовок метода, по возможности, не должен занимать более одной строки.

Заголовок метода и вызовы метода должны быть согласованы по количеству, типам и порядку перечисления параметров и аргументов.

Пример.

Использование пользовательских методов позволяет писать процедурную программу (т.е. программу без порождения объектов) на языке Java.

Пусть задана какая-то функция f(), имеющая на отрезке [a, b] ровно один корень.

Java-программа, вычисляющая этот корень приближенно методом деления пополам (бисекции, дихотомии), может быть оформлена следующим образом.

```
class Bisection1 {
  static double f(doub1e x) {  // пользовательский static метод
    return x*x*x - 3*x*x + 3;   // вид функции
  }  // конец пользовательского метода
  public static void main(String[]args){// объязательный метод класса
    double a = 0.0, b = 1.5, c, y, eps = 1e-8;
    do {
      c = 0.5 *(a + b);
      y = f(c);      //использование пользовательского метода
      if (Math.abs(y) < eps) break;
      // Корень найден. Выходим из цикла
      //  Если на концах отрезка [a; c] функция имеет разные знаки:
      if (f(a) * y < 0.0)  b = c;
      // Значит, корень здесь. Переносим точку b в точку c
      //  в противном случае:
      e1se a = c;
      //  Переносим точку a в точку c
      //  Продолжаем, пока отрезок [a; b] не станет мал
    whi1e(Math.abs(b-a) >= eps);

    System.out.println("x=", + c + ", f(" + c + ") =" + y);
   }
  }  // конец   метода main
}    //  конец класса
```

Эта программа оформлена в стиле процедурного подхода, но только на языке Java. В этом варианте весь метод решения задачи расписан внутри объязательного `static` метода **main** Java-приложения**,** а задание вида функции оформлено в виде отдельного дополнительного автономного метода `static double f(doub1e x)`. В таком варианте использование определения `static` `объязательное`.

Метод **main** всегда автоматически выполняющийся метод при запуске Java-программы. Так сказать **main** – это точка входа в Java-программу для выполния ее интерпретатором Java. Метод вызывается кодом, который находится вне программы – исполняющей системой Java.

Попробуйте написать Java-программу без использования дополнительного метода.

Лекция 8. Программные модули Java: классы.

Классы

Объекты реального мира, например, регистрационная форма, используемая для регистрации студента на курсе, может состоять из следующих *атрибутов*: идентификатор студента, имя, фамилия студента, номер учебной дисциплины, наименования учебной дисциплины, а также *вариантов поведения*: новая регистрация, отображение регистрационной карты, изменение регистрационных данных, отмена регистрации. Задача программиста состоит в том, чтобы с помощью объектно-ориентированного языка программирования преобразовать атрибуты и варианты поведения объекта реального мира в *класс*, состоящий из атрибутов и методов, которые понятны компьютеру.

Класс - это шаблон, который определяет атрибуты и методы объекта реального мира. Считайте, что класс - это форма для печенья в виде буквы *A*. Форма для печенья - это не буква *A,* она лишь определяет, как буква *A* выглядит. Если вам необходима буква *A,* поместите форму на лист теста. Если вы хотите сделать другую букву *A,* возьмите ту же самую форму и повторите процесс. С помощью это и формы вы можете сделать столько букв *A,* сколько вам надо.

Класс описывает тип, содержащий как подпрограммы (или методы), так и данные. Раз класс описывает тип, значит, мы можем создать переменные, содержащие и методы и переменные. Такие переменные являются объектами. Если вам необходим объект, представляемый классом, вы создаете экземпляр класса. Экземпляр - это то же самое, что и буква *A,* которая появляется на листе теста после того, как вы убираете формочку для печенья.

Каждый экземпляр содержит те же самые атрибуты и методы, которые определены в классе, хотя каждый экземпляр имеет свою копию этих атрибутов. Экземпляры используют одинаковые методы. Немного запутанное объяснение? Давайте вернемся к нашим формочкам для печенья, чтобы не запутаться. Вспомните, что экземпляр - это «настоящее печенье, которое было определено формочкой для вырезки печенья (шаблоном класса). Представьте формочку в виде собаки. Лапы собаки имеют некоторую ширину и длину, как и ее хвост. Это атрибуты собаки. Каждый раз, когда мы вырезаем с помощью этой формочки тесто, создается еще одно печенье в форме собаки (экземпляр). Допустим, мы сделали таким способом два печенья в форме собаки. Каждое печенье (экземпляр) имеет лапы и хвост одинаковой ширины и длины, но каждое печенье имеет свой набор (копию) лап и хвоста, которые независимы от другого печенья в форме собаки.

Метод - это поведение, которое может осуществлять печенье. Но печенье в форме собаки на самом деле не делает ничего, кроме того, что тихо сидит у вас в руках, так что нам надо воспользоваться своим воображением и притвориться, что эта собака-печенье сможет встать на лапы, если вы и только вы объясните ей, как вставать. Собака-печенье игнорирует команды кого-то еще. Каждая собака-печенье использует одинаковую копию методов (то есть ваших команд), чтобы осуществить поведение, связанное с вставанием на лапы.

Определение класса

Класс описывается с помощью определения класса (class definition). В *определении класса* перечисляются атрибуты и методы, которые являются членами класса. Определение класса состоит из трех частей:

- ключевого слова class;
- названия класса;
- тела класса.

Ключевое слово class сообщает компилятору, что вы определяете новый класс. Ключевые слова (называемые также *зарезервированными словами)* - это слова, которые имеют специальное значение для языка программирования. *Имя класса* - это последовательность символов, заданная программистом, которая уникально идентифицирует класс среди других классов. Название класса должно соответствовать объекту реального мира, который он эмулирует, а первая буква в названии должна быть *заглавной*. Вот простое определение класса на Java:

```
class RegistrationForm {
    int studentNumber;
    int courseNumber;
}
```

Здесь название класса - RegistrationForm, а сам класс представляет собой форму регистрации студентов для занятий. Перед словом *class* в определении класса может стоять один или несколько, так называемых, *модификаторов* или *спецификаторов доступа* к классу*: public, final, abstract*, но об этом позже.

Тело класса - это часть определения класса, которая находится между открывающей и закрывающей скобками. Атрибуты и методы определяются внутри этих скобок. В этом примере определены два атрибута - номер студента и номер курса. Методы не определены.

Атрибут класса – это переменная, называемая переменной экземпляра. Переменная экземпляра используется для хранения данных в памяти. Переменные экземпляра объявляются внутри определения класс с помощью оператора объявления. Оператор объявления состоит из следующих трех частей:

- тип данных;
- имя переменной экземпляра;
- точка с запятой.

Имя переменной экземпляра – это идентификатор языка Java и он должен отражать природу данных, которые хранятся в этом месте памяти. Например, studentNumber – это отличное имя для

переменной, используемой для хранения номера студента.

Можно сказать с помощью класса в Java определяется новые пользовательские типы данных.

Объединение класса и программы

Теперь, когда вы знаете, как определить класс, его атрибуты и методы, давайте поместим определение класса в программу. Определение класса помещается вне *основной части* программы. Основная часть программы на Java - это *метод main* класса для Java- приложения и *метод Applet.init* для апплетов. Методы main и init - это точки входа в программу, т.е. с них начинается выполнения программы на Java. Итак, определение класса должно находиться вне определения основного класса (с методами main и init) в Java-программе. Такое определение может стоять до или после основной части программы.

1-пример.

Итак, определение класса должно находиться вне определения класса приложения в программе на Java. Мы будем использовать программу на Java под названием MyJavaApplication, чтобы проиллюстрировать, где в программе должно находиться определение класса.

Первая часть - это класс Java-приложения, названного MyJavaApplication, а вторая - это определение класса RegistrationForm. Определение класса RegistrationForm размещено вне определения класса My]avaApplication. Эти два класса представляют разные сущности. Класс MyJavaApplication представляет программу на Java, а класс RegistrationForm - регистрационную форму, использующуюся для регистрации студентов на курс (новый тип обрабатываемых данных).

```
class MyJavaApplication {  // основная часть программы – определение  класса приложения (класса MyJavaApplication)
                           // с методом main
   public static void main (String args[]) {
        RegistrationForm regForm =  new RegistrationForm():
        regForm.dropCourse(102,1234);
   }   // конец метода main
} // конец класса  MyJavaApplication

class RegistrationForm {   // RegistrationForm - используемый класс
   void dropCourse(int courseNumber. int studentNumber) {
       System.out.println("Course " + courseNumber + " has Ьееп dropped from student " + studentNumber):
   }   //  конец метода dropCourse
} // конец класса  RegistrationForm
```

2-пример.

Предположим, имеется класс с именем Box, который определяет три переменных экземпляра: width, fcight и depth. Пока что класс Box не содержит никаких методов (но вскоре ни будут добавлены).

```
 class Box {
  double width;
 double height;
 double depth;
}
```

Как было сказано выше, класс определяет новый тип данных. В этом случае новый тип данных называется Box. Вы будете использовать это имя для объявления объектов типа Box. Важно помнить, что объявление класса создает только шаблон, а не фактический объект. Таким образом, предшествующий код не приводит к появлению каких-либо объектов типа Box. Чтобы фактически создать Box-объект, можно воспользоваться следующим утверждением:

```
mybox = new Box();       // создать Box-объект с именем mybox
```

После выполнения этого утверждения переменная mybox станет экземпляром класса Box, становясь той самой "физической" реальностью.

Всякий раз, когда вы создаете экземпляр класса, образуется объект, который содержит свою собственную копию каждой экземплярной переменной, определенной в классе. Таким образом, каждый Box-объект будет содержать свою собственную копию переменных width, height и depth. Для доступа к этим переменным необходимо использовать операцию "точка" (.) Она связывает имя объекта с именем переменной экземпляра. Например, чтобы назначить переменной width объекта mybox значение 100, нужно использовать следующий оператор:

```
Box.width = 100;
```

Этот оператор просит компилятор назначать копии переменной width, которая содержится в объекте mybox, значение 100. В общем случае, чтобы обращаться как к переменным экземпляра, так и к методам объекта, следует указывать операцию "точка". Только сначала нужно создать объект. Далее предлагается законченная программа, которая использует Box-класс.

```
/* Программа, которая использует Box-класс. Назовите этот файл BoxDemo1.java */
class Box {
  double width;
  double height;
  double depth;
}
// Этот класс объявляет объект типа Box.
class BoxDemo1  {
public static void main(String args[])  {
 Box mybox = new Box(); // Создание объекта
 double vol;
  // присвоить значения экземплярным переменным объекта mybox
  mybox.width = 10; // Обращение к переменным объекта
  mybox.height =20;
  mybox.depth = 15;
  // вычислить объем блока
  vol = mybox.width * mybox.height * mybox.depth;
  System.out.println("Объем равен"+vol); //Обращение к методу-члену объекта
 }
}
```

В этом примере используемый класс Box в тексте находится до основной части Java-программы. Вы должны назвать файл, который содержит эту программу, BoxDemo1.java, потому что метод main () находится в классе с именем BoxDemo1, а не в классе именем Box. После компиляции программы вы обнаружите, что были созданы два файла с расширением .class — один для Box-класса и один для BoxDemo1. Java-компилятор автоматически помещает каждый класс в собственный class-файл. Нет необходимости в том, чтобы классы Box и BoxDemo1 находились в одном исходном файле. Можно было поместить каждый класс в свой собственный файл с именами Box.java и BoxDemo1.java, соответственно.

Чтобы выполнить данную программу, нужно выполнить BoxDemo1.class. Когда вы сделаете это, то увидите следующий вывод:

```
Объем равен 3000.0
```

Объявление объектов

Оператор

```
Box mybox = new Box();
```

комбинирует два шага. Его можно переписать так:

```
Box mybox;
mybox = new Box();
```

Первая строка объявляет mybox как ссылку на объект типа Box. Ссылка (reference) – это что-то, что отсылает вас к чему-то еще. В данном случае ссылка отправляет вас к экземпляру класса Box. Имя ссылки mybox. Вы используете имя ссылки всегда, когда

вам необходимо обратиться к экземпляру Box. Ссылка – это не экземпляр класса. Это только строка символов, которая ссылается на экземпляр. Ссылка – это переменная, ссылающаяся на объект. После того как эта первая строка выполняется, mybox содержит значение null, которое означает, что переменная еще не указывает на фактический объект. Любая попытка использовать mybox в этой точке приведет к ошибке во время компиляции.

Команда new Box() следующей строки говорит компилятору динамически зарезервировать блок памяти равный размеру класса Box(). «Динамически» означает, что память резервируется, когда компилятор выполняет программу; это называется периодом времени выполнения (runtime). Размер класса – это размер всех атрибутов класса. После того как компилятор резервирует блок памяти, он возвращает указатель на первый адрес этого блока памяти. Указатель (pointer) – это такой же указатель, который указывает на офис вашего преподавателя. Оператор присваивания следующей строки присваивает ссылке указатель на экземпляр (т.е. начальный адрес блока памяти). В кратце следующая строка распределяет фактический объект и назначает ссылку на него переменной mybox.

После того, как вторая строка выполняется вы можете использовать mybox, как если бы это был объект Box. Но в действительности mybox просто содержит адрес ячейки-памяти фактического объекта Box. Подобную mybox переменную называют объектной переменной.

Операция *new*

Как только что было указано, операция new динамически распределяет память для объекта. Он имеет следующую общую форму:

class-var = new *classname()*;

Здесь *class-var* — переменная типа "класс", которая создается; *classname* – имя класса, экземпляр которого создается. За именем класса следуют круглые скобки, устанавливающие *конструктор* класса - метод-конструктор. Метод-конструктор класса вызывается автоматически при создании объекта (экземпляра) класса с помощью оператора new. Конструктор определяет, что происходит, когда объект класса создается. Это важная часть всех классов. Конструкторы имеют много существенных атрибутов, большинство классов явно определяют свои собственные хонструкторы (внутри своих определений). Однако если явный конструктор не определен, то Java автоматически обеспечит так называемый "конструктор по умолчанию" (default constructor) или *умалчиваемый конструктор*. Именно так обстоит дело с классом Box. Пока мы будем использовать конструктор по умолчанию, но скоро вы увидите, как можно определять ваши собственные конструкторы.

Сейчас вы могли бы задаться вопросом, почему не нужно использовать операцию new для таких типов данных, как целые числа или символы. Ответ заключается в том, что простые типы Java не реализованы как объекты. Они реализованы как "нормальные" переменные. Это сделано в интересах эффективности. Объекты имеют много свойств и атрибутов, которые требуют, чтобы Java обращалась с ними иначе, чем с простыми типами. Не применяя к простым типам тот же подход, который используется с объектами, Java может реализовывать простые типы более эффективно.

Важно понять, что new распределяет память для объекта во время выполнения. Преимущество данного подхода состоит в том, что ваша программа может создавать столько объектов, сколько требуется в течение ее выполнения.

Назначение ссылочных переменных объекта

Во время выполнения операции назначения (присваивания) ссылочные переменные объекта (object reference variables) действуют иначе, чем можно было бы ожидать. Например, как вы думаете, что делает следующий фрагмент?

```
Box b1 = new Box();
Box b2 =b2;
```

Вы могли бы предположить, что переменной Ь2 назначается ссылка на копию объекта, на который ссылается b1. То есть вы могли бы подумать, что b1 и b2 обращаются к отдельным и различным объектам. Однако это было бы неправильно. На самом деле после того, как этот фрагмент выполнится, обе переменных (b1 и b2) будут ссылаться на один и тот же объект. Назначение b1 переменной b2 не распределяет никакой памяти и не копирует какую-либо часть первоначального объекта. Эта операция просто помещает в b2 ссылку из b1. Таким образом, любые изменения, сделанные в объекте через b2 затронут объект, на которой ссылается b1, т.к. это один и тот же объект.

Добавление метода к классу *Box*

Хорошо, конечно, создавать класс, который содержит только данные, но это случается редко. Большую часть времени приходится использовать методы доступа к экземплярным переменным, которые определены в классе. Фактически, именно методы определяют интерфейс с большинством классов. Они позволяют разработчику класса скрывать специфическое размещение внутренних структур данных за более ясными абстракциями метода. Кроме методов, которые обеспечивают доступ к данным, можно также определять методы, использующиеся внутри самого класса.

Начнем с добавления метода к Box-классу. Глядя на предыдущую программу, вполне может показаться, что вычисление объема блока было бы лучше выполнять в Box-классе, а не в классе BoxDemo. И, наконец, т. к. объем блока зависит от его размера, то имеет смысл поручить это вычисление Box-классу. Для этого нужно добавить метод в класс Box, как показано в следующей программе:

```
/* Эта программа включает метод внутрь Box-класс. Назовите этот файл BoxDemo2.java */
class Box {
  double width;
  double height;
  double depth;

  // метод показа объема
  void volume() {
    System.out.print("Объем равен " );
    System.out.println(width * height * depth);
  }
}

class BoxDemo2  {
 public static void main(String args[])  {
   Box mybox = new Box(); // 1. сначала создается объект mybox класса Box

    // 2. теперь можно присваивать значения переменным объекта mybox
    mybox.width = 10;
    mybox.height =20;
    mybox.depth = 15;

    // 3. теперь можно вызвать метод volume() для вычисления объема или
    mybox.volume();//команда выполнения объектом mybox своего метода volume
  }
}
```

Когда выполняется mybox.volume(), исполняющая система Java передает управление коду, определенному внутри метода volume(). После того как операторы внутри volume() выполнятся, управление возвращается в вызывающую подпрограмму, и работа продолжается со строки кода, следующей за вызовом. В самом общем смысле метод — это способ реализации подпрограмм в языке Java.

Есть кое-что очень важное, на что нужно обратить внимание внутри метода volume(): переменные width, height и depth указаны прямо, без предшествующих им имен объектов и "точечных" операций. Когда метод использует переменную экземпляра, которая определена в его классе, он указывает ее прямо, без явной ссылки на объект и использования «точечной» операции. Как известно, метод всегда вызывается из некоторого объекта его класса. Раз этот вызов произошел, значит объект известен. Таким образом, внутри метода нет необходимости указывать объект второй раз. Этот означает, что переменные width, height и depth внутри метода volume() неявно ссылаются на копии переменных, находящихся в объекте, который вызывает этот метод.

Возврат значений

Хотя реализации метода volume() перемещает вычисление объема блока внуть Box-класса, которому этот метод принадлежит, это – не лучший способ вычисления. Лучший способ реализации метода volume() состоит в том, чтобы он вычислил объем блока и возвращал результат вызывающей программы. Следующий пример (улучшенная версия предшествующей программы) именно это и делает:

```
/* Теперь  метод volume() возвращает объем блока. Назовите этот файл BoxDemo3.java */
class Box {
  double width;
  double height;
  double depth;

  // метод показа объема
  void volume() {
    return width * height * depth;
  }
}

class BoxDemo3  {
 public static void main(String args[])  {
   Box mybox = new Box(); // 1. создание объекта класса Box
   Double vol;
    // 2. присвоение значения переменным объекта mybox
    mybox.width = 10;
    mybox.height =20;
    mybox.depth = 15;

    // 3. вычисление объема и показ его
    vol = mybox.volume();//вычисление и запоминание объема
    System.out.print("Объем равен " + vol); //показ объема
  }
}
```

Тип данных, возвращаемых методом, должен быть совместим с типом, указанным в заголовке определения метода. А также переменная, принимающая значение, возвращаемое методом, должна быть совместима с типом возвращаемого значения. Можно написать и такой вариант выдачи результата:

```
System.out.print("Объем равен " + mybox.volume());
```

Добавление метода с параметрами

Хотя некоторые методы не нуждаются в параметрах, но большинство из них •параметрами все-таки пользуется. Параметры обобщают метод. Параметризованный метод может работать на множестве данных и/или использоваться в ряде похожих ситуаций. Чтобы иллюстрировать это положение, воспользуемся очень простым примером. Имеется метод, который возвращает квадрат числа 10:

```
int square()
{
   return 10 * 10;
}
```

Хотя этот метод действительно осуществляет возврат значения 10, возведенного в квадрат, его использование очень ограничено. Однако если вы измените метод так, чтобы он имел параметр, как показано ниже, тогда вы можете сделать метод square() более полезным.

```
int square(int i)
{
   return i * i;
}
```

Теперь square() будет возвращать квадрат любого значения, с которым он вызывается. То есть square() стал универсальным методом, который может вычислять квадрат любого целого значения, а не только 10. Например:

```
int x, y;
x = square(5);  // x равно 25
y = square(9); // y равно 81
y = 2;
x = square(y); // x равно 4
```

В первом обращении к square() параметром i будет передаваться значение 5. Во втором обращении i будет принимать значение 9. Третье обращение передает значение y, которое в этом фрагменте равно 2. Как показывают эти примеры, square() способен возвращать квадрат любых данных, которые ему пересылают.

Важно различать два термина *параметр* и *аргумент. Параметр* — это переменая, определяемая методом, которая принимает значение во время вызова метода. Например, в методе square (int i) определен один параметр i типа int. *Аргумент* — это значение, которое передается методу, когда тот вызывается. Например, методу square(100) в качестве аргумента передается число 100. Внутри метода square() это значение принимает параметр i.

Вы можете использовать параметризованный метод, чтобы улучшить класс Box. В предыдущих примерах размеры каждого блока должны быть установлены отдельно при помощи последовательности следующих операторов

```
    mybox.width = 10;
    mybox.height =20;
    mybox.depth = 15;
```

Хотя этот код работает, однако возникает некоторое беспокойство. Во-первых он кажется каким-то не изящным и склонным к ошибкам. Например, легко забыть установку измерений. Во-вторых, в хорошо разработанных программах к переменным экземпляра нужно обращаться только через методы, определенные их классом. В будущем вы сможете изменять поведение метода, но вы не можете изменять поведение установленной переменной[экземпляра.

Таким образом, лучший подход к установке размеров блока состоит в том, чтобы создать метод, который берет измерения блока в свои параметры походящим образом устанавливает каждую переменную экземпляра. Эта концепция реализована следующей программой:

```
/* Эта программа использует параметризованный метод volume(). Назовите этот файл BoxDemo4.java */
class Box {
  double width;
  double height;
  double depth;

  // метод показа объема
  void volume() {
    return width * height * depth;
  }
  // установить размеры блока
  void setDim(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
```

```
  }
}

class BoxDemo4 {
 public static void main(String args[])  {
   Box mybox = new Box(); // 1. создание объекта класса Box
   Double vol;

    //2. присвоение значения переменным объекта mybox с помощью метода
    mybox.setDim(10, 20, 15);

    // 3. вычисление объема и показ его
    vol = mybox.volume();//вычисление и запоминание объема
    System.out.print("Объем равен " + vol); //показ объема
  }
}
```

Как можно заметить, метод `setDim()` используется, чтобы установить размер блока. Например, когда

```
mybox.setDim(10, 20, 15);
```

выполняется 10 копируется в параметр `w`, 20 копируется в `h` и 15 копируется в `d`. Внутри метода `setDim()` значения `w`, `h` и `d` затем назначаются переменным `width`, `height` и `depth`, соответственно.

Объектно-ориентированный вариант прогаммы Bisection1

Теперь оформим `Bisection1` алгоритм в духе объектного подхода. Для этого нужно алгоритм решения задачи расписать внутри *другого метода* (а не в методе **main** как было в `Bisection1`) или как отдельный класс.

