УНИВЕРСИТЕТ ИТМО

А.А. Дубаков

ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА JAVA

Учебное пособие

Санкт-Петербург

2016

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
УНИВЕРСИТЕТ ИТМО

А.А. Дубаков

ВВЕДЕНИЕ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА JAVA

Учебное пособие

Санкт-Петербург

2016

Дубаков А.А. Введение в объектно-ориентированное программирование на Java: учебное пособие – СПб: Университет ИТМО, 2016. – 250 с.

Настоящее пособие имеет своей целью познакомить студентов с теоретическими аспектами объектно-ориентированного подхода программирования, а также обучить практическим навыкам реализации этого подхода на языке Java.

Пособие подготовлено на кафедре “Сетевых и облачных технологий” Университета ИТМО и предназначено для бакалавров по направлению 11.03.02 «Инфокоммуникационные технологии и системы связи».

Рекомендовано к печати Ученым советом факультета инфокоммуникационных технологий. Протокол № 09/16 от 24 ноября 2016 г.

УНИВЕРСИТЕТ ИТМО

Университет ИТМО – ведущий вуз России в области информационных и фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

Оглавление

Введение

В настоящее время Java является одним из самых популярных языков программирования для различных сфер применения, от разработки игр до создания критически важных приложений, используемых для торгов на различных биржах или управления беспилотными летающими устройствами. Пособие имеет своей целью познакомить студентов с теоретическими аспектами объектно-ориентированного подхода программирования, а также обучить практическим навыкам реализации этого подхода на языке Java.

Основы языка Java

Java-технология используется для разработки приложений, предназначенных для широкого спектра систем – от мобильных устройств до корпоративных систем. Язык Java был создан в 1995 году Джеймсом Гослингом (James Gosling) в корпорации Sun Microsystems (в настоящее время дочерняя компания корпорации Oracle) с целью получения упрощенной платформенно-независимой альтернативы языку C++. Сходство языков C, C++ и Java, прежде всего, проявляется в том, что блоки кода выделяются фигурными скобками { и }, а переменные должны быть объявлены перед их использованием. Кроме того, многие языковые конструкции Java заимствованы из C, например, операторы организации цикла, условные операторы, сравнения и многие другие. Java является объектно-ориентированным (ОО), что позволяет легко связать программные конструкции с объектами-сущностями предметной области в процессе разработки программных систем.

Язык Java специально разработан для качественного рывка в создании интерактивных распределенных приложений для сети Internet и основными достоинствами языка являются:

- наибольшая среди всех языков программирования степень переносимости программ между платформами (процессорами/операционными системами). Практически любой процессор позволяет использовать программы на Java, также как и практически все операционные системы поддерживают программирование на этом языке, включая Windows, Linux, MacOS, iOS и Android;
- мощные стандартные библиотеки (frameworks), предназначенные для использования при разработке программ;
- встроенная поддержка работы в сетях (как локальных, так и Internet/Intranet – сети с использованием TCP/IP).

К недостаткам платформы Java можно отнести:

- низкое, в сравнении с другими языками, быстродействие, повышенные требования к объему оперативной памяти. В последнее время утверждение становится не таким актуальным, поскольку эффективность языка Java увеличивается с появлением новых версий;
- большой объем стандартных библиотек и технологий создает сложности в освоении языка;
- постоянное развитие языка вызывает поддержку как устаревших, так и новейших средств, имеющих одно и то же функциональное назначение.

Здесь рассматривается общее представление о платформе Java и ее компонентах. В списке литературы приводятся ссылки на дополнительные источники информации о компонентах платформы Java, которые обсуждаются в этом разделе.

Компилятор Java

Существуют различные типы языков программирования. В некоторых из них программист пишет текст программы (исходный код) и может выполнить ее в исходном виде непосредственно. Это т.н. интерпретируемые языки (например, JavaScript, Python, Ruby). При программировании на платформе Java программист пишет исходный код в файлах с расширением .java, а затем компилирует его. Компилятор проверяет код на соблюдение правил синтаксиса языка, а затем записывает *байт-коды* (*bytecode*) в файлы с расширением .class. Байт-коды - это стандартные инструкции, предназначенные для работы на виртуальной машине Java (Java Virtual Machine – JVM). Кроме проверки программы на наличие синтаксических ошибок, компилятор Java добавляет некоторые другие библиотеки (связывает) к программе после завершения компиляции (этап развертывания). Использованием этого уровня абстракции компилятор Java отличается от компиляторов других языков, которые создают инструкции для процессора, на котором впоследствии будет выполняться программа.

Java Development Kit и Java Runtime Environment

Если вы планируете использовать определенный компьютер для разработки программ на Java, то необходимо загрузить и установить Java Development Kit (JDK). Загрузив JDK, вы получите – в дополнение к компилятору и другим инструментам – полную библиотеку классов готовых утилит, которые позволят решить практически любую общую задачу разработки приложений. Если вы планируете использовать этот

компьютер только для запуска программ Java, которые были скомпилированы в другом месте, вам просто нужно Java Runtime Environment (JRE). Если JDK установлен на вашем компьютере, он включает в себя JRE. Среда исполнения JRE включает в себя JVM, библиотеки кода и компоненты, необходимые для исполнения программ на языке Java. Независимость от платформы Java исходит от того, что Java программа не знает, под какую операционную систему (ОС) или на каком аппаратном обеспечении она выполняется. Она работает в предустановленным JRE, что означает возможность исполнения на всех платформах, для которых имеется JRE. JRE можно свободно распространять с собственными приложениями в соответствии с условиями лицензии, предоставляя пользователям платформу для работы с вашим ПО.

JVM

Во время выполнения кода JVM читает и интерпретирует файлы с расширением .class и выполняет команды программы на той аппаратной платформе, для которой написана JVM. JVM интерпретирует байт-коды так же, как процессор – инструкции на языке ассемблера. Разница в том, что JVM – это программа, написанная для конкретной платформы. JVM составляет основу принципа языка Java "написано однажды – работает везде" (write-once, run-anywhere). Ваш код будет работать на любом процессоре, для которого есть реализация JVM. Реализация JVM существует для всех основных платформ, таких как Linux и Windows, а подмножества языка Java реализованы в виртуальных машинах для мобильных телефонов и встраиваемых устройств.

Java SE и EE

Прежде, чем перейти к процессу загрузки, вам нужно ознакомиться с еще двумя понятиями: Java SE (Standard Edition) и Java EE (Enterprise Edition). Последнее содержит серверные инструменты и компоненты, которые будут рассматриваться в последующих дисциплинах. Однако уже сейчас можно обозначить разновидности приложений, которые можно разрабатывать на Java, в том числе:

- приложения десктопные и распределенные, в том числе для мобильных телефонов. Простые приложения с интерфейсом командной строки или графическим интерфейсом;
- апплеты (Applet) – приложения, выполняемые на браузере. Приложение на Java, которое хранится на сервере, загружается браузером и выполняется на клиентском компьютере на виртуальной машине, расширяющей возможности браузера;

- серверные программа Java - Servlet, Java Server Pages (JSP), Java Server Faces (JSF). Серверные программы, предназначенные для выполнения серверных вычислений и организации графического интерфейса Web-приложения;
- Enterprise Java Beans (EJB) – серверные компоненты архитектуры Java EE для реализации бизнес-логики;
- Web Services - идентифицируемая веб-адресом программная система со стандартизированными интерфейсами взаимодействия.
- Распределенные сервисы, использующие RMI (remote method invocation - удаленный вызов процедур).

Ниже представлены ряд программ, которые требуют компонентов, не входящих в Standard Edition JDK (доступны в Enterprise Edition):

- Servlets;
- Web Services;
- Enterprise Java Beans.

Кроме перечисленных, в состав JavaEE входит большое количество других технологий, которые способствуют эффективной разработке корпоративных распределенных инфокоммуникационных систем. Упомянутые технологии требуют специального рассмотрения и не являются предметом настоящего пособия.

Настройка среды разработки Java

В этом разделе содержатся инструкции по загрузке и установке JDK 8 и текущей версии Eclipse IDE, а также по настройке среды разработки Eclipse.

Как мы уже упоминали, JDK содержит набор инструментов командной строки для компиляции и запуска Java-кода, включая полную копию JRE. Хотя эти инструменты, конечно, можно использовать для разработки приложений, большинство программистов выбирают дополнительные возможности, средства управления задачами и визуальный интерфейс Integration Development Environment - IDE, интегрированной среды разработки приложений. Существует достаточно большой набор IDE, поддерживающий разработку приложений на языке Java: JBuilder, Symantec Cafe, NetBeans IDE, Java Studio Creator, Java Studio Enterprise, IntelliJ IDEA, VisualJ, Eclipse) и др. В настоящем пособии для Java-разработки применяется Eclipse – популярная IDE с открытым исходным кодом. Она выполняет основные задачи, такие как компиляция кода и настройка среды отладки, позволяя сосредоточиться на написании и тестировании программ. Кроме того, Eclipse можно использовать для

организации файлов исходного кода в проекты, компиляции и тестирования этих проектов, а также для хранения файлов проектов в любом количестве хранилищ исходного кода. Чтобы использовать Eclipse для Java-разработки, необходимо, прежде всего, установить JDK.

Установка JDK 8

Для загрузки и установки JDK 8, необходимо выполнить следующие действия.

Зайдите на сайт загрузок Java SE (http://java.sun.com/javase/downloads/index.jsp) и нажмите кнопку **JDK download**, чтобы вызвать страницу загрузки последней версии JDK, как показано на Рис. 1 (на момент написания этого пособия – JDK 8, обновление 31).

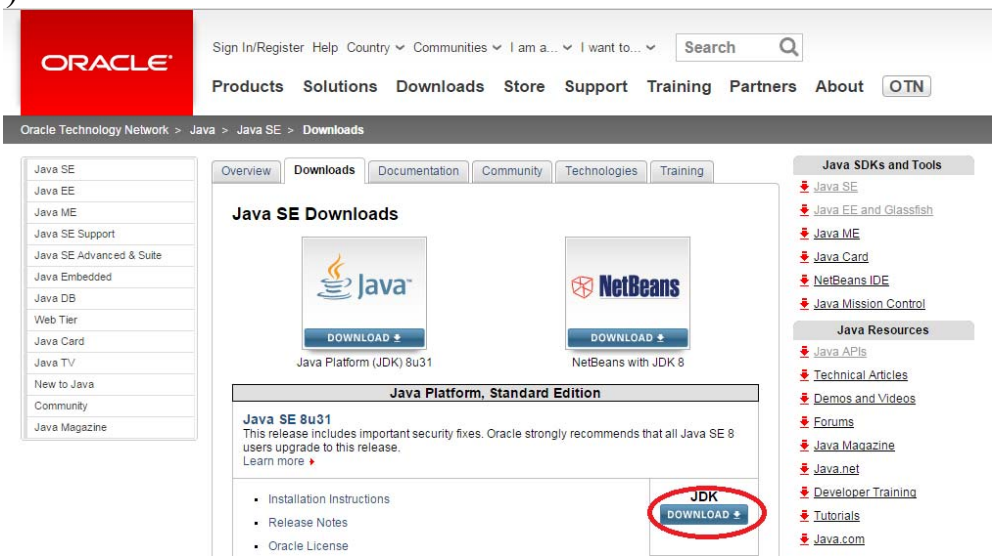


Рис. 1 Страница загрузки JDK

Выберите нужную платформу операционной системы и установите флажок Accept License Agreement. Вам может быть предложено ввести имя пользователя и пароль, введите затребованное, если у вас есть учетная запись, зарегистрируйтесь, если ее нет, или нажмите кнопку **Continue**, чтобы пропустить этот шаг и перейти к загрузке. Возможно, такого диалога и не последует, в любом случае сохраните файл на жестком диске.

Когда загрузка будет завершена, запустите программу установки - только что загруженный файл представляет собой самораспаковывающийся архив ZIP, который является программой установки. Следуйте запросам программы установки и установите JDK на жесткий диск в удобном, запоминающемся месте (например, C:\Program Files\Java\jdk1.8.0_31 или C:\home\jdk1.8.0_31 в системе Windows, или

~/jdk1.8.0_31 в системе Linux). В имени каталога установки следует указать номер версии обновления. По окончании выполнения программы установки появится диалоговое окно успешной установки JDK на вашем компьютере, в котором нажмите кнопку Close, как показано на Рис. 2.

Рис. 2. Диалоговое окно завершения установки JDK.

Установка переменной среды PATH ОС Windows

Для того чтобы использовать JDK для разработки программ на Java из любого подкаталога, необходимо установить переменную среды PATH.

Для установки переменной среды PATH выполнить следующие действия:

1. Нажать кнопку Пуск, в меню выбрать Компьютер и нажать правую кнопку. Из контекстного меню выбрать "Свойства".

2. В левой панели открытого окна "Просмотр основных сведений о вашем компьютере" выбрать "Дополнительные параметры системы".

3. В открытом диалоговом окне "Свойства системы" нажать кнопку "Переменные среды".

4. В нижней таблице "Системные переменные" прокрутить строки таблицы, выделить переменную Path и нажать кнопку Изменить...

5. В диалоговом окне в начало значения поля "Значение переменной" вставить местоположение исполняемых файлов Javac.exe Java.exe, обычно по умолчанию установки JDK в **C:\Program Files\Java\jdk1.8.0_31\bin;** (Запятая в конце обязательна как разделитель. Пробелы в конце строки не допустимы). Не вздумайте переписать ранее установленное значение Path - возникнут серьезные проблемы с другими приложениями.

6. Нажать последовательно кнопку ОК во всех открытых всплывающих окнах.

Для проверки правильности установки значения переменной PATH выполните следующие действия:

1. Нажать кнопку Пуск и в поле "Найти программы и файлы" (в нижней части меню) ввести **cmd** и нажать Enter. В результате откроется командное окно с текущей установкой в вашем Home-каталоге.

2. С клавиатуры в командную строку ввести **javac** <Enter>.

Если будут выведены сообщения из команды javac, то установка переменной среды PATH выполнена успешно, в противном случае установочные данные введены неверно, и следует повторить этапы установки переменной PATH.

Можно проверить версию установленной JVM, если ввести команду **java -version** и нажать <Enter>, как показано на Рис. 3.

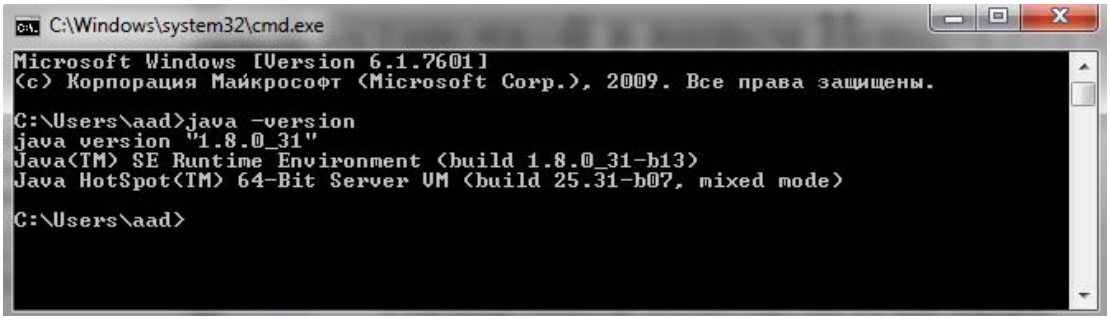


Рис. 3. Проверка версии JVM

Разработка первой программы на Java

Исторически сложилось так, что первая программа, которую Вы пишете в процессе изучения нового языка программирования, является программа HelloWorld, которая выводит сообщение Hello, World.

Чтобы написать программу на Java можно использовать любой текстовый редактор - пусть это будет Notepad. Файл, который содержит Java-код, обязательно должен быть сохранен с именем HelloWord (совпадающим с названием класса) и расширением .java.

Введите следующий код в текстовом редакторе:

```
public class HelloWorld {
     public static void main(String[] args){
          System.out.println("Hello World");
     /*комментарий-вывод привет */
     } // Комментарий - конец метода main
}
```

В командном окне, в home-каталоге создайте каталог **Practice** (используя команду **md Practice** <Enter>) и сохраните программу, которую вы только что создали в файле `HelloWorld.java` (если вы используете Notepad, выберите "все файлы" в списке "Тип файла" из выпадающего списка, чтобы избежать автоматического присоединения расширения `.txt`), как показано на Рис. 4.

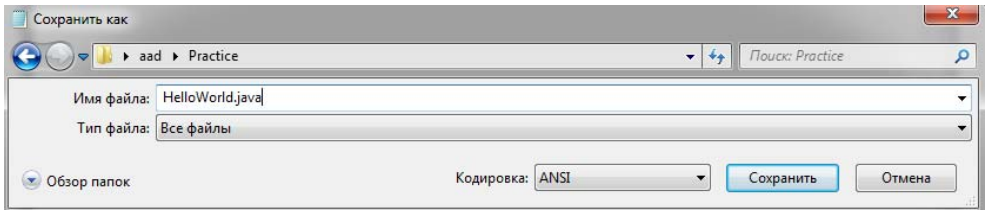


Рис. 4. Сохранение программы в Notepad

Следует помнить, что Java является языком программирования, чувствительным к регистру, в том смысле, что если вы присвоили программе имя `HelloWorld` с большой буквой H и буквой W, то не следует пытаться начать программу helloworld, это однозначно приведет к ошибке. Ваши первые синтаксические ошибки будут вызваны неправильной капитализацией.

Ниже следует очень краткое объяснение некоторых терминов и языковых элементов, используемых в программе HelloWorld. В дальнейшем вы будете чувствовать себя более уверенно с этими понятиями после освоения первых нескольких разделов этого пособия.

Наша первая программа содержит утверждение `class HelloWorld`. Имени файла Java в процессе сохранения в файловой системе должно присваиваться такое же имя, как и классу. (Могут быть и исключения из этого правила, но не в этой простой программе). Во время написания программы Java создаются классы, которые часто представляют объекты из реальной жизни. Об этом нам предстоит узнать больше несколько позже.

Класс `HelloWorld` содержит метод `main()`. Методы классов Java представляют собой функции (действия), которые класс может выполнять. Каждый метод класса имеет определенную сигнатуру, которая определяется именем метода, возвращаемым значением (возвращаемый тип, как у функции), типами формальных аргументов метода и другими формальными описателями в объявлении метода. Класс Java может иметь несколько методов, но если один из них объявляется с именем `main()` и определяется (имеет сигнатуру метода, определяющую аргументы метода), как в нашем классе, то этот Java класс становится выполняемым, то есть мы можем указать JVM выполнить данный класс как программу. Если класс не содержит метода `main()`, он может быть использован с другими классами, но вы не сможете запустить его как программу. JVM всегда ищет метод `main()` в классе и передает ему управление, выполняя операции, закодированные в этом методе.

В методе `main()` вызывается метод `println()` класса `System.out`, для отображения текста "`Hello World`" на экране. Класс `out`, в свою очередь, входит в класс `System`.

Ниже представлена сигнатура метода `main()` :

```
public static void main(String[] args).
```

Эта сигнатура метода определяет уровень доступа (`public`), вариант использования (`static`), тип возвращаемого значения (`void`), название метода (`main()`) и список аргументов (`String [] args`).

- Ключевое слово `public` означает, что метод `main()` может быть доступен любому другому классу Java.
- Ключевое слово `static` означает, что вы не должны создавать экземпляр данного класса (объект) при использовании этого метода.
- Ключевое слово `void` означает, что метод `main()` не возвращает никакого значения вызывающей программе.
- Описание (`String [] args`) определяет, что этот метод будет получать массив символьных строк, в качестве аргументов (вы можете передать внешние параметры этому методу из командной строки) для использования в методах класса.

Метод `main()` является точкой входа (entry point) программы. Можно написать программу на языке Java, которая состоит из более чем одного класса, но, по крайней мере, один из них должен содержать метод с именем `main()`, в противном случае, программа не будет выполняться JVM. Класс Java может иметь более одного метода. Например, класс `Employee` может иметь методы `updateAddress()`, `raiseSalary()`, `changeName()` и т.д.

Тело метода `main()` содержит единственное утверждение:

```
System.out.println ("Hello World");
```

Метод `println("Hello World")` класса `System.out` используется для вывода данных на системной консоли (командное окно). После кодирования имени метода в Java всегда следуют круглые скобки даже если в этих скобках ничего не определяется. Точка с запятой ";" всегда ставится в конце утверждения языка и отделяет один оператор от другого.

`System` представляет здесь еще один класс Java, а нотация с точкой `System.out` означает, что переменная класса `out` определена внутри класса `System`. Выражение `out.println("Hello World")` определяет, что существует класс с именем `out`, и в этом классе содержится метод называемый `println()`.

В дальнейшем эта т.н. точечная нотация используется для ссылки на методы или переменные классов и их объектов.

Краткое введение в массивы

Естественно, в пособии будут более подробно рассмотрены использованные элементы определения класса, но по ходу изложения материала будут предварительно вводиться некоторые языковые возможности Java, основываясь на знаниях, полученных при изучении других систем программирования. Такое ускоренное введение элементов языка позволит применять алгоритмические возможности Java безотносительно его объектно-ориентированной направленности и раньше перейти к рассмотрению более интересных примеров.

В частности, уже был упомянут массив типа `String[]` в классе `HelloWorld`. В языке Java, как и других языках программирования, массив данных (Array) используется для хранения нескольких значений одного и того же типа. Как видно из описания массива в качестве аргумента метода, он объявляется с помощью применения квадратных скобок "[]". Массив может иметь определенный при его создании размер. К элементам массива обращаются с указанием индекса, и начальный элемент массива имеет значение индекса 0. Так, например, `args[0]` и `args[1]` содержат первый и второй параметр, если они будут переданы программе из командной строки. Тип `Array` является объектным (не примитивным) типом, создается в классе фиксированного размера и для его определения в классе можно использовать конструкцию с ключевым словом `new`, например, `String [] friends = new String [20];`

Для присвоения значений элементам массива можно многократно использовать операцию присвоения типа:

```
friends[0]="Sergey";// Инициализируется первый элемент
friends[1]="Tamara";// Инициализируется второй элемент
friends[2]="Alexey";
// ...
friends[18]="Masha";
friends[19]="Natasha";
```

Если известны все значения, которые будут храниться в массиве во время его объявления, вы можете объявить и инициализировать массив в одно и то же время. Следующая строка объявляет объект класса `String` и заполняет его значениями из четырех элементов:

```
String [] friends = {"Sergey"; "Tamara"; Masha", "Alexey", "Rosa"};
```

Если каким-либо элементам массива не присвоено значение, то значение этих элементов равно значению `null`.

Для переменных объектного типа `Array` определены свойства (переменные класса) и методы (функции), которые поддерживают опе-

рации с ними. В частности, имеется свойство length, которое содержит число элементов в массиве. Следующая строка показывает, как можно получить это значение для ранее описанной переменной friends:

```
int totalElements = friends.length;
```

В этом утверждении описывается целочисленная переменная totalElements и ей присваивается число, равное размерности массива, определяемого значением свойства friends.length. Обратите внимание, что здесь также используется точечная нотация при обращении к свойству объекта.

Как известно, при работе с массивами используются операции в цикле. Циклы for используются для повторения одних и тех же действий несколько раз, когда известно заранее количество повторений.

Давайте выведем имена из массива friends.

```
int totalElements = friends.length;
for (int i=0; i < totalElements;i++){
     System.out.println("Мой друг " + friends[i]);
}
```

Предыдущий код интерпретируется следующим образом: "Вывести значение элемента i из массива friends, начиная от i = 0, и увеличивая i на удиницу (i++), пока не будет достигнута величина, равная значению totalElements". Здесь фигурные скобки используются для определения блока операций, которые выполняются в цикле. В данном случае выполняется одна операция, и скобки можно было бы опустить.

Компиляция и запуск первой программы Java

Теперь необходимо скомпилировать эту программу с помощью компилятор javac.exe, который является частью JDK. Откройте командное окно на вашем компьютере, измените текущий каталог на C:\Users\<home>\Practice и попытайтесь откомпилировать программу:

```
javac HelloWorld.java
```

В случае если на компьютере в переменной окружения Path, в числе прочих, не указан путь к подкаталогу, где располагается программа javac.exe, может выведено сообщение об ошибке. Для обеспечения возможности компиляции файла из любого подкаталога файловой системы необходимо модифицировать переменную окружения Path добавлением пути к подкаталогу с наличием компилятора

`javac.exe` (Обычно C:\Program Files\Java\jdk1.8.0_31\bin), закрыть, и вновь открыть командное окно, и повторить компиляцию.

Вы сможете увидеть подтверждение успешной компиляции в случае отсутствия ошибок, введя команду `dir`, которая покажет, что создан новый файл с именем `HelloWorld.class`, Для запуска программы введите команду `java HelloWorld`.

Обратите внимание, что на этот раз мы использовали команду `java`, которая выполняет виртуальную машину Java (JVM), выполняющую класс `HelloWorld`, и приветствие `"Hello World"` отображаются в командном окне. Имя файл `HelloWorld.class` в командной строке должно быть указано без расширения. На Рис. 5 представлен скриншот, который показывает создание подкаталога `Practice`, установку текущего подкаталога, команду компиляции, вывод содержимого папки `Practice` после компиляции и вывод программы `HelloWorld`.

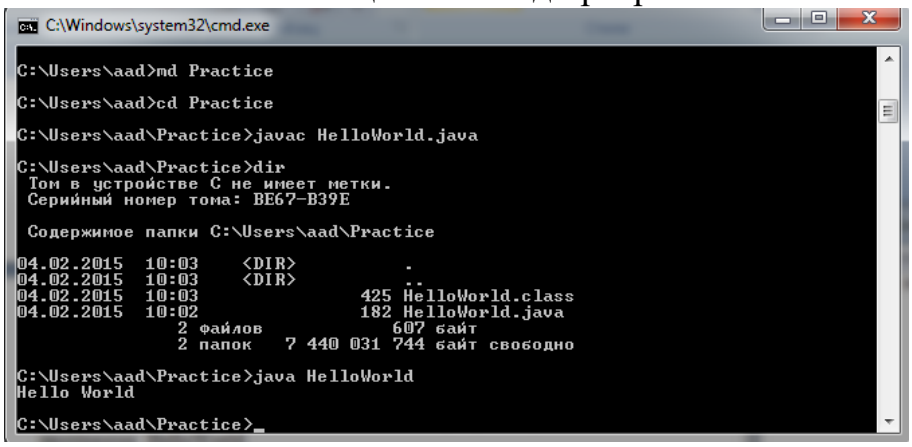


Рис. 5. Этапы разработки программы `HelloWorld`.

Последовательность шагов разработки программы из командной строки следующая:

1. Откройте текстовый редактор по вашему выбору и введите текст класса `HelloWorld`.
2. Сохраните программу в файле `HelloWorld.java`.
3. Скомпилируйте программу в командном окне, используя команду `javac HelloWorld.java`.
4. Запустите программу с помощью команды `java HelloWorld`.

Теперь измените класс `HelloWorld` следующим образом:

```
public class HelloWorld {
     public static void main(String[] args) {
     if (args.length==0)
          System.out.println("Hello, Noname");
     else
```

```
            System.out.println("Hello, "+args[0]);
        }
    }
```

Для модифицированного класса предусмотрена возможность задать в параметре командной строки выполнения программы имя приветствуемого, например, `Anatoly`. В коде класса с помощью оператора `if` проверяется наличие параметра в командной строке, и, в случае отсутствия параметра (Массив `args` - не содержит элементов если длина массива равнв `0` - `args.lenght=0`), выводится сообщение `Hello, Noname.` В противном случае, выводится приветствие `Hello,Anatoly.` Оператор условного выполнения также имеется во всех языках и не требует особых комментариев. Что можно было бы отметить, так это то, что условие обязательно должно заключаться в круглые скобки, и для формирования условия используются операция отношения, результатом которых является логическая (`boolean`) величина, принимающая значение истина (`true`) или ложь (`false`). Операция отношения может использовать следующие операторы отношений (сравнения): >, <, >=, <=, != , ==. Попутно пока можно отметить, что группа операторов с `else`, не является обязательной.

Итак, необходимо вновь скомпилировать класс `HelloWorld` с помощью компилятора `javac.exe` и выполнить его командой `java HelloWorld Anatoly`. Результат выполнения класса представлен на Рис. 6.

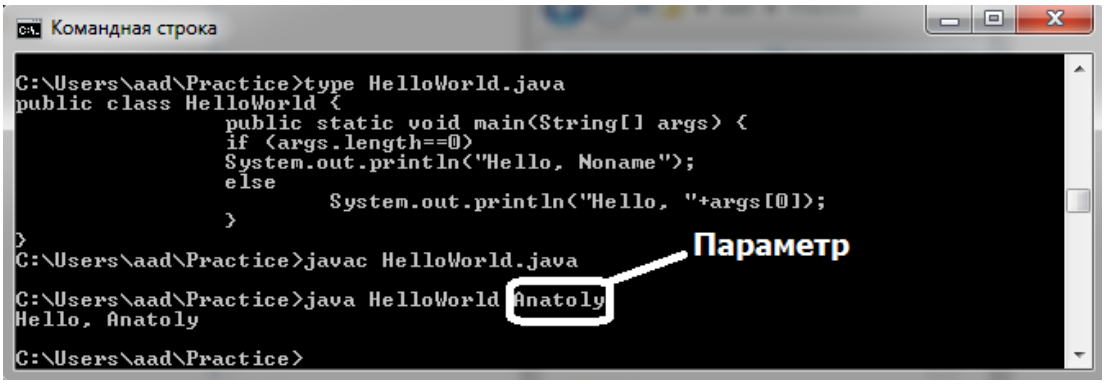


Рис. 6. Результат выполнения класса `HelloWorld` с параметром

Как правило, приложение состоит из нескольких классов и других компонентов, которые определяются в нескольких Java-файлах исходного кода. Из всех этих файлов, программист обозначает один из классов в виде исполняемого класса (содержащего метод `main()`). Программист может определить действия, которые JVM должна выполнить при запуске приложения. Например, можно определить исполняемый класс Java, который содержит код для отображения соответствующего

диалогового окна графического интерфейса пользователя (`displayGUI`), и выполнить соединение с базой данных (`openDatabaseConnection`).

На Рис. 7, классы `Window`, `UserData`, `ServerConnection` и `UserPreferences` не содержат метод `main()`, а класс `LaunchApplication` содержит метод `main()` и является исполняемым классом.

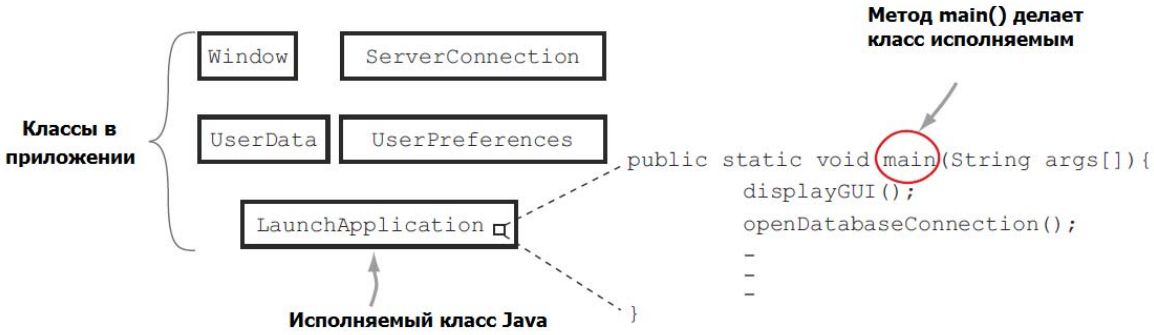


Рис. 7. Структура классов, содержащая исполняемый класс `LaunchApplication`

Eclipse IDE - среда разработки приложение Java

Написание программы, состоящей из большого количества файлов на Java, в текстовом редакторе, и компиляция из командного окна не являются продуктивным способом разработки программного обеспечения. Профессиональные программисты используют одну из интегрированных сред разработки (Integration Development Environment - IDE), которые включают в себя редактор, компилятор, опережающую помощь, отладчик и многое другое (мы рассмотрим ряд этих функций). Существует несколько популярных IDE для Java, такие как Eclipse, NetBeans, IntelliJ IDEA, и другие. Некоторые из них являются бесплатными, а некоторые - коммерческими.

Eclipse на сегодняшний день является наиболее широко используемой IDE, и будет использован для компиляции и выполнения большинства из примеров в этом пособии. При этом переход от одного IDE на другой довольно простой процесс, и если вы обнаруживаете, что в некоторых областях один IDE обеспечивает более высокую продуктивность, чем другие, просто используйте его для этой работы.

IDE Eclipse является продуктом с открытым исходным кодом, который изначально был создан на основе кода, пожертвованного IBM для сообщества Java, и с этого момента Eclipse на 100% разрабатывается сообществом. Первоначально IDE использовался для разработки программ на Java, но сегодня Eclipse является платформой, используемой

для создания тысяч инструментов и плагинов. Некоторые разработчики используют его Rich Client Platform (RCP) API для разработки пользовательских интерфейсов (UI) для приложений. Другие языки программирования, такие как C, PHP также поддерживаются Eclipse. Посетите страницу загрузки www.eclipse.org/downloads, чтобы увидеть только часть продуктов на основе Eclipse.

Помимо того, что Eclipse является IDE, он поддерживает разработку плагинов (plug-in) - функциональных компонентов, и каждый разработчик может добавить только те плагины, которые необходимы для повседневной работы. Так, например, существуют плагины для разработки UML диаграмм, другие предлагает систему отчетности, также имеются плагины для разработки приложений на языках C, Adobe Flex и других системах программирования.