Здесь мы рассматриваем первый вариант. Тогда, чтобы выполнить этот алгоритм, нужно создать объект класса, а в методе **main** предусмотреть команду выполнения объектом метода, содержащего алгоритма.

```
class Bisection2{
  private static double final eps = 1e-8;  //Константа класса
  private double  a = 0.0, b = 1.5, root;  //Закрытые поля класса

  public double getRoot() // 1-метод класса – метод доступа к полю root
    {
      return root;
    }  // конец 1-го метода

  static double f(double  x) {      // 2- метод класса
     return x*x*x - 3*x*x + 3;
   }  // конец 2-го метода

  private void bisect() {   // 3-метод класса
                            // Параметров у метода нет-
                            // метод работает с полями экземпляра
      double y = 0.0;           // Локальная переменная - не поле класса
      do {
         root = 0.5 *(a + b);
          y = f(root);   //использование пользовательского метода
        If (Math.abs(y) < eps) break;
        // Корень найден. Выходим из цикла
        // Если на концах отрезка [a; root] функция имеет разные знаки:
        if (f(a) * y < 0.0)  b = root;
        // значит, корень здесь, и мы пере носим точку b в точку root
        // в противном случае:
        else a = root;
        // переносим точку a в точку root
        // продолжаем, пока [a; b] не станет мал
        }
      while (Math.abs (b-a) >= eps);
    }  // конец 3-го метода
```

```
   public static void main(String[] args) { //4-метод (объязательный)
     Bisection2   b2 = new Bisection2();   //создание объекта класса
     b2.bisect() ;   //выполнение объектом метода, содержащего алгоритма
     System.out.println("x = " +
      b2.getRoot() +        // Обращаемся к корню через метод доступа
          ", f () = " +b2. f (b2. getRoot () ) ) ;
   }  // конец 4-го метода
}  //  конец класса
```

В описании и использовании автономного метода f() сохранен старый процедурный стиль: метод получает аргумент, обрабатывает его и возвращает результат. Можно было его вычисление расписать внутри **bisect()**. Здесь вычисление f() расписан в виде автономного метода, что приемлемо в объектно-ориентированной программе, если это необходимо для приложения.

Описание метода bisect() сделано как private (а не static как было Bisection1). Метод bisect () - это внутренний механизм класса Bisection2. Его использование связано с созданием экземпляра класса, т.е. объекта как b2. Потому его использование сделано в духе ООП.

Здесь bisect() используется как метод Java-программы. Попробуйте оформить использование bisect() как класса в Java-программе.

Лекция 9. Конструктор и его использование

Определение конструктора

Поскольку требование инициализации всех переменных экземпляра является достаточно общими, Java разрешает инициализацию объектов в момент их создания. Это автоматическая инициализация выполняется с помощью конструктора.

Конструктор инициализирует объект после создания. Он имеет такое же имя, как класс, в котором он постоянно находится и синтаксически подобен методу. Если конструктор определен, то он автоматически вызывается сразу же после того, как объект создается, и прежде, чем завершается выполнение операции *new*. Конструктор – это метод класса, который вызывается автоматически, когда объявляется экземпляр класса. Конструкторы не имеют спецификаторы возвращаемого типа. Неявным возвращаемым типом конструктора класса является тип самого класса. Работа конструктора заключается в том, чтобы инициализировать внутреннее состояние объекта так, что код, создающий экземпляр, будет полностью инициализирован и пригоден для немедленного использования объекта.

Рассмотрим определения простого конструктора, который устанавливает одинаковые значения для размеров каждого блока.

```
/*Box использует конструктор для инициализации размеров блока.*/
class Box  {
  double width;
  double height;
  double depth;

  // Это конструктор класса Box.
  Box() {
    System.out.println("Создание Box");
    width = 10;
    height = 10;
    depth = 10;
  }

  // метод вычисления объема
  double volume()  {
    return width * height * depth;
  }
} // конец класса Box
```

```
class BoxDemo5 {
public static void main(String args[]) {
/*объявить,разместить,в памяти и инициализировать два Box-объекты */
  Box myboxl = new Box(); //операции new автоматически вызывают
  Box mybox2 = new Box(); // конструктор
  double vol;

  // получить объем первого блока
  vol = myboxl.volume();
  System.out.println("Объём равен " + vol);

  // получить объем второго блока
  vol = mybox2.volume();
  System.out.println("Об'ем равен " + vol);
  }
}
```

Предложение `println` внутри `Box()`приводится только ради иллюстрации. Большинство функций конструктора ничего не будет отображать. Они просто инициализирует объект.

В предложении

```
Box myboxl = new Box();
```

в правой части фактически выполняется вызов конструктора класса Box(). Если вы явно конструктор класса не определяете, то Java создает для этого класса *конструктор по умолчанию*. Умалчиваемый конструктор автоматически инициализирует все переменные экземпляра нулями. Такой конструктор часто достаточен для простых классов, но не для более сложных. Как только в классе определяется собственный конструктор, умалчиваемый больше не используется.

Параметризованные конструкторы

В предшествующем примере конструктор инициализирует все блоки одинаковыми размерами. Чтобы конструктор устанавливал разные размеры надо использовать параметризованный конструктор.

```
/* Здесь класс Box использует параметризованный конструктор для
   инициализации размеров блока.*/
class Box  {
  double width;
  double height;
  double depth;

  // Это параметризованный конструктор класса Box.
  Box(double w, duoble h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  //  вычисление и возвращение объема
  double volume()  {
     return width * height * depth;
  }
} // конец класса Box

class BoxDemo6 {
public static void main(String args[]) {
  /* объявить, разместить, в памяти и инициализировать Box-объекты */
  Box myboxl = new Box(10, 20, 15);
  Box mybox2 = new Box(3, 6, 9);
  double vol;

  // получить объем первого блока
  vol = myboxl.volume();
  System.out.println("Объём равен " + vol);
```

```
  // получить объем второго блока
  vol = mybox2.volume();
  System.out.println("Объем равен " + vol);
  }
}
```

В строке

```
  Box myboxl = new Box(10, 20, 15);
```

Значения 10, 20 и 15 передаются конструктору Box(), когда new создает объект. Таким образом, копии переменных width, height, depth объекта myboxl будут содержать значения 10, 20, 15, соответственно.

Ключевое слово this

Как известно, метод вызывается объектом. Иногда у метода возникает необходимость обращаться к объекту, который его вызвал. Для этого Java определяет ключевое слово this. Его можно использовать внутри любого метода, чтобы сослаться на *текущий* объект. Чтобы лучше понять, на что ссылается this, рассмотрим следующую версию Box():

```
// Избыточное использование this.
Box(double w, duoble h, double d) {
  this.width = w;
  this.height = h;
  this.depth = d;
}
```

Эта версия Box() работает точно так же, как и более ранняя. Использование this избыточное, но корректное. Внутри this будет всегда ссылаться на вызывающий объект. Хотя и избыточный в данном случае, this полезен в других контекстах, один из которых объясняется в следующем разделе.

Скрытие переменной экземпляра

Как известно, в Java недопустимо объявление двух локальных переменных с одним и тем же именем внутри той же самой или включающей области действия идентификаторов. Заметим, что вы можете иметь локальные переменные, включая формальные параметры для методов, которые перекрываются с именами экземплярных переменных класса. Однако, когда локальная переменная имеет такое же имя, как переменная экземпляра, локальная переменная *скрывает* переменную экземпляра. Вот почему width, height, depth не использовались как имена переменных параметров конструктора Box() внутри класса Box. Если бы они были использованы для именования этих параметров, то, скажем width, как формальный параметр, скрыл бы переменную экземпляра width. Хотя обычно проще указывать различные имена, существует другой способ обойти эту ситуацию. Поскольку this позволяет обращаться прямо к объекту, это можно применять для разрешения любых конфликтов пространства имен, которые могли бы происходить между экземплярными и локальными переменными. Ниже представлена другая версия Box(), которая использует width, height, depth для имен параметров и затем применяет this, чтобы получить доступ к переменным экземпляра с такими же самыми именами:

```
/* Используйте этот вариант конструктора для разрешения конфликтов пространства имен. */
Box(double width, duoble height, double depth) {
  this.width = width;
  this.height = height;
  this.depth = depth;
}
```

Об определении классов и методов

Существует тесная связь между способностью ясно выражать свои мысли и способностью составлять программы для компьютера. Владеть операциями с классами в логике примерно соответствует умению преобразовывать понятия. Приобретение способности к операциям с

отношениями, что эквивалентно умению формировать и преобразовывать суждения. Наконец, при достижении уровня синтеза операций с классами и отношениями и обретение состояния интеллектуальной целостности, свободы и творчества, есть то состояние необходимое программисту. В традиционной логике это равнозначно способности к умозаключениям, т.е. получению новых истин на основании известного знания.

Созданию практической программы сопутствует огромный объем исследований и анализа поставленной, программируемой проблемы. Взаимодействия объектов проблемной области в объектно-ориентированной программе эмулируются взаимодействиями программных объектов – экземпляров классов. При идентификации и описании таких объектов прежде всего нужно учитывать потребности конечного пользователя программы. Согласно этой потребности абстрагированием образуются программные объекты и в них оставляются только те атрибуты (данные) и поведения (методы), являющиеся существенными с точки зрения постановки задачи. Описание требований конечного пользователя в словесной форме, приводят к появлению существительных и глаголов. Существительные становятся претендентами классов (объектов), а глаголы – методами классов. Принято результаты таких исследования и анализа проблемы оформлять на языке UML в виде различных *диаграмм* и *документаций*. Затем на основе диаграмм и документаций делается сначала проектирование классов, а потом только кодирование классов. Проектирование диаграмм помогает идентифицировать методы, которые будут необходимы классам и определить, что они должны делать, что они будут в качестве параметров и что будут возвращать. Уточнения вариантов методов могут делаться с помощью мозгового штурма.

После того как программист идентифицирует объекты, их атрибуты и поведение, он приступает к кодированию этих объектов в программе с помощью описании классов.

Лекция 10. Наследование

Наследование (inheritance) - один из краеугольных камней объектно-ориентированного программирования, потому что оно позволяет создавать иерархические классификации классов. Наследование позволяет объектам наследовать атрибуты и поведение других объектов, таким образом, уменьшая объем нового кода, который надо спроектировать, написать и протестировать каждый раз, когда вы разрабатываете новую программу. Иначе говоря, наследование – это прием программирования, который позволяет повторно использовать существующий код

Используя наследование, можно создать главный класс, который определяет свойства, общие для набора связанных элементов. Затем этот класс может быть унаследован другими, более специфическими классами, каждый из которых добавляет те свойства, которые являются уникальными для него. В терминологии Java класс, который унаследован, называется *суперклассом* (superclass). Класс, который выполняет наследование, называется *подклассом* (subclass). Поэтому подкласс - это специализированная версия суперкласса. Он наследует все переменные экземпляра и методы, определенные суперклассом, и прибавляет свои собственные уникальные элементы. Наследование обеспечивает распределение контроля над разработкой и поддержкой объектов. Наследование также обеспечивает ограничение доступа к атрибутам и поведению.

Управление доступом к элементам класса

Объектно-ориентированные языки программирования позволяют программистам инкапсулировать атрибуты и поведения реальных объектов и связывать их в виде класса. *Инкапсуляция* - это способ связывания атрибутов и методов для формирования класса. Основная цель использования инкапсуляции – защита. В процедурном программировании не существует способа защиты от неправильного использования атрибутов и методов. Они доступы программисту без каких-либо проверок. Можно сказать, что инкапсуляция позволяет

программисту создать эти проверки, поместив атрибуты и методы в класс, а затем в классе определить правила доступа к ним.

Вообще программисты контролируют доступ к атрибутам и методам класса с помощью спецификаторов доступа в определении класса *Спецификатор доступа* – это ключевое слово языка программирования, которое говорит компьютеру, какая часть программы может получить доступ к атрибутам и методам, являющимся членами класса. Единственный способ получить доступ к атрибутам и методам класса состоит в создании экземпляра класса, т.е. объекта.

Некоторые аспекты управления доступом относятся главным образом к наследованию или пакетам. Здесь мы рассматриваем управления доступом в применении к отдельному классу.

Спецификаторы доступа Java: *public* (общий), *private* (частный), *protected* (защищенный). Java также определяет уровень доступа, *заданный по умолчанию*.

Спецификатор доступа public определяет переменные и методы, которые доступны при использовании экземпляра класса (объекта) или, говорят, возможен доступ к ним из любой точки программы. Спецификатор доступа private определяет переменные и методы, которые доступны только методам, определенным в классе. Спецификатор достпуа protected определяет переменные и методы, которые могут быть наследованы другими классами. Поэтому спецификатор protected применяется только при использовании наследования. *Когда никакой спецификатор доступа в определении класса не используется, по умолчанию элемент класса считается public в пределах своего собственного пакета*, но к нему нельзя обращаться извне этого пакета (о пакетах позже). Цель ограничения доступа к элементам класса состоит в том, чтобы гарантировать целостность объекта, управляя тем, какие классы и части программы могут взаимодействовать с ними.

В Java каждая переменная и метод-член должен иметь собственный спецификатор доступа. Программистам требуется, чтобы отдельные переменные класса были доступны только методам-членам – это позволит проверять значения, присваиваемые этим переменным. Программист, который хочет получить доступ к отдельным переменным, вызывает метод-член, который выполняет проверку перед присвоением значений переменным.

Чтобы понимать специфику общего и частного доступа, рассмотрим следующую программу:

```
/* Эта программа демонстрирует различие между методами доступа public и private. */
 class Test {
    int a;          // доступ по умолчанию (public)
    public int b;   // общий (public) доступ
    private int c;  // частный (private) доступ

    // методы для доступа к переменной c
  void sets(int i) { // установления значения c
     c = i;
    }
    int gets() {  // получения значения c
      return c;
    }
 }

class AccessTest {
   public static void main(String args[]) {
   Test ob = new Test();

    /* OK, к переменным a и b возможен
       прямой доступ */
    ob.a = 10;
    // Не OK и вызовет ошибку
    //   ob.c = 100;   // это ошибка
    //Нужен доступ к  c  через ее методы

     ob.setc(100);  // это правильно
     System.out.print1n("a,b,c: " + ob.a + " " +
                     ob.b + " " + gets());
    }
```

```
}
```
Внутри класса `Test` используется доступ по умолчанию, который для этого примера эквивалентен указанию public. Член b явно определен как public. Для члена c задан доступ private. Это означает, что к нему нельзя обращаться из кода, находящегося вне его класса. Так внутри класса `AccessTest` переменная c не может использоваться прямо. К ней нужно обращаться через ее public-методы `setc()` и `gets()`. Если бы вы удалили символ комментария в начале следующей строки, то не смогли бы откомпилировать эту программу из-за нарушения правил доступа:

```
//   ob.c = 100;   // это ошибка
```

Итак, метод-член вызывается следующим образом: объявляется экземпляр класса, а затем используется имя экземпляра, за которым следует оператор «точка» и название метода. Доступ к переменной экземпляра осуществляется с помощью объявления экземпляра класаа, а затем используется название этого экземпляра, за которым следует оператор «точка» и имя переменной.

Основы наследования

Наследование позволяет создавать иерархические классификации классов. Иерархические отношения между классами иногда называются отношениями «*родитель-потомок*». В таком отношении потомок наследует все переменные и методы родителя, а родительские спецификаторы доступа используются для контроля того, каким образом эти унаследоанные элементы будут доступны другим классам или методам. В Java родитель называется *суперклассом*, потомок – *подклассом*.

Определение отношения «родитель-потомок» во многих ситуациях является интутивным. Для того чтобы определить существует ли отношение между клаасами, программитсы используют тест «*является*». Тест «является» определяет, является ли потомок родителем. Если отношение «является» имеет смысл, это означает, что существует отношение «родитель-потомок» и что потомок может быть унаследован от родителя. Потомок может наследовать public- и protected-члены класса-предка.

Существует три способа реализации наследования в программе: *простое, множественное и многоуровневое наследование*. Простое наследование – это когда один потомок наследуется от одного родителя. Множественное наследование используется, когда отношение включает нескольких родителей и одного потомка. Java не поддерживает множественное наследование. Многоуровневое наследование появляется, когда потомок наследуется от родителя, а затем сам становится родителем.

Имея три варианта выбора, вам, вероятно, придется поломать голову, чтобы определиться, какой использовать. Вам следует найти немного старого доброго здравого смысла и применить ваши знания о наследовании, чтобы достичь поставленной цели.

Чтобы наследовать класс, нужно просто включить определение одного класса в другое, используя ключевое слово extends. Чтобы увидеть, как это делается, начнем с короткого примера. Следующая программа создает суперкласс с именем A и подкласс с именем B. Обратите внимание, как используется ключевое слово extends, чтобы создать подкласс A. Ключевое слово extends означает, что новый класс является производным от другого класса, указанного после ключевого слова extends.

```
// Простой пример наследования.
// Создать суперкласс.
claSS A (
  int i, j;
  void showij() {
    System.out.print1n("i и j: " + i + " " + j);
  }
}
```

```
// Создать подкласс расширением класса A.
class B extends A {
  int k;
   void showk() {
     System.out.println{"k: " + k);
   }
   void sum() {
     System.out.println{"i+j+k: " + (i+j+k));
   }
}

classSimpleInheritance {
 public static void main(String args[]) {
  A superOb = new A();
  B subOb = new B();

  //Суперкласс может быть использован сам по себе.
  SuperOb.i  = 10;
  SuperOb.j  = 20;
   System.out.println("Содержимое superQb: ");
   SuperOb.showij();
   System.out.println();

  /* подкласс имеет доступ ко всем public-членам его суперкласса */
   subOb.i = 7;
   subOb.j = 8;
   suьOb.k = 9;
   System.оut.println("Содержимое of зиbOb: ");
   subOь.showij();
   subOb.showk();
   System.out.println();
   System.out.println("Сумма i, j и k в subOb: ");
   SubOb.sum();
  }
}
```

Подкласс может наследовать *public*- и *protected*- члены суперкласса. Здесь подкласс B включает все члены его суперкласса A. Вот почему объект subOb может обращаться к i и j и вызывать showij(). Поэтому же внутри sum() можно прямо ссылаться на i и j, как если бы они были частью B.

Хотя A – суперкласс для B, он тоже полностью независимый, автономный класс. Роль суперкласса для некоторого подкласса не означает, что этот суперкласс не может использоваться сам по себе. Более того, подкласс может быть суперклассом для другого подкласса.

Вывод этой программы:

```
Содержимое superOb:
 i  и j : 10 20

Содержимое superOb:
 i  и j : 7 8
k:  9

Сумма i, j и k  subOb:
i+j+k: 24
```

Ниже показана общая форма объявления класса, который наследует суперкласс:

```
class subclassName extends superclassName {
  // тело класса
}
```

где subclassName - имя подкласса, superclassName - имя суперкласса. Можно определять только один суперкласс для любого подкласса, который вы создаете. Java не поддерживает наследования множества суперклассов в одиночном подклассе. (Это отличается от C++, в котором можно наследовать множественные базовые классы.) Можно создавать иерархию наследования, в которой подкласс становится суперклассом другого подкласса. Однако никакой класс не может быть суперклассом самого себя.

Доступ к элементам и наследование

Хотя подкласс включает все элементы (члены) своего суперкласса, он не может обращаться к тем элементам суперкласса, которые были объявлены как private. Например, рассмотрим следующую простую иерархию классов:

```
/*  в иерархии классов private члены остаются private для ее классов.
Эта про грамма содержит ошибку и не будет компилироваться. */
// Создать суперкласс.
class A {
int i;                    // public по умолчанию
private int  j;      // private для A
  void setij(int x, int y)
   i = x;
   j = y;
  }
}

// j класса A здесь не доступна
class B extends A {
   int total;
   void sum() {
       total = i + j;  // Ошибка,   j здесь не доступна
    }
}
class Access {
  public static void main(String args[]) {
      B subOb = new B();
      subOb.setij(10, 12);
      subOb.sum();
      System.out.println ("Всего " + subOb.total) ;
  }
}
```

Эта программа не будет компилироваться, потому что ссылка на j внутри метода sum() класса B вызывает нарушение правил доступа. С тех пор как j объявлена с помощью private, она доступна только другим членам его собственного класса. Подклассы не имеют доступа к ней.

Член класса, который был объявлен как private, остается private для этого класса. Он не доступен любым кодам вне его класса, включая подклассы.

Статические элементы

Иногда возникает необходимость определить элемент класса так, чтобы появилась возможность пользоваться им независимо от какого-либо объекта этого класса. Обычно к элементам класса нужно обращаться только через объект этого класса. Однако можно создать элемент для использования без ссылки на определенный объект. Чтобы это сделать, укажите в начале его объявления ключевое слово static. *Когда элемент объявляется как static, к нему можно обращаться до того, как создаются какие-либо объекты его класса, и без ссылки на какой-либо объект.* Статическими можно объявлять как методы, так и переменные. Наиболее общим примером static-элемента является метод main().

Переменные экземпляра, объявленные как static, это, по существу, *глобальные переменные.* Когда создаются объекты их класса, никакой копии static-переменной не делается. Вместо этого, все экземпляры класса совместно используют одну и ту же static-переменную.

Методы, объявленные как static имеют несколько ограничений:

- Могут вызывать только другие static-методы;
- Должны обращаться только к static-данным;
- Никогда не могут ссылаться на this или super.

Если нужно вызвать static-метод вне его класса, можно воспользоваться следующей общей формой вызова:

ИмяКласса.имяМетода;

К переменной static можно обращаться таким же образом.
Например,
```
Class StaticDemo {
  Static int a = 42;
  Static int b = 99;
  Static void callme() {
     System.out.println(“a = “ + a);
  }
}

Class StaticByName {
  public static void main(String args[]) {
    StaticDemo.callme() ;
    System.out.println(“b = “ + StaticDemo.b);
  }
}
```

Здесь внутри метода main() к статическому методу callme() и статической переменной b обращаются вне их класса.

```
Вывод этой программы
a = 42
b = 99
```

Основная особенность статических методов – они работают напрямую только со статическими полями и методами класса. Статические переменные инициализируются еще до начала работы конструктора, но при инициализации можно использовать только константные выражения. Если же инициализации требует сложных вычислений, например, циклов для задания значений элементам статических массивов или обращений к методам, то эти вычисления заключают в блок, помеченный словом static, который тоже будет выполнен при загрузке класса, до конструктора:

```
static int[ ] a = new  a[10];
static  { for (int k = 0; k < a.length; k++)  a[k] =k*k; }
```

Операторы, заключенные в такой блок, выполняются только один раз, при загрузке класса, а не при создании каждого экземпляра.

Использование ключевого слова *super*

В предшествующих примерах классы, производные от Box, не были реализованы так эффективно или устойчиво, как это могло бы быть. Например, предположим пусть создан новый класс BoxWeight, который расширяет класс Box, добавляя новый член с именем weight. Таким образом, новый класс будет содержать ширину, высоту, глубину и вес блока. Пусть в этом новом классе определен также его конструктор следующим образом:

```
//Box расширяется для включения веса.
class BoxWeight extends Box  {
   double weight;          // вес блока
   // конструктор для BoxWeight        :
   BoxWeight(double w,  double h,  double d,  double m)  {
     width = w;
     height = h;
     depth = d;
     weight = m;
   }
}
```

При таком подходе конструктор класса BoxWeight явно инициализирует код своего суперкласса, что неэффективно, но это означает, что подклассу должен быть предоставлен доступ к этим членам. Однако наступит момент, когда вы захотите создать суперкласс, который сохраняет подробности своей реализации для себя (т. е. хранит свои компоненты данных как private). В этом случае у подкласса не будет никакой возможности прямого доступа или инициализации этих переменных как своих собственных. Так как инкапсуляция это первичный атрибут ООП, не удивительно, что Java обеспечивает решение этой проблемы. Всякий раз, когда

подкласс должен обратиться к своему непосредственному суперклассу, он может сделать это при помощи ключевого слова *super*.