Загрузка и установка числе Eclipse

Каждая версия Eclipse имеет имя собственное, и в настоящий момент текущей версией является Neon (2016 - v 4.6.1). С начала выпуска использовались следующие версии и имена Eclipse: Europa (2007 - v 3.3), Ganymede (2008 v 3.4), Galileo (2009 - v 3.5), Helios (2010 - v 3.6), Indigo (2011 - v 3.7), Juno (2012 - v 4.2), Kepler (2013 - v 4.3), Luna Packages (2014 - v 4.4.0), MARS (2015- v4.5). В каждой версии существует несколько вариантов конфигураций, но в пособии используется Eclipse IDE for Java EE Developers. По ссылке www.eclipse.org/downloads выберите соответствующую платформу и разрядность установленной на вашем компьютере ОС Windows и перейдите на страницу загрузки. Здесь выберите зеркало, и загрузите файл с именем, аналогичным eclipse-jee-neon-RC2-win32-x86_64 на жесткий диске вашего компьютера. Распакуйте содержимое архива в подкаталог, с легко запоминающимся именем (например, C:\Eclipse или C:\<home>\Eclipse в системе Windows).

Настройка среды Eclipse

Eclipse IDE в своей работе использует JDK, и прежде чем среду Eclipse можно будет использовать для Java-программирования, следует указать, где находится JDK. Как правило, по умолчанию Eclipse устанавливает использование JRE, но для ряда библиотек необходимо иметь доступ к полному набору JDK.

Запустите Eclipse, дважды щелкнув на eclipse.exe (или эквивалентном исполняемом файле для вашей платформы).

Появится диалоговое окно Workspace Launcher, предлагающее указать папку для хранения проектов Eclipse. Выберите папку с легко

запоминающимся именем, например, C:\home\workspace в системе Windows или ~/workspace в системе Linux и нажмите Ok, как показано на Рис. 8.

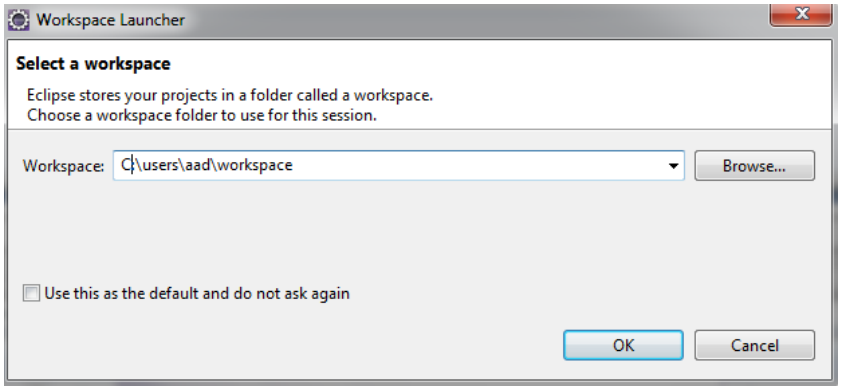


Рис. 8. Выбор места хранения проектов Eclipse

Закройте экран Welcome Eclipse, позже его можно будет открыть из главного меню **Help>Welcome**. На экране будет отображено рабочее пространство (Workbench) IDE Eclipse с пустым набором проектов в закладке Project Explorer, как показано на Рис. 9.

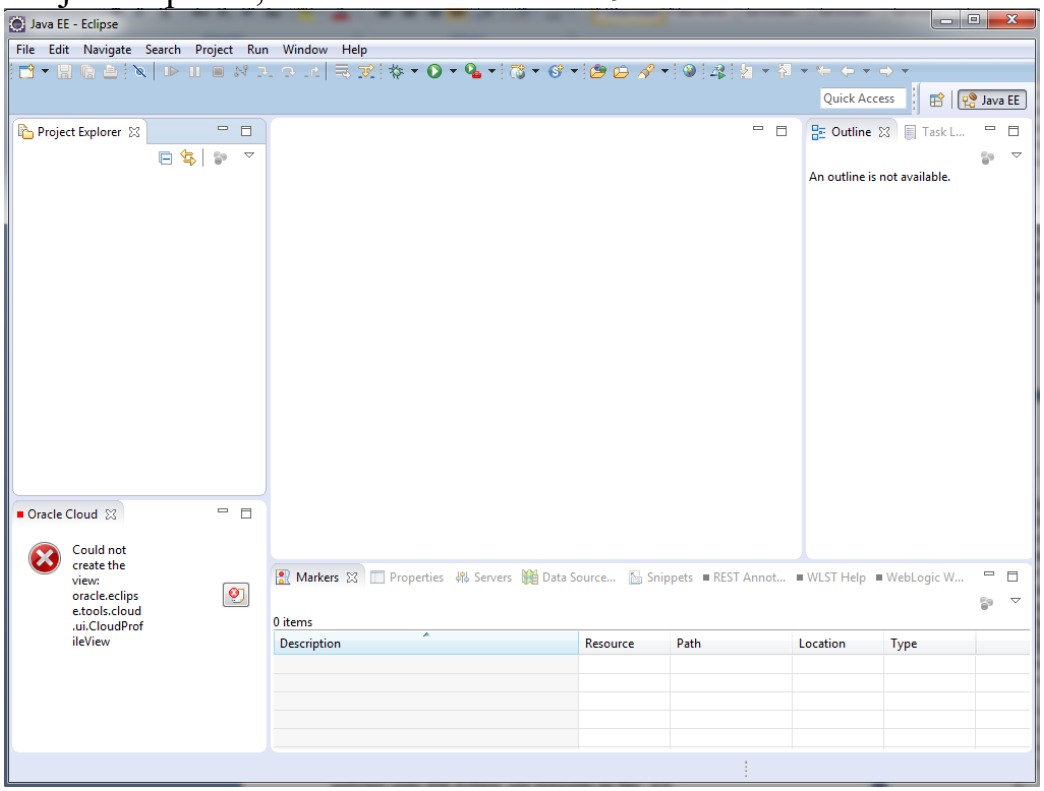


Рис. 9. Рабочее пространство IDE Eclipse

Из главного меню последовательно выберите **Window> Preferences>Java > Installed JREs**. На Рис. 10 показано диалоговое окно настройки **Installed JRE**.

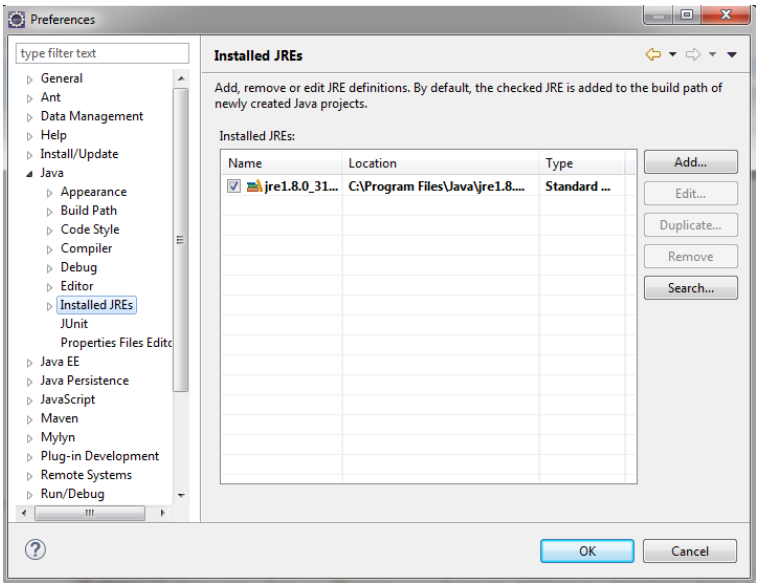


Рис. 10. Настройка JDK Eclipse

Eclipse будет указывать на установленную среду JRE. Будем использовать версию, которую вы загрузили вместе с JDK 8. Если Eclipse не может автоматически обнаружить установленный JDK, нажмите кнопку **Add...**, и в следующем диалоговом окне **JRE Type** (здесь не показано) выберите **Standard VM**, а затем нажмите кнопку **Next**.

В диалоговом окне **JRE Definition** укажите **JRE home** каталог JDK (например, C:\Program Files\Java\jdk1.8.0_31 в Windows), затем нажмите кнопку **Finish** как показано на Рис. 11.

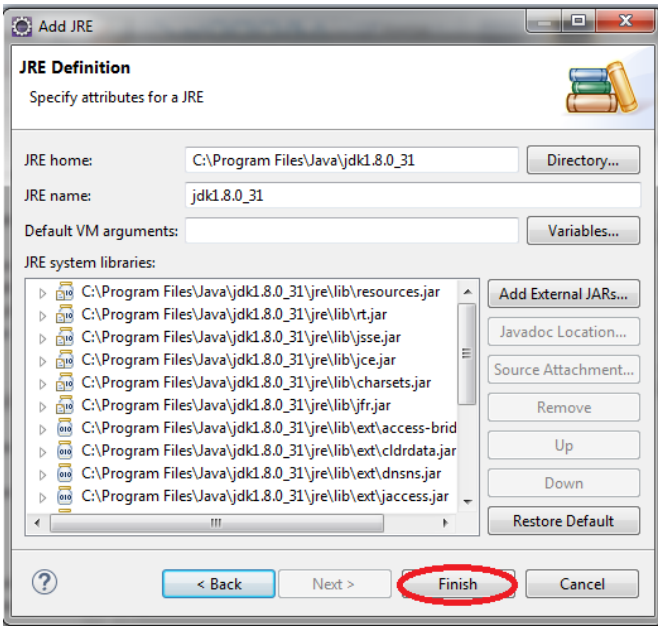


Рис. 11. Установки в диалоговом окне JRE Definition

После возвращения в диалоговое окно **Installed JRE** подтвердите установкой соответствующего флажка, что выбран только что обозначенный JDK, и нажмите кнопку **Ok** . Теперь среда Eclipse установлена и

готова к созданию проектов, компиляции и исполнению Java-кода. Следующий раздел знакомит со средой Eclipse.

Введение в Eclipse

Eclipse — это не просто IDE, это целая технологическая система разработки приложений. Данный раздел представляет собой краткое практическое введение в Eclipse для целей Java-разработки. Более подробные сведения об Eclipse можно получить по ссылкам в литературе.

Среда разработки Eclipse состоит из четырех основных компонентов:

- перспективы (perspectives);
- представления (views);
- редакторы (editors).

Перспектива – набор представлений и редакторов в рабочем окне (Workbench). Количество перспектив в окне Workbench не ограничено, т.е. имеется возможность сформировать перспективу, состоящую из любого произвольного набора представлений (диалоговых окон с данными) и редакторов. В одном окне Workbench каждая из перспектив может иметь различные наборы представлений, но все перспективы совместно используют один и тот же набор редакторов. Каждая перспектива состоит из представлений, соответствующих функциональному назначению перспективы. Так, перспектива Java состоит из представлений, связанных с разработкой проектов на Java.

Текущая перспектива обозначена в правом верхнем углу Workbench и в данный момент текущим является представление JavaEE (Рис. 12). Для переключения между перспективами используется команда основного меню **Window>Open Perspective**, которая выводит возможные в данной конфигурации Eclipse перпективы, как показано на Рис. 12.

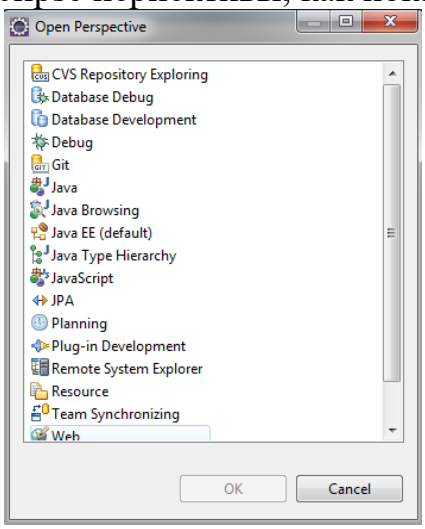
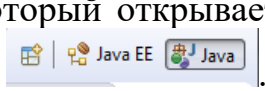


Рис. 12. Перечень возможных перспектив Eclipse

Например, для того, чтобы перейти в нужную в настоящий момент перспективу Java, просто можно выбрать из списка, представленного на Рис. 12, данную перспективу и нажать кнопку **Ok**. При этом, естественно, изменится внешний вид Workbench, определяемый составом и расположением представлений, поддерживающих операции в данном представлении.

Перечень ранее использованных перспектив доступен для оперативного переключения в правой верхней части экрана нажатием соответствующих кнопок панели, а другие перспективы также доступны для переключения с помощью списка, который открывается при нажатии на иконку  , расположенную рядом 

.

Представление – это визуальный компонент Workbench. Обычно он используется для перемещения по иерархии некоторой информации (например, структуры проекта), открытия редакторов, просмотра результатов выполнения и т.д. Все изменения, сделанные в представлениях, немедленно сохраняются. В окне Workbench может существовать только один экземпляр представления некоторого типа.

Многие представления входят в состав тех или иных перспектив и отображаются при их открытии. Тем не менее, с помощью команды меню **Window>Show View** можно открыть любое представление, независимо от текущей перспективы.

Восстановить набор представлений текущей перспективы можно с помощью команды главного меню **Window>Reset Perspective**.

Редактор также является визуальным компонентом Workbench. Используется для просмотра и редактирования некоторого ресурса, например Java-класса или файла, содержащего SQL-запросы. Изменения, делающиеся в редакторе, подчиняются модели жизненного цикла «открыть-сохранить-закрыть» (an open-save-close lifecycle model). В Workbench может существовать несколько экземпляров редакторов.

Редакторы того или иного типа обычно отображаются автоматически при двойном щелчке на соответствующий ресурс (например, Java-класс). Сохранение данных выполняется с помощью команды меню **File>Save**, либо нажатием на клавиши **Ctrl-S**.

В каждый момент времени может быть активным только одно преставление или редактор. Активным является представление или редактор, у которого подсвечен заголовок. На активный элемент будут воздействовать общие операции вырезки, копирования и вставки (cut, copy, paste). Также активный элемент обуславливает содержимое строки состояния (status line). Если закладка редактора белая, то это означает, что данный редактор неактивен, однако представления могут отобра-

жать информацию, полученную из редактора, бывшего активным последним.

Перспектива Java

На Рис. 13 показана перспектива Java, которая определяется по умолчанию, если установлен Eclipse for Java developes и она открывается, как только начинается работа в Eclipse.

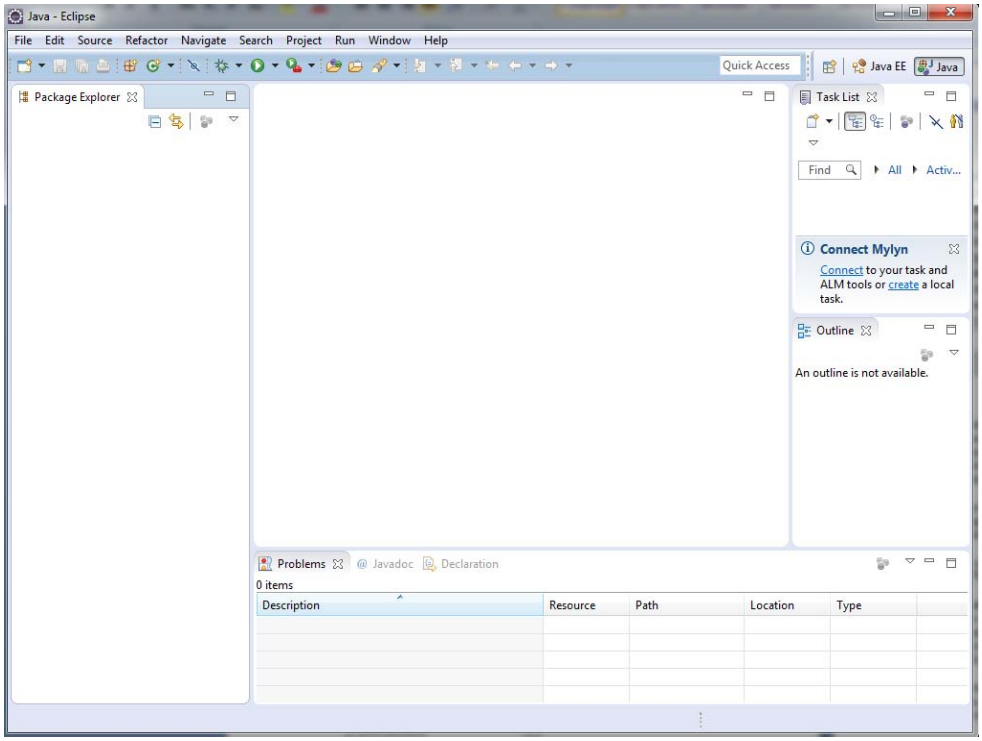


Рис.13. Java перспектива Eclipse

Перспектива Java содержит все необходимые инструменты, чтобы начать разрабатывать Java-приложения. Каждая вкладка, показанная на Рис. 13, это представление перспективы Java. Два особенно полезных представленя – это Package Explorer и Outline.

Среда Eclipse легко настраивается. Каждое представление можно перемещать по всей рабочей области Java и помещать в любое удобное место. Пока же целесообразно придерживаться перспективы и настройки представлений по умолчанию.

Создание проекта Hello в Eclipse

Ранее вы просто создали класс HelloWorld, но в Eclipse необходимо в первую очередь создать проект.

Для создания нового проекта Java, выберите из главного меню **File > New > Java Project** ..., и откроется диалоговое окно, в которое

введите название проекта **Hello,** и нажмите кнопку **Finish**, как показано на Рис. 14.

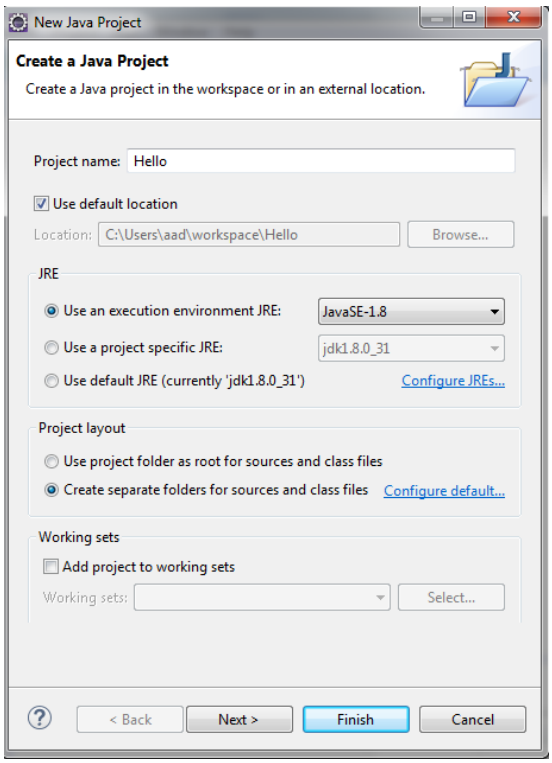


Рис.14. Мастер создания нового проекта Java

Если нужно изменить настройки проекта по умолчанию, можно нажать **Next**. (Это рекомендуется делать *только* при наличии опыта работы с Eclipse IDE).

В результате создается новый проект с именем Hello, который можно наблюдать в представлении Package Explorer, как показано на Рис. 15.

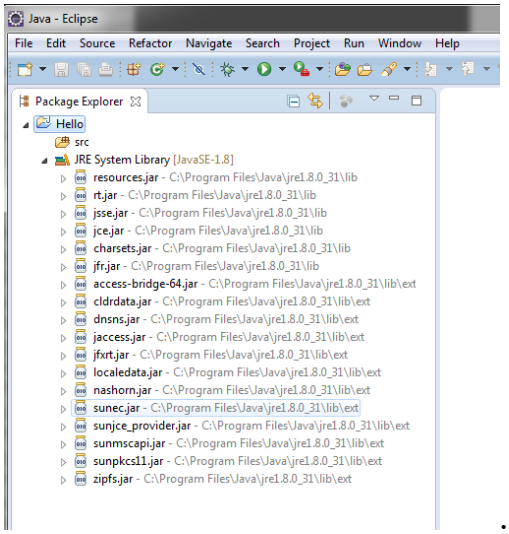


Рис. 15. Новый проект Hello в представлении Package Explorer

Этот проект содержит пустую папку, src - в которой позже будет сохранен исходный код `HelloWorld.java`, когда он будет подготовлен. Папка JRE содержит все необходимые библиотеки, поддерживающие работу JVM, в которой будет выполняться HelloWorld.

Эти библиотеки содержат файлы с расширением `.jar` в названиях. Java SDK поставляется с утилитой `jar.exe`, которая позволяет создавать архивные файлы, содержащие один или более скомпилированных классов.

В нашем проекте планируется создание одного класса с именем `HelloWorld`. Для создания класса нажмите курсором на папку `src`, и из главного меню выберите **File>New>Class.** В результате откроется диалоговое окно `New Java Class`, в котором в поле `Name` введите `HelloWorld`, в поле `Package` введите `ru.ifmo.practice`, также отметьте флажок для генерации метода `main()`, как показано на Рис. 16.

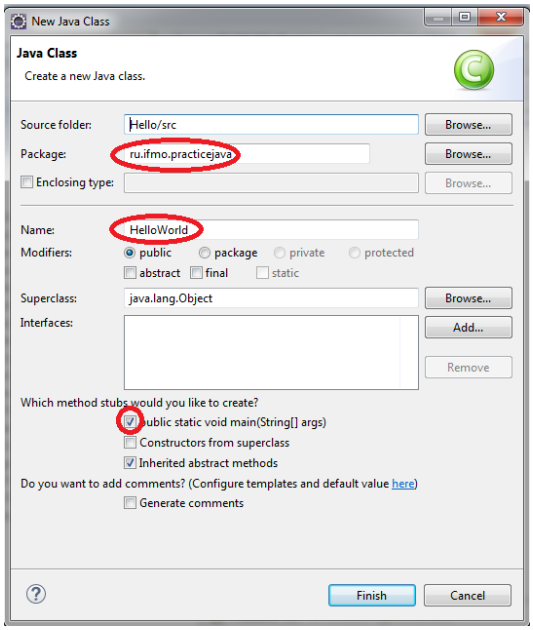


Рис. 16. Диалоговое окно определения класса

Такого понятия как Package мы не рассматривали ранее и обсудим его несколько позже. Далее в диалоговом окне и нажмите кнопку **`Finish`**, и через несколько секунд Eclipse сгенерирует для вас шаблон исходного код для класса HelloWorld, как показано на Рис. 17.

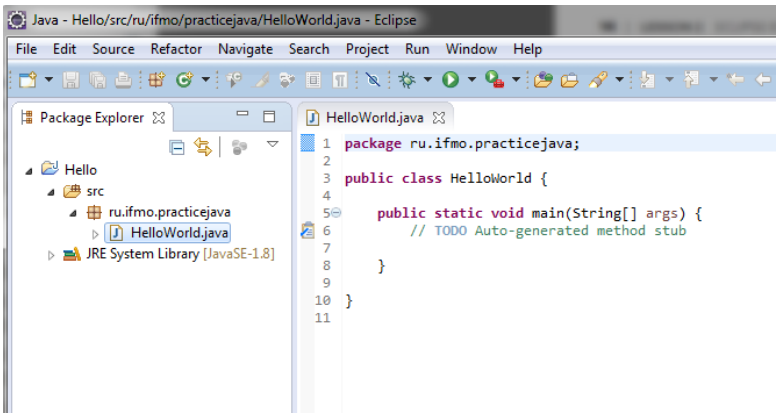


Рис. 17. Исходный код класса HelloWorld

Прежде чем приступить к написанию кода программы HelloWorld, рассмотрим введенное понятие пакетов (Package). Пакеты в Java используются, чтобы лучше организовать проекты, состоящие из большого количества файлов и для защиты их данных. Реальные проекты обычно содержать несколько сотен и тысяч классов Java, и в работе над проектом участвуют большие коллективы разработчиков. Сохранение файлов в одном каталоге не является продуктивной идеей. Следовательно, файлы целесообразно располагать в различных каталогах и подкаталогах. Существуют определенное соглашение об именовании пакетов, которое используют обратные доменные имена. Допустим, вы работаете над проектом под названием `Session` в организации Университет ИТМО, которая имеет сайт в сети Интернет на `ifmo.ru`. Тогда каждое имя пакета начнется в обратной последовательности URL организации, за которым следует имя проекта: `ru.ifmo.session`. Все Java классы, которые принадлежат к этому пакету, будут располагаться в следующей файловой структуре: `ru/ifmo/session`.

В то время как имена каталогов разделяются слешем или обратным слешем, соответствующие Java пакеты разделены точками. Для объявления пакета Java содержит специальное ключевое слово `package`, и объявление пакета должно быть первой строкой класса (комментарии в программе не в счет). Например:

```
package ru.ifmo.session;
```

Теперь предположим, что вы обучаетесь в университете ИТМО, имеющим домен `ifmo.ru` и разрабатываете классы для освоения Java в пакете с именем `practicejava`; тогда имя пакета будет `ru.ifmo.practicejava`, именно то, что определено в утверждении `package` и показано на Рис. 17. По этой же схеме классы, которые образуют какую-либо другую общность, например, предназначены для обеспечения взаимодействия с пользователем или для работы с файла-

ми, могут соответственно располагаться, например, в пакетах `ru.ifmo.interface` и `ru.ifmo.file`.

Помимо того, что пакеты используется для лучшей организации классов Java, они помогают в управлении доступом к данным класса и уникальности разрабатываемых приложений, о чем мы будем говорить далее в пособии.

Обратимся к сгенерированному коду, который представлен в окне редактора в Eclipse. Код начинается с определения пакета `ru.ifmo.practicejava`, затем следует объявление класса `HelloWorld`, и далее следует объявление метода и именем `main()`.

Введите строку `System.out.println("Hello World");` ниже комментария TODO (комментарии объясняется в следующем разделе), и сохраните код, нажав на изображение маленькой дискеты на панели инструментов или с помощью кмбинации клавиш <**Ctrl>S**.

При вводе кода Eclipse отображает контекстно-зависимую справку с предложением выбора возможных значений, сводя к минимуму предположения и опечатки. На Рис. 18, сразу после ввод `System.o` предлагается выбрать возможный вариант кодирования. Если этой подсказки автоматически не происходит, то можно использовать комбинацию клавиш **<Ctrl><Space>** для активизации подсказки.

При сохранении класса выполняется его компиляция, и, если отсутствуют синтаксические ошибки, то Eclipse создает `HelloWorld.class` в подкаталоге `bin` проекта `Hello`, который не виден в Package Explorer. В случае шибки компиляции, Eclipse ставит маленький красный круглый значок в начале проблемных строк.

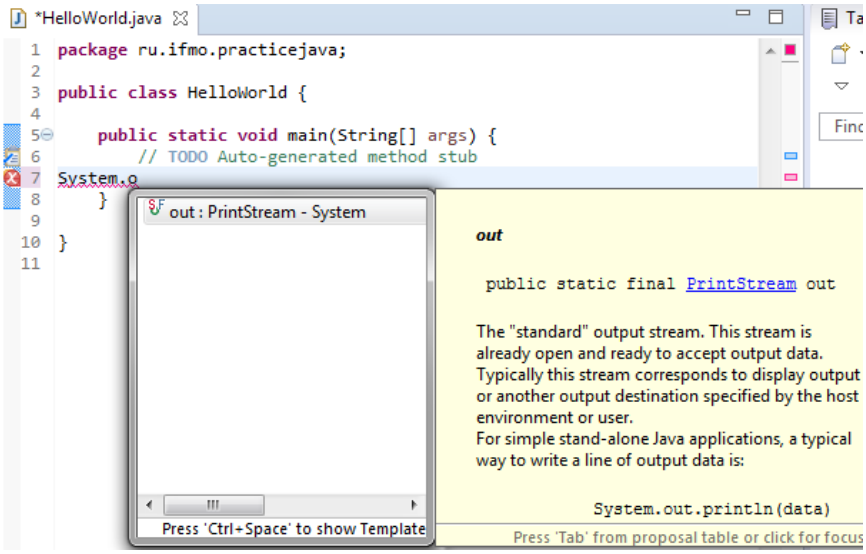


Рис. 18. Подсказка при кодировании Eclipse

Теперь можно запустить программу, нажав круглую зеленую кнопку на панели инструментов  . Вывод программы будет выполнен в представлении `Console` в нижней части Eclipse, как показано на Рис. 19.

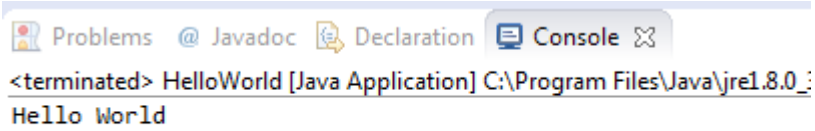


Рис. 19. Вывод программы `HelloWorld`

Рассмотрим, каким образом можно передать параметры классу в процессе выполнения, используя Eclipse. Для этого будем использовать класс `HelloWorld`, как мы это делали с использованием командной строки (Рис. 6). Для того, чтобы выполнить этот класс с параметрами, необходимо выбрать класс `HelloWorld` указателем мыши и, нажав правую кнопку в контекстном меню, последовательно выбрать **Run As>Run Configuration**. Откроется диалоговое окно Управления конфигурациями (Create, Manage and run Configuration), в котором выберите закладку Arguments, введите параметр, как показано на Рис. 20 и нажмите кнопку `Run`.

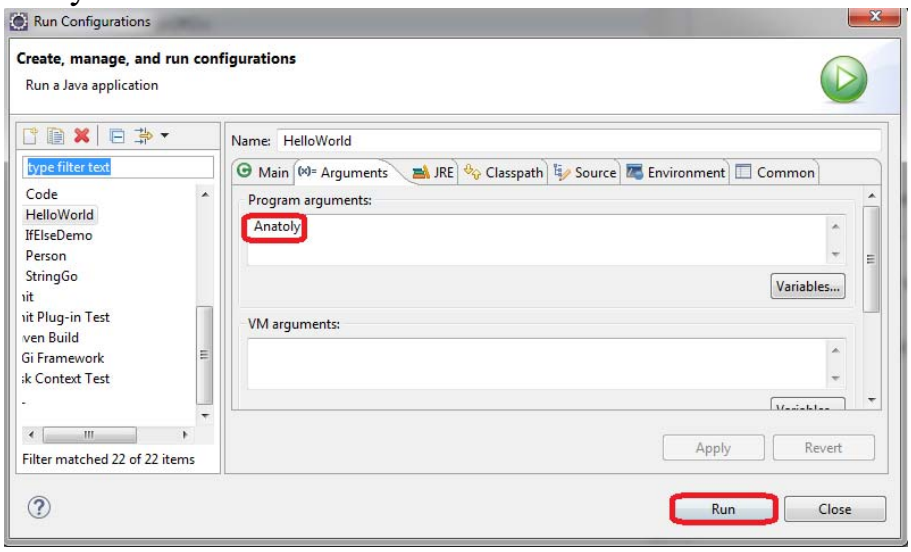


Рис. 20. Диалоговое окно Управления конфигурациями

В представлении Console будет выведено приветствие, как показано на Рис. 21.

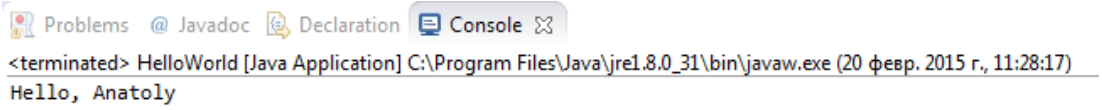


Рис. 21. Приветствие с использованием параметра

Формат этого пособия не предполагает более подробного освещения всех особенностей Eclipse, хотя по мере изложения будут описаны различные дополнительные его возможности. В списке литературы представлены ссылки на электронные материалы, а также существует множество руководств, описывающих все особенности Eclipse на `http://www.eclipse.org/documentation`.

Рассмотрим пример вычисления чисел Фибоначчи. Последовательность чисел Фибоначчи начинается с 1, и каждое последующее число представляет собой сумму двух предыдущих. Программа для вывода чисел Фибоначчи не является сложной, но она демонстрирует объявление переменных, работу еще одного простейшего цикла и выполнение арифметических операций:

```
class Fibonacci {
/** Вывод чисел Фибоначчи < 50 */
public static void main(String[] args) {
     int lo = 1;
     int hi = 1;
     System.out.println(lo);
     while (hi < 50) {
          System.out.println(hi);
          hi = lo + hi; // Изменение значения hi
          lo = hi - lo; /*
                         * Новое значение lo равно старому
                              hi, то есть сумме за
                           * вычетом старого lo
                           */
     }
  }
}
```

В этом примере объявляется класс `Fibonacci`, который, как и `HelloWorld`, содержит метод `main()`. В первых строках метода `main()` объявляются и инициализируются две переменные, `hi` и `lo`. Перед именем переменной должен указывается ее тип. Переменные `hi` и `lo` относятся к типу `int` - то есть являются 32-разрядными целыми числами со знаком, лежащими в диапазоне от -2^{31} до 2^{32-1}.

В языке Java имеется несколько встроенных (примитивных) типов данных для работы с целыми, вещественными, логическими и символьными значениями. Java может непосредственно оперировать со значениями, относящимися к примитивным типам, - в отличие от объектов, определяемых используемыми библиотеками классов и программистом. Типы, принимаемые по умолчанию, в Java отсутствуют; тип каждой переменной должен быть предварительно объявлен в классе. В Java поддерживаются следующие примитивные типы данных:

`boolean` принимает одно из двух значений: `true` или `false`
`char` 16-разрядный символ в кодировке Unicode 1.1
`byte` 8-разрядное целое (со знаком)
`short` 16-разрядное целое (со знаком)
`int` 32-разрядное целое (со знаком)
`long` 64-разрядное целое (со знаком)
`float` 32-разрядное с плавающей точкой (IEEE 754-1985)
`double` 64-разрядное с плавающей точкой (IEEE 754-1985)

В программе для вывода чисел Фибоначчи переменным `hi` и `lo` было присвоено значение 1. Начальные значения переменных можно задавать при их объявлении с помощью оператора = (это называется инициализацией). Переменной, находящейся слева от оператора =, присваивается значение выражения справа от него. В нашей программе переменная `hi` содержит последнее число последовательности, а `lo` — предыдущее число.

До инициализации переменная имеет неопределенное значение. Если вы обращаетесь к переменной до того, как ей было присвоено значение, компилятор Java будет выдавать ошибку компиляции программу.

Оператор `while (hi < 50)` в классе демонстрирует еще один вариант (наряду с `for`) организации циклов в Java, когда выполнение операторов в теле цикла зависит от значения логического выражения, заключенного в скобках. Программа вычисляет выражение, находящееся в скобках после `while`, и, если оно истинно, то выполняется операторы в блоке цикла, после завершения которого выражение проверяется снова. Цикл `while` выполняется до тех пор, пока выражение (hi<50) не станет ложным. Если оно всегда остается истинным, программа будет работать бесконечно, пока какое-либо обстоятельство не приведет к выходу из цикла - скажем, встретится оператор выхода из блока цикла `break` или возникнет исключение (`exception` - ошибка в программе), связанное, например, с появлением ошибок вычисления.

Условие, проверяемое в цикле `while`, является логическим выражением, принимающим значение `true` или `false`. Логическое выражение, приведенное в тексте программы, проверяет, не превысило ли текущее число `hi` значение 50. Если большее число последовательности (`hi`) меньше 50, то оно выводится, и программа вычисляет следующее число Фибоначчи. Если же оно больше или равно 50, то управление передается в строку программы, находящуюся после тела цикла `while`. В нашем примере такой строкой оказывается конец метода `main`(), и в нашем случае работа программы на этом завершается.

В приведенном выше примере методу `println` передается целочисленный аргумент, тогда как в `HelloWorld`, его аргументом была символьная строка типа `String`. Метод `println` является одним из многих методов, которые *перегружаются (overloaded)*, чтобы их можно было вызывать с аргументами различных типов.

1. Закодируйте и выполните выше приведенную программу из командной строки и в среде Eclipse..

2. Используя среду Eclipse, закодируйте и выполните несколько измененную программу вычисления чисел Фибоначчи, которая выглядит следующим образом:

```
public class Fibonacci {
     public static void main(String[] args) {
          int lo = 1;
          int hi = 1;
          int limit=Integer.parseInt(args[0]);
               // Переменной limit      присваивается
               // значение, ограничивающее максимальное
               //значение чисел Фибоначчи
          System.out.println(lo);
          while (hi < limit) {
               System.out.println(hi);
               hi = lo + hi; // Изменение значения hi
               lo = hi - lo; /*
                              * Новое значение lo равно
                               * старому hi,
                                * то есть сумме за
                                * вычетом старого lo
                              */
          }
     }

}
```

Обратите внимание, что в качестве верхней границы вычисления чисел фибоначчи используется величина - переменная `limit`, введенная в качестве параметра в процессе выполнения программы. Это значение передается в программу из командной строки с помощью параметра, являющегося эмементом массива `String[] args`. Вообще, таких параметров может быть несколько и к ним следует обращаться с использованием индекса, указывающего на номер элемента массива, начиная с нуля (`args[0]`). Значение параметра является символьной строкой, которая преобразуется в целочисленную величину и присваивается переменной `limit` с использованием метода *`parseInt`* класса `Integer`. Этот метод принимает значение типа `String` и возвращает целое значение, полученное на основе символов строки. Естественно,

предполагается, что такое преобразование возможно, и введенная строка состоит из символов {0-9}. При выполнении класса из командной строки необходимо ввести команду `java Fibonacci 150`, как показано на Рис. 22:

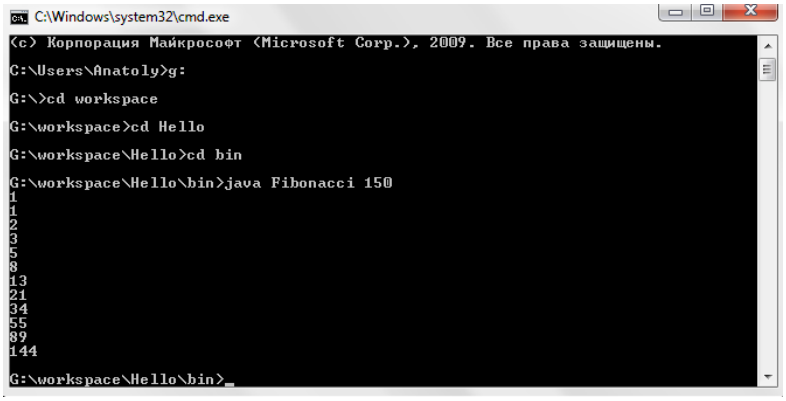


Рис. 22. Командное окно с выводом программы Fibonacci

Лексические основы языка Java

Прежде чем более подробно приступить к изучению концепции объектно-ориентированного программирования рассмотрим лексические основы построения правильных конструкций программы на Java.

Язык Java позволяет создавать первоклассные объекты, но не *всё*, что есть в этом языке, является объектом. От чисто объектно-ориентированных языков, таких, например, как Smalltalk, язык Java отличают два свойства. Во-первых, язык Java представляет собой смесь из объектов и простых (примитивных) типов. Во-вторых, он позволяет писать код, который не раскрывает внутренние детали одного объекта другому объекту, использующему его функциональность через открытый набор методов.

Язык Java дает инструменты, необходимые для следования принципам ООП и создания качественного объектно-ориентированного кода. Поскольку Java не является чисто объектно-ориентированным языком, нужно придерживаться определенных правил программирования – язык не вынуждает делать все правильно, поэтому необходимо следить за этим самостоятельно. В одном пособии не представлен весь синтаксис языка Java, поскольку его языковые возможности исключительно многообразны и понимание некоторых конструкций требует специальных знаний. В этой части основное внимание уделяется основам языка, что позволяет приобрести достаточно знаний и практики для написания простых программ.

Программа на Java представляет собой набор пробелов, комментариев, ключевых слов, идентификаторов, литеральных констант, операторов и разделителей. Язык Java допускает произвольное форматирование текста программ. Для нормальной работы программы отсутствует необходимость в выравнивании кода специальным образом, как того могут требовать другие языки программирования.

Комментарии, применяемые в процессе кодирования, поскольку компилятор игнорирует их, но при должном использовании они являются весьма полезной частью исходного кода. Комментарии делают код более читаемым и понятным программистам в процессе разработки и модификации программ, в том числе, и не разработчикам кода. Комментарии, занимающие одну строку, начинаются с символов // и заканчиваются в конце строки. Такой стиль комментирования полезен для размещения кратких пояснений к отдельным строкам кода:

```
a = 42; // если 42 - возраст клиента
```

Комментарии, размещенные на нескольких строках помещаются между символами /* и */:

```
/* возраст клиента    тоже комментарий
в несколько строк */
```

Некоторые комментарии начинаются с символов /** и заканчиваются, */ используются специальной утилитой Javadoc.exe, которая может автоматически извлекать текст из этих комментариев и создавать на их основе документацию к программе.

Чтобы получить представление о том, в каком виде Javadoc может генерировать документацию, посмотрите на документально API для Java, созданную с помощью этого инструментария http://docs.oracle.com/javase/8/docs/api/overview-summary.html.

Идентификаторы используются для именования классов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов _(подчеркивание) и $ (доллар). Идентификаторы не должны начинаться с цифры, чтобы компилятор не перепутал их с числовыми литеральными константами (2count – компилятор не пропустит такого идентификатора,). Java - язык, чувствительный к регистру клавиатуры, и, таким образом, Value и VALUE представляют собой различные идентификаторы. Для каждого идентификатора резервируется место в оперативной памяти компьютера для хранения значений идентификатора, в зависимости от объявленного типа данных.

Некоторые идентификаторы, представляющие переменные значения объектов, могут меняться с течением времени (Variable), а некоторые остаются без изменения (Constant).

Существуют соглашения об именовании идентификаторов, которых рекомендуется придерживаться, и кратко можно сформулировать следующим образом:

- `Package` кодируется строчными символами, например, `com.company.customer;`
- `Class` - первый символ имени и каждое последующее слово идентификатора кодируется с заглавной - `CustomerDriver`. Здесь используется т.н. `CamelCase` именование;
- `Interface` - аналогично классу, например, `Drawable;`
- `variable` - первый символ строчный, но каждое последующее слово с заглавной, например, `grandTotal;`
- `method` - первый символ строчный, но каждое последующее слово с заглавной, например, `computePay`. Рекомендуется для имени метода использовать глагольное выражение - `computeSum;`
- `Constant` - все символы заглавные - `MANUFACTURER.`

Зарезервированные ключевые слова - это специальные идентификаторы, которые в языке Java используются для того, чтобы идентифицировать встроенные типы, модификаторы доступа к методам и данным и средства управления выполнением программы. Эти ключевые слова совместно с синтаксисом операторов и разделителями входят в описание языка Java. По сути дела это элементы языка, смысл которых зарезервирован языковыми конструкциями языка и не разрешается их произвольное использование в качестве идентификаторов для имен переменных, классов или методов. На Рис. 23 представлены зарезервированные ключевые слова языка Java.

abstract	boolean	break	byte	byvalue
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short
static	super	switch	syncronized	this
throw	throws	transient	true	try
var	void	volatile	while	

clone	equals	finalize	getClass	hashCode
notify	notifyAll	toString	wait	

Рис. 23. Зарезервированные ключевые слова языка Java

Следует отметить, что `true`, `false` и `null` по существу не являются зарезервированными словами. Хотя они являются литералами, тем не менее, включаются в этот список, поскольку их нельзя использовать для именования Java-конструкций.

Одно из преимуществ программирования с помощью IDE Eclipse состоит в том, что для зарезервированных слов используется синтаксическая подсветка, как вы увидите далее в этом пособии.

Литеральные константы представляют собой адресную, числовую или символьную константы программы, непосредственно включаемые в операторы или команды (в отличие от переменных, обращение к которым производится посредством их идентификаторов). Язык Java позволяют задавать в программе следующие виды литералов:

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

Рассмотрим возможные варианты кодирования литералов для каждого вида:

Целочисленные литералы:

111 - десятичное (десятеричное) число

0b0010 - число 2 в двоичной форме (начинается с 0b)

056 - число 46 в восьмеричной форме (начинается с цифры 0)

0xABcDeF - число 11259375 в шестнадцатеричной форме (начинается 0x)

В языке Java возможны четыре системы счисления: десятичная (десятеричная), двоичная, восьмеричная и шестнадцатеричная. Числа в десятичной форме — это числа с основанием 10{0,9}, числа в двоичной форме — это числа с основанием 2 числа {0,1}, в восьмеричной форме — это числа с основанием 8 {0, 7}, числа в шестнадцатеричной форме — числа с основанием 16 {0,F}.

Литералы чисел **с плавающей точкой**:

```
18.01
31.4e-1
0.314e1
```

Эти литералы, как и целочисленные, могут иметь знаки "+" и "-" (т.е. быть положительными или отрицательными), иметь в записи точку, которая разделяет целую и дробную части, а также букву `e` и следом за ней - степень, в которую необходимо возвести число (если степень положительная, знак "+" можно не указывать).

Логический литерал принимает два значения `true` (истина) и `false` (ложь). Они служат для представления логического (или булева) типа данных - `boolean`.

Символьные литералы описывают один символ из набора Unicode (16-ти разрядный).

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

Символьные литералы должны заключаться в одинарные кавычки. Среди символьных литералов есть, так называемые, escape-последовательности, которые позволяют произвести какую-либо операцию, например, перевести курсор на новую строку или вывести обратную косую черту (слеш).

Они представляют собой набор последовательностей вида `\uxxxx`, где вместо `x` могут быть шестнадцатеричные символы. Однако имеются специальные символы, которые соответствуют escape-последовательностям, которые представлены ниже:

```
\a  Предупреждение (звонок)
\b  Возврат курсора на шаг
\f  Перевод страницы
\n  Следующая строка (перевод на новую строку)
\r  Возврат каретки
\t  Табуляция
```

\\ Отображение обратной косой черты
\' Отображение одинарной кавычки
\" Отображение двойной кавычки
\xxx Символ восьмеричного значения (не более 377 — т.е. 255 в десятичной системе)

Строковые литералы состоят из набора символов и записываются в двойных кавычках. Длина может быть нулевой или сколь угодно большой.

Любой символ может быть представлен специальной последовательностью, начинающейся с символа обратный слеш «\»

```
"Hello, world!\r\nHello!"
```

Ранее, использованный нами в программе `HelloWorld` метод `println ("Hello World")` для класса `System.out` выводил сообщение и переходил на следующую строку. У этого класса существует еще один метод `print`, который в отличие от предыдущего, не переходит на следующую строку после завершения вывода в текущей. Используя специальную последовательность символов, можно управлять выводом. Так, следующие утверждения Java производят эквивалентный эффект:

```
System.out.print("Hello, 1955!! \n");
System.out.println("Hello, 1955!! ");
```

В методе `print()` и `println()` используется оператор конкатенации строк, который выполняется при помощи операции «+». Буквально одна строка прибавляется к другой строке. При этом выполняется преобразование различных типов к `String`.

Null-литерал может принимать лишь одно значение: `null`

Это литерал ссылочного типа, причем эта ссылка никуда не ссылается, а объект отсутствует.

Разделители, которые влияют на внешний вид и функциональность программного кода. Используются следующие разделители: **( ) [ ] { } ; . ,**

Типы данных

В Java-программе переменные должны быть описаны перед их использованием.

Синтаксис описания:

<Тип> {<переменная - 1>[=<значение - 1>],<переменная - 2>[=<значение - 1>]}, …;

Здесь и далее для определения синтаксиса используется упрощенная форма Бэкуса-Наура (БНФ), которая была введена для описания языка программирования Алгол. Не вдаваясь в подробности, рассмот-

рим используемые в записи соглашения об обозначениях, в том числе, пока не используемые:

< > (угловые скобки) – то, что в них указано, определяет пользователь;

[ ] (квадратные скобки) – выделяют те части команды, которые могут отсутствовать;

( ) (круглые скобки) – заключают аргументы команды;

{ },… – заключенная в фигурные скобки часть конструкции может быть повторена несколько раз, причем повторы разделяются запятыми;

| (вертикальная черта) – означает альтернативный выбор.

Из описания видно, что вначале необходимо указать тип переменной. Также можно описывать переменную и инициировать начальное значение. Существует восемь примитивных типов данных в Java: Четыре из них для целочисленных значений, два для значений с десятичной точкой, один для хранения отдельных символов и один для булевских данных, который допускает только значения литералов `true` или `false` в качестве значений. Ниже приведены некоторые примеры объявления переменных и инициализации:

```
int chairs = 12;
char grade = 'A';
boolean cancelJob = false;
double nationalIncome = 23863494965745.78;
float hourlyRate = 12.50f;    // добавлен f в
// конец литерала с плавающей точкой
long totalCars = 46372836483921; // добавлен l
// в конец целочисленного литерала
```

Можно использовать символы l или L для обозначения числа типа long, однако рекомендуется использовать заглавную букву, чтобы избежать возможной путаницы.

Для хранения значений, которые никогда не изменяются, вы должны объявить константу, просто добавляя к описанию ключевое слово `final`, например, `final String MANUFACTURER;`

Значение константе может быть присвоено только один раз, и, например, при создании экземпляра конкретного автомобиля, производитель которого известен и не может изменяться в течение всего срока службы этого объекта, можно использовать значение константы `MANUFACTURER = "Volvo";`

Кроме базовых типов данных широко используются соответствующие им классы оболочки (wrapper – classes): `Boolean, Character, Integer, Byte, Short, Long, Float, Double`. Объекты этих классов могут хранить те же значения, что и соответствующие им базовые типы, но преобразуются в надлежащие классы, у кото-

рых появляется возможность использовать методы этих классов. На Рис. 24 представлен фрагмент иерархии классов, из которого видно наследование их характеристик. Здесь следует обратить внимание на то, что все классы прямо или косвенно наследуют атрибуты и методы класса `Object`, который располагается в вершине иерархии классов и его методы наследуются всеми подклассами.

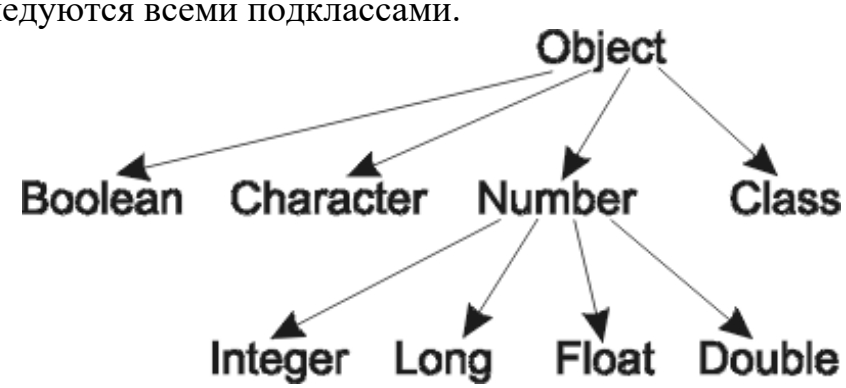


Рис. 24. Фрагмент иерархии классов Java

Ниже представлен фрагмент кода Java, показывающий создание объектной переменной `keyObj` на основе примитивного значения, присвоенного переменной `key`.

```
int key = 123;
Integer keyObj = new Integer(key);
```

Другие типы создаются аналогично. Следует иметь в виду, что значение полученного объекта неизменно, поскольку отсутствуют методы классов, позволяющие изменить их значение.

В таблице 1 приведены некоторые характеристики типов данных Java и название классов для соответствующего объекта данного типа.

Таблица 1 Характеристики типов данных

Примитивный тип	Размер bits	Min значение	Max значение	WrapperClass
byte	8	-128	127	Byte
short	16	-32,768	32,767	Short
int	32	-2,147,483,648	2,147,483,64	Integer
long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	Long
float	32	1,17549435e-38	3,4028235e+38	Float
double	64	4,9e-324	1,7976931348623157e+308	Double
char	16	Unicode 0	Unicode 2 в степени 16	Character
boolean	n/a		true (не max)	Boolean

Операторы

Как и следовало ожидать, язык Java позволяет выполнять арифметические действия, и в пособии уже использовался оператор присвоения значения переменным с использованием выражений. Теперь мы кратко рассмотрим некоторые операторы языка Java, которые потребуются для расширения возможностей программирования вычислений. В языке Java используются операторы двух типов:

- *унарные*, которым требуется только один операнд;
- *бинарные*, которым необходимы два операнда.

Арифметические операторы языка Java перечислены в Таблице 2.

Таблица 2. Арифметические операторы языка Java

Оператор	Пример использования	Описание
`+`	`a + b`	Сложение `a` и `b`
`+`	`+a`	Преобразование a в переменную типа `int`, если это переменная типа `byte`, `short` или `char`
`-`	`a - b`	Вычитание `b` из `a`
`-`	`-a`	Арифметическая инверсия `a`
`*`	`a * b`	Умножение `a` на `b`
`/`	`a / b`	Деление `a` на `b`
`%`	`a % b`	Вычисление остатка от деления a на b (оператор взятия модуля)
`++`	`a++`	Приращение `a` на 1 с предварительным вычислением значения `a`
`++`	`++a`	Приращение `a` на 1 с последующим вычислением значения `a`
`--`	`a--`	Уменьшение `a` на 1 с предварительным вычислением значения `a`
`--`	`--a`	Уменьшение `a` на 1 с последующим вычислением значения `a`
`+=`	`a += b`	Краткая форма записи `a = a + b`
`-=`	`a -= b`	Краткая форма записи `a = a - b`
`*=`	`a *= b`	Краткая форма записи `a = a * b`
`%=`	`a %= b`	Краткая форма записи `a = a % b`

Некоторые арифметические операторы нуждаются в комментариях. В частности, следует иметь в виду, что операторы инкремента (++) и декремента (--) увеличивают или уменьшают значение переменной, с

которой они используются. Например, оператор `y++;` равносилен выполнению оператора `y = y+1;` и приводит к увеличению на 1 значения переменной. Эти операторы существуют в двух формах:

- в префиксной форме (++y) значение операнда изменяется прежде, чем оно используется в выражении;
- в постфиксной форме (y++) предыдущее значение сначала используется в выражении и только потом значение операнда изменяется.

Ниже приведенные операторы демонстрируют возможности применения оператора ++ в различных вариантах.

```
a = 5;
x = a++;    // x = 5
y = ++a;    // y = 7
```

Расширенные операции присваивания, такие как `+=, -=, *=, /=` и др., все выполняются по следующей схеме:

`a += b;` равносильно `a = a + b;`

Фрагмент программы ниже демонстрирует примеры применения расширенных операторов присваивания:

```
int a = 1;
int b = 2;
int c = 3;
a += 5; // a=a+5 => 6
b *= 4; // b=b*4 => 8
c += a * b; // c= c+ a*b => 3+6*8
System.out.println("c =>"+c);
c %= 6;
System.out.println("a =>"+a+" b =>"+b+"  c =>"+c);
Результат выполнения фрагмента кода следующий:
c =>51
a =>6 b =>8  c =>3
```

Преобразование типов

При таком многообразии типов часто возникают ситуации, когда данные одного типа требуется преобразовать к другому типу. Некоторые преобразования происходят неявно. Рассмотрим такой пример:

```
double a = 3;
System.out.println(a); // выведет 3.0
```

Переменная типа `double` предусматривает наличие не только целой, но и дробной части числа, т.е. фактически в переменную запишется значение 3.0 (три целых и ноль десятых), которое потом и выведется на экран. Java выполнила преобразование 3 в 3.0 (целое число в вещественное) самостоятельно, без явного участия программиста. Такое преобразование - customizing (или, как ещё говорят, приведение) типа дан-

ных называется **неявным** или автоматическим и происходит всякий раз, когда в процессе преобразования не могут потеряться какие-либо данные (т.е. когда преобразования производится к более универсальному типу: от коротких целых `short` к длинным целым `long`, от целых `int` к вещественным `double` и т.п.). На Рис. 25 стрелками представлено направление неявного преобразования типов.

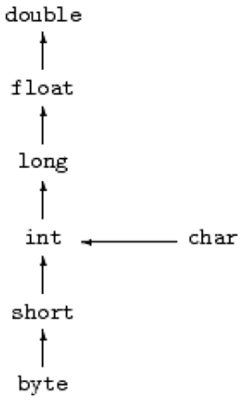


Рис. 25. Направление неявного преобразования типов

Потери значимости могут происходить, когда мы попытаемся, например, из вещественного числа получить целое. Это можно сделать, округлив число или взяв только его целую часть, но дробную часть при этом придётся отбросить, и, если она не была существенна, то данные могут потеряться.

Например, если мы попытаемся выполнить присвоение с помощью оператора `int a = 3.14;` то получим сообщение компилятора об ошибке преобразования.

Тем не менее, преобразовать вещественное значение к целому мы сможем, явно известив в программе о своём намерении. Для этого слева от исходного элемента следует в круглых скобках указать название типа, к которому исходное выражение необходимо привести, например:

```
int a = (int) 3.14;
System.out.println(a); // выведет 3
```

Такое преобразование с указанием целевого типа называется явным. Явное преобразование вещественного значения к целому типу происходит за счёт отбрасывания дробной части.

```
double b = 2.6;
int c = (int) (0.5 + b); // можно применять к целым выражениям
System.out.println(c); // выведет 3
System.out.println((int)9.59); // выведет 9
System.out.println((int)'A'); // выведет 65 — код символа «A»
System.out.println((double)3); // выведет 3.0
```

Явное преобразование может потребоваться также в тех случаях, когда значение типа, позволяющего хранить большее количество знаков, надо привести к типу, способному хранить меньшее количество знаков числа. Например, когда `long` надо преобразовать к `short` , которое содержит меньшее количество разрядов для хранения значений, следует применять явное преобразование, в противном случае обнаруживается ошибка компиляции.

Как мы увидим в дальнейшем, при помощи явного преобразования можно выполнять операцию приведения типов между объектами.

Дополнительные операторы

В дополнение к операторам, перечисленным в Таблице 2, мы использовали некоторые другие символы, которые также называются операторами в языке Java, в том числе:

- точка "`.`" указывает на имена пакетов и вызывает методы;
- круглые скобки "`()`" отделяют список параметров, разделенных запятыми, от имени метода;
- `new` создает экземпляр объекта (если за ним следует имя конструктора объекта). Конструктор - специальный метод класса, который используется для создания объекта. Класс - это некоторый шаблон, на основании которого создаются экземпляры класса - объекты.

В синтаксис языка Java входит также ряд операторов, которые используются специально для условного программирования, то есть для вычислений, которые по-разному реагируют на различные входные данные. Мы рассмотрим их в следующем разделе.

Условные операторы и операторы управления

Здесь рассматриваются операторы, которые указывают Java-программе, как действовать при различных значениях входных данных.

Операторы отношений и логические операторы

Язык Java содержит операторы, которые позволяют программе принимать решения. Операторы *отношения* (`>, >=, <, <=, ==, !=`) используются, чтобы проверить отношение между двумя операндами, и возвращают `boolean` значение "истина" (`true`) или "ложь" (`false`) в зависимости от результата сравнения значений отношения. Для формирования более сложных логических выражений используются *логические операторы* - &&, ||, !, &, |, ^ и круглые скобки "( )". Операндами логических операторов в сложном логическом выражении

должны быть значения типа boolean, получаемый результат - boolean.

В Таблице 3 приведены операторы отношений и логические операторы языка Java.

Таблица 3. Операторы отношений и логические операторы Java

Оператор	Пример использования	Возвращает значение "true", если...
>	a > b	a больше b
>=	a >= b	a больше или равно b
<	a < b	a меньше b
<=	a <= b	a меньше или равно b
==	a == b	a равно b
!=	a != b	a не равно b
&& оператор AND быстрой оценки выражений	a && b	a и b истинны, b оценивается условно (если a ложно, b не вычисляется)
|| оператор OR быстрой оценки выражений	a || b	a или b истинно, b оценивается условно (если a истинно, b не вычисляется)
! логическое унарное отрицание (NOT)	!a	a ложно
& логическое И (AND)	a & b	a и b истинны, b оценивается в любом случае
| логическое ИЛИ (OR)	a | b	a или b истинно, b оценивается в любом случае
^ логическое иселючающее ИЛИ (XOR)	a ^ b	a и b различны

В верхней части Таблицы 3 представлены отношения, которые производят boolean значение. Так, например, ниже следующий фрагмент кода Java, выводит boolean значения, вычисляемые на основе анализа соотношения сравниваемых величин:

```
System.out.println(7==6);     //  false
System.out.println(7>6); //  true
System.out.println(7<6); //  false
System.out.println(7!=6);     //   true
```

В Таблице 4 показаны результаты воздействия логических операторов на различные комбинации значений операндов A и B.

Таблица 4. Результаты операторов A *op* B

A	B	OR	AND	XOR	NOT A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true
true	true	true	true	false	false

Следующий фрагмент кода Java

```
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
```

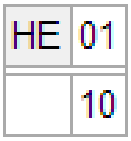
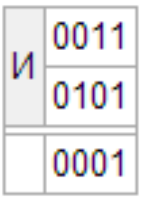
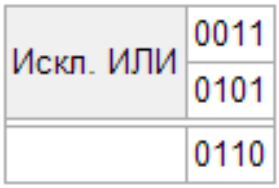
производит следующий вывод:

```
a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false
```

Целочисленные битовые операторы

Для целых числовых типов данных — `long`, `int`, `short`, `char` и `byte`, определен дополнительный набор операторов, с помощью которых можно проверять и модифицировать состояние отдельных битов соответствующих значений. В Таблице 5 приведены операторы битовой арифметики, которые работают с каждым битом как с самостоятельной величиной.

Таблица 5 Операторы битовой арифметики

Оператор	Пример использования	Результат работы
~ побитовое унарное отрицание (NOT)	~a	
& побитовое И (AND)	a & b	
| побитовое ИЛИ (OR)	a ^ b	
^ побитовое исключающее ИЛИ (XOR)	a & b	
>> сдвиг вправо	a >> b	Рассматривается позже
>>> сдвиг вправо с заполнением нулями	a >>> b	Рассматривается позже
<< сдвиг влево	a << b	Рассматривается позже

Следующий фрагмент кода Java

```
String binary[] = { "0000", "0001", "0010", "0011", "0100",
"0101", "0110", "0111", "1000", "1001", "1010", "1011",
"1100", "1101", "1110", "1111" };
          int a = 3; // 0+0+2+1 или двоичное 0011
          int b = 6; // 0+4+2+0 или двоичное 0110
          int c = a | b;
          int d = a & b;
          int e = a ^ b;
          int f = (~a & b) | (a & ~b);
          int g = ~a & 0x0f;
          System.out.println(" a = " + binary[a]);
          System.out.println(" b = " + binary[b]);
          System.out.println(" a|b = " + binary[c]);
```

```
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println(" ~a&b|a^~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
```

производит следующий вывод:

```
a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a^~b = 0101
~a = 1100
```

Приведенный выше фрагмент програмы вначале формирует массив `String`, представляющий все комбинации нулей и единиц в диапазоне от 0 (`"0000"`) до 15(`"1111"`) для того, чтобы вывести двоичный вариант результата различных битовых операторов. Каждый оператор выводит элемент массива, соответствующий значению полученных в коде переменных, в том числе, с применением операторов битовой арифметики.

Рассмотрим отложенные операторы. Оператор >> означает в языке Java сдвиг вправо. Он перемещает все биты своего левого операнда вправо на число позиций, заданное правым операндом. Когда биты левого операнда выдвигаются за самую правую позицию слова, они теряются. При сдвиге вправо освобождающиеся старшие (левые) разряды сдвигаемого числа заполняются предыдущим содержимым знакового разряда. Такое поведение называют расширением знакового разряда.

Оператор << выполняет сдвиг влево всех битов своего левого операнда на число позиций, заданное правым операндом. При этом часть битов в левых разрядах выходит за границы и теряется, а соответствующие правые позиции заполняются нулями. Здесь используется автоматическое повышение типа всего выражения до `int`, в том случае, если в выражении присутствуют операнды типа `int` или целые типы меньшего размера. Если же хотя бы один из операндов в выражении имеет тип `long`, то и тип всего выражения повышается до `long`.

Рассмотрим еще ряд примеров с целочисленными битовыми операторами, представленными в Таблице 5. Так, например, прокомментированный код, представленный ниже

```
byte b = (byte) 6&5;  //0000 0110 0000 0101->0000 0100
byte c = (byte) 5<<2; // 0000 0101->0001 0100
byte d = (byte) 6|5;  //0000 0110 0000 0101->0000 0111
byte e = (byte) 6^5 ; //0000 0110 0000 0101->0000 0011
byte f = (byte) 5>>2 ;   //0000 0101->0000 0001
byte g = (byte) 127>>4 ; //1111 1111->0000 0111
```

```
System.out.printf("Результат  6&5 = %x %n" , b);
System.out.printf("Результат  5<<2 = %x %d%n" , c,c);
System.out.printf("Результат  6|5 = %x%n", d);
System.out.printf("Результат  6^5 = %x%n", e);
System.out.printf("Результат  5>>2 = %x%n", f);
System.out.printf("Результат  127>>4 = %x%n", g);
```

выводит следующий результат, демонстрируя результаты выполненных побитовых операций:

```
Результат  6&5 = 4
Результат  5<<2 = 14 20
Результат  6|5 = 7
Результат  6^5 = 3
Результат  5>>2 = 1
Результат  127>>4 = 7
```

В этом коде вместо `println` используется `printf`, позволяющий организовать отформатированный с помощью специальных выражений вывод. В методе `printf` сначала задается шаблон, согласно которому будет отформатирована строка, а потом передаются объекты для форматирования. Вот как выглядит метод `printf` с параметрами:

`System.out.printf(String format, Object... args);`

Формат обычно заключается в двойные кавычки, и определяет содержание вывода и тип выводимых данных с помощью следующих символов, начинающихся со знака "`%`" (Далее для простоты текста вместо `System.out.printf()` пишется `printf()`):

`%s` - для типа `String`, например, `printf("Hello %s!", "World");` // выводится `"Hello World!"`

`%n` в составе формата выполняет переход на другую строку в выводе.

Для типов `byte, short, int, long`:

`%d` - вывод в десятеричном формате.

`%x` - вывод в шестнадцатеричном формате. Здесь также можно указывать ширину поля вывода, например, так: `%7d` - в десятеричном формате и минимальной шириной поля 7 знаков. Оператор `printf("%7d", 1);` // выводит `"      1"`

Можно закодировать вариант вывода с заполнением пробелов нулями:

%07d → Минимальная ширина строки 7 знаков. Начало заполняется нулями.