Ключевое слово super имеет две общие формы. Первая вызывает конструктор суперкласса. Вторая используется для доступа к элементу суперкласса, который был скрыт элементом подкласса. Далее рассматриваются обе формы.

Вызов конструктора суперкласса с помощью первой формы *super*

Подкласс может вызывать метод конструктора, определенный его суперклассом, при помощи следующей формы super:

```
super (parameter-list) ;
```

Здесь *parameter-list* — список параметров, который определяет любые параметры, необходимые конструктору в суперклассе. Похожий по форме на конструктор super() должен всегда быть первым оператором, выполняемым внутри конструктора подкласса.

Чтобы посмотреть, как super о используется, приведем следующую улучшенную версию класса BoxWeight:

```
//Box теперь использует super для инициализации Box-атрибутов.
class BoxWeight extends Box  {
   double weight;          // вес блока
   //инициализировать width, height, depth, weight, используя super()
   BoxWeight(double w,  double h,  double d,  double m)  {
     super(w, h, d); // вызвать конструктор суперкласса
     weight = m;
   }
}
```

Здесь BoxWeight() вызывает super() с параметрами w, h и d. Используя эти параметры, super () вызывает конструктор Box (), который инициализирует width, height, depth. BoxWeight больше не инициализирует эти значения самостоятельно. Он нуждается в инициализации только уникальной для него переменной — weight. Box, если пожелает, может закрыть свои переменные (сделать их private).

Использование второй формы super

Вторая форма super действует в чем-то подобно ссылке this, за исключением-того, что она всегда обращается к суперклассу подкласса, в котором ис-пвйьзуется. Общий формат такого использования siiper имеет вид:

super.member

где member может быть либо методом, либо переменной экземпляра. Вторая форма super больше всего применима к ситуациям, когда имена элементов (членов) подкласса скрывают элементы с тем же именем в суперклассе. Рассмотрим следующую простую иерархию классов:

```
// Использование super для преодоления скрытия имен
class A  {
  int  i;
}

// Создание подкласса B расширением класса A.
class B extends A  {
   int i;                          // этот i скрывает i  в A
   B(int a,  int b)  {
     super.i = b;  //  i из A
     i = b;        //  i из B
   }

    void show() {
      System.out.println("i из суперкласса: " + super.i);
      System.out.println("i из подкласса: " + super.i);
    }
}

class UseSuper {
```

```
  public static void main( String args[] ) {
  B subOb = new B(1, 2);
  SubOb.show();
  }
}
```

```
Эта программа выполняет следующий вывод:
```

```
    i из суперкласса:  1
    i из подкласса:  2
```

Хотя экземплярная переменная i класса в скрывает i класса A, super позволяет получить доступ к i, определенной в суперклассе. Кроме того, super можно также использовать для вызова методов, скрытых подклассом.

Особенности взаимодействия подкласса и суперкласса

1. Когда подкласс должен обратиться к своему непосредственному суперклассу, он может сделать это при помощи ключевого слова *super*. Ключевое слово *super* имеет две формы. Первая вызывает конструктор суперкласса. Вторая используется для доступа к элементу суперкласса, который был скрыт элементом подкласса.

2. Можно строить иерархии, которые содержит столько уровней наследования, сколько вам нравится.

3. В иерархии классов конструкторы вызываются в порядке подчиненных классов – от суперкласса к подклассу.

4. В иерархии классов, если метод в подклассе имеет такое же имя и сигнатуру типов, как метод в его суперклассе, говорят, что метод в подклассе *переопределяет* (*override*) метод в суперклассе. Когда переопределенный метод вызывается в подклассе, он будет всегда обращаться к версии этого метода, определенной подклассом.

Переопределение метода происходит только тогда, когда имена и сигнатуры типов этих двух методов идентичны. Если это не так, то оба метода просто перегружены.

Когда переопределенный метод вызывается через ссылку суперкласса, Java определяет, какую версию этого метода следует выполнять, основываясь на *типе объекта,* на который указывает ссылка в момент вызова. Это делается *во время выполнения*. Когда ссылка указывает на различные типы объектов, будут вызываться различные версии переопределенного метода. Другими словами, именно *тип объекта, на который сделана ссылка* (а не тип ссылочной переменной) определяет, какая версия переопределенного метода будет выполнена. Такой механизм Java называется динамической диспетчеризацией методов.

Переопределенные методы позволяют поддерживать полиморфизм времени выполнения.

Чтобы отменить переопределение метода, укажите модификатор *final* в начале объявления. Методы, объявленные как *final*, может использоваться еще для встраивания (inline) *final-метода* прямо в точку вызова откомпилированного кода вызывающего метода. Так как *final-методы* не являются переопределяемыми, их вызов может быть организован во время компиляции. Это называется уже *ранним связыванием*.

Иногда нужно разорвать наследованную связь классов (отменить наследование одного класса другим). Чтобы сделать это, предварите объявление класс ключевым словом *final*, что позволит неявно объявить и все его методы.

Итак, наследование – это форма *многократного использования* программного обеспечения, в которой новые классы создаются на основе уже существующих классов, поглощая их атрибуты и методы поведения и добавляя новые возможности, которые требуются в новых классах. Основное преимущество наследования заключается в возможности определения в подклассе *новых особенностей* или *замены тех, что унаследованы от суперкласса.*

5. В процессе работы над объектно-ориентированным проектом проектировщик анализирует задачу и формирует необходимых наборов суперклассов. Часто классы задачи тесно связаны.

Поэтому «выносите общий множитель за скобки», т.е. выделяйте общие атрибуты и методы анализа в суперклассы. Затем в подклассах добавляются необходимые возможности помимо тех, что унаследованы от суперклассов. Так создается необходимая иерархия классов задачи. Абстрактные классы используются как суперклассы в иерархии наследования. При разработке программ в них можно предусмотреть алгоритмы обработки объектов всех существующих классов в иерархии в общем виде – как объектов суперкласса. Все классы в иерархии могут использовать интерфейс абстрактного класса через механизм полиморфизма.

Лекции 11-12. Полиморфизм.

Мы продолжим обсуждение методов и классов, чтобы познакомиться с приемами реализации принципа полиморфизма. Мы делаем это, потому что полиморфизм в программах реализуется с помощью двух или более методов. Полиморфизм означает, что что-то может быть во многих формах - это что-то представляет собой метод в объектно-ориентированном языке программирования. В данном случае форма – это поведение, которое осуществляет метод.

Мы знаем, что определение метода определяет поведение объекта. Название метода используется для вызова метода из оператора программы, а список аргументов содержит данные, необходимые для осуществления методом его поведения. Вместе название метода и его аргументы называются *сигнатурой* метода. Тело метода содержит один или более операторов, которые выполняются, когда вызывается метод. Вот где на самом деле осуществляется поведение. Возвращаемое значение – это значение, которое возвращается программе после того, как метод закончит выполняться. Некоторым методам не требуется список аргументов или возвращаемое значение.

Полиморфизм

Полиморфизм в терминах программирования означает, что метод с одним названием может осуществлять множество вариантов поведения.

Полиморфизм – весьма существенная черта объектно-ориентированного программирования по одной причине: он позволяет базовому классу определять методы, которые будут общими для всех его производных классов, и, в то время, разрешает подклассам определять специфические реализации некоторых или всех таких методов.

Переопределяемые методы – это один из способов реализации аспекта полиморфизма «один интерфейс, множественные методы». Другой способ реализации этого аспекта – *перегрузка (overloading)* методов.

Применение переопределения методов.

Рассмотрим пример, который использует переопределение методов. Следующая программа создает суперкласс с именем Figure, который хранит размеры различных двумерных объектов. Он также определяет метод с именем area(), который вычисляет площадь объекта. Программа определяет также два Figure. Первый – Restangle, а второй – Triangle. Каждый из этих подклассов переопределяет area() так, чтобы он возвращал площадь прямоугольника и треугольника, соответственно.

```
//Использование полиморфизма времени выполнения
class Figure {
   double dim1;
   double dim2;

   Figure (double a, double b) {
    dim1 = a;
    dim2 = b;
   }
```

```
  double area() {    // в языке Java метод является виртуальной по умолчанию
    System.out.println("Площадь Figure неопределена");
    Return 0;
  }
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
     Super(a, b);
  }

  // переопределить метод area для прямоугольника
  double area() {
    System.out.println("area для Rectangle");

    return dim1 * dim2;
  }
}

class Triangle extends Figure {
   Triangle (double a, double b) {
   Super(a, b);
   }

  // переопределить area для  треугольника
  double area() {
    System.out.println("area для Triangle");
    Return dim1 * dim2 / 2;
  }
}

class FindAreas {
  public static void main(String args[]){
    Figure f = new Figure(10, 10) ;
    Rectangle r = new Rectangle(9, 5); // r ссылается на подкласс Rectangle
    Triangle t = new Triangle(10, 8);
    Figure figref; // переменная figref типа суперкласса
    figref = r;  // переменная суперкласса figref ссылается уже на подкласс
    System.out.println("Площадь = "+figref.area());
    figref = t;
    System.out.println("Площадь=" + figref.area());
    figref = f;
    System.out.println("Площадь=" + figref.area());
  }
}
```

```
Вывод этой программы

area для Rectangle
Площадь= 45
area для Triangle
Площадь= 40
ПлощадьFigure неопределена
Площадь = 0
```

Через двойные механизмы наследования и полиморфизма времени выполнения можно определить один непротиворечивый интерфейс, который используется несколькими различными, но связанными типами объектов. В этом случае, если объект является производным от Figure-объекта, то его площадь может быть получена с помощью вызова метода area(). Интерфейс этой операции один и тот же **figref.area()**, независимо от того, какой тип фигуры применяется.

Это обусловлено тем, что в объектно-ориентированном языке есть правило, что переменная суперкласса может ссылаться на экземпляр подкласса (наоборот – нет). А также всегда каждый объект «знает» свой тип и класс своего родителя.

Это используется как раз для вызова переопределенных методов на этапе выполнения. Так, если метод (в нашем случае **area()**), вызывается посредством ссылки на суперкласс (в нашем

случае это делается как **figref.area()**), то испольнительная система Java согласно типу объекта на этапе выполнения управление передает варианту метода, на который в данный момент ссылается переменная. В нашем случае по

```
    figref = r;   переменная суперкласса figref ссылается уже на подкласс Rectangle
```

потому при выполнении

```
    System.out.println("Площадь = " + figref.area());
```

делается вызов метода подкласса Rectangle, а не суперкласса Figure, хотя переменная figref – переменная типа Figure (согласно Figure figref;).

Связывание

Каждый раз, когда вы вызываете метод в своем приложении, вызов метода должен быть ассоциирован с определением метода. Программисты называют этот процесс *связыванием.* Связывание – это установление связи вызова метода с определением метода. Связывание происходит либо в процессе компиляции, либо во время выполненя. Связывание во время компиляции называется *ранним связыванием* и используется, если вся информация, необходимая для вызова метода, известна на момент компиляции приложения. Связывание *во время выполнения* называется *поздним связыванием* и используется, если какая-то информация отсутствует во время компиляции и становится известной только во время выполнения приложения.

Раннее связывание используется для вызова обычных методов. Связывание во время выполнения реализуется с помощью *виртуальных функций*, которые используют ссылки на базовый тип объекта, содержащий корректное определение метода. Слово виртуальная означает что-то, что кажется реальным, но таковым не является. В случае виртуальной функции компьютер «обманывается» определением функции, но на самом деле функция на тот момент не определена. Виртуальная функция – это «заполнитель» реальной функции. Реальная функция определяется во время выполнения программы.

Виртуальные функции могут быть «настоящими» функциями или просто заполнителями для реальных функций, которые должны быть реализованы в производных классах, иначе говоря, должны быть *переопределены* в производных классах. Выше в приведенном примере, виртуальный метод `double area()` был переопределен в поизводных классах.

Полиморфизм реализуется с помощью ***перегрузки методов*** *или с помощью* ***виртуальных функций (методов).*** В языке Java методы являются виртуальными по умолчанию, если только вы не указали ключевое слово *final*. В языке Java виртуальность метода не отмечается специальными модификаторами как в других языках. В языках ООП виртуальность метода допускает возможность его переопределения в подклассах. Реализацию полиморфизма с помощью перегрузки методов мы рассмотрим позже, а здесь мы рассмотрели реализацию полиморфизма с помощью виртуальных методов, или, иначе говоря, с помощью *переопределения метода*.

Полиморфизм еще называется *динамическим связыванием, поздним связыванием* или *связыванием во время выполнения.* Связывание всех методов в Java основано на механизме позднего связывания, только если метод не был объявлен как final. Значит, связывание всех методов в Java происходит полиморфно. Обычно вам не придется решать, где стоит использовать позднее связывание, - это происходит автоматически.

Динамический (т.е. реализуемый во время выполнения) полиморфизм – это один из наиболее мощных механизмов, привносимых объектно-ориентированным проектированием для обеспечения многократного использования кода и устойчивости к ошибкам. Способность существующих кодовых библиотек вызвать методы на экземплярах новых классов без перекомпиляции и при поддержке ясного абстрактного интерфейса – чрезвычайно мощный инструмент языка Java.

Полиморфизм позволяет улучшать организацию кода и его читаемость, а также создавать расширяемые программы, которые могут «расти» не только в процессе начальной разработки проекта, но и при добавлении новых свойств.

Если инкапсуляция создает новые типы данных, совмещая характеристики и поведение. А сокрытие реализации разделяет интерфейс и реализацию, делая детали закрытыми (private). А наследование позволяет обращаться с объектом, используя как его собственный тип, так и его базовый тип. Эта возможность очень важна, потому что она позволяет обращаться со многими типами (произведенными от одного базового типа), как с единым типом, что позволяет единому коду работать с множеством разных типов. То полиморфизм обращается с разделением в терминах типов. Невозможно понять и создать примеры с использованием полиморфизма, не используя абстракции данных и наследование.

При проектировании программной системы особое внимание должно концентрироваться не на частных случаях, а на *общей картине* – унифицировать объекты в системе по схеме *«суперкласс-подкласс»*. При этом используется приемы абстрагирования и потому этот процесс кратко называется *абстракцией данных*.

Использование абстрактных классов

Иногда необходимо создать суперкласс, определяющий только обобщенную форму, которая будет раздельно использоваться всеми его подклассами. Предоставляя каждому подклассу возможность заполнить ее своими деталями. Такой класс определяет только природу методов, конкретно реализуемых подклассами. Подобная ситуация может возникнуть, например, когда суперкласс не способен создать значимую реализацию метода. В таком случае находится класс Figure в предыдущем примере. Определение area() можно рассматривать в качестве «хранителя места». Метод не вычисляет и не отображает площадь ни для какого объекта. Он просто выдает предупреждающее сообщение. Абстракция реализуется с помощью определения абстрактного класса. В Java для этого используется ключевое слово `abstract.` Абстрактный класс – это класс, создать экземпляр которого невозможно. То есть вы не можете объявить экземпляр абстрактного класса. Вы можете определять данные и методы-члены абстрактного класса. Для того чтобы данные и поведение абстрактного класса можно было использовать в приложении, он должен быть унаследован подклассом. Методы-члены, если это необходимо, могут быть обозначены как абстрактные. Метод-член, обозначенный как абстрактный, заставляет программиста подкласса переопределить этот абстрактный метод.

Если вы определяете виртуальную функцию без тела, это означает, что оно должно быть реализовано в производном классе (выбора нет, иначе программа не откомпилируется). Классы с такими функциями называются *абстрактными классами*, потому что это не законченные классы, а, скорее, указание для создания реальных классов. В Java для создания виртуальных функций без тела используется ключевое слово `abstract.` Виртуальные функции служат для реализации полиморфизма времени выполнения (позднего связывания)

Можно также с помощью модификатора типа abstract потребовать, чтобы методы были переопределены подклассами. В таком случае подкласс должен переопределить их – он не может просто использовать версию, определенную в суперклассе. Для объявления абстрактного метода используется следующая общая форма:

```
abstract  тип(список параметров);
```

Обратите внимание, что тело метода отсутствует.

Любой класс, который содержит один и более абстрактных методов, должен также быть объявлен абстрактным. Чтобы объявить абстрактный класс, просто используйте ключевое слово `abstract` перед ключевым словом class. Нельзя создавать никакие объекты абстрактного класса. Вы не можете также объявлять абстрактные конструкторы или абстрактные статические методы. Для того чтобы данные и поведение абстрактного класа можно было использовать в

приложении, он должен быть унаследован подклассом. Любой подкласс абстрактного класса должен или реализовать все абстрактные методы суперкласса, или сам должен быть объявлен как abstract. Таким образом, программисту подкласса придется переопределять абстрактные методы, иначе он получит ошибку при компиляции.

Итак, абстрактный метод – это метод, определенный в суперклассе, который должен быть переопределен в подклассах, наследуемых от суперкласса. Абстракция используется, когда не существует никакого способа определить метод по умолчанию в суперклассе. Абстрактный метод не может быть вызван в программе. Если не переопределить абстрактный метод, при компиляции вы получите сообщение об ошибке. Абстрактный метод не может быть вызван напрямую подклассом. Программист подкласса, наследуемого от абстрактного суперкласса, должен переопределить абстрактные методы, определенные в суперклассе, даже если эти методы не вызываются в приложении. Суперкласс может содержать как абстрактные, так и неабстрактные методы. Только абстрактные методы должны быть переопределены в подклассе, который наследуется от абстрактного суперкласса.

При использовании абстракции обычно возникают три типа ошибок:

- Программист не определил абстрактный метод в подклассе.
- Попытка программиста вызвать абстрактный метод суперкласса.
- Попытка объявить в программе экземпляр абстрактного суперкласса.

Хотя абстрактные классы не могут использоваться для создания объектов, их можно использовать для создания объектных ссылок, потому что *подход Java к полиморфизму времени выполнения реализован с помощью ссылок суперкласса.* Таким образом, возможно создавать ссылку к абстрактному классу так, чтобы она могла использоваться для указания на объект подкласса. Вы увидите использование этого свойства в следующем примере.

```
//Использование абстрактных методов и классов
abstract class Figure {
   double dim1;
   double dim2;

   Figure (double a, double b) {
    dim1 = a;
    dim2 = b;
   }

   // area теперь абстрактный метод
   abstract double area(); /* функция объявлена, но ее функциональность (поведение)
                              не определена */
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
     Super(a, b);
  }

  // переопределить area для прямоугольника
  double area() {
    System.out.println("area для Rectangle");
    return dim1 * dim2;
  }

}

class Triangle extends Figure {
   Triangle (double a, double b) {
   Super(a, b);
  }

  // переопределить area для  треугольника
  double area() {
    System.out.println("area для Triangle");
```

```
    Return dim1 * dim2 / 2;
  }
}

class AbstractAreas {
  public static void main(String args[]){
  //Figure f = new Figure(10, 10);//теперь незаконно
  Rectangle r = new Rectangle(9, 5) ;
  Triangle t = new Triangle(10, 8);
  Figure figref; // ОК, объект не создается
  figref = r;
  System.out.println("Площадь = "+figref.area());
  figref = t;
  System.out.println("Площадь=" + figref.area());
  }
}
```

Как указывает комментарий внутри main(), больше невозможно объявлять объекты типа `Figure`, так как класс теперь абстрактный. Кроме того, все подклассы `Figure` должны переопределять `area()`. Чтобы доказать это, пробуйте создать подкласс, который не переопределяет `area()`. Вы получите ошибку времени компиляции.

Хотя невозможно создать объект типа `Figure`, можно создать ссылочную переменную типа `Figure`. Переменная `figref` объявлена как ссылка на `Figure`. Что означает, что она может использоваться для ссылки на объект любого класса, производного от Figure. Напомним, что именно через ссылочные переменные суперкласса выбираются переопределенные методы (во время выполнения).

Частичным залогом успешного применения полиморфизма является понимание того, что суперклассы и подклассы формируют иерархию, которая движется от меньшей к большей специализации. Суперкласс обеспечивает все элементы, которые подкласс может использовать непосредственно. Он также определяет те методы, которые производный класс должен реализовать на свой собственный манер. Это позволяет подклассу гибко определять свои методы и одновременно предписывает непротиворечивый интерфейс. Таким образом, комбинируя наследование с переопределением методов, суперкласс может определять общую форму методов, которая будет использоваться всеми его подклассами.

Наследование – это форма многократного использования программного обеспечения, в которой новые классы создаются на основе уже существующих классов, поглощая их атрибуты и методы поведения и добавляя новые возможности, которые требуются в новых классах. Основное преимущество наследования заключается в возможности определения в подклассе *новых особенностей* или *замены тех, что унаследованы от суперкласса.*

Перегрузка методов

Перегрузка (overloading) – это еще один из терминов, который вы можете услышать вместе с полиморфизмов. Он означает, что два или более метода имеют одно название, но разные списки аргументов. Перегрузка метода представляет нам способ реализовать похожее поведение дляразных типов данных с помощью написания своей версии метода для каждого используемого типа данных. Любая вариация списка аргументов делает метод отличным отдругих методов с таким же названием. То есть списки аргументов с разным количеством аргументов, типами данных аргументов и порядком аргументов считаются различными.

В языке Java в пределах одного класса можно определить два или более методов, которые совместно используют одно и то же имя, но имеют разное количество параметров. Перегружают только тесно связанные операции, хотя когда перегружают метод, допускается использовать любое действие, какое вы пожалеете. Когда это имеет место, методы называют *перегруженными*, а о процессе говорят как о *перегрузке метода*. Когда испольняющая система Java сталкивается с вызовом перегруженного метода, он просто выполняет его (метода) версию, чьи параметры соответствуют параметрам, используемым в вызове.

```
// Демонстрация перегруженного метода
class OverloadDemo {
```

```
  void test() {
    System.out.println("Параметры отсутствуют");
  }

  //Перегруженный метод test с одним int-параметром.
  void test (int a) {
    System.out.println("a =”  + a);
  }

  //Перегруженный метод test с двумя int-параметром.
  void test (int a, int b) {
    System.out.println("a,  b: “ + a+” “ +b);
  }

  //Перегруженный метод test с одним double-параметром.
  void test (double  a) {
    System.out.println("double a: “ + a);
    return a*a;
  }
}

class Overload {
  public static void rnain(String args[])
    OverloadDemo ob =  new OverloadDemo();
    double result;
    // Вызываются все версии test()
    ob. test();
    ob. test (10) ;
    ob.test (10, 20);
    result =  ob.test(123.2);
    System.out.println("Результат ob.test(123.2): “  + result);
  }
}
```

Эта программа генерирует следующий вывод:

```
Параметры отсутсвуют
a=10
a, b: 10 20
вещественное двойной точностью a: 123.2
Результат ob.test (123.2): 15178.24
```

Как можно видеть, метод `test()` перегружен четыре раза. Первая версия не имеет никаких параметов, вторая имеет один параметр целого типа, третья – два целочисленных параметра, а четвертая – один double-параметр. Когда вызывается перегруженный метод, Java ищет соответствие между аргументами вызова метода и его параметрами. Поэтому в тексте программы перегруженные методы не сопровождаются специальными спецификаторами.