`printf("%07d", 1); // "0000001"`

Для типов `float, double`.

`%f` - Десятичное число с точкой.

`%e` - Десятичное число с точкой и экспонентой.

Например, %.10f выводит с точностью 10 знаков после запятой.

В отличие от `println`, `printf` использует не конкатенацию строк, а список переменных через запятую в порядке, определенном строкой формата.

Условная операция

Язык Java поддерживает условную операцию, которая также называется тернарная и записывается следующим образом: **<условие> ? <выражение1> : < выражение2>;**

Если <условие> истинно, то результатом будет <выражение1>, в противном случае <выражение2>.

Например, `int x = a < b ? a : b;` объявляет переменную `x` и присваивает вычисленный минимум значений из `a` и `b`.

Приоритеты операций

В Java действует определенный порядок, или приоритет операций. Из элементарной алгебры известно, что умножение и деление имеют более высокий приоритет, чем сложение и вычитание. Выражения в скобках должны выполняться в первую очередь и т.д. В программировании также действует определенный порядок операций. Приоритеты операций Java, от высшего к низшему, описаны в Таблице 6. Обратите внимание, что в первой строке таблицы указаны элементы, которые, как правило, не считают символами операций: круглые и квадратные скобки и символ точки. С технической точки зрения они являются разделителями, но в выражениях они действуют подобно операциям. Круглые скобки используют для изменения порядка выполнения операций. Квадратные скобки служат для индексации массивов. А символ точки "." используется для доступа к элементам объектов, применение которых более подробно будет рассмотрено в последующих разделах.

Таблица 6. Приоритеты операций Java

Высший приоритет
( ) [ ] .
++ -- ~ !
*** / %**
+ -
>> >>> <<
== !=
&

^
|
&&
||
= op=
Низший приоритет

Поскольку высший приоритет имеют круглые скобки, вы всегда можете добавить в выражение несколько пар скобок, если у вас есть сомнения по поводу порядка вычислений или вам просто хочется сделать свой код более читабельным.

Рассмотрим выражение `int x = a >> b + 3;`

Какому из двух выражений, `a >> (b + 3)` или `(a >> b) + 3`, соответствует выражение? Поскольку оператор сложения имеет более высокий приоритет, чем оператор сдвига, правильный ответ - `a>> (b + a)`. Если же требуется выполнить операцию сдвина вначале, то необходимо задействовать скобки - `(a>>b)+ 3`.

В соответствии с приоритетами операций можно вычислять сложные выражений, например:

```
int i=3;
int var=i+++i;            //Результат = 7
```

или

```
int a = 6;
a += ++a + a++;
System.out.println(a);  //Результат = 20
```

Область видимости переменной

Переменные, описанные в классе, действуют (доступны) в течение определенного времени и имеют свою область действия. Это означает, что они могут использоваться только в определенном месте программного кода — именно там, где они описаны. Чтобы определить переменную на уровне метода, ее описание помещается в тело данного метода, и тогда это будет локальная переменная метода. Чтобы определить переменную на уровне класса и сделать её тем самым доступной для совместного применения во всех методах данного класса, следует поместить ее описание в разделе объявлений класса — перед описанием каких-либо методов. Например, если переменная `someLocalVariable` описана в теле метода с именем `someMethod`, именно этот метод и является ее областью действия. Таким образом, если имеется другой метод `someOtherMethod`, вы не можете использовать в ней эту же переменную. Если вы попытаетесь сделать это, то либо получите сообще-

ние об ошибке из-за использования неописанной переменной, либо просто получите другую переменную - с тем же самым именем, но никак не связанную с одноименной переменной из первого метода. Такая ситуация представлена на Рис. 26.

Блок, или составной оператор - это произвольный состав простых операторов языка Java, заключенных в фигурные скобки. Блоки (как и методы) определяют область видимости своих переменных. В Java запрещается иметь одноименные переменные внутри одного определения метода. Переменные, объявленные в блоке, являются локальными по отношению к этому блоку и пропадают, когда выполнение этого блока завершается. Переменную нельзя использовать вне блока, в котором она описана.

Блоки могут быть вложенными один в другой. Любая переменная в Java-программе имеет *область видимости*, или локализованное пространство имен, в котором к ней можно обращаться по имени. За пределами этой области переменная *не видима*, и попытка обращения к ней приведет к ошибке компиляции. Уровни области видимости в языке Java определяются тем, где объявлена переменная, как показано на Рис. 26.

```
1  package ru.ifmo.practicejava;
2
3  public class SomeClass {
4        private String someClassVariable;
5        public void someMethod(String someParameter) {
6          String someLocalVariable = "Hello";
7          if (true) {
8            String someOtherLocalVariable = "Howdy";
9          }
10         someClassVariable = someParameter; // правильно
11         someLocalVariable = someClassVariable; // тоже правильно
12         someOtherLocalVariable = someLocalVariable;// Переменная вне области видимости!
13       }
14       public void someOtherMethod() {
15         someLocalVariable = "Hello there";// Эта переменная вне области видимости!
16       }
17     }
```

Рис. 26. Области видимости переменных

В представленном классе с именем `SomeClass` переменная `someClassVariable` доступна для всех методов экземпляра. Внутри метода `SomeMethod` параметр `someParameter` доступен, но за пределами этого метода – нет, и то же справедливо для `someLocalVariable`. Внутри блока `if` объявляется переменная `someOtherLocalVariable`, а за его пределами она вне видимости. Область видимости подчиняется многим правилам, но наиболее важные иллюстрируются в коде класса `SomeClass`.

Обращение к методам

Приложение - набор объектов, взаимодействующих друг с другом. Кроме того, что объект хранит какие-то данные, он умеет выполнять различные операции над своими данными и возвращать результаты этих операций. Теоретически эти операции выполняются как реакция на получение некоторого сообщения данным объектом. Практически это происходит при вызове метода данного объекта, например, следующим образом - `outputArea.setText( output );`

При обращении к методу формальные аргументы описания метода заменяются на фактические (реальные) параметры. Если формальные аргументы имеют примитивные типы, то при замене аргументов на фактические параметры используется механизм вызова по значению (`call by value`). При этом формальный параметр, используемый в определении метода, является локальной переменной, которая инициализируется значением фактического параметра, и ее изменение не отражается на фактическом значении переданного параметра.

Между формальными аргументами и фактическими параметрами метода устанавливается взаимно-однозначное соответствие, т. е порядок передаваемых значений должен соответствовать предписанию сигнатуры метода. В случае рассогласования по типу осуществляется автоматическое преобразование типов: `byte->short->int->long->float->double.`

Параметры типа класса являются ссылками. Это означает, что в переменной типа класса хранится не объект этого класса, а адрес области (ссылка – reference) оперативной памяти, где расположен объект. В этой связи все типы классов являются ссылочными типами, но существуют также ссылочные типы, которые не являются типами классов (например, массивы). Таким образом, ссылочные типы передаваемых в метод значений могут непосредственно подвергаться изменению при выполнении метода.

Оператор if

Условный оператор `if` широко применяется и встречается во всех языках программирования. Оператор `if` позволяет программе в зависимости от условий выполнить оператор или группу операторов, основываясь на значении переменной типа `boolean` или логического выражения. Оператор позволяет изменять ход выполнения программы в зависимости выполнения тех или иных условий.

Оператор начинается с ключевого слова `if`, за которым должно следовать выражение, возвращающее `boolean` значение (`true` или

false), заключенное в круглые скобки. Самая простая форма оператора выглядит так:

```
if (<условие>) оператор;   // если условие истинно, то выполняется оператор
```

Если условие истинно, то оператор или группа операторов выполняется, если ложно, то оператор не выполняется. Очень часто boolean выражение в операторе if содержит какое-нибудь отношение, но можно использовать переменную типа boolean или константу: if (isRunning){...}

оператор в формате if является фрагментом кода, заключенным в фигурные скобки, который называют блоком операторов. Если используется только один оператор, то фигурные скобки не обязательны. Если выполняется более одного оператора, следует использовать фигурные скобки для формирования *составного оператора*. Составной оператор группирует несколько операторов в один блок.

Существует еще один вариант оператора if с использованием ключевого слова else:

```
if (<условие>) оператор1;  // если условие истинно,
                           //то выполняется оператор1
else оператор2;          // если условие ложно, то
                          // выполняется оператор2
```

В этом случае, если условие истинно, то выполняется оператор1, если условие ложно, то выполняется оператор2, который относится к else. Вместо операторов могут кодироваться блоки операторов, заключенные в фигурные скобки:

```
if (<условие>)
{
    оператор1;
    оператор2;
}
else
{
    оператор3;
    оператор4;
}
```

Вложенные операторы if

Вложенный оператор if используется для дополнительной проверки условий, когда условие предыдущего оператора if принимает значение true. Иными словами, вложенный оператор применяется в тех случаях, когда для выполнения действия требуется соблюдение сразу нескольких условий, которые не могут быть указаны в одном услов-

ном выражении. Необходимо помнить, что во вложенных операторах `if-else`, вторая часть `else` всегда относится к ближайшему оператору `if`, за условным выражением которого следует оператор ";" или блок операторов. Ниже небольшой фрагмент демонстрирует применение `if`:

```
if(i == 10)
{
    if(j < 20) a = b;
    if(k > 100) c = d;
    else a = c;  // else относится к if(k > 100)
}
else a = d;      // else относится к if(i == 10)
```

Цепочка операторов if-else-if

Часто используется цепочка операторов `if-else-if` - конструкция, состоящая из вложенных операторов `if`:

```
if (<условие>)
    оператор1;
else if (<условие>)
    оператор2;
else if (<условие>)
    оператор3;
.
.
.
else
    оператор n;
```

Условные выражения оцениваются сверху вниз и, как только найдено условие, принимающее значение `true`, выполняется связанный с этим условием оператор, а остальная часть цепочки пропускается. Если ни одно из условий не принимает значение `true`, то выполняется последний оператор `else`, который можно рассматривать как оператор по умолчанию. Если же последний оператор `else` отсутствует, а все условные выражения принимают значение `false`, то программа не выполняет никаких действий.

Фрагмент кода Java, приведенный ниже, демонстрирует применение оператора `else if` при расчете баллов:

```
int testscore = 76;
char grade;
if (testscore >= 90) {
     grade = 'A';
} else if (testscore >= 80) {
     grade = 'B';
} else if (testscore >= 70) {
     grade = 'C';
```

```
} else if (testscore >= 60) {
     grade = 'D';
} else {
     grade = 'F';
}
System.out.println("Grade = " + grade);
```
Результат: `Grade = C`

Оператор множественного выбора Switch

В отличие от утверждений `if`, и `if-else`, утверждение `switch` может иметь несколько возможных путей выполнения. Переключатель работает с примитивными типами `byte, short, char и int`. Он также работает с типом `Enum` (обсуждается в разделе `Enum`), классом `String` и несколькими специальными классами, которые заключают определенные примитивные типы: `Character, Byte, Short и Integer`. Оператор множественного выбора `switch` позволяет выполнять различные фрагменты программы в зависимости от того, какое значение будет иметь некоторая целочисленная переменной (её называют «переменной-переключателем», а «switch» с английского переводится как раз как «переключатель»). Схема инструкции имеет следующий вид:

```
switch (<переключатель>)  {
  case значение1:
    оператор1;
    break;
  case значение2:
    оператор2;
    break;
  …
  [default:
    оператор_по_умолчанию;]
}
```

В представленной выше конструкции:

`переключатель` — это переменная или выражение, имеющее определенный ранее возможный тип;

`значение1, значение2, …` - это соответствующие литералы, с которыми будет сравниваться значение переключателя. Если переключатель равен `значениюN`, то программа будет выполняться со строки, следующей за оператором `case значениеN:` и до ближайшего встреченного оператора `break`, либо до конца блока `switch` (если оператор `break` не встретится);

default: - это метка оператора, после которой будут выполняться в том случае, если ни одно из значений не совпало с переключателем. Метка default - необязательная: её можно не включать в блок switch меток или не выполнять после неё никаких команд;

операторN – простой или составной оператор. В случае составного оператора, несколько утверждений языка не обязательно объединять в блок, можно их просто написать друг за другом, разделяя с помощью «;» (и начиная новые строки для удобства).

Ниже представлен код примера использования оператора switch:

```
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 случае x=2 выводится сообщение "Равно 1 или 2", поскольку именно это же сообщение будет выведено при x=1. Дело в том, что код будет выполняться со строки, следующей за case 1 до ближайшего встреченного break, исходя из описания выполнения команды. Аналогичные комментарии к значениям x=3 и x=4.

Еще один пример с использованием в качестве переключателя типа String выводит номер месяца в соответствии с его наименованием, обращенным в нижний регистр с помощью мнтода toLowerCase():

```
public class StringSwitch {
      public static void main(String[] args) {
            String month = "August";
            int monthNumber = 0;
            switch (month.toLowerCase()) {
                  case "january":  monthNumber = 1;break;
                  case "february": monthNumber = 2;break;
                  case "march":    monthNumber = 3;break;
                  case "april":    monthNumber = 4;break;
                  case "may":      monthNumber = 5;break;
                  case "june":     monthNumber = 6;break;
                  case "july":     monthNumber = 7;break;
                  case "august":   monthNumber = 8;break;
```

```
                    case "september":monthNumber = 9;break;
                    case "october":  monthNumber = 10;break;
                    case "november": monthNumber = 11;break;
                    case "december": monthNumber = 12;break;
                         default:        monthNumber =
0;break;
                }
                System.out.println("Номер месяца "+month+" "
                                   +monthNumber);
            }
        }
```

Организация циклов

В дополнение к возможности применять условия ветвления программ и получать разные результаты при разных сценариях `if/else` иногда необходимо просто повторять некоторые операции многократно, пока работа не завершится. Мы уже познакомились с простой конструкцией организации цикла, теперь более подробно рассмотрим конструкции, используемые для циклического выполнения кода: циклы `for` и циклы `while`.

Цикл – это программная конструкция, которая выполняется многократно, пока справедливо некоторое условие (или набор условий). Например, можно заставить программу читать все записи до конца файла или перебирать все элементы массива, обрабатывая каждый из них. (Более подробно массивы рассматриваются в разделе, рассматривающем коллекции Java позже в настоящем пособии.)

Циклы `for`

Основной конструкцией цикла в языке Java является оператор `for`, который позволяет перебирать диапазон значений, определяя, сколько раз нужно выполнить цикл. Формат оператора цикла `for` следующий:

```
for (<инициализация>; <условие>; <итерация>) {
   //тело цикла, т.е. действия повторяемые циклично
}
```

В начале цикла выполняется оператор инициализации (несколько операторов инициализации могут следовать через запятую). В первом параметре обычно выбирают какую-то переменную, с помощью которой будет подсчитываться количество повторений цикла. Её называют счетчиком. Этой переменной задают некоторое начальное значение, (указывают, начиная с какого значения счетчик будет изменяться, обычно - 0).

Во втором параметре указывают некоторое ограничение на счётчик (указывают, до какого значения он будет изменяться). Цикл выполняется до тех пор, пока истинно выражение *условие* (условное выражение Java, которое может быть истинным или ложным).

Выражение *итерация* выполняется в конце цикла (несколько операторов в составе *итерация* могут следовать через запятую). В третьем параметре указывают выражение, изменяющее счётчик после каждого шага цикла. Обычно это инкремент или декремент счетчика, но можно использовать любые выражения. Ниже рассмотрен ряд примеров применения циклов `for`:

Следующий фрагмент кода выводит на экран числа от 1 до 100:

```
for (int i = 1; i <= 100; i++) {
        System.out.print(i + " ");
}
```

Следующий фрагмент кода выводит на экран числа от 10 до −10:

```
for (int s = 10; s > -11; s--) {
        System.out.print(s + " ");
}
```

Следующий фрагмент кода выводит на экран нечётные числа от 1 до 33:

```
for (int i = 1; i <= 33; i += 2) {
        System.out.print(i + " ");
}
```

Представленный фрагмент кода вычисляет и выводит сумму чётных элементов последовательности 2, 4, 6, 8,… 98, 100.

```
int sum = 0; // Переменная для суммирования
for (int s = 2; s <= 100; s+=2) {
        sum = sum + s;
}
System.out.println(sum);
```

Также существует очень удобная альтернатива организации цикла `for` для классов, которые реализуют интерфейс `iterable` - итерабельный (пока не будем вдаваться в подробности), такой как массивы и ряд других классов Java, которые мы будем рассматривать позже). В таких классах можно, начиная с первого, перебирать элементы друг за другом до конца, например, массива, используя следующий специальный синтаксис:

```
for (objectType varName : collectionReference) {
// Начинаем сразу обращаться к элементам ObjectType через
объектную переменную varName... }
```

Фрагмент кода ниже создает массив `friends` и выводит все его элементы на экран:

```
String [] friends = {"Masha","Sasha","Sergey", "Petr"};
```

```
for (String s:friends)
    System.out.println("I love " + s);
```

Фрагмент кода создает объектную переменную массива `friends`, и объявленная переменная `s` типа `String` внутри цикла `for` "пробегает" все значения массива `friends`, представляя всякий раз следующий элемент массива. Не правда-ли, удобно?

Циклы while

Циклы `while` предназначены для повторения каких-либо действий до тех пор, пока выполняется некоторое условие. При этом не известно, сколько итераций должно быть выполнено. Синтаксис цикла `while` следующий:

```
while (<условие>) {
//тело цикла, т.е. повторяемые действия }
```

Такие циклы присутствуют во всех языках программирования и выполняются до тех пор, пока выражение *условие* истинно. Условие, определяющее будет ли цикл повторяться снова, проверяется перед каждым шагом цикла, в том числе перед самым первым. Если *условие* истинно, то цикл продолжается. Не исключено, что цикл `while` не выполнится ни разу, если его условное выражение сразу оказалось `false`, или цикл будет выполняться бесконечно, если условие всегда будет `true`. Ниже рассмотрен ряд примеров применения циклов `while`:

Цикл выполняется 4 раза, и на экран выводится «1 2 3 4 »:

```
int i = 1;
while (i < 5) {
        System.out.print(i + " ");
        i++;
}
```

Цикл не выполнится ни разу и на экран ничего не выводится, поскольку условие цикла не выполняется сразу:

```
int i = 1;
while (i < 0) {
        System.out.print(i + " ");
        i++;
}
```

Цикл выполняется бесконечно, поскольку условие истинно всегда, а на экран выводится «1 2 3 4 5 6 7 и т.д.»:

```
int i = 1;
while (true) {
        System.out.print(i + " ");
        i++;
}
```

Как видно, цикл `while` требует несколько больше работы, чем цикл `for`. Требуется инициализировать переменную `i` и изменять ее внутри цикла.

Когда необходим цикл, который должен обязательно выполниться хотя бы один раз, а затем проверяется условное выражение, используется цикл `do-while` следующего синтаксиса:

```
do {
//тело цикла, т.е. действия повторяемые циклично
}while (условие);
```

Условное выражение `условие` не проверяется до конца цикла.

Ниже рассмотрен ряд примеров применения циклов `do-while`:

Цикл выполняется 4 раза, а на экран выводится «2 3 4 5 »:

```
int i = 1;
do {
        i++;
        System.out.print(i + " ");
} while (i < 5);
```

Цикл выполняется 1 раз, а на экран выводится «2»:

```
int i = 1;
do {
        i++;
        System.out.print(i + " ");
} while (i < 0);
```

Завершение цикла

Иногда бывает необходимо завершить работу цикла до того, как условное выражение окажется ложным. Например, если вы ищите конкретное значение в массиве `String`, то при обнаружении этого значения нет необходимости продолжать обрабатывать оставшиеся элементы массива. Цикл можно завершить досрочно, если внутри тела цикла поместить оператор `break`. При этом происходит моментальный выход из цикла, не будет закончен даже текущий шаг (т. е. если после оператора `break` в теле цикла присутствовали какие-то ещё операторы, то они не выполняются). Оператор `break` ведет к следующему исполняемому оператору, расположенному вне цикла, в котором он находится. Код ниже показывает, как завершается цикл при достижении элемента массива со значением "Sasha":

```
String [] friends = {"Masha", "Sasha", "Sergey",
"Petr"};
for (String s:friends) {
     System.out.println("I love " + s);
     if (s=="Sasha") break;}
}
```

Продолжение цикла

Также бывает необходимо возобновить следующий цикл до выполнения оставшихся в теле цикла операторов. Например, если вам не требуется выполнять полный цикл обработки при выполнении каких-либо условий. Цикл можно возобновить досрочно, если внутри тела цикла использовать оператор `continue`. При этом происходит возврат к заголовку цикла, и возобновляется следующая итерация, начиная с проверки условия завершения цикла. Оставшаяся часть тела цикла после оператора `continue` игнорируется.

В коде ниже не выполняется вывод сообщения остатка цикла (оператора `System.out.println..`) и продолжается выполнение цикла. В результате не выводится сообщение для элементов массива со значениями `"Sasha"` и `"Sergey"`.

```
String[] friends = {"Masha","Sasha","Sergey","Petr"};
for (String s : friends) {
     if (s == "Sasha" | s == "Sergey")
          continue;
          System.out.println("I love " + s);
}
```

Операторы ввода данных

Очевидно, что в составе библиотек (фреймворков) Java имеется полный набор средств для организации ввода и вывода данных как командного, так и графического взаимодействия. Мы рассмотрим простейшие из них, с тем, чтобы расширить возможности примеров программирования, применяя взаимодействие программы с пользователем и обеспечивая ввод данных в процессе выполнения программы.

При использовании процедур ввода/вывода следует иметь в виду, что язык Java предусматривает стандартный поток ввода (с клавиатуры) (`static InputStream`) в Java представлен классом - `System.in`, а стандартный поток вывода (на дисплей) (`static PrintStream`)- уже знакомым клвссом `System.out`. Имеется также стандартный поток для вывода ошибок (`static PrintStream`) - `System.err`, в который можно помещать сообщения об ошибках. Пока о представленных модификаторах потоков `static` будем думать как о возможности использовать методы класса без создания соответствующих объектов - это существенно и будет обсуждаться позднее в пособии. Класс `System` входит в класс `java.lang`, который доступен без использования утверждений импортирования классов.

Для ввода данных может быть использован класс `java.util.Scanner` из библиотеки классов Java, который необходимо импортировать в программе. В классе содержатся методы для чтения очередного значения заданного типа из стандартного потока ввода (клавиатуры), а также для проверки существования такого значения.

Для работы с потоком ввода создается объект класса `Scanner`, для которого указывается, с каким потоком ввода он будет связан. Для создания объекта объект класса `Scanner` обычно применяется утверждение вида: `Scanner sc = new Scanner(System.in);`

Ввод данных с клавиатуры заканчивается нажатием на клавишу <Enter>. Рассмотрим простой пример ввода целого числа с клавиатуры - стандартный поток ввода данных, и вывода квадрата введенного значения на экран - стандартный поток вывода данных.

```
import java.util.Scanner; // импортируем класс
public class InputInt {
public static void main(String[] args) {
     Scanner sc = new Scanner(System.in); // создаём
объект                                    // класса
Scanner
     int i = 2;
     System.out.print("Введите целое число: ");
     if (sc.hasNextInt()) { // возвращает true если из
                            // потока ввода можно
                            // считать целое число
          i = sc.nextInt(); // считывает целое число из
                 // потока ввода и сохраняет в переменной
          System.out.println("Квадрат введенного числа
="+ i*i);
     } else {
          System.out.println("Вы ввели не целое чис-
ло");
          }
     }
}
```

Аналогично можно было бы использовать метод `hasNextDouble()` объекта класса `Scanner` для проверки возможности преобразования потока ввода в вещественное число типа `double`, и метод `nextDouble()` для считывания значения. Если попытаться считать значение без предварительной проверки введенного типа, то во время исполнения программы может возникнуть исключение (`exception`) - ошибка во время выполнения программы, в случае, если с клавиатуры будет введена последовательность символов, которая не

может быть преобразована в соответстующий тип, например, нецифровые символы при вводе целого числа.

Имеется также метод `nextLine()`, позволяющий считывать последовательность символов в объект типа `String`. В следующем примере создаётся два объекта типа `String`, которым присваиваются введенные пользователем строки, и затем на экран выводится их конкатенация.

```
import java.util.Scanner;
public class Main {
     public static void main(String[] args) {
          Scanner sc = new Scanner(System.in);
          String s1, s2;
          s1 = sc.nextLine();
          s2 = sc.nextLine();
          System.out.println(s1 + s2);
     }
}
```

Для преобразования строки в тот или иной тип значения существуют соответствующие методы, в частности:

```
Integer.parseInt(String s); // возвращает int целое
Float.parseFloat String s); // возвращает float
Double.parseDouble(String s);// возвращает double
```

При использовании этих методов могут также возникнуть ошибки выполнения в случае невозможности соответствующего преобразования.

Упражнение 1. Написать программу с именем KeyboardScanner, которая предлагает пользователю ввести `int`, `double` и `String`. Диалог програамы должен выглядеть следующим образом (ввод выделен полужирным шрифтом):

```
Введите целое число: 12
Введите вещественное число: 33.44
Введите Ваше имя: Anatoly
Привет, Anatoly, сумма 12 и 33.44 = 45.44
```

Шаблон для программы `KeyboardScanner` приведен ниже:

```
import java.util.Scanner;  // для использования  Scan-
ner
public class KeyboardScanner {
   public static void main(String[] args) {
      int num1;
      double num2;
      String name;
      double sum;
```

```
            // Создание объекта in класса Scanner для обработки клавиатуры (System.in)
            Scanner in = new Scanner(System.in);
            System.out.print("Введите целое число: ");
            num1 = in.nextInt(); // nextInt() для int
            System.out.print("Введите вещественное число: ");
            num2 = in.nextDouble();  // nextDouble() для double
            System.out.print("Введите Ваше имя: ");
            name = in.next();         // next() для String

            // Организация вывода
            ......
        }
    }
```

Рассмотрим пример программы с именем `CheckPassFail`, которая в цикле вводит от пользователя целое (`int`) число в переменную `mark` и выводит на экран символы "`PASS`", если значение переменной `mark` больше либо равно 60; или выводит "`FAIL`", в противном случае. Условием завершения ввода оценок является ввод нуля или отрицательной величины. Если введена нечисловая величина, то выводится сообщение об ошибке формата ввода и предлагается повторить ввод. По окончании ввода данных вычисляется и выводится средний балл по введенным данным. Программа, реализующая постановку задачи, приведена ниже:

```
import java.util.Scanner;
public class CheckPassFail {
     public static void main(String[] args) {
          int mark, sm = 0, count = 0;
          Scanner sc = new Scanner(System.in);
          // объект класса Scanner
          while (true) {
               System.out.print("Введите оценку : ");
               if (sc.hasNextInt()) { // возвращает true
                         // если из потока ввода можно
                         // извлечь целое число
                    mark = sc.nextInt();
               // считываем и сохраняем в переменной mark
                    if (mark > 0) {
                         count += 1;
// увеличиваем счетчик оценок
                         sm += mark;
// увеличиваем сумму оценок
                         if (mark >= 60)
                              System.out.println("Pass = " +
mark);
```

```
                        else
                            System.out.println("Fail = " +
mark);
                    } else {
                        System.out.println("Средняя = " + sm
/ count);
                        break;
                    }
                } else {
                    System.out.println("Ошибка формата
ввода");
                    sc.nextLine(); // Очищаем поток
                    continue;
                }
            }
        }
}
```

Упражнение 2. Разработать программу с именем `CheckOddEven`, которая в цикле вводит от пользователя `int` значение, присваивает его переменной `number` и выводит на экран "Нечетное число", если значение нечетное, и “Четное число”, в противном случае. Окончание работы цикла происходит при вводе строки "end". Если введена нечисловая величина, отличная от "end", то выводится сообщение "Ошибка формата ввода" и предлагается повторить ввод. По окончании ввода данных выводится количество подсчитанных нечетных и четных введенных чисел.

Замечание: для определения четности использовать операцию (`n % 2`).

Упражнение 3. Написать программу с именем `PrintNumberInWord`, которая вводит от пользователя целое число, выделяет составляющие число символы, и выводит на экран соответствующие символам строки `"Один", "Два",... , "Девять", "Другое"`, если значение `char` переменной `c - 1, 2,... , 9`, или другое, соответственно. Использовать "вложенные-else-if" утверждения .

Для ввода можно использовать следующий фрагмент кода:

```
Scanner in = new Scanner(System.in);
String input;
System.out.print("Enter an integer: ");
    input = in.next();    // next для ввода строки
```

В цикле для каждого символа следует вывести его символьное выражение. Для переменной типа `String` с именем `input`, можно использовать метод `input.length()` для получения длины `String`; и

input.charAt(index) для получения char в позиции index, где index начинается с 0.

```
public class PrintNumberInWord {
   public static void main(String[] args) {
      String input; char c;
     // Сюда вставить ввод числа
     // в цикле выделяется символ с из строки
    // Использование вложенного - if
     if (c == '1') {
        System.out.println("Один");
     } else if (......) {
        ......
     } else if (......) {
        ......
        ......
     } else {
        ......
     }
}
```

Концепции объектно-ориентированного программирования

В данном разделе детально рассматривается понятие классов и объектов, с тем, чтобы в дальнейшем не ограничиваться примитивными типами данных в контексте рассмотрения возможностей программирования на языке Java.

Классы и объекты

Поскольку Java является объектно-ориентированным языком, то он поддерживает объектные типы данных, и содержит языковые конструкции для представления классов и объектов.

Для тех, кто не имел дела с объектно-ориентированным языком, его концепции на первый взгляд могут показаться необычными. Этот раздел представляет собой краткое введение в концепции языка ООП с использованием для сравнения методов структурного программирования.

Языки структурного программирования, такие как C и Pascal, следуют совсем иной парадигме программирования, чем объектно-ориентированные языки. Парадигма программирования определяет совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий алгоритмизацию и структурирование вычис-

лений, выполняемых компьютером. Структурное программирование - методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры модулей, в которой одни модули вызывают другие модули и передают набор параметров. Структурное программирование в значительной степени ориентировано на данные, и это означает, что, с одной стороны, есть структуры данных и имеются программы, воздействующие на эти данные. Структурное программирование придерживается принципов модульности, нисходящего и пошагового проектирования программ, одновременного проектирования программ и структур данных.

Нисходящий подход предполагает первоначальное определение целей и задач программы, проектирование ведется методом "сверху вниз", от общего к деталям. Решения задачи разбивается на несколько более простых частей. В виде блок-схемы определяют головную и подчиненные подзадачи и связи между ними, т. е. интерфейс взаимодействия между программными модулями системы. При этом технология структурного проектирования программ не регламентирует свойства данных и, прежде всего, уделяет внимание разработке алгоритма, в котором процедуры (функции) являются ведущими в этой связке: как правило, функции вызывают функции, передавая данные друг другу по цепочке.

В объектно-ориентированных языках, таких как Java, данные и код программы (логика программы) объединяются в классы и объекты. В технологии ООП взаимоотношение данных и алгоритма обработки имеют более систематический характер:

- Класс (базовое понятие этой технологии) объединяет в себе данные (структурированные переменные) и методы (функции).
- Принципиально отличается порядок взаимодействия функций и данных. Метод (функция), вызываемый для одного объекта, как правило, не вызывает другую функцию непосредственно. Для этого он должен получить доступ к другому объекту либо классу (создать, получить указатель, использовать внутренний объект в текущем объекте и т.д.), после чего он может вызвать один из известных методов.

Таким образом, структура программы определяется взаимодействием различных объектов и классов между собой. Как правило, имеет место иерархия классов, а технология ООП иначе может быть названа как программирование "от класса к классу".

Класс - это шаблон поведения объектов определенного типа с заданными атрибутами, определяющими состояние объектов. Все экземпляры одного класса (объекты, порожденные от одного класса) имеют

один и тот же набор свойств (атрибутов) и типичное поведение, то есть одинаково реагируют на одинаковые сообщения (вызовы методов). Концептуально, объект - это мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной в предметной области. С точки зрения программирования объект является автономным модулем со своими атрибутами и поведением, и представляет модель объекта предметной области.

Каждая программа Java имеет, по крайней мере, один класс, который знает, как выполнить определенные действия самостоятельно, и как использовать методы некоторого другого класса или объекта для выполнения других действий. Например, самый простой класс, `HelloWorld`, знает, как сделать приветствие миру, обращаясь с методу `println()` класса `out`, находящегося в составе другого класса `System`.

Классы в Java могут иметь собственные методы (функции) и поля (известные также как атрибуты или свойства), которые определяют ответственность класса и его состояние, соответственно. Давайте создадим и рассмотрим класс с именем `Car`. Этот класс, будучи объектной моделью автомобиля, будет иметь методы, описывающие, что этот тип транспортного средства может сделать, например, запустить двигатель, выключить его, ускорить, затормозить, запереть двери, и так далее.

Этот класс также будет иметь некоторые атрибуты: цвет кузова, количество дверей, цена и так далее.

```
class Car{
     String color;
     int numberOfDoors;
     void startEngine {
          // Здесь находится некоторый код
     }
     void stopEngine {
          int tempCounter=0;
          // Здесь находится некоторый код
     }
}
```

Класс `Car` представляет общую модель для различных автомобилей: все автомобили имеют такие свойства, как цвет и количество дверей, и все они выполняют аналогичные действия. Если возникает необходимость иметь дело с более конкретным автомобилем, то можно создать другой класс Java с именем, например, `JamesBondCar`. Он попрежнему остается автомобилем, наследуя свойства класса `Car`, но с некоторыми дополнительными свойствами, характерными для модели Джеймса Бонда. Можно сказать, что класс `JamesBondCar` является

подклассом Car, или, используя синтаксис Java, JamesBondCar расширяет класс Car.

```
class JamesBondCar extends Car{
     int currentSubmergeDepth;
     boolean isGunOnBoard=true;
     final String MANUFACTURER;
     void depth { // Глубина погружения
          currentSubmergeDepth = 50;
          // Здесь находится некоторый код
     }
     void surface { // Внешний вид
          // Здесь находится некоторый код
     }
}
```

Но даже после определения всех свойств и методов для класса JamesBondCar вы не можете управлять им, даже на экране компьютера. Класс Java, как чертеж в строительстве или машиностроении, и, до тех пор, пока не будут созданы реальные объекты, основанные на этом чертеже, вы не сможете их использовать.

Создание объектов, как экземпляров на основе классов эквивалентно сборки реальных автомобилей по чертежам. Создание экземпляра класса означает создание объекта в памяти компьютера на основе определения класса. Чтобы создать экземпляр класса (установить еще один автомобиль в гараже), объявляется переменная этого типа класса, и используется оператор new для каждого нового экземпляра автомобиля:

```
JamesBondCar carFirst= new JamesBondCar();
JamesBondCar carSecond = new JamesBondCar();
```

Теперь переменные carFirst и carSecond используются для обозначения первого и второго экземпляра JamesBondCar, соответственно. Чтобы быть точным, объявлять переменные, указывающие на экземпляры класса необходимо, если вы планируете обращаться к этим экземплярам в программе. Утверждение new JamesBondCar() тоже создает экземпляр, который можно использовать в контексте другого утверждения Java, не присваивая экземпляру имя. Вы можете создать несколько автомобилей, на основе одной и той же спецификации. Даже если все они представляют собой один и тот же класс, они могут иметь разные значения своих свойств - некоторые из них красные, некоторые из них имеют две двери, а другие четыре, и т.д.

Таким образом, как только определен некоторый класс, программист может создавать сколько угодно объектов этого класса, или, как их

еще называют, экземпляров (class instance) класса и манипулировать ими так, как будто они представляют собой элементы решаемой задачи.

В итоге можно определить принципы программирования на Java:

1. Всё является объектом - Все данные и программы хранятся в объектах.
2. Каждый объект создается (имеются средства для создания объектов), существует (живет) какое-то время, потом уничтожается (автоматически).
```
Car car = new Car();     // Создание объекта
                         // класса Car
```
3. Программа представляет собой набор классов и объектов, взаимодействующих друг с другом. Кроме того, что объект хранит какие-то данные, он умеет выполнять различные операции над своими данными и возвращать результаты этих операций. Теоретически эти операции выполняются как реакция на получение некоторого сообщения данным объектом. Практически, это происходит при вызове метода данного объекта, например, следующим образом: `car.startEngine()`;
4. Каждый класс и объект имеет свою память, состоящую из других объектов и/или примитивных данных.
5. Каждый объект имеет свой предопределенный тип (класс). Все объекты одного типа могут выполнять один и тот же набор методов. Т.е. в ООП не имеется возможности создания произвольного объекта, состоящего из того, например, что мы укажем в момент его создания.

Родительские и дочерние объекты

Родительский объект служит в качестве структурной основы для получения более сложных *дочерних объектов*. Дочерний объект повторяет родительский, но является более специализированным. Объектно-ориентированная парадигма позволяет многократно использовать общие атрибуты и поведение родительского объекта, добавляя к ним новые атрибуты и поведение дочерних объектов. (Подробнее о *наследовании* в последующих разделах настоящего пособия.)

Связь между объектами и управление - Model-View-Controller

Объекты взаимодействуют друг с другом, отправляя сообщения (на языке Java *вызывают методы* - *methods call*). При этом, в объектно-ориентированных приложениях имеется класс (программа), который управляет взаимодействием между объектами в рамках решения задачи

в контексте данной предметной области и выполняет функции управляющей программы. Одним из широко используемых шаблонов (pattern) проектирования систем является Model-View-Controller (MVC, «модель-представление-поведение», «модель-представление-контроллер», «модель-вид-контроллер»). Шаблон проектирования или паттерн (англ. design pattern) - повторяющаяся архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

В рамках шаблона MVC модель данных приложения, пользовательский интерфейс и управляющая логика разделены на три отдельных компонента таким образом, чтобы модификация одного из компонентов оказывала минимальное воздействие на остальные. Данная концепция проектирования часто используется для построения архитектурного каркаса при переходе к реализации в конкретной предметной области. На Рис. 27 представлена схема MVC.

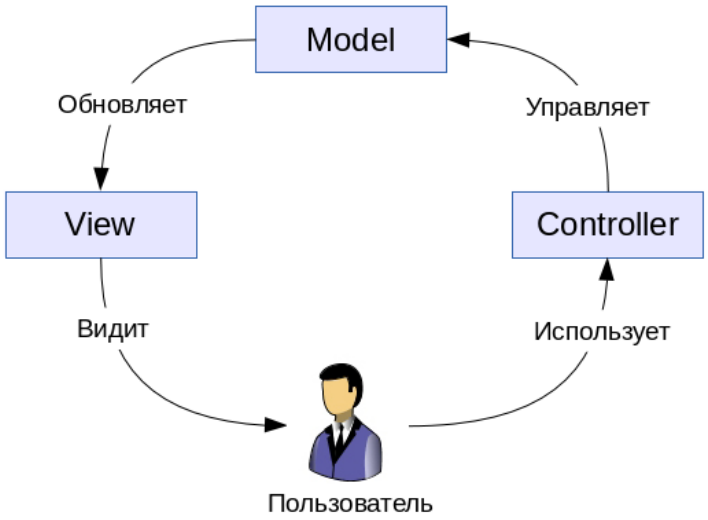


Рис.27. Схема MVC

Концепция MVC позволяет разделить данные, представление и обработку действий пользователя на три отдельных компонента:

- Модель (Model). Модель представляет знания: данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Не содержит информации, как эти знания можно визуализировать.
- Представление, вид (View). Отвечает за отображение информации (визуализацию). Часто в качестве представления выступает форма (диалоговое окно) с графическими элементами (Поля, кнопки и др.).

- Контроллер (Controller). Обеспечивает связь между пользователем и системой: контролирует ввод данных пользователем и использует модель и представление для реализации необходимой реакции.

Важно отметить, что как представление, так и контроллер зависят от модели. Однако модель не зависит ни от представления, ни от контроллера. Тем самым достигается назначение такого разделения, что имеется возможность строить модель независимо от визуального представления, а также создавать несколько различных представлений для одной модели (таблицы, графики,..).

За счет разделения бизнес-логики (модели) от её визуализации (представления, вида) повышается возможность повторного использования. Наиболее полезно применение данной концепции в тех случаях, когда пользователь должен видеть те же самые данные одновременно в различных контекстах и/или с различных точек зрения.

При этом разработчики могут специализироваться только в одной из областей: либо разрабатывают графический интерфейс, либо разрабатывают бизнес-логику. Поэтому возможно добиться того, что программисты, занимающиеся разработкой бизнес-логики (модели), вообще не будут осведомлены о том, какое представление будет использоваться.

Этапы разработки ОО приложения

Объектно-ориентированный подход к программированию включает в себя 3 этапа жизненного цикла:

- Объектно-ориентированный анализ (OOA). Определяется функциональность системы, которую приложение должно выполнять для достижения требований пользователей. Кроме того, на этом этапе (1) определяется набор объектов, из которых будет состоять система и описывается взаимодействие и связи между различными объектами (такое взаимодействие принимает форму сообщений между объектами); (2) описываются внутренние процессы, которые происходят внутри каждого объекта в ответ на сообщения от других объектов и инициация сообщений другим объектам
- Объектно-ориентированное проектирование (OOD). Формируется архитектура классов, эффективно обеспечивающая выявленную на этапе анализа, функциональность приложения. Строго говоря, граница между OOA и OOD не определена, и применяется итеративный пошаговый процесс к раз-

работке. Модели строятся на этапе анализа и улучшаются на этапе проектирования.

- Объектно-ориентированное программирование (ООР). Определяется действительная реализация приложения средствами языка Java.

Задача создания системы, обеспечивающей эффективное удовлетворение потребностей пользователей в рамках объектно-ориентированной парадигмы, является нетривиальной, и в этой работе участвуют различные специалисты, в том числе, будущие пользователи системы. Как правило, для документирования и обсуждения решений на каждом этапе жизненного цикла создаются различные модели, которые поддерживаются современными Computer Aid Software Engeneering (CASE) - технологиями. Следует отметить, что два коллектива профессионалов в области объектно-ориентированного подхода создадут различные архитектуры классов, для решения одной и той же задачи.

Стандартом по "де-факто" для разработки моделей-диаграмм на всех этапах жизненного цикла создания объектно-ориентированной системы является Unified Modeling Language (UML). UML позволяет разрабатывать 18 различных диаграмм, начиная от Use Case (диаграмма вариантов использования), заканчивая Deployment Diagram (диаграмма развертывания). На Рис. 28 представлена структура типов UML - диаграмм, которые можно использовать в процессе создания объектно-ориентированной системы.

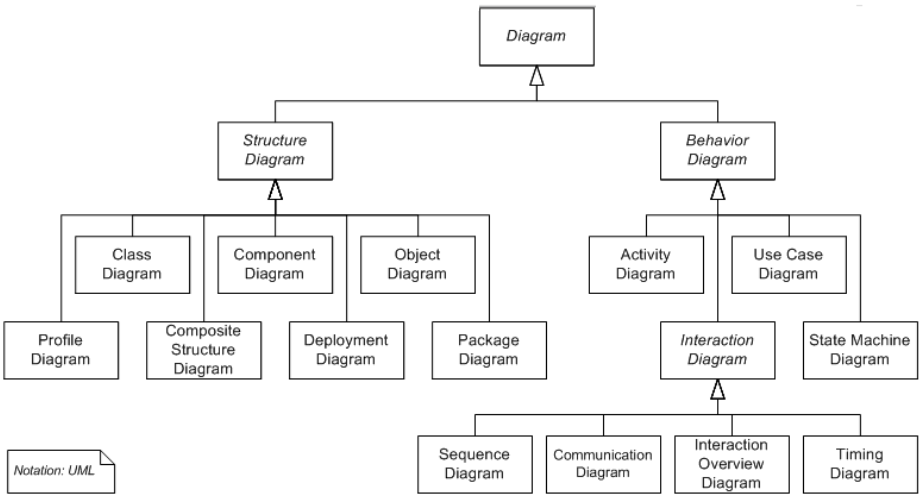


Рис. 28. Структура типов UML - диаграмм.

Вопросы детального рассмотрения моделей выходят за рамки настоящего пособия, и в качестве примера рассмотрим диаграмму классов (Class Diagram), состоящую из двух классов (Customer и CustomerDriver), которая представлена на Рис. 29.

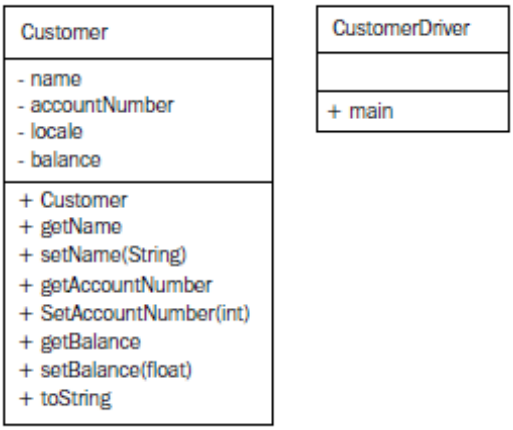


Рис. 29. Пример диаграммы классов UML

На Рис. 29 каждый класс представляется в виде прямоугольника, разделенного на три части. В верхней части указывается имя класса, в средней части - список переменных (полей, атрибутов) класса, в нижней части - методы класса. Символы, предшествующие переменным и методам определяют видимость этих членов класса. Здесь используются:

• -: Private (доступны только внутри класса)

• +: Public (доступны за пределами класса)

Обычно диаграмма классов состоит из многих классов и различных соединительных линий, показывающих связи между классами, например, как показано на Рис. 30.

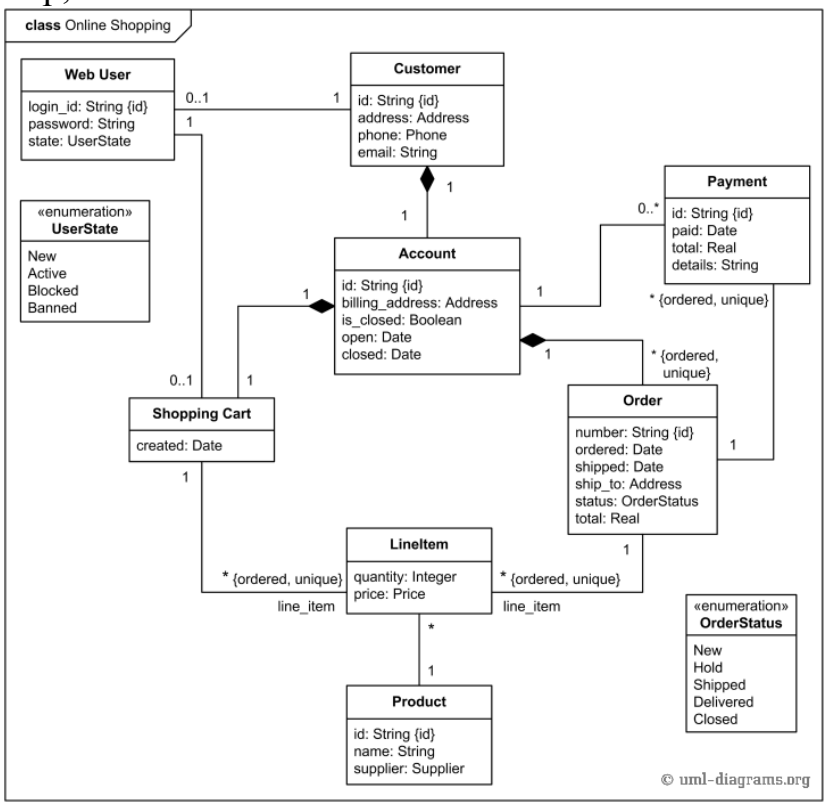


Рис. 30. Диаграмма классов для Интернет-магазина

По желанию разработчика на диаграмме классов могут указываться не все элементы (на Рис. 30 не показаны методы, реализуемые классами), вплоть до указания только наименований классов и связей между ними. На диаграмме каждый клиент (`Customer`) имеет уникальный идентификатор и связан ровно с одной учетной записью (`Account`). `Account` содержит `Shopping cart` и один или несколько классов заказов (`Order`). Клиент может зарегистрироваться в качестве веб-пользователя, чтобы иметь возможность делать покупки в Интернет. Клиент не обязан быть веб-пользователем, поскольку покупки можно также сделать по телефону или заказав по каталогам. Веб-пользователь имеет Имя (`login_id`), которое также служит в качестве уникального идентификатора. Веб-пользователь может находиться в нескольких состояниях (`UseStatus`): нового (`New`), активного (`Active`), заблокированного (`Blocked`) или запрещенного (`Banned`), и присвязаны к корзине. Корзина принадлежит к `Account`.

Более детальное рассмотрение UML выходит за рамки настоящего пособия, хотя в дальнейшем будут приводиться диаграммы классов и рассматриваться конкретные случаи их применения.

Сведения об объектах

Как уже отмечалось, ОО программа состоит из образованных классов и объектов. Объект представляет собой конкретный экземпляр на основе шаблона класса. Как мы увидим в дальнейшем понятие класса и объекта взаимозаменяемы, поскольку и класс и его объекты могут иметь переменные и методы. Количество и содержание классов является предметом проектирования и общие рекомендации по выделению классов следующие:

- класс должен имеет четкие границы, и содержать данные и методы, определяемые его предметной направленностью;
- для класса должен быть определять обозримый набор методов; Запросы (сообщения), которые программист может отправлять экземпляру класса, определяются его интерфейсом, и именно тип (класс) определяет интерфейс – набор методов, их сигнатуру и последовательность использования методов при решении конкретных задач;
- "знает" только о своих данных и любых других объектах, которые нужны для его деятельности.

По сути, объект - это дискретный модуль, который обладает только необходимыми зависимостями с другими объектами для решения собственных задач.

Теперь рассмотрим, как выглядит объект.

Пример объекта Person

Рассмотрим пример приложения, компонентом которого является физическое лицо, представляемое объектом класса `Person`.

Напомним, что объект состоит из двух основных элементов: атрибутов (данных) и методов (поведения), и определим эти элементы применительно к классу `Person`.

Атрибуты

Определим, какой перечень атрибутов (полей) может иметь физическое лицо. К число наиболее общих атрибутов можно отнести следующие: имя, возраст, рост, вес, цвет глаз, пол. В некоторой специализированной предметной области возможно понадобились бы дополнительные атрибуты, например, размер обуви, группа крови, но для начала достаточно уже перечисленных.

Поведение

Поведение (behavior) объекта представляют собой действия и реакции объекта, выраженные в терминах отклика на сообщения (message) (выполнение действий - выполнение методов) и изменения состояния - видимая извне и повторяемая активность объекта. Каждый объект уникален и характеризуется значениями набора свойств объекта (переменных или атрибутов) и это отличает его от других объектов.

Например, в объектно-ориентированной операционной системе, для каждого объекта существует определенный набор действий, которые с ним можно произвести. Например, возможные действия с некоторым файлом операционной системы ПК:

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

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

Обычно говорят, что взаимодействие между объектами в программе происходит посредством передачи сообщений между ними.

В терминологии объектно-ориентированного подхода понятия "действие", "сообщение" и "метод" являются синонимами. Т.е. выражения "выполнить действие над объектом", "вызвать метод объекта" и "послать сообщение объекту для выполнения какого-либо действия" эквивалентны. Последняя фраза появилась из следующей модели ООП.

Программу, построенную по технологии ООП, можно представить себе как виртуальное пространство, заполненное объектами, которые условно "проживают" свою жизнь. Их активность проявляется в том, что они вызывают друг у друга методы, или посылают друг другу сообщения. Внешний интерфейс объекта, или открытий набор его методов - это описание того, какие сообщения он может принимать.

Физическое лицо в контексте взаимодействующих объектов может выполнять много полезных действий, но применительно к контексту бизнес-приложения объект класса `Person` может отвечать, например, на запрос: "Сколько вам лет?", возвращая значение атрибута `"возраст"`. Естественно, объект `Person` может обеспечивать более сложное поведение, например, предоставить свое резюме, но ограничим поведение `Person` ответами на анкетные запросы:

Какое зовут?
Сколько лет?
Каков рост?
Каков вес?
Какого цвета глаза?
Какой пол?

Состояние объекта

Состояние является важным понятием в ООП. Каждый объект имеет определенное время жизни. В процессе выполнения приложения, или функционирования какой-либо реальной системы, могут создаваться новые объекты и уничтожаться уже существующие. Каждый объект имеет состояние, обладает четко определенным поведением и уникальной идентичностью (т.е. каждый объект, в принципе, отличается от ему подобных). В каждый момент времени состояние объекта характеризуется значением его атрибутов, свойств объекта в текущий момент времени (обычно статический набор, определенный в момент создания объекта).

Состояние является совокупным результатом поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно на основе значения атрибутов. Например, в системе отслеживания движением самолетов https://www.flightradar24.com/ объект САМОЛЕТ в какой-то момент

времени стоит в ангаре на обслуживании; в другой момент времени объект САМОЛЕТ выруливает на ВПП и на борту находится 152 пассажира; объект САМОЛЕТ летит на высоте 10000 метров из Петербурга в Москву,…

Применительно к объекту `Person` состояние определяется такими атрибутами, как имя, возраст, рост и вес и т.д. В случае необходимости представить список из значений нескольких атрибутов, можно использовать объект класса `String`.

Таким образом, использование понятия состояния и тип `String` позволяют запросить у объекта `Person`: "Представьте себя, предъявив перечень (как выражение `String`) значений своих атрибутов". Позже в пособии мы создадим класс `Person`, содержащий рассмотренные атрибуты и поведение.

Принципы объектно-ориентированного программирования

Одним из основополагающих принципов ООП является **абстракция (abstaction)**. В объектно-ориентированном подходе предметы и понятия реального мира заменяются моделями, т.е. определенными формальными конструкциями набора реальных объектов (сущностей) предметной области (служащие, проекты, публикации, …). Модель содержит не все признаки и свойства представляемого ею предмета или понятия, а лишь те из них, которые существенны с точки зрения разрабатываемой программной системы. Упрощение модели (абстрагирование от ненужных деталей) по отношению к реальному предмету позволяет сделать ее формальной, благодаря чему при разработке можно четко выделить все зависимости и операции над ними в создаваемой программной системе. Это упрощает как изучение (анализ) и разработку моделей, так и их реализацию на компьютере.

Другими механизмами, которые должны быть реализованы в составе ООП, являются **инкапсуляция, наследование и полиморфизм**.

Инкапсуляция (encapsulation) - это механизм, который связывает код вместе с обрабатываемыми им данными и сохраняет их в безопасности, как от внешнего влияния, так и от ошибочного использования. При этом сокращается уровень сложности программного обеспечения за счет образования вокруг объекта защитной оболочки, предохраняющей код и данные от произвольного доступа из других методов, определенных вне этого объекта. Другим термином, используемым для выражения замкнутости и защищенной природы объектов, является сокрытие деталей реализации объекта - hiding. От пользователей классов скрывается информация, не имеющая для него значения, и показывается

полезная. Таким образом, инкапсуляция скрывает реализацию класса, предоставляя внешний интерфейс (набор открытых методов), посредством которого обеспечивается и контролируется доступ к данным и коду внутри капсулы.

При проектировании классов программист должен определить, какие части объекта должны быть доступны для пользователя, а какие следует изолировать в объекте. Детали класса, остающиеся невидимыми для пользователя, называются инкапсулированными в классе.

При использовании ООП не рекомендуется применять прямой доступ к свойствам какого-либо класса из методов других классов. Для этого используется интерфейс (открытый набор методов) объекта, посредством которого осуществляется все взаимодействие с этим объектом. Для доступа к свойствам объекта принято использовать специальные методы этого класса для получения и изменения его свойств. Эти методы называются: `getter` для получения значений переменных объекта, `setter` для установки значений переменных объектов. Вместе эти методы называются `mutator` и предназначены для контроля изменения в переменных объекта.

Применительно к ООП, важно то, что объект отделяет свое состояние и поведение от внешнего мира и, как и в реальном мире, объекты, используемые в программировании, поддерживают различные типы отношений с различными категориями объектов в приложениях, в которых они используются.

На платформе Java можно использовать *модификаторы доступа* (о которых позже - есть и другие варианты, кроме здесь упомянутых), чтобы менять характер отношений объекта с *публичных* на *частные*. Публичный (public) доступ открывает доступ к переменным и методам объекта для всех прочих объектов, тогда как частный (private) доступ обозначает, что переменные и методы объекта доступны только внутри самого объекта.

Граница "public/private" поддерживает объектно-ориентированный принцип инкапсуляции. Языковые средства Java позволяют варьировать положение этой границы для каждой пары объектов в зависимости от системы доверия. Инкапсуляция — это мощная функция языка Java.

Наследование

В структурированном программировании часто копируют структуру, присваивают ей новое имя и добавляют или изменяют свойства, вследствие чего создают новую сущность (например, запись `Account`),

которая отличается от своего оригинала. Со временем такой подход порождает большое количество дублированного кода, что увеличивает проблемы сопровождения.

ООП вводит понятие *наследования*, согласно которому специализированные объекты могут "наследовать" атрибуты и поведение исходных объектов без добавления кода. Наследование опирается на инкапсуляцию. Оно позволяет строить на основе базового класса новые, добавляя в классы специальные поля данных и дополнительные методы. Исходный класс называется суперклассом(superclass), прародителем (ancestor), новые классы - его покдклассами(subclass), потомками (descendants). От потомков-подклассов, в свою очередь, можно наследовать атрибуты и методы, получая потомки-подклассы следующего уровня. Наследование вводит иерархию "общее/частное" между классами, в которой подкласс наследует от одного или нескольких более общих суперклассов их свойства и методы. Класс, стоящий на вершине иерархии, от которого унаследованы все остальные (прямо или косвенно), называется базовым классом иерархии. В Java все классы являются потомками класса `Object`. Если атрибуты или поведение некоторого класса нужно частично модифицировать, то вводится класс, который наследует родительский и переопределяет соответствующие атрибуты и методы. Переопределяемые методы (`overriding`) - это методы, которые находятся в родительском классе, так же как и в его потомке. При этом изменяются (переопределяются) только те методы, которые необходимо изменить для создания специализированного класса.

Использование наследования способствует уменьшению количества кода, созданного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода. Наследование исключительно важно, так как позволяет объектно-ориентированным программам увеличивать сложность линейно, а не геометрически, как в структурированном программировании за счет лучшей локализации изменений. Производный класс (потомок) наследует описание базового кода, делая ненужным повторную разработку и тестирование кода. Там где используется суперкласс может использоваться и класс.

Предположим, что разрабатывается приложение для отдела кадров и для нового объекта `Employee` в качестве основы хотите использовать объект `Person`. Дочерний по отношению к `Person` объект `Employee` будет иметь все атрибуты объекта `Person`, а также дополнительные атрибуты:

- Дата приема на работу,
- Размер заработной платы,

- Количество дней отпуска,
- Использованные дни отпуска.

Наследование позволяет легко создавать новый класс `Employee` без необходимости копировать весь код `Person`. Подкласс `Employee` наследует структуру и поведение своего суперкласса `Person`. На Рис. 31 представлена диаграмма классов UML, на которой обозначено отношение между классами `Person` и `Employee`, демонстрирующее наследование атрибутов и поведения с помощью иерархии классов обобщения и специализации. Наследуемые свойства явно не показаны в прямоугольнике, обозначающем подкласс `Employee`. Для описания наследования в классе `Employee` применяется ключевое слово `extends` с указанием имени суперкласса `Person`. Ниже представлены фрагменты кода для классов `Person` и `Employee`:

```
public class Person {
     private String fullName;
     private Date dateOfBirth;
     public Person() {...}
     public int age(){
          return getYear() - getYear(dateOfBirth);
     }
}
public class Employee extends Person{
     private Date dateHired;
     private int salary;
     private int leaveEntitlement;
     private int leaveTaken;
     public Employee()    {...}
     public int remainingLeave(){
               return leaveEntitlement - leaveTaken;
     }
}
```

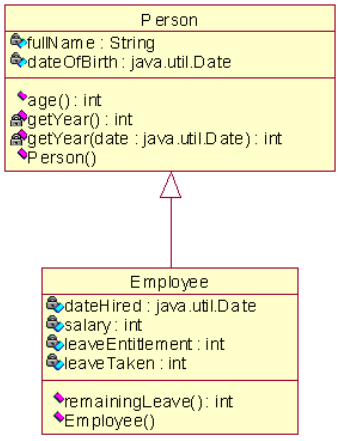


Рис. 31. Пример отношения обобщения-специализации

Полиморфизм

Полиморфизм – более сложная для понимания концепция ООП, которая опирается как на инкапсуляцию, так и на наследование. В сущности, она означает, что объекты, принадлежащие к одной той же ветви иерархии, получая одно и то же сообщение (то есть, когда им предписывается выполнять один и тот же метод), могут действовать поразному. Целью полиморфизма, применительно к ООП, является использование одного имени метода для задания общих для класса действий. При этом выполнение каждого конкретного действия будет определяться типом класса.

Зачастую метод, унаследованный подклассом, напрямую используется этим подклассом. Операция `age()` используется аналогично в объектах классов `Person` и `Employee` на Рис. 31. Однако, иногда необходимо, чтобы операции в подклассе были переопределены (overridden) в соответствии с семантическими вариациями подкласса. Например, операция `Employee.remainingLeave()` (длительность отпуска сотрудника) у разных категорий сотрудников может вычисляться различным образом. Например, сотрудник, являющийся менеджером, имеет право на ежегодное получение дополнительного отпуска. Тогда, если добавить в обобщенную иерархию класс `Manager` (как показано на Рис. 32), операция `manager.remainingLeave()` должна заместить операцию `employee.remainingLeave()` , как это показано с помощью использования одного и того же имени метода в подклассе. При этом говорят, что операция `remainingLeave()` – полиморфна, которая в зависимости от типа класса будет выполняться с помощью различного метода, имеющего одинаковое имя. Фрагмент исходного текста класса `Manager` представлен ниже:

```
public class Manager extends Employee
{
   private Date dateAppointed;
   private int leaveSupplement;
   public Manager()
   {...}
   public int remainingLeave()
   {
          int mrl;
          mrl=super.remainingLeave()+leaveSupplement;
          // super - обращение к суперклассу
          return mrl;
   }
}
```

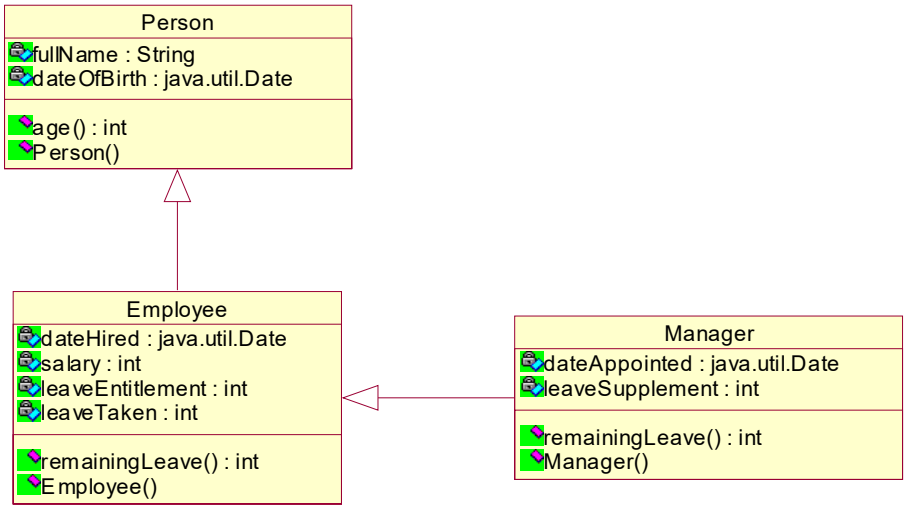


Рис. 32. Модель полиморфизма

При правильном применении полиморфизм, инкапсуляция и наследование комбинируются так, что создают некую среду программирования, которая обеспечивает намного более устойчивые масштабируемые программы. Удачно спроектированная иерархия классов является базисом для повторного используемого кода. Инкапсуляция позволяет реализациям мигрировать из проекта в проект без разрушения кода, который зависит от public-интерфейса классов. Полиморфизм позволяет создавать ясный, хорошо модифицируемый и читабельный код.

Хотя были рассмотрены принципы объектно-ориентированного программирования по отдельности, работают они вместе и практически не существуют отдельно друг от друга. Для того чтобы разработать систему, необходимо создать некоторую иерархию классов. При этом базовый класс должен быть спроектирован с учетом возможного наследования. Если базовый класс изменится, это повлечет за собой соответствующие изменения в функционировании производных классов. Понятно, что внутренняя структура как базового, так и производных классов должна быть скрыта от пользователя классов. Таким образом, совместно работают инкапсуляция и полиморфизм. Некоторые методы в производных классах могут уточняться (переопределяться). Имена этих методов будут одинаковы как в базовом классе, так и в производных классах, и, таким образом, начинают совместно работать наследование и полиморфизм.

Структура класса Java

Мы уже упоминали, что класс представляет собой дискретный модуль со своими атрибутами и поведением. Это означает, что он имеет

четкие границы и состояние и может выполнять предписанные ему действия, если к нему правильно обратиться. В каждом объектно-ориентированном языке существуют свои правила определения классов.

Ниже представлен формат определения класса, в котором полужирным шрифтом выделены ключевые слова, в треугольных скобках "<>" указываются кодируемые значения операторов, в квадратных скобках "[ ]" указываются элементы, которые не являются обязательными. Сами же квадратные скобки (в отличие от { и }) не являются частью синтаксиса Java.