Итак, перегрузка методов – один из способов, с помощью которого Java реализует полиформизм. На языках, которые не поддерживают перегрузку методов, каждому методу необходимо давать уникальное имя. Значение *перегрузки (и полиморфизма)* заключается в том, что она позволяет осуществлять доступ к связанным методам при помощи *общего имени*. При использовании полиморфизма несколько имен были сокращены до одного. Это помогает программисту управлять большей сложностью. Возможно также перезагружать методы конструкторов.

Полиморфизм особенно эффективен при разработке многоуровневых систем программного обеспечения. Полиморфизм работает и когда новые классы добавляются в систему. Поэтому увеличивает многократность использования программного кода.

Лекция 13. Пакеты и библиотека классов Java

Программы Java состоят из частей, называемых классами. В свою очередь классы состоять из частей, называемых методами, которые выполняют (решают) определенные задачи и возвращают информацию по завершении своей рабты.

В практике программирования задач принято разработанные и испытанные классы группировывать и собирать в пакеты. Иначе говоря, могочисленные предопределенные классы Java группируются по категориям связанных классов, и эти группы называются *пакетами.* Пакет кроме классов еще могут содержат *интерфейсы*. Все вместе эти пакеты объединяются термином *библиотека классов Java*, или *программный интерфейс приложений Java (API Java).*

Таблица некоторых пакетов Java API

Имя пакета	Содержимое
java.applet	Классы для реализации апплетов
java.awt	Классы для работы с графикой, текстом, окнами и GUI
java.awt.datatransfer	Классы для обеспечения передачи информации (Copy/Paste)
java.awt.event	Классы и интерфейсы для обработки событий
java.awt.image	Классы для обработки изображений
java.awt.peer	GUI для обеспечения независимости от платформы
java.beans	API для модели компонентов JavaBeans
java.io	Классы для различных типов ввода-вывода
java.lang	Классы ядра языка (типы, работа со строками, тригонометрические функции, обработка исключений, легковесные процессы)
java.lang.reflect	Классы Reflection API
java.math	Классы для арифметических операций произвольной точности
java.net	Классы для работы в сети Интернет (сокеты, протоколы, URL)
java.rmi	Классы, связанные с RMI (удаленный вызов процедур)
java.rmi.dgc	Классы, связанные с RMI
java.rmi.registry	Классы, связанные с RMI
java.rmi.server	Классы, связанные с RMI
java.security	Классы для обеспечения безопасности
java.security.acl	Классы для обеспечения безопасности
java.security.interfaces	Классы для обеспечения безопасности
java.text	Классы для обеспечения многоязыковой поддержки
java.text.resources	Классы для обеспечения многоязыковой поддержки
java.util	Разноообразные полезные типы данных (стеки, словари, хэш-таблицы, даты, генератор случайных чисел)
java.util.zip	Классы для обеспечения архивации

Классы пакета называются еще *классами повторного использования*, потому что пользователь может импортировать в свою программу любой класс из пакета. А пакет **java.lang** импортируется в каждую программу Java по умолчанию, ибо там содержатся классы, необходимые для выполнения на компьютере простейшей программы Java. Другая выгода использования пакетов состоит в том, что они позволяют поддерживать *уникальные имена классов*. Поскольку программируют во всем мире, существует большая вероятность того, что имена, которые вы подберете для своих классов, совпадут с именами, которые другие программисты присвоили своим классам. Каждый *пакет* имеет свое пространство имен, что позволяет создавать одноименные классы в различных *пакетах*. Таким образом, названия пакетов также служат для того, чтобы можно было использовать два класса с одинаковыми именами, если имена их.

Пакет является механизмом как именования, так и управления видимостью. Вы можете определять внутри пакета классы, которые не доступны кодам вне этого пакета, Вы можете также определять члены класса, доступные только другим членам того же самого пакета. Так все классы, входящие в данный пакет, будут иметь доступ к защищенному

protected-методу класса пакета, а классы, не входящие в пакет, не будут иметь доступа к этому методу.

Определение пакета

Создать пакет очень легко: просто включите оператор **package** в начало исходного файла Java. Любые классы, объявленные в пределах того файла, будут принадлежать указанному пакету. Оператор package определяет пространство имен, в котором сохраняются классы. Общая форма инструкции package:

```
package имяпакета;
```

Например, следующая инструкция создает пакет с именем MyPackage.

```
package MyPackage;
```

Чтобы хранить пакеты, Java использует каталоги файловой системы. Например, class-файлы для любых классов, которые вы объявляете как часть пакета MyPackage, должны быть сохранены в каталоге с именем MyPackage.

Можно создавать иерархию пакетов. Для этого необходимо просто отделить каждое имя пакета от стоящего выше при помощи операции «точка». Например, пакет объявленный как

```
package  java.awt.image;
```

тогда должен быть сохранен в каталоге java\awt\image. Нельзя переименовывать пакет без переименования каталога, в котором хранятся классы.

Вы должны знать, что размещением **корня** любой иерархии пакетов в файловой системе компьютера управляет специальная переменная окружения CLASSPATH. Переменная окружения CLASSPATH определяет место, где компилятор Java и JRE будут искать пользовательские пакеты. Значение CLASSPATH содержит список папок и (или) имена jar-файлов. Поэтому этот факт нужно учитывать при использовании класса из пакета.

Пример простого пакета

```
//Простой пакет
package myPackage;
class Balance {
  String name ;
  double bal;
  Balance (String n, double b) {
    Name = n;
    Bal = b;  }
  void show() {
    if  (bal < 0)
     System.out.print("—» ");
     System.out.println(name + ":  $"  + bal); }  }
class AccountBalance  {
  public static void main(String args[])  {
  Balance current[] = new Balance[3];
  current[O]  = new Balance("K...J.  Fielding",  123.23);
  current[1] = new Balance("Will Tell",  157.02);
  current[2] = new Balance("Tom Jackson",  -12.33);
  for(int i=0;  i<3;  i++)  current[i].show(); }
}
```

Если в тексте программы не будет указано объявление пакета, (например, `package MyPackage;`), то классы такой программы будут рассматриваться как часть пакета по умолчанию. Для компиляции такой программы нужно перейти в каталог, в котором содержится файл AccountBalance.java, и набрать команду (в предположении, что путь к javac известен системе):

```
javac AccountBalance.java
```

Если в тексте программы будет указано объявление пакета (например, `package myPackage;`), то в созданном пользователем каталоге, например, `C:\myjava`, где пользователь сохраняет свои `.java` файлы, нужно создать подкаталог `myPackage`, совпадающий с именем пакета. Файл

AccountBalance.java должен быть теперь сохранен в этом подкаталоге. Под Windows получится файл с именем

```
C:\myjava\myPackage\ AccountBalance.java
```

Создав структуру каталогов и разместив в ней программу, вы должны сообщить компилятору и интерпретатору Java, где ее искать. Компилятору и интерпретатору нужно знать только выбранный вами базовый каталог (в нашем случае это `C:\myjava`); файл AccountBalance они будут искать в его подкаталоге, ориентируясь по имени пакета. Чтобы указать Java направление поиска, вы должны присвоить значение переменной окружения CLASSPATH способом, соответствующим вашей операционной системе. Для Windows, если исходный файл сохраняется в папке `C:\myjava`, можно использовать следующую команду:

```
C:\>set CLASSPATH=.,c:\myjava
```

Эта команда предлагает Java искать программу сперва в текущем каталоге (.), а затем в каталоге `c:\myjava`

Можно автоматизировать этот процесс, включив команду установкм переменной CLASSPATH в стартовый файл, в autoecex.bat под Windows.

Файл, соответствующий тексту программы, назовите AccountBalance.java и поместите его в каталог с именем `myPackage`.. Далее, откомпилируйте файл и удостоверьтесь, что результирующий .class-файл также находится в каталоге `myPackage`. Для запуска программы следует вызвать интерпретатор Java, но этот раз нам придется указать полностью квалифицированное имя программы, чтобы интерпретатору стало в точности известно, какую именно программу вы хотите исполнить:

```
java myPackage.AccountBalance
```

Установив значение CLASSPATH, вы можете запускать интерпретатор Java из любого каталога вашей системы, и он всегда найдет нужную программу.

Итак, класс AccountBalance.class теперь является частью пакета `myPackage`. Это означает, что он не может быть выполнен отдельно. То есть вы не можете использовать следующую командную строку:

```
java AccountBalance
```

AccountBalance должен быть квалифицирован именем своего пакета. Если набор вручную таких длинных имен утомляет, вы, возможно, изготовите для себя подходящий файл, который будет выполняить это автоматически.

Создание повторно используемого класса

Мы познакомились как нужно создавать свои собственные пакеты и использовать классы из этого пакета. Теперь познакомимся как надо создавать класс, который будет повторно использоваться. Для этого нужно выполнить следующме шаги:

1. Определить открытый класс. Если класс не объявить как открытый, то его могут использовать только другие классы из этого же самого пакета.
2. Указать корректное имя пакета. С целью обеспечения уникальных имен для каждого пакета, компания определила соглашение для имен пакетов, которому должны следовать все программисты на Java. Каждое имя пакета должно начинаться с вашего имени домена в Интернете в обратном порядке. После того как сформирована доменная часть имени пакета, вы можете выбирать любые другие имена, которые вы захотите использовать для вашего пакета.
3. Откомпилировать **.java**-файл и созданный **.class**-файл скопировать (или сохранить) в папку со структурой, которая соответствует названию пакета. Иначе говоря, откомпилировать класс так, чтобы он попал в соответствующую структуру каталогов пакета. Имена каталогов, указанные в операторе package, становятся частью имени класса, когда класс компилируется. Затем присвоить переменной окружения CLASSPATH значение, которое бы указывало на папку, содержащую **корень** структуры папок. Также включите папку **«.»** (текущая

папка). После того как класс откомпилирован, вы можете использовать это полное составное имя в ваших программах.

4. Начать создания Java–приложения с добавления оператора **package**, в котором определяется класс для последующего многократного использования..
5. Импортировать повторно используемый класс, в программу и использовать его. Для этого после первого package-предложения нужно написать второе предложение вида:
   import польноеИмяПакета.имяПовторноИспользуемогоКласса;
   Затем расписать остальные действия работы с классом и при этом можно в тексте программы использовать только имя класса.

Итак, в общем случае исходный файл Java (т.е. исходный текст Java-программы) может содержать любую (или все) из следующих четырех внутренних частей: в начале стоит одиночный package-*оператор* (если он необходим); затем любое число *import-операторов* (не обязательно); а затем одиночное объявление *общего (public) класса* с методом main (объязательно требуется); и любое число *частных классов* (не обязательно). Только один из определямых классов может быть открытым. Другие классы, определенные в файле, также помещаются в пакет, но они могут использоваться только в других классах из этого пакета. Они не могут импортироваться в классы из другого пакета. Они выполняют в пакете вспомогательную роль – поддерживают класс повторного использования, определяенный в файле.

Защита доступа

Ранее вы познакомились с различными аспектами механизма управления доступом Java и его спецификаторами доступа. Например, вы уже знаете, что доступ к private-члену класса предоставляется только другим членам того же класса. Пакеты добавляют еще одно измерение к управлению доступом. Java обеспечивает достаточно уровней защиты, чтобы допустить хорошо разветвленный контроль над видимостью переменных и методов в пределах классов, подклассов и пакетов.

Классы и пакеты, с одной стороны, обеспечивают инкапсуляцию, а с другой — поддерживают пространство имен и области видимости переменных и методов. Пакеты действуют как контейнеры для классов и других зависимых пакетов. Классы действуют как контейнеры для данных и кода. Класс – самый мелкий модуль абстракции языка Java.

Повторим механизм управлений доступом Java Спецификаторами доступа к члену класса могут быть ключевые слова: public, private, protected, abstact, final, static. Все, объявленное как public, может быть доступно отовсюду. Методы public класса часто называют общедоступными сервисами класса или общедоступным интерфейсом. Эти методы используются клиентами класса для работы с объектами класса. Всё, объявленное как private, не может быть видимо извне своего класса. Элементы класса, объявленные с модификатором доступа private, доступны только методам класса. Классы часто предоставляют методы типа public, дающие возможность клиентам устанавливать (set) или получать (get) переменные объектов типа private. Когда элемент не имеет явных спецификаций доступа, он видим в подклассе, также как в других классах в том же самом пакете. Это — доступ, заданный по умолчанию. Если вы хотите позволить элементу быть видимым извне вашего текущего пакета, но только в классах, являющихся прямыми подклассами вашего класса, то объявите этот элемент protected. Модификатор abstact не применим к полям-данным. Final-полю в объявлении должно быть присвоено начальное значение и значение не может быть изменено позже, а final-метод не может быть переопределен. Static-поле и static-метод доступны без образования объекта, в то время как для остальных необходимо создания объекта.

Сам класс имеет лишь два возможных уровня доступа — по умолчанию и общий (public). Когда класс объявлен как public, он доступен любым другим классам, даже вне его пакета. Если класс имеет доступ по умолчанию (т.е. модификатор public опущен), то к нему возможен доступ

только из кодов того же пакета (т.е. обращаться смогут только классы из этого пакета). Если класс объявлен с модификатором final, то класс не может наследоваться. Класс с методами-членами final сам не объязательно является final. Если класс объявлен с модификатором abstract, то нельзя объявлять экземпляры класса, класс должен быть использован в качестве суперкласса. Если класс определяет абстрактные члены, он сам должен быть объявлен как абсрактный. Этот модификатор является взаимоисключающим с final

Лекция 14. Интерфейсы

Термин интерфейс в программировании используется в связи с аппаратным, программным и пользовательским интерфейсами. Интерфейсы для объектно-ориентированного программирования служат тем же целям, но на совершенно другом уровне, а именно, увеличивая гибкость разработки программного обеспечения.

Использование множественного наследования в практике ООП пораждало множество проблем..Анализ показал, что проблему создает только реализация методов, а не их описание (иначе, говоря интерфейсная часть). Поэтому разработчики Java предложили множественное наследование заменит множественным интерфейсом. Интерфейсы могут подражать множественному наследованию, но программисты все равно должны писать код для интерфейсов в каждом классе, которые определяют и реализуют их, ибо интерфейсы не представляют повторно используемый код.

В Java, применяя ключевое слово interface, вы можете полностью освободить интерфейс класса от его реализации. То есть, используя interface, вы можете определять *, что* класс должен делать, но не *как* он это делает. Ключевое слово interface развивает понятие абстрактного класса еще дальше, вообще запрещая определения любых функций. Интерфейсы синтаксически подобны классам, но в них нет экземплярных переменных, и их методы объявляются без тела. Практически, это означает, что вы можете определять интерфейсы, которые не делают предположений относительно того, как они реализованы. Как только интерфейс определен, то реализовать его может любое число классов. И наоборот, один класс может реализовать любое число интерфейсов.

Для реализации интерфейса класс должен создать полный набор методов, определенных интерфейсом. Однако каждый класс волен сам определять детали своей реализации. Ключевое слово interface позволяет полностью использовать аспект полиморфизма, декларируемый как "один интерфейс, множественные методы".

Определение интерфейса

Определение интерфейса во многом подобно определению класса. Общая форма интерфейса выглядит так:

```
access interface name {
   return-type method-name1 (parameter-list);
   return-type method-name2(parameter-list);
   type final-varname1 = value;
   type final-varname2 = value;
   // ...
   return-type method-nameN(parameter-list);
   type final-varnameN = value;
}
```

Здесь access - спецификатор доступа (или public или не используется). Если никакой спецификатор доступа не включен, тогда используется доступ по умолчанию, и интерфейс доступен только другим членам пакета, в котором он объявлен. При объявлении с public интерфейс может использоваться любым другим кодом. name - имя интерфейса, им может быть любой допустимый идентификатор. Ключевое слово `interface` используется для объявления `name` как интерфейса, а не как класс.

Обратите внимание, что объявленные методы не имеют тел. Они заканчиваются точкой с запятой после списка параметров. Это, по существу, абстрактные методы. В пределах интерфейса для них нет никаких умалчиваемых реализаций. Каждый класс, который включает интерфейс, должен реализовать все его методы.

Внутри объявлений интерфейсов можно объявлять переменные. Они неявно считаются final и static (это означает, что они не могут быть изменены реализующим классом), а также должны быть инициализированы постоянными значениями. Все методы и переменные интерфейсов - неявно общие (public), если интерфейс сам не объявлен как public.

Пример определения интерфейса:

```
interface Callback {
   void callback(int param);
}
```

Здесь объявлен простой интерфейс, содержащий один метод с именем callback (), который имеет единственный целый параметр.

Реализация интерфейсов

Когда интерфейс определен, он может реализовываться одним или несколькими классами. Для реализации интерфейса в определение класса включают предложение с ключевым словом implements и затем создают методы, определенные в интерфейсе. Общая форма класса, который включает implements предложение, выглядит примерно так:.

```
access class classname [extends superclass]
       [implements interfase [,interfase…]]  {
    // тело-класса
}
```

Здесь access - спецификатор доступа (public или не используется). Ключевое слово `implements` означает, что новый класс предоставляет необходимые функции для реализации `interfase` (интерфейса). Если класс реализует более одного интерфейса, они разделяются запятой. Если класс реализует два интерфейса, которые объявляют один и тот же метод, то клиенты любого интерфейса будут использовать один и тот же метод. Методы, которые реализуют интерфейс, должны быть объявлены как public. Кроме того, сигнатура типа реализующего метода должна точно соответствовать сигнатуре типа, указанной в определении интерфейса.

Пример небольшого класса, который реализует интерфейс Callback, показанный ранее:

```
classClient implements Callback {
   //реализация Callback-интерфейса
   public void callback(int p) {
      System.out.println("callback вызван с аргументом" + p);
   }
}
```

Обратите внимание, что callback ( ) объявлен со спецификатором доступа public. Замечание:. при реализации метода интерфейса он должен быть объявлен как public.

Для классов, которые реализуют интерфейсы, обычно допустимо определят дополнительные собственные члены. Например, следующая версия класс Client реализует метод callback () и добавляет метод nonIFaceMeth () :

```
class Client implements Callback {
     // реализация Callback интерфейса
     public void callback(int p) {
        System.out.println("callback вызван с аргументом" + p);
     }
     void nonlfaceMeth() {
     System.out.println("классы, реализующие интерфейсы" +
                        "должны  также определять другие члены.");
     }
}
```

Реализации доступа через интерфейсные ссылки

Можно объявлять переменные как объектные ссылки, которые использую интерфейсный тип, а не тип класса. В такой переменной можно сохранят всякий экземпляр любого класса, который реализует объявленный интерфейс. Когда вы вызываете метод через ссылки такого рода, будет вызываться его правильная версия, основанная на актуальном экземпляре интерфейс

Это - одно из ключевых свойств интерфейсов. Выполняемый метод отыскивается динамически (во время выполнения), что позволяет создавать классы позже кода, который вызывает их методы. Кодом вызова можно управлять через интерфейс, ничего не зная об объекте вызова. Этот процесс подобен использованию ссылки суперкласса для доступа к объекту подкласса, как описано ранее.

Поскольку динамический поиск метода во время выполнения приводит к существенной потере производительности по сравнению с нормальным вызовом следует проявлять осторожность и беспричинно не использовать интерфесы там, где нужна повышенная эффективность.

В следующем при мере вызов метода callback () выполняется через ссылочную переменную интерфейса:

```
class TestIfase {
    public static void main(String args[]) {
        Callback c = new Client();
        c.callback(42);
   }
}
Вывод этой программы:
Callback вызван с аргументом 42
```

0братите внимание, что переменной с, объявленной с типом интерфейса Callback, был назначен экземпляр класса Client. Хотя разрешается использовать с для обрашения к методу callback(), она не может обращаться к любым другим членам класса Client. Переменная интерфейсной ссылки обладает знаниями только методов, объявленных в соответствующей интерфейсной декларации. Таким образом, с нельзя использовать для обращения к методу nonlfaceMeth(), т. к. он определен классом Client, а не интерфейсом Callback.

Хотя предыдущий пример отражает чисто технически, как переменная интерфейсной ссылки может обращаться к объекту реализации, он не демонстрирует полиморфную мощь такой ссылки. Чтобы показать это, сначала создайте другую реализацию Callback, как показано ниже:

```
// Другая реализация Callback.
class AnotherClient implements Callback {
   //  Implement Callback's interface
   public void callback(int p) {

           System.out.println("Другая версия callback");
           System.out.println("Квадрат p равен" + (p*p));

   Теперь протестируйте следующий класс:
class TestIface2 {
    public static void main(String args[]) {
      Callback c = new Client();
      AnotherClient ob = new AnotherClient();
      c.callback(42);
      c = ob;
   }
}

 Вывод этой программы:
Callback вызван с аргументом 42
Другая версия callback
Квадрат p равен 1764
```

Здесь видно, что вызываемая версия callback () определяется типом объекта, к которому обращается с во время выполнения.

Интерфейсы поддерживают динамический вызов методов во время выполнения. Обычно для вызова метода одного класса из другого нужно, чтобы оба класса присутствовали во время компиляции и компилятор Java мог проверить совместимость сигнатур методов. Само по себе это требование предъявляется. к статической и нерасширяемой среде классификации. В иерархической же многоуровневой системе, где функциональные возможности обеспечиваются длинными цепочками связанных в иерархию классов этот механизм неизбежно используется все большим и большим числом подклассов. Интерфейсы разработаны для того, чтобы обойти эту проблему. Они исключают определение метода или набора методов из иерархии наследования. Так как интерфейсы находятся вне иерархии классов, то для несвязанных в этой иерархии классов появляется иная, более эффективная возможность реализации интерфейсов. В этом и заключена действительная их польза.

Именно интерфейсы добавляют во многие приложения те функциональные возможности, которые обычно потребовали бы использования *множественного нвследоввния* языков типа C++.

Частичные реализации

класс включает интерфейс, но полностью не реализует методы, определенные этим интерфейсом, то этот класс должен быть объявлен как abstract (абстрактный). Например:

```
abstract class Incomplete implemeпts Callback {
   int a, b;
   void show() {
       System.out.priпtlп(a + " " +b);
   }
   //….
}
```

Здесь класс Incomplete не реализует callback() и должен быть объявлен как абстрактный. Любой класс, который наследует Incomplete, должен реализовать cal1back() или объявить себя как abstract.

Итак, интерфейс определяет, что должен делать класс, но не ко нкр етизир уе т, как о н это должен делать. Синтаксически интерфейсы аналогичны классам, но методы интерфейсов объявляются без тела. Класс реализует интерфейс путем создания набора методов, определенных в интерфейсе. Интерфейсы подобны множественному наследованию.

Интерфейс может содержать одну или более метод-член или объявление констант, и он до некоторой степени подобен абстрактному суперклассу. Интерфейсы отличаются от суперклассов по следующим параметрам.

- Интерфейс не может реализовать никакие методы-члены, тогда как абстрактный класс может.
- Класс может реализовывать множество интерфейсов, но имеет при этом только один суперкласс.
- Интерфейс не является частью какой-либо иерархии.

Лекция 15. Ошибки при работе программы. Исключения (Exceptions)

При выполнении программы могут возникать ошибки. В одних случаях это вызвано ошибками программиста, в других - внешними причинами. Например, может возникнуть ошибка ввода/вывода при работе с файлом или сетевым соединением. В процедурных языках программирования, например, в С требовалось проверять некое условие, которое указывало на наличие ошибки, и в зависимости от этого предпринимать те или иные действия.

Например:

. . .

```
int statusCode = someAction();
if (statusCode){
   ... обработка ошибки
} else {
   statusCode = anotherAction();
   if(statusCode) {
      ... обработка ошибки ...
   }
}
...
```
Кратко ошибка времени выполнения называется **исключением.**

Возникновение ошибки – это исключительная ситуация в терминологии Java. Для создания Интернет-приложений потребовалось ввести в язык Java многопотоковость и обработку исключительных ситуаций. Поэтому в языке Java предусмотрены механизмы обработка исключительных ситуаций. Работа с массивами и обработка исключительных ситуаций в языке Java организованы не так, как в других языках программирования. Для этого каждой исключительной ситуации поставлен в соответствие некоторый класс. Если подходящего класса не существует, то он может быть создан разработчиком. Исключения являются наследниками суперкласса **Throwable** и его подклассов **Error** и **Exception** из пакета **java.lang**. Поэтому особенностью языка Java является то, что при возникновении исключительной ситуации всегда автоматически генерируется объект соответствующего типа, который должен быть перехвачен и обработан определенным для этого блоком кода текста программы. Кроме того, при создании метода можно сделать указание программисту включить в код обработку исключений, которые может генерировать этот метод.

Причины возникновения ошибок

Существует три причины возникновения исключительных ситуаций.

- Попытка выполнить некорректное выражение. Например, деление на ноль, или обращение к объекту по ссылке, равной null, попытка использовать класс, описание которого (class-файл) отсутствует, и т.д. В таких случаях всегда можно точно указать, в каком месте произошла ошибка, - именно в некорректном выражении.
- Выполнение оператора throw. Этот оператор применяется для явного порождения ошибки. Очевидно, что и здесь можно указать место возникновения исключительной ситуации.
- Асинхронные ошибки во время исполнения программы. Причиной таких ошибок могут быть сбои внутри самой виртуальной машины (ведь она также является программой), или вызов метода stop() у потока выполнения Thread. В этом случае невозможно указать точное место программы, где происходит исключительная ситуация. Если мы попытаемся остановить поток выполнения (вызвав метод stop()), нам не удастся предсказать, при выполнении какого именно выражения этот поток остановится.

Таким образом, все ошибки в Java делятся на синхронные и асинхронные. С первыми сравнительно проще работать, так как принципиально возможно найти точное место в коде, которое является причиной возникновения исключительной ситуации. Конечно, Java является строгим языком в том смысле, что все выражения до точки сбоя обязательно будут выполнены, и в то же время ни одно последующее выражение никогда выполнено не будет. Важно помнить, что ошибки могут возникать как по причине недостаточной внимательности программиста (отсутствует нужный класс, или индекс массива вышел за допустимые границы), так и по независящим от него причинам (произошел разрыв сетевого соединения, сбой аппаратного обеспечения, например, жесткого диска и др.).

Асинхронные ошибки гораздо сложнее в обнаружении и исправлении. Обычному разработчику очень трудно выявить причины сбоев в виртуальной машине. Это могут быть ошибки создателей JVM, несовместимость с операционной системой, аппаратный сбой и многое

другое. Все же современные виртуальные машины реализованы довольно хорошо и подобные сбои происходят крайне редко (при условии использования качественных комплектующих).

Аналогичная ситуация наблюдается и в случае с принудительной остановкой *потоков исполнения* (thread).

Еще бывает *поток данных* (stream) – это просто объект, из которого или в который данные могут последовательно считываться или записываться. Для того чтобы отвлечься от особенностей конкретных устройств ввода и вывода, в Java употребляется понятие поток данных. Считается, что в программу идет входной поток символов Unicode или просто байтов, воспринимаемый в программе методами read(). Из программы методами write(), print() выводится выходной поток символов или байтов. При этом не важно, куда направлен поток: на консоль, на принтер, в файл или в сеть или откуда поступает в программу. В Java ввод и вывод осуществляются при помощи классов и интерфейсов потоков пакета java.io. Поскольку такие действия выполняются операционной системой, никогда нельзя предсказать, в каком именно месте остановится поток. Это означает, что программа может многократно отработать корректно, а потом неожиданно дать сбой просто из-за того, что поток остановился в каком-то другом месте. По этой причине принудительная остановка не рекомендуется.

При возникновении исключительной ситуации управление передается от кода, вызвавшего исключительную ситуацию, на ближайший блок `catch` (или вверх по стеку) и создается объект, унаследованный от класса `Throwable`, или его потомков (см. диаграмму иерархии классов-исключений), который содержит информацию об исключительной ситуации и используется при ее обработке. Собственно, в блоке `catch` указывается именно класс обрабатываемой ситуации. Подробно обработка ошибок рассматривается ниже.

Иерархия, по которой передается информация об исключительной ситуации, зависит от того, где эта исключительная ситуация возникла. Если это

- метод, то управление будет передаваться в то место, где данный метод был вызван;
- конструктор, то управление будет передаваться туда, где попытались создать объект (как правило, применяя оператор new);
- статический инициализатор, то управление будет передано туда, где произошло первое обращение к классу, потребовавшее его инициализации.

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

Обработка исключительных ситуаций. Конструкция try-catch

В общем случае конструкция выглядит так:

```
try {
   ...
} catch(SomeExceptionClass e) {
   ...
} catch(AnotherExceptionClass e) {
   ...
}
```

Работает она следующим образом. Сначала выполняется код, заключенный в фигурные скобки оператора `try`. Если во время его выполнения не происходит никаких нештатных ситуаций, то далее управление передается за закрывающую фигурную скобку последнего оператора `catch`, ассоциированного с данным оператором `try`.

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

Возникла исключительная ситуация, класс которой указан в качестве параметра одного из блоков `catch`. В этом случае производится выполнение блока кода, ассоциированного с

данным `catch` (заключенного в фигурные скобки). Далее, если код в этом блоке завершается нормально, то и весь оператор `try` завершается нормально и управление передается на оператор (выражение), следующий за закрывающей фигурной скобкой последнего `catch`. Если код в `catch` завершается не штатно, то и весь `try` завершается нештатно по той же причине.

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

Конструкция try-catch-finally

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

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

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

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

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

Рассмотрим пример применения конструкции `try-catch-finally`.

```
try {
   byte [] buffer = new byte[128];
   FileInputStream fis =
      new FileInputStream("file.txt");
   while(fis.read(buffer) > 0) {
      ... обработка данных ...
   }
} catch(IOException es) {
      ... обработка исключения ...
} finally {
   fis.flush();
   fis.close();
}
```

Если в данном примере поместить операторы очистки буфера и закрытия файла сразу после окончания обработки данных, то при возникновении ошибки ввода/вывода корректного закрытия файла не произойдет. Еще раз отметим, что блок `finally` будет выполнен в любом случае, вне зависимости от того, произошла обработка исключения или нет, возникло это исключение или нет.

В конструкции `try-catch-finally` обязательным является использование одной из частей оператора `catch` или `finally`. То есть конструкция

```
try {
   ...
} finally {
   ...
}
```

является вполне допустимой. В этом случае блок `finally` при возникновении исключительной ситуации должен быть выполнен, хотя сама исключительная ситуация обработана не будет и будет передана для обработки на более высокий уровень иерархии.

Если обработка исключительной ситуации в коде не предусмотрена, то при ее возникновении выполнение метода будет прекращено и исключительная ситуация будет передана для обработки коду более высокого уровня. Таким образом, если исключительная ситуация произойдет в вызываемом методе, то управление будет передано вызывающему методу и обработку исключительной ситуации должен произвести он. Если исключительная ситуация возникла в коде самого высокого уровня (например, методе `main()`), то управление будет передано исполняющей системе Java и выполнение программы будет прекращено (более точно - будет остановлен поток исполнения, в котором произошла такая ошибка) и выдано на консоль сообщение о произошедшем исключении.

Использование оператора throw

Помимо того, что предопределенная исключительная ситуация может быть возбуждена исполняющей системой Java, программист сам может явно породить ошибку. Делается это с помощью оператора `throw`.

Например:

```
...
public int  calculate(int theValue) {
   if( theValue < 0) {
   throw new Exception(
     "Параметр для вычисления не должен быть отрицательным");
   }
}
...
```

В данном случае предполагается, что в качестве параметра методу может быть передано только положительное значение; если это условие не выполнено, то с помощью оператора `throw` порождается исключительная ситуация. (Для успешной компиляции также требуется в заголовке метода указать `throws Exception` - это выражение рассматривается ниже.)

Метод должен делегировать обработку исключительной ситуации вызвавшему его коду. Для этого в сигнатуре метода применяется ключевое слово `throws`, после которого должны быть перечислены через запятую все исключительные ситуации, которые может вызывать данный метод. То есть приведенный выше пример должен быть приведен к следующему виду:

```
...
public int  calculate(int theValue)
  throws Exception {
    if( theValue < 0) {
      throw new Exception(
       "Some descriptive info");
   }
}
...
```

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

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

В этом случае ошибка появляется вторично.

Например:

```
...
try {
   ...
} catch(IOException ex) {
   ...
   // Обработка исключительной ситуации
```

```
    ...
    // Повторное возбуждение исключительной
    // ситуации
    throw ex;
}
```

Рассмотрим еще один случай.

Предположим, что оператор `throw` применяется внутри конструкции `try-catch`.

```
try {
    ...
    throw new IOException();
    ...
} catch(Exception e) {
    ...
}
```

В этом случае исключение, возбужденное в блоке `try`, не будет передано для обработки на более высокий уровень иерархии, а обработается в пределах блока `try-catch`, так как здесь содержится оператор, который может это исключение перехватить. То есть произойдет неявная передача управления на соответствующий блок `catch`.

Проверяемые и непроверяемые исключения

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

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

Ошибки, порожденные от `Exception` (и не являющиеся наследниками `RuntimeException`), являются *проверяемыми*. Т.е. во время компиляции проверяется, предусмотрена ли обработка возможных исключительных ситуаций. Как правило, это ошибки, связанные с окружением программы (сетевым, файловым вводом-выводом и др.), которые могут возникнуть вне зависимости от того, корректно написан код или нет. Например, открытие сетевого соединения или файла может привести к возникновению ошибки и компилятор требует от программиста предусмотреть некие действия для обработки возможных проблем. Таким образом повышается надежность программы, ее устойчивость при возможных сбоях.

Исключения, порожденные от `RuntimeException`, являются *непроверяемыми* и компилятор не требует обязательной их обработки. Как правило, это ошибки программы, которые при правильном кодировании возникать не должны (например, `IndexOutOfBoundsException` - выход за границы массива, `java.lang.ArithmeticException` - деление на ноль). Поэтому, чтобы не загромождать программу, компилятор оставляет на усмотрение программиста обработку таких исключений с помощью блоков `try-catch`.

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

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

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

```
catch(UserException ue) {
   ...
}
```

В данном примере при возникновении исключительной ситуации (класс, порожденный от `Exception`) будет выполняться всегда только первый блок `catch`. Остальные не будут выполнены ни при каких условиях. Эта ситуация отслеживается компилятором, который сообщает об `UnreachableCodeException` (ошибка - недостижимый код). Правильно данная конструкция будет выглядеть так:

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

В этом случае будет выполняться последовательная обработка исключений. И в случае, если не предусмотрена обработка того типа исключения, которое возникло (например, `AnotherUserException`), будет выполнен блок `catch(Exception e){:}`

Если срабатывает один из блоков `catch`, то остальные блоки в данной конструкции `try-catch` выполняться не будут.

Создание пользовательских классов исключений

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

Пример:

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

Соответственно, данное исключение будет создаваться следующим образом:

```
 throw new UserException(
   "Дополнительное описание");
```

Обработка исключений чрезвычайно важна для написания надежных программ.

Генерации исключений на примере вычисления факториалов.

/* Этот клас демонстрирует рекурсивный способ вычисления факториалов. Этот метод многократно вызывает сам себя, основываясь на формуле : n! = n * (n-1)! */

```
public class Factorial2 {
 public static long factorial(long x)  {
   if (x < 0) throw new IllegalArgumentException("x должен быть >= 0");
   if (x <= 1) return 1;       // Здесь рекурсия пекращается
   else return x * factorial(x-1);   // Шаг рекурсии - вызов самого себя
 }
}
```

В этом примере показано, как генерируются исключения

```
    throw new IllegalArgumentException("x должен быть >= 0");
```

Исключения – это вид объектов Java. Они, как и массивы, создаются при помощи ключевого слова new. Если программа генерирует объект-исключение при помощи оператора throw, это означает, что возникло неожиданное обстоятельство или ошибка. Когда возникает исключение, управление в программе

передается ближайшей секции catch оператора try/catch. Эта секция должна содержать код для обработки исключений. Если исключение не будет перехвачено, исполнение программы прекращается с сообщением об ошибке. В примере возникает исключение, уведомляющее вызывающую процедуру о том, что переданный ею аргумент должен быть больше равно ноль.

Раз уж факториал вычислен его значение можно сохранить для будущего употребления, т.е. можно делать *кэширование*. Напишем вариант вычисления факториала с кэшированием результатов.

/* Этот класс вычисляет факториалы и кэширует результаты в таблице для дальнейшего употребления. 20! – самый большой факториал, который мы можем вычислить с применением типа данных long, поэтому проверим рагумент и выдадим исключение, если аргумент окажется слишком большим или слишком маленьким */

```
public class Factorial3 {
   //Создаем массив для хранения факториалов от 0! до 20!
   static long[ ]  table = new long[21];
   “Статический инициализатор” : инициализируем первый элемент массива
   static { table[0] = 1;}  // Факториал 0 равен 1
   // Запоминаем номер последнего вычисленного факториала
   static int last = 0;
   public static long factorial(int x)  throws IllegallArgumentException {
     // Проверяем, не слишком ли мал или велик x.
     // Выдаем исключение, если это оказывается так
     if (x >= table.length) // «.length» возвращает длину любого массива
     throw new IllegalArgumentException(“Переполнение : x слишком велик.”);
     if (x < 0) throw new IllegalArgumentException(“x должен быть неотрицательным.”);
     // Вычисляем и кэшируем все пока ещее несохраненные значения
     while (last < x) {
       table[last + 1] = table[last] * (last + 1);
       last++;
     }
     // Возвращаем кэшированный факториал x
     return table[x];
   }
}
```

ЛЕКЦИЯ 16. КОМПОНЕНТЫ JAVA

Программный интерфейс JavaBeans предоставляет среду для разработки многократно используемых, встраиваемых, модульных программных компонентов. Спецификация JavaBeans дает следующее определение компонента (bean): «многократно используемый программный компонент, которым можно манипулировать в визуальных средах разработки». На самом простом уровне компонентами JavaBeans являются все отдельные графические компоненты, тогда как на гораздо более высоком уровне в качестве компонента может также функционировать встраиваемое приложение с электронными таблицами. Однако большинство компонентов находятся где-то между этими двумя крайностями.

Графический интерфейс пользователя или GUI представляет собой превосходный пример модульности и многократного использования программного обеспечения (компонентов). GUI почти всегда собираются из готовых строительных блоков (компонентов), хранящихся в библиотеках. Стандартной библиотекой простых компонентов GUI в Java является пакет java.awt, java.swing. Апплет тоже является оконным приложением с графическим интерфейсом пользователя (см. лекцию 4).

Одной из целей модели JavaBeans является обеспечение взаимодействия с аналогичными компонентными средами. Так, например, обычная Windows-программа может с помощью соответствующего моста или компонента-обертки пользоваться компонентом Java так, как будто бы она является компонентом COM или ActiveX.

Если вы пишете приложения, в которых используются компоненты, разработанные другими программистами, или используете контейнер компонентов для сборки этих компонентов в одно приложение, вам в действительности не понадобится знакомство с интерфейсом JavaBeans. Вам потребуется только знание документации по отдельным используемым вами компонентам.

Основы компонентов

Любой объект, удовлетворяющий определенным базовым правилам и соглашениям об именах, может считаться компонентом JavaBeans. Не существует никакого класса Bean, который все компоненты должны были бы иметь в качестве базового.

Любой компонент экспортирует свойства, события и методы. *Свойство* (property) – это часть внутреннего состояния компонента, которую можно программно устанавливать и опрашивать, обычно через стандартную пару методов доступа, начинающихся с слов «set» и «get». Компоненты можно *конфигурировать т.е. задавать значения свойствам,* вызывая различные методы, имена которых начинаются с «set». Компонент может генерировать *события* (events). Компонент определяет событие путем предоставления методов для добавления и удаления *объектов-слушателей событий* из списка слушателей, заинтересованных в данном событии. *Методы,* экспортируемые компонентом, являются простыми открытыми (public) методами.

Кроме этих обычных типов свойств интерфейс JavaBeans обеспечивает поддержку индексных (indexed), связанных (bound) и ограниченных (constrained) свойств. *Индексное свойство* – это любое свойство, имеющее значение типа «массив» и для которого компонент предоставляет методы для извлечения и установки отдельных компонентов этого массива, а также методы для считывания и записи массива целиком. *Связанное свойство* – это свойство, которое рассылает уведомление при изменении своего значения, тогда как *ограниченное свойство* – это то, которое рассылает уведомление при изменении своего значения и позволяет слушателям запретить это изменение.

Компонент, снабженный этими и другими необходимыми качествами, в технологии Java называется компонентом JavaBaens. В него может входить один или несколько классов. Как правило, файлы этих классов упаковываются в jar-архив и отмечаются в файле MANIFEST.MF как Java-Baen: True. Jar-архивы создаются с помощью классов пакета java.util.jar или посредством утилиты командной строки jar. Архивные файлы очень удобно использовать в апплетах и в приложениях, поскольку весь архив загружается по сети сразу же, одним запросом.

Создание GUI программно на основе Java происходит в четыре основных этапа:

1. *Создание и конфигурирование компонентов*

   Компонент GUI дается, как и любой другой объект в Java, путем вызова его конструктора. Необходимо изучить документацию на индивидуальный компонент, чтобы узнать, каких аргументов ожидает конструктор. Например. Для создания Swing-компонент JButton с надписью «Quit», следует просто написать:

   ```
   JButton quit = new JButton(“Quit”);
   ```

   Создав компонент, можно сконфигурировать его, задав одно или несколько его свойств. Например, чтобы задать шрифт, который должен использовать компонент `JButton`, можно написать:

   ```
   quit.setFont(new Font(“sansserif”, Font.BOLD, 18);
   ```

   И опять, следует познакомиться с документацией на используемый компонент, чтобы узнать, какие методы можно применять для его конфигурирования.

2. *Помещение компонента в контейнер*

   Все компоненты должны размещаться в *контейнере*. Все контейнеры в Java являются подклассами java.awt.Container. К обычно используемым контейнерам принадлежиат классы JFrame и JDialog, представляющие окна верхнего уровня и диалоговые окна соответственно. Класс java.applet.Applet представляющий собой базовый класс для апплетов, также является контейнером и, следовательно. Может содержать и отображать компоненты GUI. Контейнер – это вид компонента, поэтому контейнеры могут быть помещены и часто помещаются в другие контейнеры. Контейнер JPanel часто используется следующим образом. При разработке GUI фактически создается иерархия контейнеров: окно верхнего уровня или апплет содержат контейнеры, которые могут содержать другие контейнеры, которые в свою очередь содержат компоненты. Чтобы поместить компонент в контейнер, его просто передают методу add() контейнера. Например, кнопку quit можно поместить в контейнер “buttonbox” посредством кода подобного следующему:

```
buttonbox.add(quit);
```

3. *Размещение или компоновка компонентов*
   Помимо определения того, какой компонент внутри какого контейнера находится, необходимо также задать расположение и размер каждого компонента в его контейнере так, чтобы GUI хорошо смотрелся. Хотя можно включить положение и размер каждого компонента в код программы, гораздо чаще применяют объект LayoutManager для автоматического размещения компонентов в контейнере в соответствии с определенными правилами компоновки, задаваемыми конкретным, выбранным программистом объектом LayoutManager.
4. *Обработка событий, генерируемых компонентами.*
   Описанные выше шагов достаточно для создания GUI, который хорошо смотрится на экране, но наш графический «интерфейс пользователя» не закончен, поскольку он еще не реагирует на действия пользователя. При взаимодействии пользователя с компонентами GUI при помощи клавиатуры или мыши эти компоненты генерируют, или активизируют, события. Поэтому обработка любого действия пользователя реализуется как обработка событий. Обработка событий основывается на *событиях* (events) и *слушателях событий* (event listeners). Событие – это просто объект, содержащий информацию о действиях пользователя. Событие нельзя обработать произвольно написанным методом. У каждого события есть свои методы, к которым обращается испольняющая система Java при его возникновении. Они описаны в *интерфейсах – слушателях*. Для каждого типа событий есть свой интерфейс. Чтобы задать обработку определенного типа, надо в тексте программы реализовать соответствующий интерфейс. Класс, реализующий такой интерфейс – класс-обработчик события. Методы объекта такого класса «слушают», что происходит в потенциальном источнике события, т.е. в компоненте грфического интерфейса.

   При возникновении события исполняющая система Java автоматически создает объект соответствующего событию класса и методы слушателя автоматически выполняются, получая в качестве аргумента объект-событие и используя при обработке сведения о событии, содержащиеся в этом объекте. Таким образом, компонент-источник, в котором произошло событие, не занимается его обработкой. Он обращается к экземпляру класса-слушателя, умеющего обрабатывать события, делегирует ему полномочия по обработке.

   Последним шагом при создании GUI является добавление *слушателей событий* (event listeners) – объектов, которым посылаются уведомления о том, что произошло событие, и которые адекватно отвечают на это событие. Например, кнопке quit нужен слушатель, который вызовет выход из приложения.

Многие программисты предпочитают разрабатывать приложения с графическим интерфейсом пользователя с помощью визуальных средств разработки: IntelIiJ IDEA, Eclipse, JBuilder и др. Эти средства позволяют помещать компоненты в контейнер (окно-форма) визуально графически, с помощью мыши. Чтобы поместить компонент в форму, надо щелкнуть кнопкой мыши на ярлыке компонента, перенести курсор мыши в нужное место формы и щелкнуть кнопкой мыши еще раз. Далее следует определить свойства компонента, затем задают обработку событий.

Главное достоинство компонентов, оформленных как JavaBaens, в том, что они без труда встраиваются в любое приложение. Более того, приложение можно собрать из готовых JavaBaens как из строительных блоков, остается только настроить их свойства. Специалисты пророчат большое будущее компонентному программированию. Они считают, что скоро будут созданы тысячи компонентов JavaBaens на все случаи жизни и программирование сведется к поиску в Интернете нужных компонентов и сборке из них приложения. Из-за того что Java разрешает загружать классы динамически, контейнерные программы могут загружать неизвестные им классы.

Построение оконного приложения с графическим интерфейсом пользователя

Мы используем класс JFrame из пакета javax.swing для создания собственного окна, в которое поместим программно компоненты графического пользовательского интерфейса, с которыми пользователь может взаимоодействовать для управления программой. Класс JFrame расширяет класс Frame графической библиотеки AWT. Это полноценное самостоятельное окно верхнего уровня, снабженное рамкой и строкой заголовка с кнопками системного меню Свернуть,

Развернуть и Закрыть, как принято в графической оболочке операционной системы. Класс JFrame будет играть роль контейнера для компонентов. Шаблон для оконного приложения с графическим пользовательским интерфейсом, использующего библиотеку Swing, будет следующим:

```
import java.awt.*;   //Базовые классы
import javax.swing.*; // Основные классы
public class имяПриложения  extends JFrame {
 public  имяПриложения(String title) {
  super(title);  // 1. Создаем основное окно
  Container c = getContentPane();  // 2. Получаем контейнер верхнего уровня
  c.add(xxxx);  // 3. Помещаем компонент в контейнер
  // 4. Конфигурирование компонента, задав его свойств. Присоединение к компоненте слушателя событий
  setSize(width, height); // 5. Задание начальной ширины и высоты окна
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);//6.Завершаем работу приложения при закрытии
  setVisible(true); // 7. Выводим окно на экран
 }
 // 8. Задание метода обработки  события, возникшего в компоненте-источнике.
 public static void main(String[] args) {
    new  имяПриложения("Заголовок основного окна");
 }
}
```

Мы познакомимся с оконным приложением TokenTest.java (Рис. 16.1). Эта программа также познакомит вас с классом StringTokenizer (пакет javax.util). Когда вы читаете предложение, вы разбиваете его на отдельные слова, которые называются лексемами, каждое из которых имеет определенное значение. Компиляторы также осущесествляют разбивку текста программы на лексемы. Они разбивают текст программы на отдельные лексемы, такие как идентификаторы, ключевые слова и другие элементы языка программирования. Класс StringTokenizer разбивает строку на составляющие лексемы. Лексемы отделяются другой с помощью разделителей, в качестве которых обычно выступают пробельные символы: пробел, табуляция, перевод строки и возврат каретки. В качестве разделителей лексем могут также применяться и другие символы.

```
1  // Рис. 16.1. TokenTest.java
2  // Тестирование класса StringTokenizer из пакета java.util
3  import javax.swing.*;
4  import java.util.*;
5  import java.awt.*;
6  import java.awt.event.*;
8 public class TokenTest extends JFrame
9                          implements AetionListener {
10    private Jlabel prompt;
11    private JTextField input;
12    private JtextArea output;
13
14    public TokenTest()
15    {
16       super( "Testing Class StringTokenizer");
17
18       Container c = getContentPane();  // Получение контейнера с класса
19       c.setLayout( new FlowLayout() );// В каждом контейнере есть свой менеджер размещения компонентов
20      // Установка другого (FlowLayout) менеджера производится методом setLayout.
21       prompt = new Jlabel ("Enter a sentence and press Enter"); // Создание компонента Jlabel
22       c.add( prompt);    // Помещение компонента Jlabel в контейнер
23
24       input = new JTextField( 25 ); // Создание компонента JTextField
25       c.add( input );  // Помещение компонента JTextField в контейнер
26      input.addActionListener(this); // Присоединение к компоненте JTextField класса-слушателя TokenTest
27                          // Ибо в методе указан аргумент this, значит класс сам слушает свой компонент
28       output = new JtextArea( 10, 25 ); // Создание компонента JtextArea
29       output.setEditable( false ); // Конфигурирование компонента JtextArea
```

```
30      c.add( new JscrollPane( output ) ); // Помещение компонента JtextArea в контейнер
30
31       setSize( 300, 260 );   // задание размеров окна-наследника
33      show();    // отображение окна и начало взаимодействия пользователя с графическим интерфейсом
34   }             // Такое взаимодействие продолжается до тех пор пока пользователь не закроет окно
35                 // графического пользовательского интерфейса
36   public void actionPerformed( ActionEvent e) // Метод интерфейса-слушателя - метод обработка события
37   {    // реализован в этом же классе, поэтому этот класс сам слушает свой компонент
38     String stringToTokenize = e.getActionCommand();
39     StringTokenizer tokens =
40          new StringTokenizer( stringToTokenize );
41
42     output.setText( "Number of elements:," +
43     tokens.countTokens() + "\nThe tokens are:\n" );
44
45     while ( tokens.hasMoretokens() )
46          output.append( tokens.nextToken() + "\n" );
47   }
48
49   // с метода main начинается выполнения программы,
50   // в нем создается объект класса TokenTest.
51   public static void main< String args[] )
52   {
53   TokenTest app = new TokenTest(); // TokenTest строит объект-окно. Помещается на окно компоненты.
54  // Делается конфигуризация, соединение со слушателем, отображние окна, ожидание действия пользователя
55   app.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ); // Сработает, когда пользователь закроет окно
56   }
57 }
```

Строки с 3 по 6

```
import  javax.swing.*;
import  java, util.*;
import java.awt.*;
import java.awt.event.*;
```

указывают компилятору, какие библиотечные классы используются в этсм приложении. Первый оператор import указывает, что программа использует классы из пакета **javax.swing** (в частности, классы JFrame, JLabel, JScrollPane, JTextField, JTextArea). Второй оператор import указывает, что программа использует из пакета **java.util** (конкретно, класс StringTokenizer). Третий оператор указывает, что программа использует классы из пакета java.awt (конкретно классы Container и FlowLayout). Последний оператор import указывает, что программа использует классы иа пакета java.awt.event. Этот пакет содержит множество типов данных, которые дают возможность программе обрабатывать взаимодействие пользователя с графическим пользовательским интерфейсом программы. В этой программе мы используем типы данных ActionListener и ActionEyent из пакета import java.awt.event

Каждая Java-программа основана по крайней мере на одном определении класса, который расширяет и дополняет существующее определение класса посредством наследования.

В строках 8 и 9

```
public class TokenTest extends JFrame
                    implements ActionListener   {
```

указывается, что класс TokenTest наследует *(extends)* классу JFrame и реализует (*implements)* интерфейс *ActionListener.* Класс может наследовать атрибуты и поведение (методы) другого класса, указываемого справа от ключевого слова extends в определении класса. Путем расширения класса JFrame мы создаем класс TokenTest как новый тип окна. В отношении расширения JFrame считается *суперклассом,* а TokenTest — *подклассом* (или *производным классом)* JFrame. Применение наследования здесь приводит к новому определению класса, который имеет *атрибуты* (данные) и *поведение* (методы) как класса JFrame, так и сво и

собственные, которые мы добавляем в наше определение класса TokenTest (в частности, способность разбивать предложение, введенное пользователем, на лексемы).

Основное преимущество расширения класса JFrame состоит в том, что кто-то ранее уже определил «что означает окно». За счет простого использования оператора **extends** для наследования класса JFrame все методы JFrame теперь становятся частью класса TokenTest.

Мы создаем объект класса TokenTest для управления этой программой из метода main, который будет рассмотрен позднее.

В дополнение к расширению одного класса, класс может реализовать один или несколько *интерфейсов.* Интерфейс задает одно или несколько поведений (т.е. методов), *которые вы должны определить* в определении класса. Интерфейс ActionListener указывает, что этот класс *должен определять метод*

```
public void actionPerformed( ActionEvent e )
```

Задача этого метода — обслуживать взаимодействие пользователя с классом JTextField. Когда пользователь вводит предложение и нажимает клавишу *Enter*, этот метод будет автоматически вызываться в ответ на действие пользователя. Этот процесс называется *обработкой события. Событие* — это действие пользователя (нажатие клавиши *Enter). Обработчик события* — это метод actionPerformed, который автоматически вызывается в ответ на событие. Мы вкратце обсудим детали этого взаимодействия и метод actionPerformed.

В строках 10-12

```
private  JLabel prompt;
private JTextField input;
private JTextArea output;
```

объявляются ссылки на .компоненты графического интерфейса пользователя, используемые в этом приложении. Ссылка prompt относится к объекту JLabel. Объект JLabel содержит строку символов, подлежащих отображению на экране. Обычно JLabel используется для указания назначения другого элемента графического пользовательского интерфейса. Ссылка input относится к объекту JTextField, в который пользователь будет вводить предложения. Объекты типа JTextField используются для получения от пользователя строки данных, введенной с клавиатуры, либо для отображения информации на экране. Ссылка output относится к объекту JtextArea, в котором будут отображаться результаты.

Заметим, что три ссылки, объявленные в строках 10-12, являются также *объявлениями переменных экземпляров* - каждый экземпляр (объект) класса TokenTest содержит свою собственную копию объекта. Важное преимущество переменных экземпляра состоит в том, что их идентификаторы могут быть использованы на протяжении всего объявления класса (т.е. во всех методах класса). До сих пор мы объявляли все переменные в методе **main** приложения. Переменные, определенные в теле метода, называются *локальными переменными* и могут быть использоваться только в теле метода, в котором они определены. Другое отличие между переменными экземпляра и локальными переменными состоит в том, что переменным экземпляра компилятором всегда присваивается значение по умолчанию, а локальным переменным — нет. Значением по умолчанию для этих ссылок является null (т.е. компоненты графического пользовательского интерфейса еще не существуют).

Строки 14-34 определяют *конструктор* класса TokenTest. Этот метод автоматически вызывается, когда объект (экземпляр) класса TokenTest создается с помощью оператора new. В строке 16

```
super(   "Testing Class StringTokenizer"   );
```

вызывается конструктор класса JFrame, ему передается строка. Эта строка отображается в строке заголовка окна, когда окно TokenTest отображается на экране. Конструктор TokenTest также создает объекты компонентов графического пользовательского интерфейса (см. лекцию 4).

Чтобы компонент отобразился на экране, его надо поместить в объект-контейнер класса Container. Поэтому в составе графического пользовательского интерфейса должен присутствовать хотя бы один контейнер. В строке 18

```
Container  c = getContentPane();
```

объявляется ссылка с типа Container, ей присваивается результат обращения к методу getContentPane. Метод getContentPane возвращает ссылку на панель содер*жимого* окна - объект,

в который мы должны поместить компоненты графического пользовательского интерфейса, чтобы они отображались должным образом.

В строке 19

```
c.setLayout( new FlowLayout() );
```

используется метод *setLayout* объекта типа Container для определения менеджера *компоновки* для организации графического интерфейса пользователя в окне TokenTest. Менеджеры; компоновки предусмотрены для организации компонентов пользовательского интерфейса в объекте типа Container (пакет java.awt)

В строке 21

```
prompt = new JLabel( "Enter a sentence and press Enter" );
```

создается новый объект JLabel (т.е. вызывается его конструктор), он инициализируется и ссылка на него присваивается к prompt. Это создает надпись для объекта. JTextFieldinput (поле ввода) интерфейса пользователя, чтобы пользователь мог понять назначение текстового поля. В строке 22 объект типа JLabel, на который ссылается prompt, добавляется к содержимому панели окна с помощью метода add. В строке 24

```
input = new JTextField( 25 );
```

создается новый объект JTextField (текстовое поле), для него устанавливается ширина в 25 символов, он присваивается переменной input. Этот объект получает ввод от пользователя программы. В строке 25 объект типа JTextField, на который ссылается input, добавляется в панель содержимого окна. К строке 26 мы вернемся чуть позже.

В строках с 28 по 30 создается объект JTextArea (текстовая область) и добавляется в панель содержимого. В данном случае область JTextArea имеет 10 строк высоту и 25 символов в ширину. В строке 29 указано, что объект JTextArea должен быть нередактируемым (т.е. пользователь не сможет вводить текст в эту область). В строке 30

```
c.add( new JScrollPane( output ) );
```

текстовая область помещается в объект *JScrollPane,* который добавляется к интерфейсу пользователя. Класс JScrollPane предоставляет полосы прокрутки, которые могут использоваться для прокрутки содержимого текстовой области. Полосы прокрутки не отображаются до тех пор, пока содержимое не окажется слишком, большим по ширине или по высоте, чтобы целиком поместиться в текстовой области. С этого момента полосы прокрутки отображаются автоматически.

В строках 32 и 33 задается размер окна, а также осуществляется его отображение. Если вы не вызовите методы setSize и show, окно не будет отображено на экране.

В строке 26

```
input.addActionListener( this );
```

указано, что *данное (this)* приложение должно *прослушивать* события объекта типа JTextField с именем input. Ключевое слово this дает возможность объекту класса TokenTest ссылаться на самого себя. Когда пользователь взаимодействует с текстовым полем ввода input, приложению посылается *событие.* Событие является сигналом, указывающим на то, что пользователь программы нажал клавишу *Enter,* находясь в текстовом поле. Это указывает приложению, что *действие было выполнено* пользователем для объекта типа JTextField, после чего автоматически вызывается метод actionPerformed для обработки действия пользователя.

Подобный стиль программирования называется *программированием с управлением по событиям.* Пользователь взаимодействует с компонентом пользовательского интерфейса, программа уведомляется о событии и обрабатывает событие. Взаимодействие пользователя с графическим интерфейсом управляет ходом выполнения программы. Методы, вызываемые при наступлении события, называют *методами обработки событий.* При наступлении в программе события пользовательского интерфейса среда выполнения Java создает объект, содержащий информацию о событии, которое произошло, и *автоматически вызывает* соответствующий метод обработки события. До того как любое событие будет обработано, каждый компонент пользовательского интерфейса должен знать, какой объект в программе определяет метод обработки события, который будет вызван при наступлении события. В строке 26 используется метод *addActionListener* класса JTextField, чтобы сообщить компоненту input (поле ввода), что приложение TokenTest (this) может *прослушивать событие,* для чего определен метод

actionPerformed. Это называется *регистрацией обработчика события* для компонента пользовательского интерфейса. Чтобы среагировать на события, мы должны определить класс, который реализует (implements) обработчик ActionListener (при этом необходимо, чтобы класс определял метод actionPerformed), а также должны зарегистрировать обработчик события для компонента пользовательского интерфейса .

Метод actionPerformed (строки 36-47) представляет собой один из нескольких методов, которые обрабатывают взаимодействия между пользователем и компонентами пользовательского интерфейса. Первая строка метода

```
public void actionPerformed( ActionEvent e );
```

указывает, что actionPerformed — это public метод, который ничего не возвращает (void) по завершении выполнения своей задачи. Метод actionPerformed принимает один параметр - ActionEvent — при автоматическом вызове в ответ, на действие пользователя. Параметр ActionEvent содержит информацию о произошедшем действии.

Когда пользователь вводит предложение в текстовое поле JTextField и нажимает клавишу *Enter,* активизируется метод actionPerformed (строка 36). В строке 38 используется параметр, переданный методу actionPerformed, чтобы получить строку, введенную пользователем в текстовое поле с помощью метода getActionCommand. В строках 39 и 40 создается объект типа StringTokenizer с именем tokens, конструктору объекта передается текстовая строка, полученная в строке 38 листинга. В строках 42 и 43

```
output.setText( “number of elements: “ +
                    tokens.countTokens()  +  "\nThe tokens are:\n" );
```

используется метод setText объекта JTextArea для отображения строки, переданной в качестве параметра JTextArea. В предыдущих строках листинга

```
tokens.countTokens ()
```

используется метод countTokens объекта StringTokenizer для определения числа лексем в строке, подлежащей лексическому разбору.

В цикле while строках 45 и 46

```
While( tokens.hasMoreTokens()   )
     output.append( tokens.nextToken()   + "\n"  );
```

используется условие tokens.hasMoreTokens() для определения, имеются ли еще лексемы в строке, подлежащей лексическому разбору. Если да, вызывается метод append для объекта типа JTextArea с именем output для добавления следующей лексемы в строку, которая отображается в объекте JTextArea. Следующая лексема получается путей вызова tokens.nextToken(), который возвращает объект типа String. Лексема выводится с завершающим символом перевода строки, чтобы последовательности лексем отображались в отдельных строках.

Если нужно сменись разделитель, используемый при лексическом:разборе строки, это можно сделать, указав новую строку для разделителя при вызове nextToken следующим образом:

```
Tokens.nextToken (  newDelimeterString ) ;
```

Этот прием в программе не используется.

Метод" main, (строки 51-55) запускает программу на выполнение путем создания экземпляра класса TokenTest. Этот объект открывает основное окно приложения. Когда строка 53 исполняется, осуществляется вызов конструктора, определенного в строке 14, для создания окна и его отображения. Далее ведется работа пользователя с программой через это окно. В строке 55 метод

```
setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
```

закрытии окна следует выйти из программы (т.е. программа должна быть завершена). Вы можете закрыть окно, щёлкнув на *кнопке закрытия в* правом верхнем углу окна.

2. Лабораторные занятия

Lab1. Знакомство с Java программой.

1. Программы Java состоят из частей, называемых *классами*. В свою очередь классы состоят из частей, называемых *методами* , которые выполняют определенные задачи (сложные действия) и возвращают данные-результаты по завершении своей работы. Так что, методы – это реализация подпрограммы-функции в Java. В свою очередь классы группируются в части, называемые *пакеты*, а различные пакеты в совокупности образуют, так называемую *библиотеку классов* Java. Писать программу на Java не возможно без использования библиотеки классов Java. Библиотеки классов содержат готовые программы для многократного использования.

```
/* Это простая Java-программа. Она состоит только из одного класса. При сохранении
назовите этот файл "Example.java" */

class Example {
  // Программа начинает выполняться с вызова метода main().
  public static void main (String args [])  {
   System.out.println{"Это простая Java-программа.");
  }
}
```

Ввод программы

Для большинства компьютерных языков, имя файла, который содержит исходный код программы, является произвольным. Для языка Java это не так. Первое, что вы должны изучить в языке Java, это то, что имя, которое вы присваиваете исходному файлу, очень важно. Например, имя исходного файла должно быть Example java. Посмотрим почему.

В Java исходный файл официально называют *единицей компиляции*. Это текстовый файл, который содержит одно или несколько определений классов. Компилятор Java требует, чтобы исходный файл использовал расширение java. Заметим, что расширение имени файла имеет длину в четыре символа. Как вы можете заметить, ваша операционная система должна поддерживать длинные имена файлов.

Глядя на программу можно заметить, что имя класса, определенное в программе, тоже Example. Это не совпадение. В Java весь код должен находиться внутри класса. По соглашению имя этого класса должно быть согласовано с именем файла, который содержит программу. Вы должны также убедиться в согласовании прописных букв в именах файла и класса. Причина в том, что язык Java чувствителен к регистру клавиатуры. Здесь соглашение о том, чтобы, имена файлов соответствовали именам классов, может показаться произволом. Однако это соглашение делает более простым поддержку и организацию, ваших, программ.

Компиляция программы

Чтобы откомпилировать программу Example, нужно запустить компилятор javac, указав в параметре командной строки имя исходного файла:

C:\>javac Example.java

Чтобы на экране появилась командная строка надо запустить, например, NortonCommander или Far-manager, и набрать в командной строке >cmd-Enter. Эту команду также можно выполнить так: Пуск-Выполнить-cmd-OK. Программы, запускающиеся с командной строки называются консольными приложениями. Они разрабатываются для выполнения на серверах, там где не требуется интерективная связь с пользователем.

Компилятор javac создает файл с именем Example.class, который содержит программу в виде байт-кода. Байт-код Java — это промежуточное представление программы, состоящее из инструкций, которые будет выполнять *интерпретатор Java*. Таким образом, результат работы .компилятора javac не является непосредственно выполняемым кодом. Для действительного выполнения программы следует использовать Java-интерпретатор с именем java. Командная строка запуска интерпретатора выглядит так:

C:\jaya Example

После выполнения программы на экран выводится строка:

Это простая Java-программа.

После компиляции исходного кода каждый индивидуальный класс помещается в собственный выходной файл, имя которого совпадает с именем этого класса, и расширением .class. Присвоение исходному файлу того же имени что и содержащемуся в нем классу - неплохая идея, так как имя исходного файла будет согласовано с именем class-файла. Когда вы запускаете интерпретатор Java, вы в действительности специфицируете имя класса, который должен исполнить интерпретатор. Он автоматически отыскивает файл с тем же именем и расширением .class. Если интерпретатор находит такой файл, то он выполняет код, содержащийся в указанном классе.

Хотя программа Example.java очень короткая, она содержит некоторые ключевые свойства, общие для всех Java-программ. Рассмотрим поближе каждую часть программы.

/* Это простая Java-программа.Она состоит только из одного класса. При сохранении назовите этот файл "Example.java" */

Это *комментарий.* Как и большинство других языков программирования, Java позволяет вводить замечания в исходный программный, файл. Содержимое комментария игнорируется компилятором. Комментарий описывает и объясняет работу программы всем, кто читает исходный код. В нашем случае комментарий описывает программу и напоминает, что исходный файл должен быть назван Example.java. Конечно, в реальных приложениях комментарий, в общем случае, объясняет, как 'работает некоторая часть программы или какими специфическими свойствами она обладает.

Java поддерживает три стиля комментариев. Первый, показанный в начале нашей программы, называют *многострочным комментарием.* Этот тип комментария должен начинаться символами /* и заканчиваться */. Все символы, находящиеся между этими парами, игнорируются компилятором. Как понятно из названия, многострочный комментарий может состоять из нескольких строк.

Следующая строка кода программы:

class Example {

Эта строка использует ключевое слово class для объявления, что определяет новый класс. Example — это идентификатор, являющийся именем класса. Полное определение класса, включающее все его члены, размещается между открывающей ({) и закрывающей (}) фигурными скобками.

Следующая строка в программе — *однострочный комментарий:*

Вывод нескольких строк текста одним оператором вывода.

```
Например, при выполнении следующего файла
// Файл Welcome3. Java
  public class Welcome3 {
    public static void main( String args[])
    {
       System.out.println(
             "Добро\nпожаловать\nв мир\nпрограммирования на  Java!");
     }  // конец метода main
   } //конец класса Welcome3
будет на экране следующая выдача
```

```
Добро
Пожаловать
В мир
Программирования на Java!
```

Обычно символы строки выводятся друг за другом, как они располагаются в строке. Когда обратный слэш (\), называемый знаком перехода или escape-символом, встречается в строке символов, следующий символ комбинируется с обратным слэшем и формирует управляющую последовательность (escape-последовательность) Управляющая последовательность \n означает вставку новой строки **.**

2. **Задание**. Напишите программу, которая выводит в командную строку изображения прямоугольника, построенного из символов звездочки.

Lab2: Составление Java программы.

1. Повторить и закрепить понятия первого задания
2. **Задание**. Самостоятельно решить следующие задачи:

   Напишите программу, которая выводит в командную строку изображения овала, стрелки и ромба, составленные из символов звездочки.

   Основываясь только на материала первой лабораторной работы, создайте программу, в которой рассчитываются квадратные и кубические значения целых чисел от 1 до 10. Полученные результаты должны выводиться в виде таблицы.

Lab3: Использование диалогового окна для ввода-вывода.

1. Изучить, понять текст программы и выполнить ее на компьютере

```
// Файл:  Welcome4.java
// Вывод нескольких строк в окне диалога пакеты расширений

import javax.swing.JOptionPane; // импорт класса JOptionPane
public  class Welcome4  {
  // Выполнение приложения Java начинается с метода main
  public static void main (  String args[]  ) {
     JOptionPane.showMessageDialog(  null,
           "Добро\nпожаловать\nв мир\nпрограммирования на Java!");
     System.exit ( 0 );    // завершение приложения
     }    // конец метода main
} // конец класса
```

Одно из достоинств Java заключается в том, что этот язык имеет богатый набор предопределенных классов, которые программисты могут использовать повторно и не заниматься «изобретением колеса во второй раз». Мы будем постоянно использовать многие из этих; классов в нашей книге. Многочисленные предопределенные классы Java группируются по категориям связанных классов, и эти группы называются *пакетами.* Все вместе эти пакеты объединяются термином *библиотека классов Java* или *программный интерфейс приложений Java (API Java).* Пакеты API Java разделяются на *базовые пакеты* и *пакеты расширений.* Названия пакетов начинаются со слов "java" (базовые пакеты) или "javax" (пакеты расширений). Многие базовые пакеты и пакеты расширений включены в состав Java 2 Software Development Kit. Поскольку Java продолжает развиваться, то новые пакеты разрабатываются как пакеты расширений. Эти расширения, как правило, можно загрузить с сайта **java.sun.com** и использовать для расширения возможностей Java. В этом примере мы используем класс JOptionPane, который в Java определяется в пакете *javax.swing.* Строка

```
import javax.swing.JOptionPane; // импорт класса JOptionPane
```

это пример оператора .import. Компилятор использует операторы import для идентификации и загрузки классов, используемых в программах Java. Операторы помогают компилятору локализовать классы, которые вы намереваетесь исполььзовать. Для каждого нового

класса API Java, который вы намереваетесь использовать, вы должны указывать пакет, в котором можно найти этот класс. Эта информация о пакетах очень важна: она помогает вам, в том числе, найти описания пакета и класса в *документации по API Java.* Web-версию этой документации вы можете найти по следующей ссылке:

`java.sun.com/j2se/l.3/docs/api/index.html`

Кроме того, вы можете загрузить эту документацию на ваш компьютер, используя следующую ссылку:

`java.sun.com/j2se/l.3/docs.html`

Операторы `import` должны размещаться только перед определениями классов, внутри и после нельзя.

В пакете javax.swing содержится много классов, которые программисты могут использовать для создания *графического интерфейса пользователя (GUI) в своих* приложениях. *Компоненты графического интерфейса пользователя* облегчают ввод данных пользователем вашей программы и позволяют улучшить форматирование и представление данных, выводимых вашей программой.

Класс Java *JOptionPane* обеспечивает стандартные диалоговые окна, которые могут быть использованы в программах для отображения сообщений пользователю. Такие окна называются *окном сообщений*. В данной программе результат работы программы выводится на таком диалоговом окне.

В строке

```
JOptionPane.showMessageDialog( null,
    "Добро\nпожаловать\nв мир\nпрограммирования на Java!");
```

вызывается метод *showMessageDialog* класса JOptionPane. Метод имеет два параметра. В том случае, когда метод требует несколько параметров, то эти параметры отделяются *запятыми* (,). Первый параметр в наших примерах всегда будет получать значение *null* — пустой указатель. Второй параметр — это выводимая строка. Используя первый' параметр, мы можем определить приложению Java место, в котором выводить диалоговое окно. Когда первый параметр имеет значение null, то диалоговое окно появляется в центре экрана монитора компьютера. Большинство приложений, которые вы запускаете на вашем компьютере, выполняется в отдельном окне (например, программы для работы с электронной почтой, браузеры Web и текстовые процессоры).

Метод JOptionPane.showMessageDialog — специальный метод класса JOptionPane, называемый *статическим методом.* Такие методы всегда вызываются, используя имя класса, за которым следует операция точки (.) и имя метода, т.е.

ClassName.methodName{ параметры)

Многие из предопределенных методов, которые мы используем, представляют собой статические методы. Выполнение оператора приведет к выводу диалогового окна, которой содержит кнопку ОК. Java позволяет разбивать большие операторы на несколько строк. Однако вы не имеете права разбивать оператор где-то на середине идентификатора или символьной строки.

В строке

```
System.exit(0);    //  завершение приложения
```

Вызывается статический метод exit класса System, завершающий выполнение приложения. Эта строка должна присутствовать в любом приложении, использующем графический интерфейс пользователя, для завершения приложения. Обратите еще раз на синтаксис вызова метода: сначала имя класса (System), затем символ операции точка (.) и за ним — имя метода (exit). Напоминаем вам еще раз что, идентификаторы, начинающиеся с заглавной буквы, обычно используются для имен классов. Поэтому вы можете предположить, что System — это имя класса.

Класс System входит в пакет *java.lang.* Обратите внимание, что класс System *не* импортируется с помощью оператора import в начале программы. По умолчанию пакет java.lang импортируется в каждую программу Java. Пакет java.lang — это единственный пакет API Java для работы с которым вы не должны использовать оператор import.

Аргумент 0 в вызове метода указывает на то, что приложение успешно завершило свою работу. (Значение, отличное от нуля, обычно указывает на то, что в процессе выполнения приложения произошла ошибка.) Это значение передается в командное окно. в котором выполняется программа. Этот параметр может быть полезен, если программа выполняется из *пакетного файла* (в операционных системах: Windows 95/98/ME/NT/2000) или из *сценария оболочки* (в операционных системах UNIX или Linux). В пакетных файлах или сценариях оболочки несколько программ часто выполняются последовательно, друг за другом. Когда первая программа завершает свое выполнение, автоматически начинает выполняться следующая программа. В таких командных файлах можно использовать значение этого параметра для того, чтобы определить, должны ли выполняться вызовы остальных програм

2. Сложение целых чисел

В нашем следующем приложении пользователем вводятся с клавиатуры два *целых числа* (речь идет о целых числах вида -22, 7 или 1024), вычисляется сумма этих значений и результат выводится на экран. В этой программе используется другое предопределенное диалоговое окно из класса JOptionPane, называемое *диалогом ввода*. Это окно применяется, когда пользователь должен ввести в программу значение. В программе, кроме того, используется диалоговое окно для вывода сообщения, в которое выводится вычисленное значение суммы целых чисел.

```
// Файл:  Addition.java
// Сложение целых чисел. пакеты расширений Java
import javax.swing.JOptionPane;    // импорт класса JOptionPane
public class Addition  {
  // Выполнение приложения Java начинается с метода main
  public static void main(  String args[]  )  {
    String firstNumber;    // первая строка,  введенная пользователем
    String secondNumber;  // вторая строка,  введенная пользователем
    int numberl;>             // первое слагаемое
    int number2;              // второе слагаемое
    int sum;                  //  сумма Чисел numberl и number2
    // ввод первого числа в виде строки
    firstNumber = JOptionPane.showInputDialog(  "Введите первое число"  );
    // ввод первого числа в виде строки
    secondNumber = JOptionPane.showInputDialog(  "Введите второе число"  );
    // преобразование введенных значений из типа String к типу int
    numberl = Integer.parselnt(  firstNumber  );
    number2 = Integer.parselnt(  secondNumber );
    // сложение чисел
    sum = numberl + number2;                    ,
    // вывод полученного результата
    JOptionPane.showMessageDialog(
               null, "Сумма равна " + sum, "Ответ:", JOptionPane.PLAIN_MESSAGE );
    System.exit( 0 );      // завершение приложения
    }    // конец метода main
 }  // конец класса Addition
```

3. Задание

3.1. Написать программу получения единой фразы соединением двух слов.

3.2. Написать программу сложения целых двух чисел, используя инициализацию переменных при объявлении.

Lab4: Простой апплет - вывод текстовой строки

Апплеты – это небольшие программы Java, которые могут помещаться HTML-документы, т.е. Web-страницы. Когда браузер загружает Web-страницу, содержащую апплет, апплет загружается в браузер Web и начинает выполняться. Браузер, который выполняет апплет, выступает в данном случае в качестве контейнера апплета. Комплект разработки J2SDK включает в себя контейнер апплета appletviewer, в котором удобно проводить тестирование апплетов, прежде чем вы будете размещать их в Web-страницах.

1. Изучить и выполнить следующий текст простого апплета, который повторяет программу, представленную в Lab3, и выводит сообщение Добро пожаловать в мир программирования!.

```
// Файл: WelcomeAppLet.java
// Ваш первый апплет Java.
//  Базовые пакеты Java
import java.awt.Graphics;//загрузка класса Graphics
//  Пакеты расширений Java
import javax.swing.JApplet;//загрузка класса JApplet
public class WelcomeAppLet extends Japplet
{
  // вывод текстовой строки в апплете
  public void paint( Graphics g ) {
    // вызов унаследованной версии метода paint
    super.paint( g );
    //вывод строки в точке с координатами x=25 и y=25
    g.drawstring(
    "Добро пожаловать в мир программирования!",25,25);
   }   // окончание метода paint
} // конец определения класса WelcomeApplet
```

Строка

`import java.awt.Graphics;//загрузка класса Graphics`

это оператор import, который указывает компилятору имя класса, которое нужно загрузить *(Graphics)* из пакета *java.awt,* и использовать в этом Java-апплете. Класс `Graphics` используется в апплетах Java для вывода графики типа линий, прямоугольников, овалов и строк символов. Класс также позволяет приложениям рисовать изобра-жения.

В строке

`import javax.swing.JApplet;//загрузка класса JApplet`

в операторе import выполняется загрузка класса *JApplet* из пакета javax.swing. Когда вы создаете апплет в Java, то обычно импортируете класс JApplet. *[Заменание:* в пакете *java.applet* имеется более ранняя версия класса по имени Applet, который не используется с новыми компонентами графического интерфейса пользователя Java из пакета javax.swing. По этой причине мы используем класс JApplet для создания апплетов.

Со строки

`public class WelcomeApplet extends JApplet {`

начинается определение класса WelcomeApplet. В конце строки левая фигурная скобка ({) открывает тело определения класса. Соответствующая правая фигурная скобка (}) в конце текста обозначает конец определения класса. Определение класса начинается с ключевого слова class. WelcomeApplet — это имя класса. Ключевое слово *extends* указывает на то, что класс WelcomeApplet наследует существующий код другого класса. Класс, наследником которого является `WelcomeApplet` (т.е. JApplet), указывается справа от ключевого слова *extends.* В этих отношениях наследования класс JApplet называется *суперклассом* или *базовым классом,* а

`WelcomeApplet` называется *подклассом* или *производным классом.* Использование наследования здесь приводит к тому, что класс WelcomeApplet будет иметь атрибуты (данные) и *поведение* (методы) класса JApplet, а кроме того, может иметь новые особенности, которые мы добавим в определение нашего класса `WelcomeApplet` (в частности, способность выводить строку Добро пожаловать в мир программирования! в окне апплета).

Класс JApplet обеспечивает возможности построения пустого окна апплета на экране, а наша программа просто дописывает указанную строку текста в этой заготовке. Так что программистам не нужно еще раз «изобретать велосипед», а именно заново писать программу построения пустого окна апплета. Указав всего лишь одно ключевое слово extends, вы можете воспользоваться механизмом наследования и быстро создавать на основе класса JApplet новые апплеты.

Классы используются как «шаблоны» или «проекты» *создания объектов,* которые могут использоваться в программе. Объект или *экземпляр класса* находится в памяти компьютера и содержит в себе информацию, используемую программой. Термин объект обычно подразумевает, что с ним связаны определенные атрибуты (данные) и способы поведения (методы). Методы объекта, используя атрибуты, обеспечивают функциональность, необходимую *клиенту объекта* (т.е. коду программы, которая вызывает методы объекта).

2. Компиляция и выполнение WelcomeApplet.

Как в случае классов приложений, вы должны откомпилировать классы апплета прежде, чем апплет будет выполняться. После определения класса WelcomeApplet и сохранения его в файле WelcomeApplet.java, откройте командное окно, перейдите в каталог, в котором вы сохранили определение класса апплета re команду

```
javac WelcomeApplet.java
```

чтобы выполнить компиляцию класса WelcomeApplet. Если компилятор не обнаружит синтаксических ошибок, полученный байт-код будет сохранен в файле WelcomeApplet.class.

Этот WelcomeApplet.class теперь не может быть выполнен интерпретатором java, ибо нет метода main в тексте. Теперь вместо интерпретации надо дать указание браузеру для запуска апплета. Как известно, все указания браузеру даются тегами на языке HTML. Поэтому прежде чем вы сможете выполнить апплет, вы должны создать *файл HTML на языке гипертекстовой разметки,* который может использоваться для загрузки апплета в контейнер апплета (appletviewer или браузер). Как правило, HTML имеют расширения имени файла *.html* или *.htm.* Браузеры отображают содержание текстовых документов, которые также называют *текстовыми файлами.* Чтобы выполнить апплет Java, его нужно указать в текстовом HTML файле, который контейнер апплета может загрузить и выполнить таким образом апплет. Указание на запуск апплета дается в теге <applet>. В нем обязательно задается имя файла с классом апплета параметром code, ширина и высота панели апплета в пикселях.

Простой HTML-файл — WelcomeApplet.html, — который загружает апплет в контейнер апплета будет следующим

```
<html>
  <applet code = "WelcomeApplet.class" width = "300"  height = "45">
 </applet>
</html>
```

Программа appletviewer понимает только теги HTML <applet> и </applet>, поэтому его можно назвать «минимальным браузером», поскольку он игнорирует все остальные теги HTML, appletviewer — идеальное средство тестирования апплетов, которое может гарантировать вам, что проверенные на нем апплеты будут но выполняться в браузерах Web. После того как

апплет проверен, вы можете добавить теги HTML, в которых указывается апплет, в файлы HTML, которые будут просматривать пользователи Интернет. Оба файла WelcomeApplet.html и WelcomeApplet.class должны помещаться в один каталог на сервере.

Чтобы выполнить апплет WelcomeApplet в appletviewer, вам нужно открыть командное окно, перейти в каталог, содержащий ваш апплет и HTML-файл, и ввести команду

```
appletviewer WelcomeApplet.html
```

Еще раз укажем, что appletviewer для загрузки апплета *требуется* файл HTML, Это несколько отличается от интерпретации java-приложений, когда нужно указывать только имя класса приложения. И указанная команда должна выдаваться из текущего каталога, в котором находятся отображаемые HTML-файл и файл **.class** апплета. Интернет браузер просматривая HTML-файл, выполнит тег <applet> и загрузит апплет. После загрузки апплет появится в окне браузера.

2. Задание. Переписать приложение из Lab3 в виде апплета и его выполнить

Lab5: Повторение, управляемое счетчиком

1. Изучить, понять и выполнить на компьютере

```
//  Файл:  Sum.java
// Повторение,  управляемое  счетчиком,
//  реализованное в  структуре  for
// Пакеты расширений Java
import javax.swing.JOptionPane;
public class Sum {
  // Выполнение приложения Java начинается с метода main
  public static void main( String args[]) {
     int  sum =0;
     //Суммирование четных чисел от 2  до  100
     for(int number=2;number<=100;number+=2)
       sum += number;
    // Вывод результатов
    JOptionPane.showMessageDialog(  null, "Сумма равна  "  +  sum,
           "Суммирование четных чисел от 2 до 100",
            JOptionPane.INFORMATION_MESSAGE  );
    System.exit(0);
 }
}
```

2. В следующем примере структура for используется для вычисления суммы сложных процентов, начисляемых в конце года. Задача формулируется следующим образом:

Клиент банка открыл счет и внес на него $1000.00. По счету начисляются проценты из расчета 5% годовых. В предположении, что все начисленные проценты остаются на счету, рассчитайте и выведите сумму на счете в конце каждого года за десятилетний период. Используйте следующую формулу для расчета процентов:

$a = p\ (1 + г)^a$, *где*

p — первоначально внесенная на счет сумма, г — годовая процентная ставка, *n* — число прошедших лет, *a* — сумма на депозите на конец n-го года. Для решения этой задачи потребуется цикл, в котором выполняются указанные четы суммы депозита на конец каждого года за период в 10 лет. Приложение, в котором решается эта задача, приведено ниже.

```
//Файл:Расчет сложных процентов
```

```
// Базовые пакеты Java
import Java.text.NumberFormat;
import Java.util.Locale;
// Пакеты расширений-Java
import javax.swing.JOptionPane;
import javax.swing.JTextArea;
public class Interest {
  public static void main(String args[]){
     double amount,principal = 1000.0,  rate =0.05;
     // Создаем ссылку на объект класса NumberFormat для
     //форматирования чисел с плавающей точкой
     //с двумя дробными десятичными разрядами
     NumberFormat moneyFormat = NumberFormat.getCurrencyInstance( Locale.US );
     //Создаем объект JTextArea для вывода результатов
     JTextArea outputTextArea = new JTextArea();
     // Выводим первую строку текста в outputTextArea
     outputTextArea.setText('Год\tСумма депозита\n"  );
     //calculate amount on deposit for each of ten years
     for (int year=l; year<=10; year++ )  {
        // Расчет суммы на депозите на каждый год
        amount = principal*Math.pow(1.0 + rate, year );
        //Добавляем строку текста в outputTextArea
        outputTextArea.append(year + "\t" +
                     moneyFormat.format( amount )  +  "\n"  );
     }   // конец структуры for
  // Вывод результатов
  JOptionPane.showMeseageDialog( null,   outputTextArea,
          "Расчет сложных процентов",  JOptionPane.INFORMATrON_MESSAGE );
   System.exit( 0 ); // завершение работы приложения
    }   // конец метода main
} // конец класса
```

Ссылка **moneyFormat** на объект класса NnmberFormat, которая здесь же инициализируется при помощи вызова статического метода getCurrencyInstance класса **NumberFormat.** Этот метод возвращает объект **NumberFormat,** с помощью которого можно форматировать числовые значения как денежные значения (например в Соединенных Штатах Америки денежные обычно начинаются с символа доллар — $). Параметр метода — Locale.US указывает на то, что денежные значения должны отображаться с предшествующим знаком доллара ($), целая часть от дробной должна отделяться десятичной точкой, а тысячи в целой части должны разделяться запятой (например, 1.56).

Ссылка **outputTextArea** на объект класса JTextArea, указывает на вновь создаваемый объект класса **JTextArea** (из пакета **.swing). JTextArea** — это компонент графического интерфейса пользователя, который может отображать набор строк текста. Диалоговое окно сообщения, в котором отображается компонент **JTextArea,** определяет ширину и высоту **JTextArea** на основании объекта **String,** который в нем содержится. Мы представляем компонент графического интерфейса пользователя сейчас, потому что мы будем широко использовать его в наших примерах, в которых выводится достаточно много строк текста. Этот компонент графического интерфейса пользователя позволяет нам прокручивать строки текста, так что мы можем спокойно просмотреть весь вывод программы. Методы, используемые для задания текста в объекте **JTextArea**, включают в себя ***setText и append.***

3. Задание. Эти программы переписать с использованием операторов while и do while, затем проверить на компьютере

Lab6. Использование операций break, continue

1. Изучить, понять и выполнить на компьютере

```
//Файл: BreakTest.java
```

```
// Использование оператора break в структуре
// повторения for
//Пакеты расширений Java
import javax.swing.JOptionPane;
public class BreakTest {
  public static void main ( String args[] ) {
     String output = "";
     int count;
     // Цикл повторяется 10 pas
     for ( count = 1; count <= 10; count++ )  {
        // Когда count равно 5, то выполнение цикла
        // прекращается
        if ( count == 5 ) break; //выполнение цикла прекращается
           output += count + " ";
      }    // окончание структуры for
    output += "\nВыход из цикла, когда count = " + count;
    JOptionPane.showMessageDialog( null, output );
    System.exit( 0 ); // завершение работы приложения
  }//конец метода
}  //конец класса
```

2.

```
// Файл: BreakLabelTest.java
// Использование оператора break с меткой
// Пакеты расширений Java
import javax.swing.JOptionPane;
public class BreakLabelTest  {
  public static void main(  String args[]  ) {
    String output =  "";
    stop:  {    // помеченный блок операторов
    //  Цикл должен повторяться 10 раз
    for ( int row = 1;  row <= 10;  row++  )   {
      //  Цикл по  5  столбцам
      for(int column=1;column<=5;column++) {
        // если номер строки равен 5, выполнение блока "stop"  прерывается
        if  (  row == 5  ) break stop; //переход в конец блока "stop"
          output  +=  "*";
      } // окончание внутренней структуры for
      output +=  "\n";
    } //окончание внешней структуры for
    // Следующий оператор никогда не выполняется
    // output += "\nЦикл завершен нормально";
    }    //  конец помеченного блока
    JOptionPane.showMessageDialog(null,output,"Использование break с меткой",
    JOptionPane.INFORMATION_MESSAGE  );
    System.exit(0); //  завершение работы приложения
  }    //  окончание метода main
}   //  окончание класса BreakLabelTest
```

3.

```
//Файл:ContinueLabelTest.java //Использование помеченного оператора continue
//Пакеты расширений Java
import  javax.swing.JOptionPane;
public  class ContinueLabelTest  {
  public static void main(String args[]){
    String output =  "";
    nextRow:// целевая метка оператора continue
    // Цикл из  10  строк
    for  ( int row = 1; row <= 5; row++ ) {
      output +=  "\n";
```

```
      //Цикл из 10 столбцов для каждой строки
      for(int column=1; column <= 10; column++ )   {
        //Если номер столбца превышает номер строки, начать новую строку
        if ( column > row ) continue nextRow; //следующая итерация
                                              //помеченного цикла
           output += "*";
      }
    }
    JoptionPane.showMessageDialog( null,
       output,"использование continue с меткой",
       JoptionPane.INFORMMATION_MESSAGE );
    System.exit(0);
  }
}
```

4. Задание. Изученное применить в программировании следующего вычисления:
y = 1+2+3+5+6+7+8+9+10

Lab7. Создание и использование пользовательского метода

1. Изучить, понять и выполнить на компьютере

```
// Файл: SquarelntegerS. java
// Метод square - разработанный пользователем метод
// Базовые пакеты Java
import java.awt.Container;
// Пакеты расширений Java
import javax.swing.*;
public class SquarelntegerS extends JApplet {
  // Определение GUI-компонента и расчет квадратов
  // чисел от 1 до 10
  public void init(){
    // Объект JTextArea будет использоваться для вывода результата
    JTextArea outputArea = new JTextArea();
    // Определение контейнера апплета
    Container container = getContentPane();
    // Связывание outputArea с container
    container.add( outputArea );
    int  result;
    // В переменной сохраняется результат вызова метода square
    String jutput = "";
    //  Строковое представление результатов
    // Цикл из  10 итераций
    for (int counter = 1;,counter<=10; counter++)  {
    // Расчет квадратного значения переменной  counter
    result = square(counter);
      // Значение  result объединяется со строкой output
      output +=  "Квадрат значения  "  +
                 counter + "  равен "  + result,+  ''\n";
    }    // окончание структуры for
    outputArea.setText(  output  );
    // результаты помещаются в объект JTextArea
  }    // окончание метода init

  //  определение метода  square
  public int square( int y ) {
    return y*y;//возвращается квадратное значение y

  } //окончание метода square

} // конец класса SquareIntegers
```

2.

```
//Файл:  Factorial.java
/*   Этот класс не определяет метод main() и поэтому не является самостоятельной программой. Он, тем не менее, определяет полезный метод, который можно использовать в других программах */
public class Factorial  {
  public static int factorial(int x)  {
    int fact = 1;
    for(int i = 2; i <= x; i++) // цикл
      fact *= i;   // краткая запись для fact = fact * i;
    return fact;
  }
}
```

3.

```
//Файл:  Maximum.java
// Нахождение максимального из трех значений
import java.awt.Container;
// Пакеты расширений Java
import javax.swing.*;
public class Maximum extends  JApplet  {
  //Инициализация апп. ввод данных пользователем
  public void init () {  // ввод данных
    String si = JOptionPane.showInputDialog (
              "Введите первое число с плавающей точкой ");
    String s2 = JOptionPane.showInputDialog(
               "Введите второе число с плавающей точкой");
    String s3 = JOptionPane.showInputDialog(
               "Введите третье число с плавающей точкой ");
    // Преобразование введенных значений
    double numberl = Double.parseDouble( si );
    double number2 = Double.parseDouble( s2 );
    double number3 = Double.parseDouble( s3 );
    // Вызов метода maximum для определения
    //  максимального значения
    double max = maximum( numberl, number2, пишЬегЗ );
    // Создание объекта JTextArea для вывода результатов
    JTextArea outputArea = new JTextArea();
    // Вывод чисел и найденного максимального значения
    outputArea.setText( "первое число:  " + numberl +
        "\nвторое число:  " + number2 + "\nтретье
         число:  " + пишЬегЗ + "\nмаксимальное значение:  " + max );
    // Получение ссылки на контейнер апплета
    Container container = getContentPane();
    // Связывание outputArea с Container
    container.add( outputArea );
  }   // окончание метода init
  // Метод maximum использует метод max класса Math
  // для определения максимального значения
  public double maximum( double x,double y,double z ) {
    return Math.max(  x,  Math.max(  y,  z  1  );
  }    //  окончание метода maximum
} //  окончание класса Maximum
```

3. Задание. Изученное применить к решению задачи.

Y= max(a+c, b) + max(a, b, c)

Lab8. Создание класса и его использование.

1. Изучить, понять и выполнить на компьютере

```
/* Программа, которая использует Box-класс. Назовите этот файл BoxDemo.java */
class Box {
  double width;
  double height;
  double depth;
} // конец класса Box

// Этот класс объявляет объект типа Box.
class BoxDemo  {

  public static void main(String args[]) {

    Box mybox = new Box();

    double vol;
    // присвоить значения экземплярным переменным объекта mybox
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;
    // вычислить объем блока
    vol=mybox.width * mybox.height * mybox.depth;
    System.out.println("Объем равен " + vol);
   } // конец метода main
 } // конец класса BoxDemo
```

2.

```
//Эта программа включает метод внутрь класса
class Box {
  double width;
  double height;
  double  depth;

  // показать объем блока
  void volume ()  {
    System.out.print ("Объем равен");
    System.out.println(width * height * depth);
}

class BoxDemo3 {
  public static void main (String args [ ]){
    Box myboxl = new Box ();
    Box mybox2 = new Box();
    // присвоить значения переменным myboxl
    myboxl. width =10;
    myboxl.height = 20;
    myboxl.depth = 15;
    /* присвоить другие значения переменным экземпляра mybox2  */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;
    // показать объем первого блока
    myboxl.volume();
    // показать  объем второго блока
    mybox2.volume();
   }
}
```

3. Добавление метода в класс с возвратом значения

```
class Box {
  double width;
  double height;
  double  depth;

  // вычислить и возвратить объем блока
  double volume ()  {
    return width * height * depth;
  }
} // конец класса Box

class BoxDemo3 {
  public static void main (String args [ ]){
    Box myboxl = new Box ();
    Box mybox2 = new Box();
    Double vol;
    // присвоить значения переменным myboxl
    myboxl. width =10;
    myboxl.height = 20;
    myboxl.depth = 15;
    /* присвоить другие значения переменным экземпляра mybox2  */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;
    // получить объем первого блока
    vol = myboxl.volume();
    System.out.println( vol );
    // получить объем второго блока
    vol = mybox2.volume();
    System.out.println( vol );
  }
}
```

Вопрос:

1.Можно ли вместо

```
  vol = mybox2.volume();
```

и

```
  System.out.println( vol );
```

использовать один оператор

```
  System.out.println( mybox2.volume()); ?
```

2. Пусть дан следующий метод

```
void setDim(double w, double h, double d ) {
    width = w;
    height = h;
    depth = d;
 }
```

Что делает метод setDim при вызове?

3. Задание. Классу Box добавить метод, устанавливающий начальные значения переменным экземпляра класса, и написать приложение использования такого нового класса.

Lab9. Создание и использование конструктора

1. Изучить, понять и выполнить на компьютере

/*Box использует конструктор для инициализации размеров блока.*/

```
class Box  {
  double width;
  double height;
  double depth;

  // Это конструктор класса Box.
  Box() {
    System.out.println(”Создание Box”);
    width = 10;
    height = 10;
    depth = 10;
  }

  // метод вычисления объема
  double volume()  {
     return width * height * depth;
   }
} // конец класса Box

class BoxDemo6 {
  public static void main(String args[]) {
    /* объявить, разместить, в памяти и инициализировать Box-объекты */
    Box myboxl = new Box() ;
    Box mybox2 = new Box();
    double vol;
    // получить объем первого блока
    vol = myboxl.volume();
    System.out.println("Объём равен " + vol);
    // получить объем второго блока
    vol = mybox2.volume();
    System.out.println("Об6ъем равен “ + vol);
  }
}
```

3. **Задание.**

- Используя следующий вид параметризованного конструктора:

```
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
```

написать приложение, вычисляющее объем Box-объектов, указанного вами размеров.

- Используя объектный подход, написать программу сложения двух целых чисел

Lab10. Использование наследования для расширения класса

1. Изучить, понять и выполнить на компьютере

Версия класса Box расширяется, чтобы включить четвертый компонент (переменную) weight.

```
/*Программа использует наследование для расширения Box.*/
class Box (
  double width;
  double height;
  double depth;

  // создать  клон объекта
  Box(Box ob) { // передать  объект конструктору                ,
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
```

```
  /* конструктор, используемый, когда указаны все размеры */
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  /* конструктор, используемый, когда размеры не указаны */
  Box() {
    width = -1;   //использовать -1 для указания
    height = -1;  // неинициализированного
    depth = -1;   // блока
  }

 /*  конструктор, используемый для создания куба */
 Box(double len) {
   width = height = depth = len;
 }

  // вычислить и вернуть объем
  double volume() {
     return width * height * depth;
  }
}

// Box расширяется для включения веса.
class BoxWeight extends Box {
  double weight;      // вес блока

  // конструктор для BoxWeight
  BoxWeigh (double w, double h, double d, double m) {
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
 }

class DemoBoxWeight {
 public static void main(String args[]) {
   BoxWeight myboxl=new BoxWeight(10,20,15,34.3);
   BoxWeight mybox2=new BoxWeight(2,3,4,6.076);
   double vol;
   vol = myboxl.volume();
   System.out.println("Объем myboxl равен " + vol);
   System.out.println("Вес myboxl равен "  +  myboxl.weight);
   System.out.println();

   vol= mybox2.volume();
   System.out.println("Объем mybox2 равен  " + vol);
   System.out.println("Вес mybox2 равен  "   + mybox2.weight);
 }
}
```

BoxWeight наследует все характеристики Box и прибавляет к ним компонент weight. Для Boxweight нет необходимости заново создавать все свойства Box надо просто расширить Box, чтобы достичь своих собственных целей.

Главное преимущество наследования состоит в том, что, как только вы создаете суперкласс, который определяет общие для набора объектов атрибуты, его можно использовать для создания любого числа более специфичных подклассов. Каждый подкласс может добавить свою

собственную классификацию. Например, следующий класс наследует Box и прибавляет цветовой атрибут:

```
// расширяется для включения цвета.
class ColorBox extends Box  {
  int color;  // color of box
 ColorBox(double w, double h, double d, int c){
   width = w;
    height = h;
    depth = d;
    color = c;
  }
}
```

Помните, как только вы создали суперкласс, который определяет общие аспекты объекта, этот суперкласс может наследоваться для формирования специализированных классов. Каждый подкласс просто прибавляет свои собственные, уникальные атрибуты. В этом сущность наследования.

Замечание. Конструктор класса `BoxWeight` явно инициализирует поля метода `Box()`.

2. Задание.

Подкласс может вызывать метод конструктора, определенный его суперклассом, при помощи следующей формы `super`:

```
super(список параметров);
```

Переделать программу, используя метод конструктора определенный его суперклассом, и выполнить.

Lab11. Использование абстрактного класса для построения полиморфного метода.

1. Изучить, понять и выполнить на компьютере

```
abstract class Pet {
  abstract void voice();
  }

class Dog extends Pet {
  int k = 10;
  void voice() {
  System.out.println("Gav-gav! ");
 }
}

class Cat extends Pet {
  void voice() {
  System.out.println("Miaou!");
  }
}

class Cow extends Pet {
  void voice(){
    System.out.println("Mu-u-u! ");
    }
  }

  public class Chorus {

public static void main(String[] args) {
  Pet [] singer = new Pet[3];
  singer[0) = new Dog();
       singer[1] = new Cat();
```

```
   singer[2] = new Cow();
   for ( int i = 0; i < singer.length; i++)
    singer[i].voice();
 }
}
```

В тексте программы, хотя задается одна общая форма записи команды singer[i].voice();, но тем не менее при выполнении программы печатаются возгласы разных голосов. Все дело в определении поля singer[i]. Хотя массив ссылок singer[] имеет тип Pet, каждый его элемент ссылается на объект своего типа Dog, Cat, Cow. При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки. Так в Java реализуется полиморфизм.

4. **Задание.** Написать самим новый пример использования абстрактного класса.

Lab12. Перегрузка метода и конструктора

1. Перегрузка метода

В языке Java в пределах одного класса можно определить два или более методов, которые совместно используют одно и то же имя, но имеют разное количество параметров. Когда это имеет место, методы называют перегруженными, а о процессе говорят как о перегрузке метода. Перегрузка методов – один из способов, с помощью которого Java реализует полиформизм.

// Демонстрация перегруженного метода

```
class OverloadDemo {
  void test() {
    System.out.println("Параметры отсутствуют");
  }

  //Перегруженный метод test с одним int-параметром.
  void test (int a) {
    System.out.println("a =" + a);
  }

  //Перегруженный метод test с двумя int-параметром.
  void test (int a, int b) {
    System.out.println("a,  b: " + a+" " +b);
  }

  //Перегруженный метод test с одним double-параметром.
  void test (double  a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}

class Overload {
   public static void main(String args[]) {
     OverloadDemo ob =  new OverloadDemo();
     double result;
     // Вызываються все версии test()
     ob. test();
     ob. test (10) ;
     ob.test (10, 20);
     result =  ob.test(123.2);
     System.out.println("Результат ob.test(123.2): "  + result);
  }
}
```

Эта программа генерирует следующий вывод:

```
Параметры отсутсвуют
a=10
```

```
a, b: 10 20
double a: 123.2
Результат ob.test(123.2): 15178.24
```

Как можно видеть, метод test() перегружен четыре раза.

2. Задание. Аналогично можно также перегружать методы конструкторов. Придумайте для конструктора класса Box перегруженные варианты и используйте их.

Lab13. Создание и использование пакета

Каждый класс и интерфейс принадлежать определенному пакету. Пакеты облегчают повторное использование программного кода, поскольку программы могут импортировать классы из других пакетов.

Следующее приложение иллюстрирует, как нужно создавать ваши собственные пакеты и использовать в программе классы из этого пакета.

1. Создание пакета

```
//
package myPack;
class Balance {
  String name;
  double bal;

  Balance (String n,  double b)  {
     name = n;
     bal = b;
  }
  void show() {
    if(bal<0) System.out.print("→ ");
    System.out.println(name + ":  $"  + bal);
  }
}

class AccountBalance  {
  public static void main(String args[])  {
    Balance current[] = new Balance[3];
    current[O] = new Balance("K.J.Fielding",23.23);
    current[l] = new Balance("Will Tell",  157.02);
    current[2] = new Balance("Tom Jackson", 12,33);

    for(int i=0;  i<3;  i++)  current[i].show();

  }

}
```

Назовите этот файл AccountBalance.java и поместите его в каталог с именем myPack. Далее откомпилируйте файл и удостоверьтесь, что результирующий .class файл также находится в каталоге myPack. Затем попробуйте выполнить класс AccountBalance, используя следующую командную строку

java myPack.AccountBalance

Помните, когда вы выполняете эту команду, то должны или быть на уровень выше каталога myPack, или иметь подходящую установку переменной CLASSPATH.

Класс AccountBalance теперь является частью пакета myPack. Это означает, что он не может быть выполнен отдельно. То есть вы не можете использовать следующую командную строку:

java AccountBalance

AccountBalance должен быть квалифицирован именем своего пакета.

2. Создание повторно используемого класса.

Для создания класса, который будет *повторно использоваться*, нужно выполнить следующие шаги:

1. Определите класс как открытый (public). Если не объявить как открытый, то его могут использовать только другие классы из этого же самого пакета.
2. Выберите имя для пакета и добавьте оператор package в файл исходного текста, в котором определяется класс для последующего многократного использования.
3. Откомпилируйте класс так, чтобы он попал в соответствующую структуру каталогов пакета.
4. Импортировать повторно используемый класс в программу и использовать его.

3. Пример использования

Теперь если нужно, чтобы класс Balance пакета myPack, показанный ранее, был автономным классом общего пользования (многократного использования) вне пакета myPack, то необходимо объявить его как public и поместить в собственный файл, как показано ниже:

```
//
package myPack;
public class Balance (
  String name;
  double bal;
  public Balance (String n,  double b)  {
     name = n;
     bal = b;
  }
  public void show() {
    if(bal<0) System.out.print("→ ");
     System.out.println(name + ":  $"  + bal);
  }
}
```

Как вы видите, класс Balance,теперь – с общим (public) доступом. То же можно сказать о его конструкторе и методе show(). Это означает, что они доступны любым типам кода вне пакета myPack. Например, в следующей программе класс TestBalance импортирует пакет myPack и получает возможность использовать класс Balance:

```
import myPack.*;
class TestBalance {
  public static void main(String args[])  {
    Balance test = new Balance("K.J.Field",23.23);
    Test.show();
  }
}
```

4. Задание.

- В качестве эксперимента удалите спецификатор public из класса Balance .и затем попробуйте откомпилировать TestBaiance. Как объяснялось ранее, в результате вы получите сообщение об ошибке.
- Изучить текст следующей программы, понять и составить блок-схему алгоритма:

```
/* Этот класс демонстрирует сортировку чисел при помощи простого алгоритма */
public class SortNumbers {
   public static void sort(double[ ] nums) {
   for(int i = 0, i < nums.length; i++) {
     int min = i;
     for(int j = i; j < nums.length; j++){ if (nums[j] < nums[min]) min = j;}
     double tmp;
```

```
        tmp = nums[i];
        nums[i] = nums[min];
        nums[min] = tmp;
      }
    }
    /* Это простая тестирующая программа для вышеприведенного алгоритма*/
     public static void main(String[ ] args) {
        double[ ] nums = new double[10];
        for(int i = 0; i < nums.length; i++)  nums[i] = Math.random() * 100;
        sort(nums);
        for(int i = 0; i < nums.length; i++) System.out.println(nums[i]);
     }
   }
```

- Вставив в тело класса, например, public static class Test { }, полный текст метода main, можно превратить этот метод, так называемый *внутренний класс*, который будет содержаться в классе SortNumbers. В результате компиляции получиться два файла класса SortNumbers.class и SortNumbers$Test.class. Для запуска тестовой программы тогда необходимо вызывать интерпретатор java, используя символ $ вместо символа точка в имени класса (SortNumbers$Test). Сделав еще необходимые уточнения, протестировать модифицированный вариант программы.

Lab14. Использование интерфейса для организации полиморфизма

1. Изучите и выполните

```
interface Voice{
  void voice();
}

class Dog implements Voice{
    public void voice() {
      System. out. println ("Gav-gav! ");
    }
}

class Cat implements Voice{
  public void voice() {
    System.out.println("Miaou!");
  }
}

class Cow implements Voice{
   public void voice(){
     System. out. println ("Mu-u-u! ") ;
   }
}

public class Chorus{
   public static void main(String[] args) {
       voice [] singer = new Voice[3];
        singer [0] = new Dog();
        singer[1] = new Cat();
         singer[2] = new Cow();
         for ( int i = 0; i < singer.length; i++)
```

```
            singer[i].voice();
    }
}
```

2. Сопоставьте с примером из Lab11. Вопрос: Зачем в Java есть и абстрактные классы, и интерфейс? Нельзя ли было обойтись одной из этих конструкций?
3. Задание. Придумайте пример необходимости использования интерфейса.

Lab15. Обработка исключительных ситуаций

1. Программа без обработки исключений

```
class SimpleExt {       // В классе используется метод Integer.parseInt() для преобразования в число строки символов,
  public static void main(String[] args) {    // заданный в командной строке
    int n = Integer.parseInt(args[0]);
    System.out.println("10 / n ~ " + (10 /n));
    System.out.print1n("After all actions");
  }
}
```

2. Программа с блоками обработки исключений

```
class SimpleExt1 {
  public static void main( String[] args) {
    try {   // в секции try содержится блок кода, который может выдать исключение.
      int n = Integer.parseInt(args[O]);
      System. out. println ("After parseInt () ) ;
      System.out.println(" 10 / n = " + (10 / n);
       System.out.println("After results output");

    } // за секции try следует секции catch, в каждй секции catch заботится об определенном типе исключений.
    catch(ArithmeticException ae) {
       System.out.println("From Arithm.Exc.catch: " + ae);
    }
    catch(ArrayIndexOutOfBoundsException arre) {
       System.out.print1n("From Array.Exc. catch: " + arre);
    }
    finally {
      System.out.println("From finally");
    }
    System.out.println("After all actions");
  }
}
```

3. Выбрасывание исключения из метода

```
class SimpleExt2 {
  private static void f(int n) {
    System.out.println(" 10 / n = " + (10 / n);
  }

  public static void main( String[] args) {
    try {
      int n = Integer.parseInt(args[O]);
      System. out. println ("After parseInt () ) ;
      f(n);
```

```
    System.out.println("After results output");

   }
   catch(ArithmeticException ae) {
    System.out.println("From Arithm.Exc.catch: " + ae);
   }
   catch(ArraylndexOutOfBoundsException arre) {
    System.out.print1n("From Array.Exc.catch: " + arre);
  }
  finally {
   System.out.println("From finally");
  }
  System.out.println("After all actions");
 }
}
```

Замечание. Откомпилировав и запустив эту программу убедимся, что вывод программы не изменился, он такой же как для программы 2. Исключение, возникшее при делении на нуль в методе f(), «пролетело» в метод main(), там перехвачено и обработано.

4. Интерактивный ввод

/* В этой программе показана техника, применяемая для считывания пользовательского ввода с клавиатуры. Для этого в ней используется метод readLine() объекта BufferedReader. Также показано применение метода equals() к объекту line типа String для проверки, не набрал ли пользователь “quit”. Только одна секция catch обрабатывает все объекты исключений, принадлежащих типу Exception. Exception – это суперкласс исключений всех типов, поэтому вызывается одна секция catch вне зависимости от типа возникшего исключения. */

```
import  java.io.*;
public class FastQuoter {
  public static void main(String[ ] args) throws IOException {
   // Так подготовливается чтение строк, вводимых пользователем
   BufferedReaderi=newBufferedReader(new InputStreamReader(System.in));
   // Бесконечный цикл
   for(;;)  {
     // Отображается подсказка для пользователя
     System.out.print(“FastQuoter>  “);
     //Считывается введенная пользователем строка
     String line = in.readLine();
     // Если считан конец файла,
     //  или если пользователь набрал «quit», то конец
     if ((line == null) || line.equals(“quit”)) break;
     //  Пытаемся проанализировать строку
     try {
     int x = Integer.parseInt(line);
     System.out.println( “Было введено “ +  x);
     }
     // Если что-то не в порядке, отображается общее сообщение об ошибке
     catch(Exception e) { System.out.println(“Invalid Input”); }
   }
  }
}
```

5. Задание. Изучите ранее составленные тексты программ и выберите одну, вставьте в нее команды обработки исключительных ситуаций и выполните ее с вводом некорректных данных.

Lab16. Обработка событий

Стандартная поставка J2SE JDK включает в себя богатейшую библиотеку классов, обеспечивающих создание графического интерфейса пользователя GUI. Основное средство построения графического интерфейса пользователя в технологии Java – это библиотека Swing. Она содержит более пятисот классов и интерфейсов. При воздействии пользователя на компонент интерфейса приложения возникает *событие*. Объект, в котором произошло событие, называется источником. Все события классифицированы. У каждого события есть свои методы, к которым обращается исполняемая система Java при его возникновения. Они описаны в *интерфейсах-слушателях*. Имена интерфейсов составляются из имени события и слова "Listener". Классы, реализующие такой интерфейс, классы-обработчики события. При возникновении события исполняющая система Java автоматически создает *объект* соответствующего событию класса и обращается к *экземпляру класса-обработчика*, умеющего обрабатывать события. Методы обработки события объекта автоматически выполняются, получая в качестве аргумента объект-событие и используя при обработке сведения о событии, содержащиеся в этом объекте.

1. Пример создания обработчика события. Пусть в контейнер типа `JFrame` помещено поле ввода `tf` типа `JTextField`, не редактируемая область ввода `ta` типа `JTextArea` и кнопка `b` типа `JButton`. В поле ввода `tf` вводится строка, после нажатия клавиши <Enter> или шелчка кнопкой мыши по кнопке `b` строка переносится в область `ta`. После этого можно снова вводить строку в поле `tf` и т.д. Здесь при нажатии клавиши <Enter>, и при шелчке кнопкой мыши возникает событие класса `ActionEvent`, причем оно может произойти в двух компонентах-источниках: поле `tf` или кнопке `b`. Обработка события в обоих случаях заключается в получении сроки текста из поля `tf` (например, методом `tf.getText()`) и помещения ее в область `ta` (скажем, методом `ta.append()`). Значит, можно написать один обработчик событию `ActionEvent`, реализовав соответствующий интерфейс, который называется `ActionListener`. В этом интерфейсе есть всего один метод `actionPerfomed()`, который надо определить. Итак, пишем:

```
class TextMove implements ActionListener {
  private JTextField tf;
  private JTextArea ta;
  TextMove(JTextField tf, JTextArea ta) {
      This.tf = tf; this.ta = ta;
   }
  public void actionPerformed(ActionEvent ae) {
     ta.append(tf.getText() + "\n" );
  }
}
```

Обработчик события готов. При наступлении события типа `ActionEvent` будет создан экземпляр класса-обработчика `TextMove`, конструктор получит ссылки на конкретные поля объекта-источника, метод `actionPerformed`, автоматически включившись в работу, перенесет текст из одного поля в другое.

2. Теперь напишем класс-контейнер `MyNotebook`, в котором находятся источники `tf` и `b` события `ActionEvent`, и подключим к ним слушателя этого события `TextMove`, передав им ссылки на него методом `addActionListener()`, как показано ниже:

```
import java.awt.*;
import java.awt.event.*;
import java.swing.*;
class MyNotebook extends JFrame{
  MyNotebook(String title){
    Super(title);

    Container c = getContentPane();

    JtextField  tf = new JtextField("Вводите текст", 50);
    c.add(tf, BorderLayout.NORTH);
```

```
    JTextArea ta = new JtextArea();
    ta.setEditable(false);
    c.add(ta);

   JPanel p = new JPanel();
   c.add(p,  BorderLayout.SOUTH);

   JButton b = new Jbutton("Перенести");
   p.add(b);

   tf.addActionListener(new TextMove(tf, ta));
   b.addActionListener(new TextMove(tf, ta));

   setSize(300, 200);
   setVisible(true);
  }
  public static void main(String[] args){
    JFrame f = new MyNotebook("  Обработка ActionEvent");
    f.setDefaultCloseOperation(Jfame.EXIT_ON_CLOSE);
  }
}

// Текст класса TextMove
…..
```

Здесь в методах `addActionListener()`создаются два экземпляра класса `TextMove` – для прослушивания поля `tf` и для прослушивания кнопки `b`. Можно создать один экземпляр класса `TextMove`, он будет прослушивать оба компонента:

```
TextMove tm1 = new TextMove(tf, ta);
tf.addActionListener(tm1);
b.addActionListener(tm1);
```

3. Задание. Чтобы класс, содержащий источники события, сам обрабатывал события надо реализовать соответствующий интерфейс прямо в самом классе-контейнере. Составить такой класс и испытать на компьютере.

Литература

1. Дейтел Х. М., Дейтел П. Дж. Как программировать на Java Книга 1-2. СП. 2003
2. Ноутон П., Шилдт Г. Java 2. СП., 2004
3. Дж. Кьоу, М. Джеанини. Объектно-ориентированное программирование. П. 2005.
4. Хабибуллин И. Java 2. СП. 2005.
5. Флэнаган Д. Java в примерах. Справочник., СП. 2003
6. Эккель Б. Философия Java, СП. 2001.