```
package  <имя пакета>;
import  <импортируемые классы>;
<модификатор доступа>  class <имя класса> {
  <модификатор доступа> <тип данных> <имя переменной> [= <начальное значение>];
  <модификатор доступа> <имя класса> ([<список аргументов>]) {
    <утверрждения конструктора>
  }
  <модификатор доступа> <тип возвращаемого значения> <имя метода>([<список аргументов>]) {
    <утверрждения метода>
  }
 }
```

Рассмотрим кодирование утверждений класса более подробно.

Оператор package

Язык Java позволяет произвольно выбирать имена классов, например, `Account`, `Person` или `Car`. Такая возможность в рамках большого проекта может привести к ситуации, когда одно и то же имя класса должно выражать два разных понятия, известное как *конфликт имен*. Для разрешения таких конфликтов в языке Java используются *package (пакеты),* которые мы уже упоминали в связи с классом `HelloWorld`.

Пакет Java – это механизм организации пространства имен (namespace): ограниченной области, в которой имена уникальны, но за пределами которой, они не существуют. Чтобы определить уникальный класс, используемый в проекте, нужно указать полную его спецификацию в пространстве имен. Повторим, что особую пользу пакеты предоставляют при построении более сложных приложений из дискретных единиц функциональности.

Как уже отмечалось, для определения пакета используется ключевое слово `package`, за которым следует формальное имя пакета, закан-

чивающееся точкой с запятой. Обычно имена пакетов разделяются точками и следуют такой схеме:

package *orgType.orgName.appName.compName;*

Это наименование пакета расшифровывается следующим образом:

orgType – это тип организации, такой как `ru`, `com` или `net`.

OrgName – доменное имя организации, как, например, `ifmo`, `emc` или `ibm`;

AppName – сокращенное имя приложения, например, `session; account;`

compName – имя компонента приложения, например, `dbms; gui`.

Язык Java не заставляет следовать этому соглашению о пакетах, и можно использовать любые имена в иерархии пакетов. Более того, определять пакет вообще не обязательно, но тогда все классы должны иметь уникальные имена и будут находиться в пакете по умолчанию. Однако для более эффективной работы рекомендуется определять все свои классы Java в пакетах, и в пособии будем придерживаться этого правила.

Иерархия пакетов классов должна соответствовать иерархии каталогов файловой системы, в которых эти классы (и интерфейсы) определяются в коде. На Рис. 33 представлена иерархия подкаталогов файловой системы проекта `Hello` для оператора `package ru.ifmo.practicejava;`

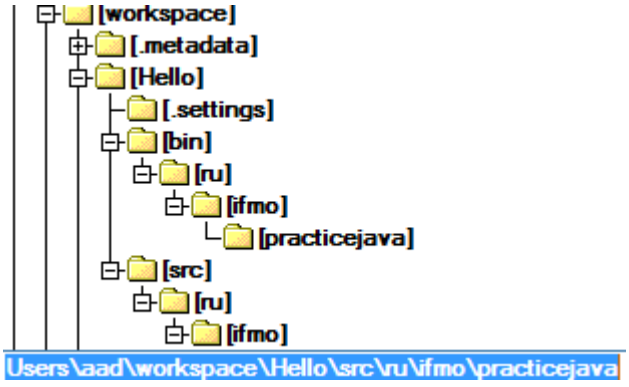


Рис. 33. Иерархия подкаталогов проекта `Hello`

Следует также отметить, что оператор `package` всегда должен располагаться первым в описании класса и любое изменение его положения приведет к ошибке компиляции. Приведет к ошибке и появление в коде класса двух операторов `package`.

Если в файле с расширением *.java* представляется несколько классов (такое возможно, если все классы имеют модификатор доступа, отличный от *public*), то все эти классы должны входить в один пакет, определенный оператором *package* в первой строке:

```
// содержимое файла Multiple.java
package ru.ifmo.practicejava;
interface Printable {
//.. Здесь находится некоторый код
}
class MyClass {
//.. Здесь находится некоторый код
}
interface Movable {
//.. Здесь находится некоторый код
}
class Car {
//.. Здесь находится некоторый код
}
```

Здесь, наряду с классом (`class`) представлен интерфейс (`interface`), о котором речь пойдет позже.

Оператор `import`

Следующим в определении класса должен следовать оператор `import`. Этот оператор информирует компилятор Java, где находятся классы, на данные или методы которых, имеется ссылка в коде класса. Любой класс использует другие классы и объекты для выполнения некоторых функций, и оператор `import` позволяет сообщить о них компилятору Java для разрешения (resolving) ссылки. Если оператор `import` не используется, тогда придется при обращении к каждому используемому классу в коде приписывать полную спецификацию имени пакета. Так, для использования класса `Date` в коде класса, придется всякий раз при обращении к классу `Date` кодировать его пакет, `java.util.Date`.

В операторе за ключевым словом `import` следуют наименование пакета класса, который нужно импортировать, с последующей точкой с запятой. Имя класса должно быть полным, и включать наименование пакета, например, `import java.util.Date;`

Чтобы импортировать все классы из пакета, после имени пакета можно закодировать `.*`. Например, следующий оператор импортирует все классы пакета `java.util`: `import java.util.*;`

JDK содержит исчерпывающий набор полезных классов в различных пакетах, например, пакеты `java.lang.String` и `java.lang` нет необходимости импортировать, поскольку они всегда используются компилятором Java по умолчанию, остальные пакеты следует импортировать.

Хорошая новость состоит в том, что при написании кода в редакторе Eclipse можно ввести имя класса, а затем нажать комбинацию клавиш Ctrl+Shift+O. Eclipse определяет, какие классы нужно импортировать, и добавляет их автоматически. Если Eclipse находит два класса с одним и тем же именем, он выводит диалоговое окно с запросом, показывающим список классов, из которых можно выбрать класс для добавления, как показано на Рис. 34.

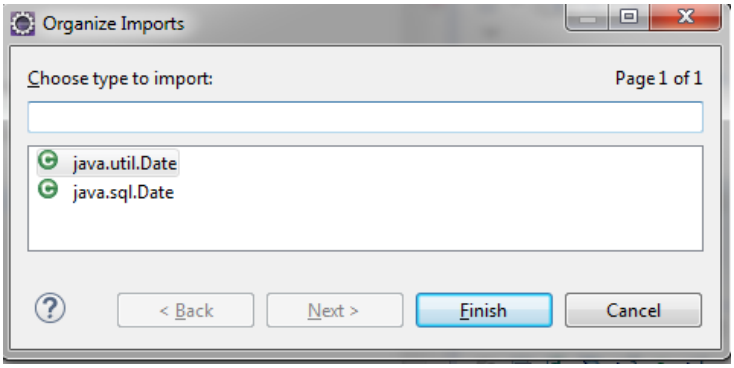


Рис 34. Подсказка для выбора класса для импортирования

В Eclipse существует еще один вариант для выбора импортируемого класса, который инициируется нажатием на красный маркер слева от введенной строки с нераспознанным именем класса. При этом открывается контекстное меню, предлагающее варианты устранения ошибки, как показано на Рис. 35.

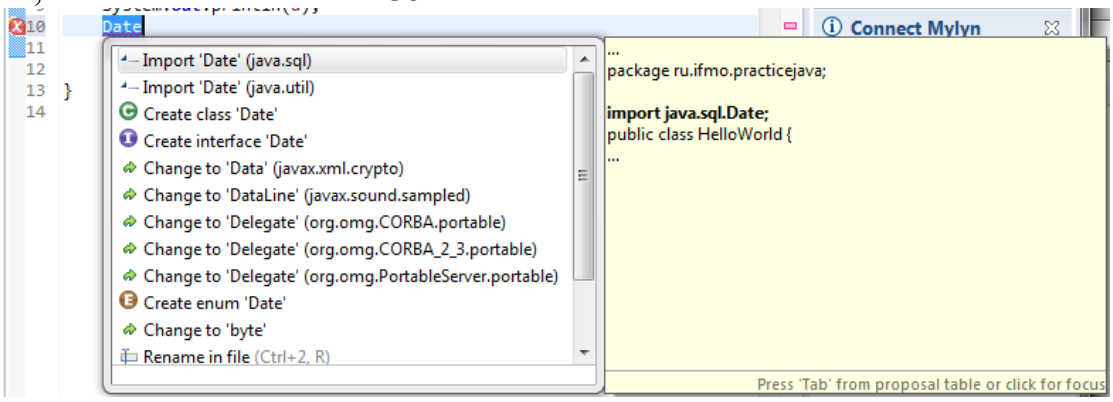


Рис. 35. Меню выбора вариантов исправления ошибки

Также можно подвести указатель мыши к подчеркнутому красной чертой идентификатору и откроется окно для выбора вариантов, как показано на Рис. 36.

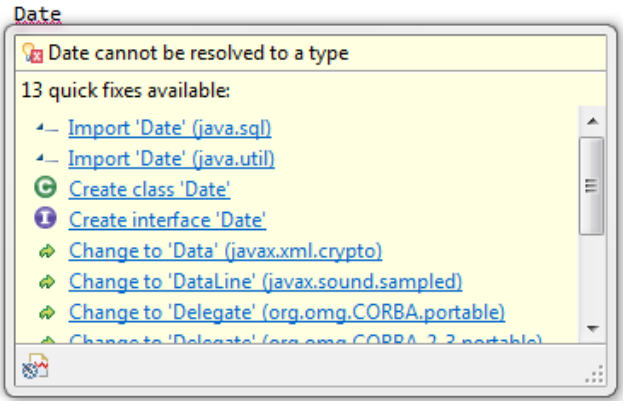


Рис. 36. Меню выбора вариантов исправления ошибки

Объявление класса

Для того чтобы определить объект в языке Java, сначала он должен быть объявлен как экземпляр класса. Класс можно рассматривать как некоторый тип или шаблон объектов, как форму для штамповки изделий. Класс определяет базовую структуру объекта, и во время выполнения программы обычно создается экземпляр этого класса. Слово `объект` часто используется как синоним слова `класс`. Как мы увидим в дальнейшем, класс может иметь переменные и методы, определяемые на уровне класса (т.н. статические переменные и методы класса, которые можно использовать без создания объекта). Строго говоря, класс содержит определенную структуру данных и методов, которые воплощается (или не воплощается) в конкретном экземпляре объекта.

Модификатор доступа касается идентификаторов классов, переменных и методов класса, и может принимать следующие значения:

- `public`: Любой объект в любом пакете «видит» (может обращаться) к идентификатору с таким модификатором. (Использовать переменные с модификатором `public` не рекомендуется, но в ряде случаев это может быть необходимо, поэтому такая возможность существует.)
- `protected`: Любой объект, определенный в этом же пакете или подклассе (определенным в любом пакете) может «видеть» идентификатор.
- Не специфицировано (также называется `friendly` или `package private access`): только объекты и классы, определенные в том же пакете, могут «видеть» идентификатор.
- `private`: только класс, содержащий идентификатор, может «видеть» его.

Замечание: Локальные идентификаторы (определяемые на уровне методов) и параметры метода не могут быть определены с применением модификаторов доступа. Попытка использовать их приведет к ошибке компиляции.

Класс, как правило, состоит из элементов двух типов: **переменные** и **методы**.

Важно понимать, что значения переменных класса различаются в различных экземплярах этого класса и определяют его состояние. Эти значения часто называют *переменными экземпляра*. Формат определения переменных класса определяется следующим образом:

<модификатор доступа> <тип данных> <имя переменной> [= *<начальное значение>*];

Например:

```
private String name;
private int age;
private int height;
private int weight;
private String eyeColor;
private String gender;
```

При необходимости переменная может содержать начальное значение, например:

```
private String name ="NoName";
private int y = 1;
```

Модификаторы доступа, применительно к переменным, действуют аналогичные класса: `public`, `protected`, "`не специфицирован`" и `private`.

Наряду с модификатором доступа можно использовать ключевое слово `static` для статической переменной, метода или блока кода. Статическая переменная, метод или блок кода в классе - не определяются для конкретного экземпляра и вызываются с помощью имени класса. В этой связи нет необходимости создавать экземпляр класса, чтобы обратиться к статическому элементу класса. Например, объявленная переменная `static int x = 0`; означает, что есть только одна переменная `x`, независимо от того, сколько экземпляров класса существует. Возможно существование нескольких экземпляров класса, одного, или вообще ни одного - существует точно одна переменная `x`. Четыре байта памяти, занятой `x`, выделяются при загрузке класса. Позже мы рассмотрим примеры работы с переменной, описанной ключевым словом `static`.

Тип данных переменной класса зависит от допустимых значений переменной, и может быть примитивным или иметь тип другого класса, например:

```
private String name; // Здесь name - объектный тип,
private int age;     // а age - примитивный,
```

Вне зависимости от того, является ли тип данных, определенный в JDK, типом системы программирования (как, например, `String` или `Date`) или класс определяется программистом, синтаксис определения переменных абсолютно одинаковый.

Имя переменной <*имя переменной*> присваивается разработчиком класса с использованием рассмотренных ранее соглашений об именовании *CamelCase*, но имя начинаются со строчной буквы.

Начальное значение переменной присваивается, исходя из удобства дальнейшего использования класса. В случае отсутствия начального значения, компилятор создаст значение по умолчанию, которое будет устанавливаться переменной при создании экземпляра класса.

Прежде чем перейти к рассмотрению методов класса, рассмотрим пример, обобщающий уже рассмотренные понятия. Ниже приводится определение класса `Person`, который в представленном виде абсолютно бесполезен. Он содержит только атрибуты класса, причем с модификатором `private`, которые доступны только методам класса, которые отсутствуют. Класс `Person` окажется полезнее, когда у него появится поведение – то есть методы, выполняемые в классе.

```
package ru.ifmo.intro;
public class Person {
  private String name;
  private int age;
  private int height;
  private int weight;
  private String eyeColor;
  private String gender;
}
```

Методы класса определяют его поведение, в смысле реакции на получаемые сообщения. Иногда поведение выражается в возвращении текущего значения переменной, в других случаях поведение может быть исключительно сложным.

Существуют две категории методов:

- `методы-конструкторы`, которые используется только для создания экземпляра класса;
- `другие методы`, которые используются для предопределенного классом поведения. Методов может быть много, но все они должны быть объединены целью поддержки функций класса и определяются семантикой класса и его назначением.

Формат определения методов класса записывается следующим образом:

```
      <модификатор доступа> <имя класса> ([<список
аргументов>]) {
        <утверрждения конструктора>
      }
      <модификатор доступа> <тип возвращаемого значения>
<имя метода>([<список аргументов>]) {
        <утверрждения метода>
      }
```

Сигнатурой метода (***signature)*** называется сочетание структурных элементов в определении метода (имя метода, тип возвращаемого значения, количество аргументов, типы аргументов, порядок следования аргументов).

Рассмотрим подробнее обе разновидности методов, начиная с конструкторов. Конструкторы указывают, каким должен создаваться экземпляр класса - объект. Отличительной особенностью конструктора является совпадение имени конструктора с именем класса. Кроме того, в сигнатуре конструкторе должно отсутствовать определение типа возвращаемого значения, обязательное для всех прочих методов. В классе может быть несколько конструкторов с разными сигнатурами. Пример реализации конструктора класса `Employee` приведен ниже:

```
    public class Employee {
      private String employeeName;
      private String employeeAddress;
      private float annualSalary;

      public Employee(String employeeName,
            String      employeeAddress, float annualSala-
ry) {
            super();// обращение к конструктору суперкласса
            this.employeeName = employeeName;
            this.employeeAddress = employeeAddress;
            this.annualSalary = annualSalary;
      }
    }
```

Обратите внимание на использование ключевого слова `this` в операторах присвоения значений переменным. В Java `this` означает `"this object"` (этот объект) и служит для обращения к переменным с одинаковыми именами (как в данном случае, когда `age` – это и параметр конструктора и переменная класса), а также помогает компилятору преодолеть неопределенность при неоднозначности ссылок.

Мы уже отмечали, что все классы неявно наследуют свойства класса с именем `Object`. Если один класс наследуется от другого, каж-

дый из них может иметь свой собственный явно определенный конструктор. Как и любой другой метод класса, конструктор подкласса может переопределить конструктор суперкласса. Но иногда может понадобиться добавить в конструкторе подкласса некоторую функциональность, которая должна быть выполнена после выполнения кода конструктора суперкласса. В этом случае можно просто добавит явный вызов конструктора суперкласса вида `super();`, как показано в предыдущем коде. Вызов конструктора суперкласса должен быть первой строкой в конструкторе.

Если конструктор отсутствует, компилятор создаст конструктор по умолчанию (*без аргументов*). Но компилятор не станет генерировать конструктор, если уже указан другой конструктор, отличный от конструктора без аргументов. Таким образом, если в составе класса имеется конструктор с параметрами, то необходимо позаботиться о создании конструктора без аргументов, если его использование предполагается в системе. Рис. 37 демонстрирует процедуру генерации конструктора по умолчанию.

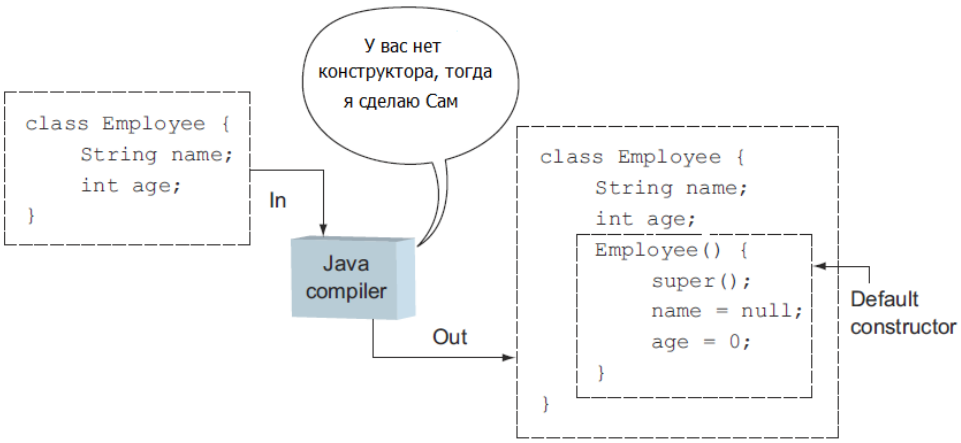


Рис. 37. Генерация конструктора по умолчанию

Модификатор доступа для конструктора применяется такой же, как и для переменных. Имя конструктора должно совпадать с именем класса. Так что если назвали свой класс `Employee`, то и имя конструктора должно быть `Employee`.

Для любого конструктора, кроме конструктора по умолчанию, создается список аргументов, который должен содержать один или более аргументов, разделенных запятыми. При этом два аргумента не могут иметь одинаковые имена, а тип аргументов может быть либо примитивным, либо ипом класса (так же, как в случае с типами переменных).

Ниже представлен код, который обеспечивает возможность создания объекта `Person` двумя способами: с использованием конструктора без аргументов и с инициализацией неполного списка атрибутов:

```
package ru.ifmo.intro;
public class Person {
  private String name;
  private int age;
  private int height;
  private int weight;
  private String eyeColor;
  private String gender;
  public Person() {
    super();
  }
public Person(String name, int age, int height, String
eyeColor, String gender) {
    this.name = name;
    this.age = age;
    this.height = height;
    this.eyeColor = eyeColor;
    this.gender = gender;
  }
}
```

Объект `Person` станет более полезным и интересным, если ему добавить более сложное поведение (функциональность), т.е. добавить методов.

Конструктор – это метод особого назначения с функцией создания экземпляра класса - объекта с определенными значениями переменных. Конструирование объектов потенциально сопряжено с большими накладными расходами, поэтому желательно при создании объекта сразу устанавливать правильное исходное состояние (значение атрибутов). Некоторые классы не обладают целесообразным начальным состоянием, если не инициировать значения объектных переменных. При конструировании объектов некоторых видов классов определение исходного состояния оказывается самым удобным и естественным способом.

Другие методы, выполняющие конкретные обязанности в Java-классах, выглядят аналогично методам-конструкторам, за исключением того, что им можно присваивать имена по своему усмотрению и эти методы должны указывать тип возвращаемого значения, в том числе `void`, когда значение не возвращается. Тип возвращаемого методом значения может быть как примитивным, так и объектным.

При именовании методов рекомендуется придерживаться `CamelCase` соглашений, и, в этой связи, начинать имя со строчной буквы, а для каждого последующего слова составного имени использовать заглавную букву.

Ниже представлена версия класса `Person` с добавлением нескольких методов, которые позволяют получать и устанавливать значе-

ния переменных экземпляра класса - объекта. Обратите внимание на то, что если метод не возвращает значение, нужно сообщить это компилятору, указав тип возврата `void` в его сигнатуре. (Для краткости конструкторы не показаны.)

```
package ru.ifmo.intro;
public class Person {
  private String name;
  private int age;
  private int height;
  private int weight;
  private String eyeColor;
  private String gender;
// Конструкторы ...
  public String getName() { return name; }
  public void setName(String value) { name = value; }
  // Комбинации getter/setter для других переменных...
}
```

Также обратите внимание на комментарий в листинге о "комбинациях геттеров/сеттеров". Для инкапсуляции данных классов и защиты от других объектов переменные объявляются `private` и обеспечиваются методами `mutator`: `getter` методы для получения значений атрибутов, `setter` методы для модификации значений атрибутов. Впоследствии мы будем часто иметь дело с этими методами. В коде выше показана только одна комбинация `getter/setter` (для атрибута `name`), но аналогичным образом можно определить и другие методы доступа.

Именование этих методов известно как JavaBeans шаблон, в котором предполагаемое имя `foo` имеет `getter` с именем `getFoo()`, а `setter` называется `setFoo(<аргумент>)`. Для переменных типа `boolean` имена обозначаются: `isFoo()` и `setFoo()`, соответственно. Имя `foo` используются для обозначения сущностей, таких как переменные, функции и команды, цель которого не имеет значения и служат лишь для демонстрации концепции.

Атрибут обычно объявляется `private` доступом, а модификаторы доступа для методов `getter` и `setter` объявляются `public`, обеспечивая возможность доступа к переменным класса из других классов.

Метод `getter` не принимает параметров и возвращает значение соответствующего типа. Метод `setter` принимает только один параметр типа атрибута и не возвращает значения, как показано в коде ниже применительно к переменной с именем `foo`:

```
private String foo;
```

```
public String getFoo() {   return foo; }
public void setFoo(String value) {
   foo = value;
}
```

Методы экземпляра и методы класса

Ранее мы упоминали о статических членах класса - атрибутах и методах. Существуют два основных типа методов (кроме конструкторов): методы экземпляра и статические методы. Выполнение метода экземпляра зависит от состояния конкретного экземпляра класса и может выполняться после создания объекта. Для вызова метода экземпляра класса необходимо ссылаться на объект и после точки указать имя метода и параметры, которые должны быть переданы, например:

```
objectReference.someMethod();
objectReference.someOtherMethod(parameter);
```

Статические методы иногда еще называют методами класса, поскольку их выполнение не зависит от состояния какого-либо конкретного объекта. Для определения метода класса используется ключевое слово static. Поведение статического метода определяется на уровне класса.

Статические методы широко используются для создания утилит и их можно рассматривать как способ создания глобальных методов (как в языке C) с сохранением при этом самого кода, сгруппированного с классом, которому они принадлежат.

Для вызова метода класса необходимо делать ссылку на класс и после точки указать имя метода и параметры, которые должны быть переданы, например:

```
classReference.someMethod();
classReference.someOtherMethod(parameter);
```

Ниже приводится пример использования метода экземпляра класса showArea и методов класса area.

```
public class PlayCircle {
     public static final double PI = 3.14159;//
Константа PI
     private double diameter;// переменная реализации

     public void setDiameter(double newDiameter) {
          diameter = newDiameter;
     }

     public static double area(double radius) {
          return (Math.PI * radius * radius);
     }
```

```
        public void showArea() {
                System.out.println("Площадь равна"
    +area(diameter/2));
        }// вызов статического метода внутри нестатическо-
    го
        public static void areaDialog() {
                double newDiameter = 4;
                System.out.println("Площадь равна " + ar-
    ea(newDiameter));

        } // в статическом методе
        public static void main(String[] args) {
                PlayCircle circle = new PlayCircle(); // Со-
    здается объект circle
                circle.setDiameter(2); // переменной реализа-
    ции diameter присваивается 2
                System.out.println("Диаметр круга равен 2,");
                // Вызов нестатического метода и статического
    внутри нестатического
                circle.showArea(); // вызов статического ме-
    тода внутри нестатического showArea
                System.out.println("Диаметр круга равен 4.");
                 // Вызов статического метода
                 PlayCircle.areaDialog(); // Вызов статиче-
    ского метода
        }
    }// end PlayCircle
```

В приведенном примере выполняется вызов нестатических методов `setDiameter(2)` и `showArea()` созданного объекта `circle`, а также статического метода `areaDialog()` путем ссылки на класс `PlayCircle`.

Вывод из программы `PlayCircle` будет следующий:

```
Диаметр круга равен 2,
Площадь равна 3.141592653589793
Диаметр круга равен 4.
Площадь равна 50.26548245743669
```

Статические переменные

Наряду со статическим методами существуют статические переменные - переменные, определенные с ключевым словом `static`. Статические переменные обределяются на уровне класса и существует только одна копия статической области в классе, независимо от того, сколько экземпляров класса создано, и совместно используются всеми объектами класса. Они называются переменными класса, т.е. переменными относящимися ко всему классу, в отличие от переменных, отно-

сящихся к его отдельным объектам. Статические переменные инициализируются еще до начала работы конструктора, но при инициализации можно использовать только констаные выражения. Если же инициализация требует сложных вычислений, например, циклов для задания значений элементам статических массивов или обращений к методам, то эти вычисления заключают в блок, помеченный ключевым словом `static`, который тоже будет выполнен до запуска конструктора. Пример такого кода представлен тиже:

```
static int var2 = 0;
static void setVar2(int var)
{
 var2 = var;
}
```

Подобно переменным реализации, статические переменные обычно объявляются закрытыми (`private`). Возможность чтения и изменения их значений вне класса должна предоставляться только посредством определения соответствующего доступа.

Ниже представлен пример, в котором выполняются операции увеличения на единицу статической переменной `numberOfInvocations`, выполняемые различными методами, в том числе и статическим методом (`numberSoFar`). Вне зависимости от места выполнения операции, все они выполняются с единственным экземпляром переменной, о чем свидетельствует выведенное результирующее значение этой переменной.

```
public class StaticDemo {
     private static int numberOfInvocations = 0;
     public static void main(String[] args) {
          int i;
          StaticDemo object1 = new StaticDemo();
          for (i = 1; i <= 10; i++)
               object1.outPutCountOfInvocations();
          StaticDemo object2 = new StaticDemo();
          for (i = 1; i <= 10; i++)
               object2.justADemoMethod();
          System.out.println("Общее количество вызовов
= " + numberSoFar());
      }
     public void justADemoMethod() {
          numberOfInvocations++;
     }
     public void outPutCountOfInvocations() {
          numberOfInvocations++;
     System.out.println("из outPutCountOfInvocations =
"+numberOfInvocations);
```

```
    }
    public static int numberSoFar() {
        numberOfInvocations++;
        return numberOfInvocations;
    }
}
```

Полученный вывод класса `StaticDemo` будет следующий:

```
из outPutCountOfInvocations = 1
из outPutCountOfInvocations = 2
из outPutCountOfInvocations = 3
из outPutCountOfInvocations = 4
из outPutCountOfInvocations = 5
из outPutCountOfInvocations = 6
из outPutCountOfInvocations = 7
из outPutCountOfInvocations = 8
из outPutCountOfInvocations = 9
из outPutCountOfInvocations = 10
Общее количество вызовов = 21
```

В примере выше утверждения метода `main()` создают объект `object1` класса `StaticDemo` и в цикле обращаются к нестатическому методу `outPutCountOfInvocations`, в котором добавляется 1 к статической переменной `numberOfInvocations` и результат выводится на консоль. Далее создается объект `object2` класса `StaticDemo` и в цикле выполняется вызов нестатического метода `justADemoMethod`, в котором вновь добавляется 1 к той же статической переменной `numberOfInvocations`, но значение не выводится. В завершении выполнения метода в контексте выполнения утверждения вывода выполняется вызов статического метода `numberSoFar`, в котором еще раз добавляется 1 к статической переменной `numberOfInvocations` и полученное значение возвращается для вывода на консоль. Таким образом, все методы обращаются к единственной статической переменной `numberOfInvocations`, изменяя ее значение.

Статический блок

Язык Java допускает блок инициализации - код между фигурными скобками, который выполняется всякий раз перед созданием объекта класса до того как выполняется конструктор. Также допускается статический блок инициализации - блок кода, заключенный в круглые скобки и использующий ключевое слово, `static`, который выполняется только один раз, когда загружается класс и может только инициализировать статические члены данного класса.

Рассмотрим пример простого класса с двумя блоками инициализации и один из них статический следующего вида:

```
public class CheckInit {
 // Статический массив
 static int[] values = new int[10];
      // Блок инициализации 1
 static {
      System.out.println("Выполнение блока 1...");
      for (int i = 0; i < 10; i++) {
           values[i] = (int) (100.0 * Math.random());
      }
 }
 {
      // Блок инициализации 2
      System.out.println("Выполнение блока 2...");
 }
     CheckInit(){
           System.out.println("Выполнение конструктора
 класса CheckInit...");
 }
 // List values in the array for an object
 void listValues() {
      for (int i : values)
           System.out.print(i + " "); // Вывод значений
      System.out.println(); // новая строка
 }
 public static void main(String[] args) {
      CheckInit o1 = new CheckInit();
      System.out.println("o1:");
      o1.listValues();
      CheckInit o2 = new CheckInit();
      System.out.println("o2:");
      o2.listValues();
 }
}
```

В процессе выполнения приведенной выше программы будет совершен следующий вывод:

```
Выполнение блока 1...
Выполнение блока 2...
Выполнение конструктора класса CheckInit...
o1:
96 74 62 56 60 16 61 57 43 47
Выполнение блока 2...
Выполнение конструктора класса CheckInit...
o2:
96 74 62 56 60 16 61 57 43 47
```

В выше приведенном примере класс `CheckInit` содержит статический массив `values` , значениями которого является массив из 10 целых величин. Эти значения формируются в статическом блоке с помощью функции `Math.random()`, которая генерирует случайные числа.

В методе `main()` последовательно создаётся два объекта (o1 и o2) типа `CheckInit`, и затем в каждом из них вызывается метод `listValues()` для вывода значений статического массива `values`, ранее образованного статическим блоком. Вывод значений массива из различных объектов совпадает. Вывод также показывает, что инициализация статического блока выполняется только однажды, а нестатический блок выполняется при каждом создании класса перед выполнением конструктора класса.

Существуют очевидные ограничения по использованию статических переменных и методов. В частности:

- Если класс содержит переменные реализации, то к ним нельзя обращаться в утверждениях статических методов. Иным словами в статическом методе нельзя обращаться к нестатическим переменным.
- Внутри определения любого статического метода нельзя вызывать нестатический метод. Для вызова нестатического метода необходимо создать новый объект этого класса, а затем использовать этот объект, применяя в качестве вызывающего объекта для нестатического метода.
- В статическом методе нельзя использовать ссылки `this`.

Создание класса Java в Eclipse

Теперь, когда мы рассмотрели полный набор типов переменных Java - примитивные и класса, перейдем к программированию. Здесь, применяя IDE Eclipse, объявим класс и добавим к нему переменные и методы. Кроме того, создадим в классе статический метод `print`, к которому можно обращаться на уровне класса, не создавая экземпляр класса, позволяющий выводить значения переменных, передаваемых этому методу в качестве аргумента, и метод `main()` для проверки работоспособности класса `Person`.

Создание проекта

Как уже упоминалось ранее, разрабатываемые ресурсы Java располагаются в рамках проекта и, таким образом, перед началом разработки приложений необходимо создать проект и присвоить ему имя. Чтобы создать новый проект Java, выберите из главного меню **File >**

New > Java Project ..., и откроется диалоговое окно, в которое введите название проекта **Labs** и нажмите кнопку **Finish**.

В представлении Package Explorer должно появиться название только что созданного нового проекта Labs, и его содержимое можно посмотреть при нажатии на символ расширитель " ▷ ", как показано на Рис. 37. При этом символ расширителя преобразуется в символ свертывания " ◢ ", при нажатии на который свертывается содержимое пакета. Такие символы (иконки) аналогично применяются ко всем уровням ресурсов проекта.

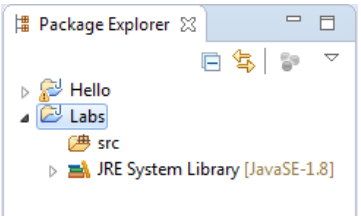


Рис. 37. Начальное содержимое проекта Labs

Создание пакета

Перед созданием класса, создадим пакет для его размещения, который, как известно, предназначен для создания автономного пространства имен, отображающихся в структуру каталогов файловой системы.

Вместо использования пакета по умолчанию (всегда рекомендуется использовать пакеты), создадим пакет для своего кода в проекте. Убедитесь, что проект Labs является текущим и выберите **File > New > Package**, чтобы отобразить диалоговое окно **New Java Package**, в поле **Name** введите `ru.ifmo.intro` и нажмите кнопку **Finish** как показано на Рис. 38.

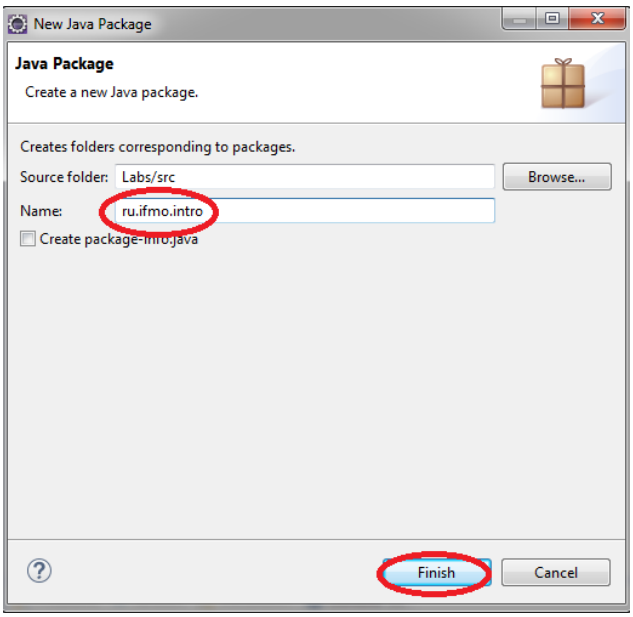


Рис. 38. Диалоговое окно New Java Package

После завершения этой процедуры в Package Explorer появится новый пакет, как показано на Рис. 39.

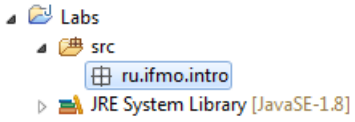


Рис. 39. Созданный пакет в представлении Package Explorer

Объявление класса

Существуют различные способы создания класса в Eclipse, но самый естественный - выбрать правой кнопкой мыши образованный ранее пакет и из появившегося контекстного меню выбрать **New > Class...**. В результате выполненных действий появится диалоговое окно **New Java Class**, в котором введите `Person` в поле **Name**. В разделе **Which method stubs would you like to create?** (Шаблон какого метода нужно создать?) отметьте `public static void main(String[] args)` , для того, чтобы Eclipse в составе создаваемого класса определил метод `main()` и нажмите кнопку **Finish,** как показано на Рис. 40.

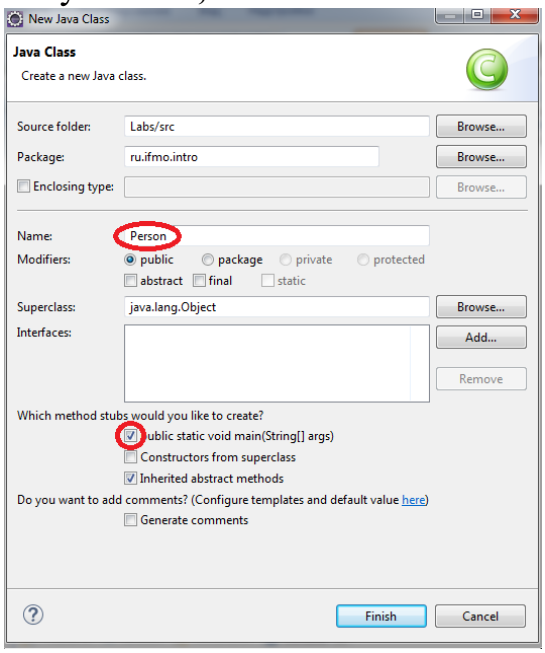


Рис. 40. Диалоговое окно создания класса

После завершения процедуры заполнения диалогового окна New Java Class в окне редактирования появится код создаваемого класса с указанием соответствующего пакета, как показано на Рис. 41.

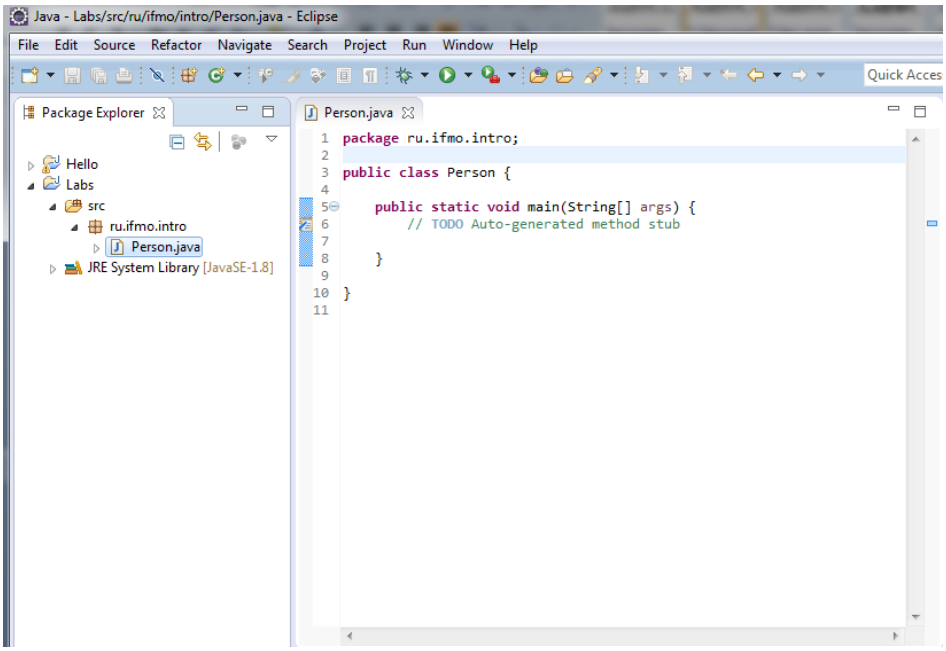


Рис. 41. Код класса `Person` в окне редактирования

Eclipse генерирует код класса с заданным именем, который содержит оператор `package` и затребованный метод `main()` с комментариями. Представленному коду класса соответствует файл `Person.java`, определенный в файловой структуре пакета, и задача состоит в том, чтобы доопределить код класса в соответствии с замыслом его создания. Как только изменяется содержимое окна редактирования, рядом с именем в заголовке окна редактирования появляется звездочка (*), указывающая на то, что внесены изменения, и код не сохранен.

Прежде чем приступим к определению переменных и методов экземпляра класса ниже метода `main()` введите код статического метода `print` следующего вида:

```
public static void print(String str) {
    System.out.println(str);
}
```

В коде метода выполняется единственный оператор языка Java, который выводит на экран строковое содержимое, полученное в аргументе `String str`. Поскольку сигнатура этого метода содержит ключевое слово `static`, то в дальнейшем из класса `Person` мы сможем обращаться к нему, просто указывая имя метода (*`print`*), а из других классов, которых пока нет в нашем проекте, по имени класса и имени метода (`Person.`*`print`*). К методам экземпляров класса обращаются по имени объектов, которые создаются на основе класса.

Метод `main()` можно использовать в качестве средства тестирования работоспособности класса. В принципе, не запрещается содержать метод `main()` в любом классе для того, чтобы имелась возможность проверить работу методов этого класса с помощью JVM. Однако не всякий класс обязательно должен содержать метод `main()` - на самом деле, в большинстве классов такой метод отсутствует.

В реальном программировании используются специальные библиотеки тестирования, но в рамках настоящего пособия применяется метод `main()` в качестве средства тестирования.

Добавление переменных класса

Войдите в редактор исходного кода Eclipse для класса `Person` и непосредственно под заголовком класса добавьте следующий код java для определения переменных класса:

```
private String name;
private int age;
private int height;
private int weight;
private String eyeColor;
private String gender;
```

Среди большого набора возможностей Eclipse содержит средства генератора, в том числе, кода для создания геттеров и сеттеров атрибутов, описанных в классе. Для этого щелкните правой кнопкой мыши в любом месте редактора кода `Person` и в открывшемся контекстном меню выберите **Source > Generate Getters and Setters...**. Когда откроется диалоговое окно, нажмите кнопку **Select All**, как показано на Рис. 42.

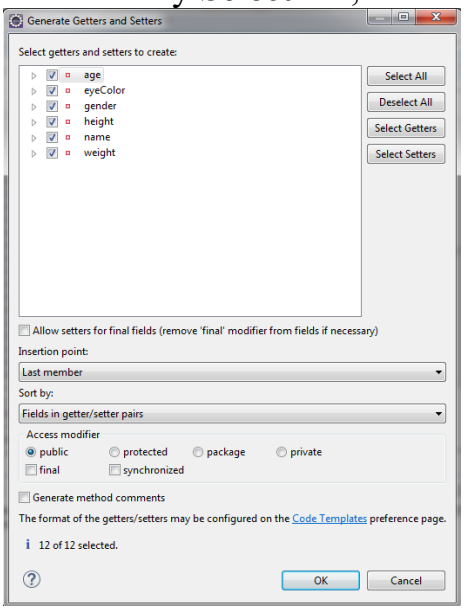


Рис. 42. Диалоговое окно генерации геттеров и сеттеров

Чтобы определить точку размещения генерируемого кода, выберите **Last member** (Последний член) в поле со списком **Insertion Point** (Точка внедрения) и нажмите кнопку **OK**. Обратите внимание, что методы геттеров и сеттеров появляются после нашего метода `print ()` - последнего метода класса. Естественно, можно выбрать другой вариант размещения методов.

Добавим к классу `Person` два конструктора. Напомним, что конструктор - это специальный метод, который используется для создания объектов класса. Введите в окне редактора исходного кода в верхней части определения класса (под строкой `public class Person ()`) следующий код, определяющий конструктор:

```
public Person(String name, int age, int height,int
      weight, String eyeColor, String gender) {
      this.name = name;
      this.age = age;
      this.height = height;
      this.weight = weight;
      this.eyeColor = eyeColor;
      this.gender = gender;
}
```

Убедитесь в отсутствии красных волнистых линий, означающих ошибки компиляции. Далее в классе введите конструктор по умолчанию (без параметров) следующего вида:

```
public Person() {
      super();
}
```

Кодирование метода `main()`

Для того чтобы создать экземпляр класса `Person`, присвоить значение переменным экземпляра, а затем вывести их значения на консоль мы будем использовать метод `main()`, которому JVM передает управление при выполнении класса. Модифицируйте метод `main()` и сделайте его таким, как показано ниже:

```
public static void main(String[] args) {
      // TODO Auto-generated method stub
        Person p = new Person("James Bond",42,183,82,"Brown",
        "MALE");
        print("Name:" + p.getName());
        print("Age:" + p.getAge());
        print("Height (cm):" + p.getHeight());
        print("Weight (kg):" + p.getWeight());
        print("Eye Color:" + p.getEyeColor());
        print("Gender:" + p.getGender());
```

}

В этом методе создается объект с именем p с заданными значениями переменных экземпляра класса с помощью утверждения вида:

```
  Person p = new Person("James Bond", 42, 173, 82, "Brown",
"MALE");
```

Здесь выполняется утверждение вида *<Тип> <имя переменной>=new <имя конструктора>(),* которое применяется для создания объекта класса. В отличие от описания примитивной переменной добавляется ключевое слово new.

Следующая группа однотипных утверждений в методе main() выводит текущие значения переменных экземпляра образованного класса p с помощью статического метода *print*, для которого в качестве параметра формируется конкатенация из литерала типа String в двойных апострофах и значения соответствующей переменной объекта p, полученного с помощью нестатического метода getXXX, например, `print("Name:"+p.getName());` для переменной name.

Статический метод *print* доступен внутри класса без создания объекта, а ссылка к нестатическим методам и атрибутам объекта происходит с указанием ссылки на экземпляр класса p.

Следует также иметь в виду, что создание объекта класса не ограничивается используемым способом обращения к конструктору этого класса, и объект может быть возвращен любым методом, в сигнатуре которого определен соответствующий тип в качестве возвращаемого значения. Например, если существует метод findPerson, возвращающий объект класса Person по значению его идентификатора с описанием public Person findPerson(int id), то объект может быть получен следующим образом: Person p = findPerson(c_id);

Выполнение кода в Eclipse

Для запуска Java-программы в Eclipse поместите указатель мыши в окне редактирования класса Person и нажмите кнопку **Run** (на панели инструментов). Альтернативным вариантом выполнения класса является указание мышью класса Person в представлении Project Explorer, нажатие правой кнопки и выбор в контекстном меню **Run as->Java Application**). В результате выполнения класса будет выведена информация в представлении Console, представленная на Рис. 43.

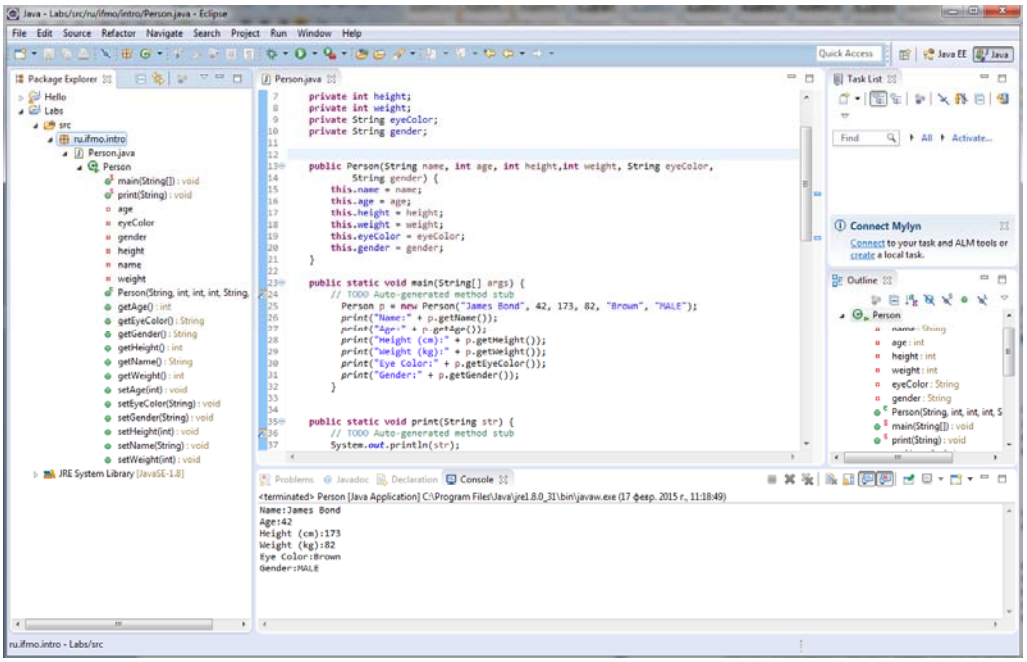


Рис. 43. Результат выполнения класса `Person`

Обратите внимание, что представление `Console` открывается автоматически и показывает результат работы программы. Представление `Outline` в правой части перспективы Java отображает базовую структуру класса `Person`.

Упражнение 4.

Постановка задачи. Класс `Employee` создается для описания нового сотрудника в процессе приема на работу. Класс содержит три поля: `employeeName, employeeAddress и annualSalary`, а также метод `displayDetails()`, предназначенный для вывода всех значений переменных класса для объекта и метод, который для любого другого объекта класса `Employee` выводит некоторое выражение String. Для этого выполните следующие действия:

- Используя Eclipse, создайте в любом созданном Java-проекте пакет с произвольным именем.
- Создайте класс `Employee` и отметьте флажок создания метода `main()`. Определите в нем переменные класса (`employeeName, employeeAddress и annualSalary`) с модификатором доступа `private` соответствующего типа (`String` или `float`).
- Создайте методы `get` и `set` для переменных класса (Следует использовать среду Eclipse: правая кнопка в поле окна редактирования -> **Source** -> **Generate Getters and Setters** ->для всех полей).

- Создайте метод - конструктор для инициализации переменных класса.(Следует использовать среду Eclipse: правая кнопка в поле окна редактирования -> **Source** -> **Generate Constructor using Fields**). *Конструктор - это метод, имя которого совпадает с именем класса и который используется при создании объекта данного класса.* Если такой конструктор (с аргументами) создается, то необходимо в случае необходимости явно создать конструктор по умолчанию `Employee()`, который в противном случае создается компилятором Java.
- Создайте метод `toString()`. (Следует использовать среду Eclipse: правая кнопка в поле окна редактирования -> **Source** -> **toString**). (Можно указать все поля для включения в строку вывода). Этот метод по умолчанию вызвается, когда вы указываете объект класса `Employee` , например, в операторе `System.out.println(o2)`, и выводит значение, возвращаемое методом `toString()`. В данном случае мы переопределяем наследуемый всеми классами от объекта `Object` метод `toString()`, с тем, чтобы выводились полезные нам значения об объекте класса вместо мало информативной ссылки на объект по умолчанию. Увидеть вывод можно, позже удалив созданный вами метод `toString()` и повторить выполнение класса еще раз. Перед описанием метода будет автоматически установлена аннотация `@Override`, которая сообщает компилятору, что метод переопределяется, заменяя наследуемый код. Аннотации введены с 5-й версии SDK и широко используются различными компонентами Java.
- Самостоятельно создайте метод `displayDetails()`, который возвращает значения полей класса с использованием единственного оператора `return employeeName + " " + employeeAddress+ " "+ annualSalary;`
- Также самостоятельно создайте метод, который возвращает данные объекта того же класса (`Employee`), который обращается к данному методу текущего объекта. То есть один объект данного класса может обратиться к некоторому методу другого объекта, передавая в качестве параметра ссылку на себя. Определите сигнатуру для метод `public String askMeeting(Employee e)`. Как видно из сигнатуры, метод возвращает значение `String`, которое пусть будет сообщением-приглашением на встречу, формируемым с помощью

оператора `return "Дорогой " + e.getEmployeeName() + " я буду рад(а) встрече с Вами по адресу " + employeeAddress;`. Корректным будет обращение только из объекта класса `Employee`, который поддерживает метод `getEmployeeName()`.

В подготовленном Eclipse методе `main()`, следует выполнить следующие действия:

1. Создать три разных объекта с именами `o1, o2, o3` с данными своих знакомых с помощью операторов:

```
Employee o1 = new Employee (<....>);
Employee o2 = new Employee (<....>);
Employee o3 = new Employee (<....>);
```

Например так:

```
Employee o1 = new Employee("Anatoly", "St. Petersberg", 30000);
```

2. Вывести сведения о значениях объектных переменных для каждого объекта `o1, o2, o3`, используя метод `displayDetails()`, например, `System.out.println(o2.displayDetails());`

3. Выведите на экран годовую зарплату всех созданных объектов как сумму соответствующих полей объектов `o1,o2 и o3`. Например, так:

```
System.out.println("Сумма = "+(o1.annualSalary + o2.annualSalary + o3.annualSalary));
```

4. В конце метода `main()` выведите сообщение, полученное при выполнении метода `askMeeting` обращением от объекта `o1` к объекту `o3`, например, так:

```
System.out.println(o3.askMeeting(o1));
```

5. И в заключение выполните команду `System.out.println(o2)`, в которой неявно используется вызов переопределенного метода `toString()`, имеющийся в каждом объекте. Кроме того напишите оператор, который создает объект только на время использования его в операторе, например так: `System.out.println((new Employee("Peter", "New-York", 40000)annualSalary));`

Выполните созданный вами класс (Правая кнопка и в контекстном меню выбрать **Run as->Java Application**), проанализируйте полученный вывод вашего класса.

Мы рассмотрели базовые концепции ООП и продолжим рассмотрение классов, расширяющих вычислительные возможности языка Java.

Тип String и операторы

В пособии неоднократно использовали переменные типа `String`, базируясь на интуитивных знаниях, заимствованных из других систем программирования - все языки программирования поддерживают аналогичный тип данных для хранения строк произвольных символов.

Работа со строками на языке С - весьма трудоемкий процесс, поскольку программисту приходится манипулировать массивами 8-битных символов. На языке Java строки – это объекты типа `String`, для управления которыми используются специальные методы.

Обычно используются для создания экземпляра класса `String` и установления его значения оператор типа

```
String greeting = "hello";
```

Поскольку тип String является объектным, можно использовать эквивалентный оператор `String greeting=new String("hello");`

Для создания экземпляров `String` можно использовать оператор `new`. Задание значения переменной типа `String` приводит к тому же результату, потому что язык Java создает объект `String` для хранения литерала, а затем присваивает этот объект экземпляру переменной.

Переменные класса `String` широко используются в приложениях Java и поддерживают множество полезных и удобных для применения методов. Даже еще не используя методы, мы уже сделали кое-что интересное с двумя строками, объединив или конкатерировав их, используя оператор сложения: `print("Height (cm):" + p.getHeight());` На языке Java знак оператор "+" означает конкатенацию `String`.

Класс `String`, инкапсулирующий структуру данных соответствующую строке, расположен в пакете `java.lang`. Класс `String` - объектное представление **неизменяемой** символьной строки (статической строки). Классы `StringBuffer/StringBiulder` в отличие от `String`, используется, когда строку после создания требуется изменять (динамическая строка). В этих классах содержатся методы, которые позволяют сравнивать строки, осуществлять в них поиск и извлекать определенные символы и подстроки и т.п..

Существует несколько других способов создания переменных типа `String`, используя соответствующий конструктор класса, некотрые из которых приводятся ниже:

- `String()` — создается объект с пустой строкой;

- `String (String str)` — из одного объекта создается другой;
- `String (StringBuffer str)` — преобразованная копия объекта класса `BufferString`;
- `String (byte[] byteArray)` — объект создается из массива байтов byteArray (ASCII);
- `String (char [] charArray)` — объект создается из массива charArray символов Unicode;
- `String (byte [] byteArray, int offset, int count)` — объект создается из части массива байтов `byteArray`, начинающейся с индекса `offset` и содержащий `count` байтов;

Ниже представлен код класса `StringGo` и его вывод, демонстрирующие применение различных конструкторов для создания переменной типа `String`. В этом коде применяются массивы, о которых речь пойдет далее.

```
     public class StringGo {
          public static void main(String[] args) {
          // TODO Auto-generated method stub
          char chars[] = { 'a', 'b', 'c', 'a' };// Массив
chars                System.out.println("Массив char -
>"+chars.toString());
          String s = new String(chars);
          System.out.println("String на основе массива->
"+s);
          byte[] bytesData = s.getBytes(); // Из символов
создали                                        //массив byte
          System.out.println("Для массива выводится ссылка->
"                    +bytesData.hashCode());
          String a = new String(bytesData); // Из масиива
byte
     //образовали строку
          System.out.println("String из массива byte ->
"+a);
          byte charb[] = { 65, 66, 67, 68 };// Массив byte
из чисел
          String b = new String(charb);
          System.out.println("Еще String на основе массива -
> "+b);
          String c = new String (bytesData,0,2);
          System.out.println("String на основе части массива
-> "+ c);
          }
```

```
}
```

Вывод, полученный в процессе выполнения класса `StringGo`:

```
Массив char ->[C@15db9742
String на основе массива-> abca
Для массива выводится ссылка-> 1829164700
String из массива byte -> abca
Еще String на основе массива -> ABCD
String на основе части массива -> ab
```

Комментарии в классе `StringGo` и соответствующий вывод достаточно полно описывают выполненные операции в коде.

В классе `String` существует масса полезных методов, которые можно применять к строкам (перед именем метода указывается тип значения, которое он возвращает):

- `int length()` - возвращает длину строки (количество символов в ней);
- `boolean isEmpty()` - проверяет, пустая ли строка - возвращает истинну, если строка не содержит символов;
- `String replace(char oldChar, char newChar)` - возвращает строку, в которой все символы, совпадающие со значением переменной `oldChar`, заменяются в новой копии строки на символ `newChar`;
- `String toLowerCase()` - возвращает строку, в которой все символы исходной строки преобразованы к строчным;
- `String toUpperCase()` - возвращает строку, в которой все символы исходной строки преобразованы к прописным;
- `boolean equals(String s)` - возвращает истинну, если строка к которой применён метод, совпадает со строкой s указанной в аргументе метода (с помощью оператора == строки сравнивать нельзя, как и любые другие объекты);
- `int indexOf(char ch)` - возвращает индекс символа `ch` в строке (индекс это порядковый номер символа, но нумероваться символы начинают с нуля). Если символ не будет найден, то возвращается -1. Если символ встречается в строке несколько раз, то вовзвратит индекс его первого вхождения.
- `int lastIndexOf(char ch)` - аналогичен предыдущему методу, но возвращает индекс последнего вхождения, если смивол встретился в строке несколько раз.

- `int indexOf(char ch,int n)` - возвращает индекс символа ch в строке, но начинает проверку с индекса n (индекс это порядковый номер символа, но нумероваться символы начинают с нуля).
- `char charAt(int n)` - возвращает код символа, находящегося в строке под индексом n (индекс это порядковый номер символа, но нумероваться символы начинают с нуля).
- `String substring(int n[,int m])` - возвращает подстроку из строки, начиная с начального индекса n до конечного индекса m строки. Можно указать только индекс первого символа подстроки - тогда будут скопированы все символы, начиная с указанного и до конца строки. Также можно указать и начальный, и конечный индексы - при этом в новую строку будут помещены все символы, начиная с первого указанного, и до (но не включая его) символа, заданного конечным индексом.

Код класса `StringMethods` демонстрирует примеры использования методов класса `String` с полученным выводом:

```
public class StringMethods {
    public static void main(String[] args) {
        String s1 = "Hello, World";
        System.out.println(s1.toUpperCase());
        // выведет  «HELLO, WORLD»
        String s2 = s1.replace('o', 'a');
        System.out.println(s2); // выведет «Hella,
Warld»
        System.out.println(s2.charAt(1)); // выведет
«e»
        int i;
        i = s1.length();
        System.out.println(i); // выведет 12
        i = s1.indexOf('f');
        System.out.println(i); // выведет -1
        i = s1.indexOf('r');
        System.out.println(i); // выведет 9
        i = s1.lastIndexOf('f');
        System.out.println(i); // выведет -1
        i = s1.indexOf('t');
        System.out.println(i); // выведет -1
        i = s1.indexOf('r',3);
        System.out.println(i); // выведет 9
        System.out.println(s1.substring(4,8));
        // выведет  «o, W»
    }
```

```
}
```
Вывод класса `StringMethods`:
```
HELLO, WORLD
Hella, Warld
e
12
-1
9
-1
-1
9
o, W
```

Тип StringBuffer (StringBuilder)

Класс `String` представляет собой неизменяемые последовательности символов постоянной длины, и частое использование объектов этого класса ведет к неэффективному использованию памяти компьютера. Класс `StringBuffer` (также как и `StringBuilder`) представляет расширяемые и доступные для изменений последовательности символов, позволяя вставлять символы и подстроки в существующую строку в любом месте. Данные классы гораздо экономичнее в плане потребления памяти и настоятельно рекомендуется к использованию. Методы для этих типов классов совершенно одинаковы, и класс `StringBuilder` может использоваться при синхронизации данных различными потоками (thread), что предполагает дополнительные накладные расходы (overhead) в процессе работы.

`StringBuffer` может хранить количество символов, определенное его размером. Если `StringBuffer` расширяется, то автоматически расширяется пространство для дополнительных символов.

Класс `StringBuffer` также используется для реализации операций конкатенации `+` и `+=` для `String`.

Существует четыре конструктора класса `StringBuffer`:

- `StringBuffer()` - резервирует место под 16 символов без перераспределения памяти.
- `StringBuffer(int capacity)` - явно устанавливает размер буфера в аргументе `capacity`.
- `StringBuffer(String string)` - устанавливает начальное содержимое в соответствии со `string` и резервирует 16 символов без повторного резервирования.
- `StringBuffer(CharSequence cs)` - создаёт объект, содержащий последовательность символов `cs`, и резервиру-

ет место ещё под 16 символов. `CharSequence` является читаемой последовательностью значений `char`. Этот интерфейс обеспечивает унифицированный доступ только для чтения ко многим различным видам символьных последовательностей.

Существуют следующие методы классов:

- `int length()` - возвращает текущую длину объекта. Например:

```
StringBuffer sb = new StringBuffer("Привет");
System.out.println("Длина: " + sb.length());
// Вывод: Длина: 6
```

- `int capacity()` - возвращает текущий объём выделенной памяти. Например:

```
StringBuffer sb = new StringBuffer("Привет");
System.out.println("Объём памяти: "+ sb.capacity());
    // Вывод: Объём памяти: 22
```

Обратите внимание, что хотя слово состоит из шести символов, в памяти зарезервировано пространство для дополнительных 16 символов. Для такой простейшей операции выигрыша нет, но в сложных примерах, когда приходится активно работать с множеством строк, производительность резко возрастает.

- `void ensureCapacity(int minimumCapacity)` - предварительно выделяет память для определённого аргументом `minimumCapacity` количества символов.

- `void setLength(int length)` - устанавливает длину строки. Значение аргумента `length` должно быть неотрицательным.

- `charAt(int index)` и `setCharAt(int index, char ch)` - извлекает значение отдельного символа с помощью метода `charAt()` и устанавливает новое значение символа с помощью метода `setCharAt()`, указавая индекс символа и его значение. Например,

```
StringBuffer sb = new StringBuffer("Cat");
sb.setCharAt(1, 'o');
```

Фрагмент кода выше заменяет символ "a" на "o".

- `getChars(srcBegin, srcEnd, dst, dstBegin)`- копирует подстроку из объекта класса `StringBuffer` в массив. Необходимо предусмотреть массив достаточного размера для приёма указанного количества символов подстроки. Первый символ должен быть скопирован с индексом `srcBegin`; последний символ должен быть скопирован с индексом `srcEnd-1`. Общее число символов, которое требуется скопировать `srcEnd-srcBegin`. Символы копируются в подмассив

dst, начиная с индекса dstBegin и заканчивая индексом dstbegin + (srcEnd-srcBegin) - 1.

- append() - объединяет представление любого другого типа данных. Существует несколько перегружаемых версий метода:

```
StringBuffer append(String string)
StringBuffer append(int number)
StringBuffer append(Object object)
```

В процессе выполнения строковое представление каждого параметра получается через метод String.valueOf(), и затем полученные строки добавляются в итоговую строку. Например,

```
String str1 = "Cat has ";
String str2 = " paws";
int paws = 4;
StringBuffer sb = new StringBuffer(20);
sb.append(str1).append(paws).append(str2);
System.out.println (sb); // Cat has 4 paws
```

- insert() - вставляет одну строку в другую. Также можно вставлять значения других типов, которые будут автоматически преобразованы в строки. Метод возвращает модифицированный объект StringBuffer.

Ниже приведены сигнатуры метода для каждого примитивного типа данных:

```
public StringBuffer insert(int offset, boolean b)
public StringBuffer insert(int offset, char c)
public insert(int offset, char[] str)
public StringBuffer insert(int index, char[] str, int
offset, int len)
public StringBuffer insert(int offset, float f)
public StringBuffer insert(int offset, int i)
public StringBuffer insert(int offset, long l)
public StringBuffer insert(int offset, Object obj)
public StringBuffer insert(int offset, String str)
```

Во всех форматах следует указать индекс позиции (offset), с которой будет вставляться строка. Например,

```
StringBuffer sb = new StringBuffer("I cats");
sb.insert(2, "like ");
System.out.println (sb); // I like cats
```

- StringBuffer reverse() – изменяет порядок символов на обратный. Например,

```
StringBuffer sb = new StringBuffer("АБРАКАДАБРА");
sb.reverse();
System.out.println (sb); // АРБАДАКАРБА
```

- StringBuffer delete(int start, int end) и StringBuffer deleteCharAt(int index) - удаляет последовательность символов; указывается индекс первого символа (start), который надо удалить, а также индекс символа, следующего за последним из удаляемых (end). Метод deleteCharAt() удаляет один символ в указанной позиции index.

- StringBuffer replace(int start, int end, String str) - Позволяет заменить один набор символов на другой. Требуется указать начальный (start) и конечный (end) индексы, а также строку замены (str).

- StringBuffer substring((int start[,int end]) возвращает часть содержимого. Существует две формы метода. В первом варианте нужно указать индекс началльной позиции, с которой нужно получить подстроку до конца. Во втором варианте указывается начальный и конечный индексы.

Выполним ниже следующее упражнение для апробации

Упражнение 5 . Написать класс Arithmetic, который принимает три аргумента командной строки: два целых, за которыми следует арифметический оператор (+, -, x или /). Программа должна выполнить соответствующий оператор над двумя целыми значениями и вывести результат. Например:

```
> java Arithmetic 3 2 +
3+2=5

> java Arithmetic 3 2 -
3-2=1

> java Arithmetic 3 2 /
3/2=1
```

В этом классе метод main(String[] args) принимает аргумент массив String, который часто (но не обязательно) имеет имя args. Этот аргумент захватывает параметры командной стоки, представленные пользователем, когда программа вызывается. Например, если пользователь вводит:

```
> java Arithmetic 12345 4567 +
```

Три параметра командной строки "12345", "4567" и "+" будут представлены в массиве String {"12345", "4567", "+"} и переданы в метод main() как аргумент args. При этом,

```
args = {"12345", "4567", "+"}; // args - массив String
args.length = 3;       // длина массива
args[0] = "12345";     // 1-й элемент массива String
args[1] = "4567";      // 2-й элемент массива String
```

```
args[2] = "+";          // 3-й элемент массива String
args[0].length()=5;     //длина 1-го элемента массива
args[1].length()=4;     // длина 2-го элемента массива
args[2].length()=1;     // длина 3-го элемента массива
```

Шаблон программы может иметь следующий вид:

```
public class Arithmetic {
  public static void main (String[] args) {
    int operand1, operand2;
    char theOperator;
   // Проверка количества параметров в командной строке
   //  массива  args[]  с  использованием  переменной
length.
    if (args.length != 3) {
      System.out.println("Пользователь: java Arithmetic
int1 int2 op");
      System.exit(0);
    }
   // Преобразовать 3 Strings args[0], args[1], args[2]
в int и char.
   // Используем Integer.parseInt(aStr) для преобразо-
вания String в int.
    operand1 = Integer.parseInt(args[0]);
    operand2 = ......
    // Получить оператор, предполагая, что он - 1-й
символ
    //   3-ей строки. Используйте метод charAt() для
String.
    theOperator = args[2].charAt(0);
    System.out.print(args[0]  +  args[2]  +  args[1]  +
"="); // Вывод операнодов и операцию между ними
      switch(theOperator) {
      case  ('-'):  System.out.println(operand1  -  oper-
and2); break;
      case ('+'): ......
      case ('x'): ......
      case ('/'): ......
      default:
      System.out.println("Error: invalid operator!");
    }
  }
}
```

Массивы

Мы уже кратко рассмотрели концепцию *массивов* для хранения наборов экземпляров сущностей, и в этом разделе рассмотрим более подробно работу с массивами. Массив – это не что иное, как набор эле-

ментов одного и того же типа. Массив не может хранить более чем один примитивный или объектный тип.

Квадратные скобки в утверждениях Java представляют собой необходимую часть синтаксиса коллекций Java, а *не* указывают на необязательные элементы. В иерархии классов массивы (`Arrays`) непосредственно расширяют класс `Object`, наряду с коллекциями (`Collections`), как показано на Рис. 44.

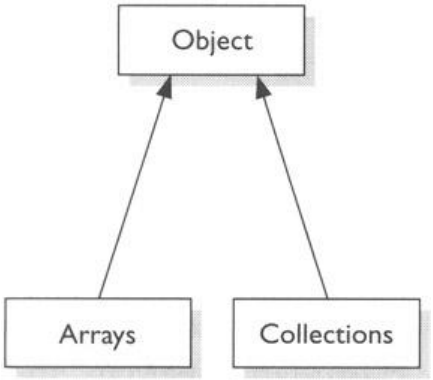


Рис. 44. Иерархия массивов и коллекций

Объявление массива

Чтобы создать массив, необходимо объявить переменную соответствующего типа и инициализировать экземпляр объект массива, на который ссылается переменная. Вы можете объявлять и создавать экземпляр массива в отдельных операциях, или вы можете объединить объявление и инициализацию массива в одном операторе. Чтобы объявить переменную массива, вы кодируете квадратные скобки после типа или имени переменной. Большинство программистов предпочитают кодирования скобки после типа массива.

Чтобы создать экземпляр массива, можно использовать ключевое слово `new` и указать длину, или размер массива в скобках после типа массива. Вы можете указать длину путем кодирования значения литерала или с помощью константы или переменной типа `int`.

Когда инициализируется массив примитивных типов, числовые типы устанавливаются в нули и логические типы в `false`. Когда создается массив объектов, элементам массива присваивается `null`.

Общая формула объявления массива следующая: `<тип элементов>[] <имя переменной> [= new <тип элементов> [размер]];`. Жирными скобками обозначена возможность отложенной инициализации массива. Массив можно объявить и инициализировать двумя способами:

- с определенной размерностью, фиксированной на весь срок жизни массива;
  ```
  // создает пустой массив на 5 элементов:
  int[] integers = new int[5];
  ```
- с определенным набором начальных значений. Размерность этого множества определяет размер массива – он будет в точности таким, чтобы вместить все указанные в фигурных скобках значения. Его размер будет оставаться неизменным в течение всего срока службы массива Например,
  ```
  // создает массив из 5 элементов со значениями:
  int[] integers = new int[] { 1, 2, 3, 4, 5 };
  ```
  Оператор выше создает массив и сразу его инициализирует. Начальные значения заключены в фигурные скобки и разделяются запятыми.

Создание массивов

Более трудоемким способом создания массива является его объявление и инициализация определенного размера с последующим присвоением в цикле необходимых значений элементам массива. Например, после выполнения утверждения `int[] integers = new int[5];` инициируется целочисленный массив из пяти элементов с начальным значением 0 для каждого. После инициализации массива обычно в цикле присваиваются действительные значения элементов массива:

```
int[] integers = new int[5];
for (int i = 0; i < integers.length; i++) {
  integers[i] = i;
}
```

Чтобы присвоить значения массиву, нужно перебрать целые числа от 0 до длины массива без 1 (длину можно определить, вызвав свойство `.length` этого массива, которое мы упоминали ранее).

Если попытаться вставить в массив больше пяти элементов, среда исполнения Java выдаст *исключение (exception) ArrayIndexOutOfBoundsException - ошибку выполнения,* о которых речь пойдет далее.

Описание и инициализация массива могут быть разделены в коде. Так, вначале можно объявить массив, например, с помощью утверждения `int integers [];`. При этом переменная `integers` будет неопределена и не может использоваться в других утверждениях, кроме утверждения инициализации, например, `integers = new int[] { 1, 2, 3, 4, 5, 6 };`

Индекс элемента

Массив можно представить себе как последовательность элементов, в каждой из которых находится значение определенного в масииве типа. Доступ к каждой ячейке осуществляется с помощью *индекса*: `element = arrayName [elementIndex];`

Таким образом, чтобы извлечь элемент, нужно обратиться к массиву (по имени) и указать индекс местоположения элемента. В качестве индекса должны быть использованы литерал или переменная целого типа.

Свойство `length`

Мы уже использовали полезное свойство массива `length`. Поскольку это не метод, а свойство, то синтаксис не включает круглых скобок. Достаточно после установленной точки после имени массива ввести слово `length`, и возвращается длина массива.

В языке Java массивы начинаются с нулевой позиции. Так что первый элемент некоторого массива с именем `array` всегда находится по индексу `array [0]`, а последний – `array[array.length - 1]`.

Статические методы класса Array

Класс `Array` пакета `java.util` содержит несколько статических методов, которые можно использовать для сравнения, сортировки и поиска в массивах. Кроме того, вы можете использовать этот класс для присвоения значений для одного или нескольких элементов массива. Ниже представлены статические методы и их использование:

- `Arrays.fill (arrayName, value)`- Заполняет все элементы заданного массива указанным значением. Например,

```
int[] quantities = new int [5] ;
Arrays.fill(quantities, 1);// Все элементы = 1
```

- `Arrays.fill (arrayName, indexl, index2, value)` - Заполняет элементы указанного массива указанным значением, начиная с элемента `indexl`, но не включая элемент `index2`. Например,

```
int[] quantities = new int [5];
Arrays.fill(quantities, 1, 4, 100);// элементы 1, 2, и 3=100
```

- `Arrays.equals (arrayNamel, arrayName2)` - Возвращает логическое значение `true`, если оба массива имеют и тот же тип, и все элементы в массива равны друг другу. Допустим, имеется два совершенно одинаковых массива:

```
String[] titlesl = {"Война и мир", "Унесенные ветром"};
```

```
String[] titles2 = {"Война и мир", "Унесенные ветром"};
```

Приведем фрагмент кода, сравнивающий эти массивы:

```
 if (titles1 == titles2)
     System.out.println( "titles1 == titles2 is true");
 else
     System.out.println("titles1 == titles2 is false");
if (Arrays.equals(titles1, titles2))
     System.out.println                              ("Ar-
rays.equals(titles1,titles2) is true"); else
     System.out.println("Arrays.equals(titles1,titles2)
is false");
```

Фрагмент кода выше выводит:

```
titles1 == titles2 is false
Arrays.equals (titles1, titles2) is true
```

Оператор == при сравнении двух объектов не обнаруживает тождествнности, поскольку переменные представляют собой различные объекты (занимающие различные места в памяти). Этот оператор "признал" равенство в случае, если бы инициализация `titles2` выполнялась бы при помощи оператора, например, `String[] titles2 = titles1;`

Оператор `Arrays.equals` сравнивает исключительно содержание массивов и признает их эквивалентными.

- `Arrays.copyOf (arrayName, length)` - Копирует указанный массив, усекая или добавляя значениями по умолчанию, в случае необходимости, чтобы копия имела заданную длину. Метод введен с JDK 6. Например, код

```
double[] grades = {92.3, 88.0, 95.2, 90.5};
double[] percentages = Arrays.copyOf(grades,
grades.length+1);
percentages[1] = 70.2;    // не изменяет grades[1]
System.out.println("grades[1]=" + grades[1]);  // выво-
дит 88.0
for (double grade : percentages) {
     System.out.print(grade + " ");
}
```

выводит:

```
grades[1]=88.0
92.3 70.2 95.2 90.5 0.0
```

В предложенном фрагменте кода массив `percentages` образован на основе массива `grades` с указанием размера создаваемого массива на 1 больше исходного. Затем в созданном массиве изменяется второй элемент, и выводятся значения полученного массива `percentages`.

Для выполнения операции копирования до версии JDK 1.6 можно применять метод `arraycopy` класса `System` следующего формата:

```
System.arraycopy(fromArray, intFromlndex, toArray, intToIndex, intLength);
```

Ниже следующее утверждение выполнит такое же копирование, как только что рассмотренное методом `copyOf`:

```
System.arraycopy(grades, 0, percentages, 0, grades.length+1);
```

- `Arrays.copyOfRange (arrayName, indexl, index2)` - Копирует указанный диапазон указанного массива в новый массив.

- `Arrays.sort (arrayName)` - Сортирует элементы массива в порядке возрастания. Например, фрагмент кода

```
int[] numbers = { 2, 6, 4, 1, 8, 5, 9, 3, 7, 0 };
Arrays.sort(numbers);
for (int num : numbers) {
     System.out.print(num + " ");
}
```

выводит:

```
0 1 2 3 4 5 6 7 8 9
```

- `Arrays.sort (arrayName, indexl, index2)` - Сортирует элементы массива в порядке возрастания от `indexl` элемента, но не включая элемент `index2`. Результат применения метода `Arrays.sort(numbers,1,4);` к только что рассмотренному массиву `numbers` выведет результат `2 1 4 6 8 5 9 3 7 0`, в котором сортировка выполнена к указанному диапазону элементов массива.

- `Arrays.binarySearch (arrayName, value)` - Возвращает целое значение индекса указанного значения в указанном массиве. Возвращает отрицательное число, если указанное значение не найдено в массиве. Для правильной работы этого метода необходимо предварительно отсортировть массив методом sort. Например, фрагмет кода

```
String[] productCodes = {"dbms", "jsps", "Java"};
Arrays.sort(productCodes);
int index = Arrays.binarySearch(productCodes, "jsps");
System.out.println("index of jsps = "+index);
```

выводит:

```
index of dbms = 2
```

Многомерные массивы

Массив, который в качестве своих элементов содержит другие массивы, называется многомерным массивом. Они строятся по принципу "массив массивов". Чаще всего используются двумерные массивы. Такие массивы можно легко представить в виде матрицы. Каждая строка матрицы является обычным одномерным массивом, а объединение всех строк - двумерным массивом, в каждом элементе которого хранится ссылка на определенную строку матрицы.

Трёхмерный массив можно представить себе как набор матриц, каждую из которых мы записали на отдельном листе. Тогда чтобы добраться до конкретного числа сначала нужно указать номер листа (первый индекс трёхмерного массива), потом указать номер строки (второй индекс массива) и уже потом в строке (третий индекс).

Для того, чтобы обратиться к элементу n-мерного массива нужно указать n индексов.

Объявляются массивы так:

```
int[] d1; //Обычный, одномерный
int[][] d2; //Двумерный
double[][][]d3; //Трёхмерный
int[][][][][]d4; //Пятимерный
```

При создании массива можно указать явно размер каждого его уровня:

```
d2 = int[3][4]; // Матрица из 3 строк и 4 столбцов
```

Но можно указать только размер первого уровня:

```
int[][] dd2 = int[5][]; /* Матрица из 5 строк. Сколько элементов будет в каждой строке пока не определено. */
```

В последнем случае, можно создать двумерный массив, который не будет являться матрицей из-за того, что в каждой его строке будет различное количество элементов. Например:

```
for(int i=0; i<5; i++) {
  dd2[i] = new int[i+2];
}
```

В результате получим двумерный массив, как показано ниже:

```
0 0
0 0 0
0 0 0 0
0 0 0 0 0
0 0 0 0 0 0
```

Можно создать массив, явно указав его элементы:

```
int[][] ddd2 = {{1,2}, {1,2,3,4,5}, {1,2,3}};
```

При этом можно обратиться к элементу с индексом 4 во второй строке `ddd2[1][4]`, но если мы обратимся к элементу `ddd2[0][4]` или `ddd2[2][4]` — произойдёт ошибка, поскольку таких элементов не существует. Эта ошибка будет происходить уже во время исполнения программы (т. е. компилятор её не увидит).

Для того, чтобы узнать размер строки двумерного массива `ddd2`, следует обратиться к соответствующему свойству `length` следующим образом `ddd2[0].length, ddd2[1].length,...`

Обычно используются двумерные массивы с равным количеством элементов в каждой строке, и для их обработки используются два вложенных друг в друга цикла с разными счётчиками, например, так:.

```
static final int ROWS = 2; static final int COLS = 3;
public static void main(String[] args) {
     int grades[][] = new int[ROWS][COLS];
     grades[0][0] = 0;
     grades[0][1] = 1;
     grades[0][2] = 2;
     grades[1][0] = 3;
     grades[1][1] = 4;
     grades[1][2] = 5;
     for (int rows = 0; rows < ROWS; rows++) {
       for (int cols = 0; cols < COLS; cols++) {
          System.out.printf("%d ", grades[rows][cols]);
       }
       System.out.println();
     }
}
```

В результате выполнения приведенного кода будет выведено:

```
0 1 2
3 4 5
```

В данном случае организован цикл по строкам, но аналогично можно организовать цикл по столбцам, если того требует алгоритм обработки:

```
for (int cols = 0; cols < COLS; cols++) {
     for (int rows = 0; rows < ROWS; rows++) {
         System.out.printf("%d ", grades[rows][cols]);
     }
     System.out.println();
}
```

Вывод в этом случае будет выглядеть следующим образом:

```
0 3
1 4
2 5
```

Массив объектов

Как уже упоминалось, кроме примитивных значений, массивы могут содержать объекты. Поэтому массив в языке Java является наиболее часто употребляемой коллекцией.

Создание массива объектов типа `java.lang.Integer` не отличается существенным образом от создания массива простого типа `int`. Так же можно сделать двумя способами создание массива:

```
// создает пустой массив на 5 элементов:
Integer[] integers = new Integer[5];
```

В отличие от простого типа элементами только что инициированного массива будут `null`, поскольку элементам массива не присвоена ссылка на конкретные объекты.

```
// создает массив из 5 элементов со значениями:
Integer[] integers = new Integer[] { Integer.valueOf(1), Integer.valueOf(2),Integer.valueOf(3), Integer.valueOf(4), Integer.valueOf(5)};
```

Упаковка и распаковка

Как мы уже отмечали, для каждого простого типа в языке Java имеется аналог объектной переменной, как видно из Таблицы 1.

Каждый класс содержит методы для анализа и преобразования из своего внутреннего представления в соответствующий примитивный тип. Например, следующий фрагмент кода преобразует десятичное значение 234 в `Integer`:

```
int value = 234;
Integer packedValue = Integer.valueOf(value);
```

Эта операция называется *упаковкой*, поскольку примитивный тип помещается в обертку, или в конверт.

Аналогично, для преобразования представления `Integer` обратно в его аналог `int`, следует *распаковать*:

```
Integer packedValue = Integer.valueOf(234);
int intValue = packedValue.intValue();
```

Автоматическая упаковка и распаковка

Вообще говоря, нет необходимости явно упаковывать и распаковывать простые типы. Вместо этого можно использовать функции автоматической упаковки и распаковки языка Java, например:

```
int intValue = 234;
Integer packedValue = intValue;
// работает простое присвоение
intValue = packedValue;
```

Анализ и преобразование упакованных типов

Мы рассмотрели, как получить упакованный тип, но как поместить значение `String`, в соответствующий конверт? Классы JDK

предусматривают методы для выполнения таких преобразований, например:

```
String strNumeric = "234"; // Обатите внимание на тип
Integer convertedValue = Integer.parseInt(strNumeric);
```

Также имеется возможность преобразовать содержимое упакованного типа в тип `String`:

```
Integer packedValue = Integer.valueOf(234);
String characterNumeric = packedValue.toString();
```

Обратите внимание, что при использовании оператора конкатенации в выражении `String` примитивный тип автоматически упаковывается, а упакованные типы автоматически вызываются посредством метода `toString()`, наследованного от класса `Object`.

Коллекции Java

Большинство реальных приложений имеет дело с группами сущностей: файлы, переменные, записи в файлах, результаты запросов к базе данных и т.п. До сих пор нам известен только один способ хранения коллекции объектов - массивы Java, которые хороши для хранения, но проявляют недостатки, когда размер коллекции не фикстрован и необходимо динамически добавлять или удалять данные, или сортировать их. Язык Java имеет мощную структуру интерфейсов и классов в `java.util`, которые позволяет создавать и управлять группами объектов разного типа. Типичный класс коллекции реализует несколько интерфейсов, которые представляют собой хорошо разработанную иерархию. Например, `ArrayList` реализует интерфейс `List`, который расширяет `Collection`. Данный раздел не является полным описанием коллекций Java; он знакомит лишь с некоторыми наиболее употребительными классами и рассматривает способы их применения.

`Collection` является корневым интерфейсом в иерархии, из которого производятся интерфейсы `List`, `Queue` и `Set`, реализующиеся в кассах на нижнем уровне иерархии, как представлено на Рис. 45.

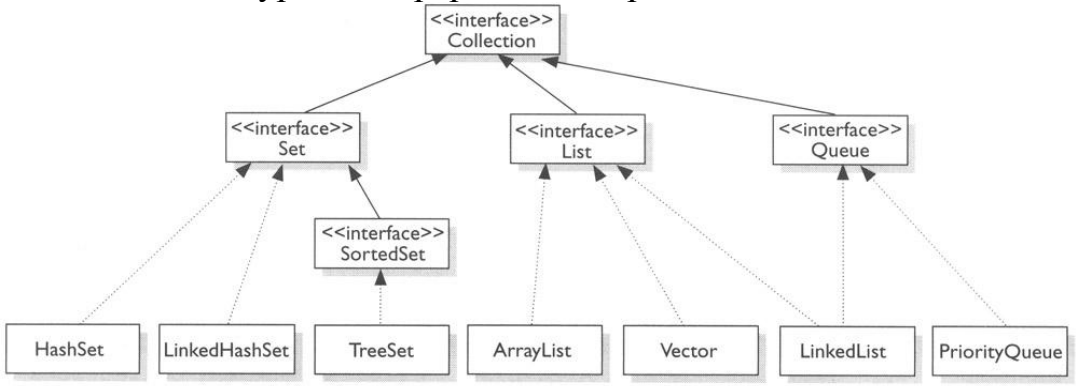


Рис. 45. Иерархия интерфейсов и классов `Collection`

В представленной иерархии показаны следующие интерфейсы коллекции:

- `Set` - коллекция набора, не содержащая дубликатов.
- `List` - упорядоченная коллекция списка, содержащего дубликаты.
- `Queue` - обычно коллекция очереди FIFO (первым пришел - первым обслужись - first-in, first-out), которая моделирует обслуживание в типичной очереди - каждый новый элемент добавляется в конец очереди, и элементы извлекаются с начала очереди. Могут быть определены другие порядки обслуживания (LIFO - стек (последним пришел - первым обслужись - last-in, first-out), которая извлекает из очереди последнего добавленного в нее. Существуют многие другие дисциплины обслуживания. Здесь интерфейс `Queue` приводится для полноты, но не рассматривается в дальнейшем.

Отображение `Map` не является потомком интерфейса `Collection` и образует свою иерархию, представленную на Рис. 46.

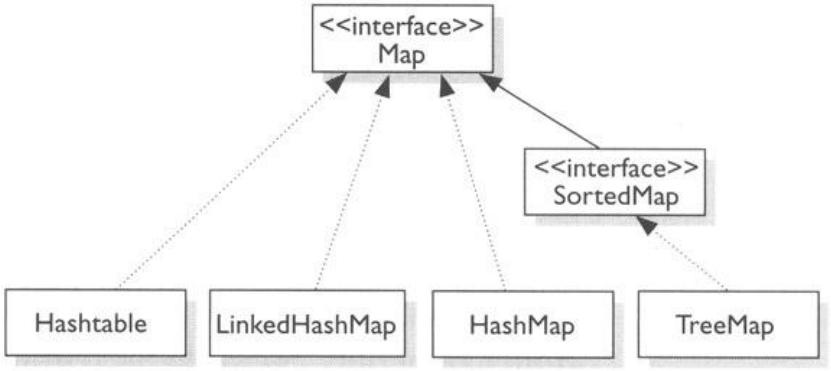


Рис. 46. Иерархия интерфейсов и классов `Map`

`Map` - представляет собой связанный список, который связывает ключи (обычно строковое значение) с объектами и не может содержать дубликаты ключей.

List - Список

Список `List` представляет собой коллекцию, которая, по определению, упорядочена, то есть является *последовательностью*. Поскольку `List` упорядочен, можно полностью управлять местом размещения его элементов. Коллекция Java `List` может содержать только объекты и строго регламентирует их поведение. Используя коллекции, программист применяет существующие структуры данных, не заботясь о том, как они реализованы.

List – не является классом в привычном понимании (это интерфейс), поэтому его экземпляр нельзя создать непосредственно.

Интерфейс являются основой для т.н. контракта, который должен быть реализован в классах. Класс, использующий интерфейс (реализующий интерфейс), должен выполнять все обязательства, определенные в интерфейсе, т.е реализовать все методы интерфейса. В терминах языка Java, класс, который применяет интерфейс должен реализовать все методы, определенные интерфейсом. Интерфейс содержит подготовленные структуры данных, другие интерфейсы и процедуры для обработки этих структур данных. Более подробно мы рассмотрим интерфейсы и их применение позже в настоящем пособии.

Используя List, обычно работают с его реализацией в ArrayList и описывют следующим образом:

```
List<Object> listOfObjects = new ArrayList<Object>();
```

Обратите внимание, что объект ArrayList присваивается переменной типа List. Язык Java позволяет присваивать переменную одного типа переменной другого типа, если слева переменная суперкласса или интерфейс, реализованный присваиваемой переменной. Более подробно рассмотрим процесс присвоения переменных наследуемых типов позже в этом пособии.

Однако можно описывать списки, используя определение на уровне класса, например:

```
ArrayList <Object> listOfObjects = new ArrayList<Object>();
```

Последнее замечание касается всех поддерживаемых коллекций в языке Java.

Объявление формального типа

<Object> - это *формальный тип*, который указывает компилятору на то, что список List содержит коллекцию типа Object, то есть в данном описании в List можно помещать любой объект, поскольку любой объект является потомком объекта <Object>.

Если нужно сделать более строгие ограничения на список объектов, помещаемый в список List, то можно переписать определение List, например, следующим образом:

```
List<Person>listOfPersons=new ArrayList<Person>();
```

Здесь список List может содержать только экземпляры (объекты) класса Person, и компилятор не допустит добавления в список объекта, отличного от типа Person.

Использование `List`

Использовать `List` достаточно просто, как и все коллекции Java в целом. Ниже представлены некоторые методы, которые можно выполнять со списками:

- `add([int index,] variable)` - поместить некоторый объект в `List`;
- `size()` - узнать, какой размер списка `List` в текущий момент;
- `get(int index)` - извлечь объект из списка `List`.

Рассмотрим примеры использования этих методов. Вы уже рассмотрели, как создать экземпляр `List`, создав экземпляр его типа реализации `ArrayList`, с этого списка и начнем.

Чтобы поместить объект в список `List`, необходимо использовать метод `add()`:

```
List<Integer> listOfIntegers = new ArrayList<Integer>();
listOfIntegers.add(Integer.valueOf(234));
```

При выполнении метода `add()` элемент добавляется в конец списка. Можно указать индекс, указывающий место размещения элемента:

```
listOfIntegers.add(2, Integer.valueOf(234));
```

Значение индекса должно быть в диапазоне используемых индексов элементов или на единицу. Если указывается индекс на единицу больше, то элемент добавляется в конец списка, если указывается несуществующий индекс списка, в процессе выполнения выводится сообщение об ошибке нарушения индекса - IndexOutOfBoundsException.

Чтобы узнать длину списка, нужно вызвать метод `size()`:

```
System.out.println("Текущий размер списка listOfIntegers "+ listOfIntegers.size());
```

Чтобы извлечь элемент из списка, нужно вызвать метод `get()` и указать индекс требуемого элемента:

```
System.out.println("Item at index 0 is:" + listOfIntegers.get(0));
```

Интерфейс `Iterable`

На практике список содержит записи файлов или бизнес-объекты, и для их обработки может потребоваться полный перебор элементов. В этой связи важной особенностью коллекций в Java является наличие т.н. итераторов. Итератор (Iterator)— особый класс, который позволяет организовать перемещение по элементам коллекции и удалять её элементы в цикле. Все коллекции из `java.util` реализуют интерфейс `Col-`

lection, который, в свою очередь, расширяет интерфейс java.lang.Iterable. Если коллекция реализует интерфейс java.lang.Iterable, она называется *перебираемой (iterable) коллекцией*. Это означает, что можно обращаться по порядку последовательно ко всем элементам коллекции. Упомянутый итератор и позволяет пробежать по элементам коллекции.

В интерфейсе Iterable описан только один метод: iterator(), который является конструктором объекта типа Iterator, т.е. объекта, который поочерёдно возвращает все элементы коллекции. Применительно к конкретной имеющейся коллекции, например, List<Integer> listOfIntegers = new ArrayList<Integer>(); итератор может описан следужщим образом: Iterator iterator = listOfIntegers.iterator();

В дальнейшем можно использовать переменную iterator для обхода всех элементов списка, например, следующим образом:

```
while (iterator.hasNext()) {
    System.out.println(iterator.next());

}
```

Рассмотрим подробнее методы возвращаемого класса Iterator:

- boolean hasNext() - метод возвращает true, если в коллекции ещё остались элементы и false, если достигнут конец коллекции.
- <объект типа коллекции> next() - метод возвращает текущий элемент. Поскольку итератор настраивается на работу с объектами определенных классов, то этот метод нам будет возвращает не Object, а сразу тот тип, который содержится в списке.
- void remove()- метод удаляет из коллекции последний возвращенный итератором элемент. Этот метод может быть вызван только однократно на один вызов next().

Можно использовать укороченный цикл for-each, с классами, которые реализуют интерфейс java.lang.Iterable. В пособии уже приводился специальный синтаксис для перебора элементов массива, рассмотрим его еще раз применительно к коллекциям, которые также реализуют интерфейс Iterable:

```
for (objectType varName : collectionReference)   {
    // Начинаем сразу использование ObjectType
    //   (через переменную varName)...
}
```

В приведенном выше формате цикла по элементам коллекции объявленная переменная `varName` представляет поочередно объекты коллекции, пробегая значения всех элементов коллекции. К объектам коллекции в цикле обращаться следует через переменную `varName`.

Перебор элементов `List`

Начнем рассмотрение списков с простой программы, в которой создается список олимпийских видов спорта и затем выводится на консоль. Ниже следующий класс выполняет поставленную задачу:

```
import java.util.*;
public class Olympics {
     public static void main(String[] args) {
          // Несколько олимпийских видов спорта
          // предополагается хранить в классе ArrayList
          ArrayList<String> olympicSports = new Ar-
rayList<String>();
          // метод add для добавления в класс
olympicSports
          olympicSports.add("Стрельба из лука");
          olympicSports.add(0,"Бокс");
          olympicSports.add(0,"Крикет");
          olympicSports.add(0,"Прыжки в воду");
          // Несколько олимпийских видов спорта
          System.out.println("В списке представлены " +
olympicSports.size() + " олиспийских видов спорта:");
          // Организация перебора и вывода элементов
списка
          for (String sport: olympicSports) {
               System.out.println(sport);
          }
     }
}
```

Оператор `ArrayList<String> olympicSports = new ArrayList <String>();` описывает и инициализирует список olympicSports. Оператор `olympicSports.add("Стрельба из лука");` добавляет вид спорта в конец списка. Последующие операторы add добавляют элементы в начало списка, смещая на единицу элементы по порядку.

В результате будет выведена следующая информация:

```
В списке представлке 4 олиспийских видов спорта:
Прыжки в воду
Крикет
Бокс
Стрельба из лука
```

Рассмотрим еще один пример, в котором используется метод fibonacci, возвращающий список List некоторым образом полученных числовых величин, пусть это будут числа Фибоначчи, и сигнатура этого метода будет следующей:

```
public List<Integer> fibonacci(int limit);
```

В методе fibonacci создается список List<Integer> itemsIds = new ArrayList<Integer>(); и добавляются вычисленные числа Фибоначчи с использованием метода add() для списка itemsIds. Тогда фрагмент кода в методе main() этого же класса может инициализировать список listOfIntegers, обращаясь к методу fibonacci и выводить полученные числа Фибоначчи:

```
List<Integer> listOfIntegers = fibonacci(50);
for (Integer i : listOfIntegers) {
          System.out.println("Число Фибоначчи : " + i);
}
```

Короткий фрагмент кода выше выполняет то же самое, что и более длинный фрагмет кода ниже:

```
List<Integer> listOfIntegers = fibonacci(50);
for (int i = 0; i < listOfIntegers.size(); i++) {
       Integer n = listOfIntegers.get(i);
     System.out.println ("Число Фибоначчи " + n);
}
```

Также можно использовать итератор как в следующем фрагменте кода:

```
List<Integer> listOfIntegers = fibonacci(50);
Iterator<Integer> iter = listOfIntegers.iterator();
while (iter.hasNext()) {
     System.out.println ("Число Фибоначчи " + iter.next());
}
```

В первом фрагменте используется сокращенный синтаксис: в нем нет индексной переменной для инициализации и отсутствует вызов метода get() по отношению к listOfIntegers. Вообще говоря, такая упрощенная конструкция представляет собой т.н. "синтаксический сахар", который при компиляции "разворачивается" в традиционный оператор for.

Полный код класса SampleFibonacci для выполнения описанной постановки задачи приводится ниже:

```
import java.util.ArrayList;
import java.util.List;
public class SampleFibonacci {
     // Метод fibonacci формирует и возвращает
ArrayList
     public static List<Integer> fibonacci(int limit) {
```

```
                // Инициализируется список ArrayList
                List<Integer> itemsIds = new Ar-
rayList<Integer>();
                int lo = 1;
                int hi = 1;
                itemsIds.add(lo);
                while (hi < limit) {
                     itemsIds.add(hi); // Добавляется
                     hi = lo + hi; // Изменение значения hi
                     lo = hi - lo; /*
           Новое значение lo равно старому hi, то есть сумме
за   вычетом старого lo  */
                }
                return itemsIds;
           }
           public static void main(String[] args) {
                List<Integer> listOfIntegers = fibonacci(50);
                for (int i = 0; i < listOfIntegers.size();
i++) {
                     Integer I = listOfIntegers.get(i);
                     System.out.println ("Число Фибоначчи " +
I);
                }
           }
     }
```

Здесь используется статический метод `fibonacci`, с тем, чтобы не создавать объект класса `SampleFibonacci`, а обращаться к методу класса.

Set - Набор

Набор (`Set`) – это коллекция, которая, по определению, содержит только уникальные элементы – в ней отсутствуют дубликаты объектов. Если `List` может содержать одни и те же объекты многократно, то `Set` может содержать данный объект только один раз. Java коллекция `Set` может содержать только объекты и строго регламентирует их поведение. Поскольку `Set` является интерфейсом, нельзя создать его экземпляр непосредственно, необходимо использовать одну из реализаций интерфейса `Set`: `HashSet`, `LinkedHashSet` или `TreeSet`.

Все три рассматриваемые коллекции реализуют один тот же интерфейс, но, естественно, отличаются друг от друга. Главное отличие состоит в порядке хранения элементов. `HashSet` хранит элементы в случайном (на первый взгляд) порядке. Однако элементы внутри набора `HashSet` упорядочены в соответствии со значением `hashCode`, которое рассчитывается для каждого элемента. По значению `hashCode`

также осуществляется поиск элемента в наборе. `TreeSet`, в отличие от `HashSet`, хранит элементы упорядоченно, то есть в какой бы последовательности не добавлялись и не удалялись элементы, коллекция останется строго упорядоченной. `LinkedHashSet` используется в том случае, если необходимо помнить порядок добавления элементом. Поиск по этой коллекции происходит также по значению `hashCode`, но порядок будет всегда совпадать с очерёдностью добавления.

Рассмотрим некоторые методы, которые позволяет выполнять интерфейс `Set`:

- `boolean add([index,] variable)` - поместить некоторый объект `variable` в набор `Set`; Возвращает `true`, если набор не содержит элемент. Можно указать порядковый номер элемента в наборе в переменной `index;`
- `int size()` - возвращает целое, указывающее какой размер списка `Set` в текущий момент;
- `addAll(Collection c)` - добавляет все элементы коллекции c (если их ещё нет);
- `clear()` - удаляет все элементы коллекции;
- `boolean contains(Object o)` - возвращает `true`, если элемент есть в коллекции;
- `boolean containsAll(Collection c)` - возвращает `true`, если все элементы содержатся в коллекции
- `boolean isEmpty()` - возвращает `true`, если в коллекции нет ни одного элемента;
- `boolean remove(Object o)` - удаляет первое вхождение указанного элемента из этого списка, если он существует. Если список не содержит элемент, он остается неизменным.
- `boolean removeAll(Collection c)` - удаляет из этого набора все его элементы, которые содержатся в указанном наборе. Если указанная коллекция также набор, эта операция эффективно изменяет этот набор, так что его значение является ассиметричной разностью двух множеств;
- `boolean retainAll(Collection c)` - оставляет только элементы в этом наборе, которые содержатся в ука-

занном наборе. Другими словами, удаляет из этого множества все элементы, которые не содержатся в указанном наборе. Если указанная коллекция также набор, эта операция эффективно изменяет этот набор, так что его значение является пересечением двух множеств;

- Object`[] toArray()` - возвращает массив, содержащий элементы коллекции.

Использование Set

В качестве примера использования методов `Set` продемонстрируем различие в способе хранения наборов различных типов. Для этого создадим три упомянутые набора, заполним эти наборы случайными числовыми значениями в диапазоне от 0 до 100 и выведем полученные наборы на консоль. Кроме того, случайные значения будем помещать в фиксированный массив для последующего анализа значений наборов. Поставленную задачу решает представленный ниже класс `SetsSamples`:

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Random;
import java.util.TreeSet;
public class SetsSamples {
     static int count = 20;
     static HashSet<Integer> hashSet = new
HashSet<Integer>();
     static TreeSet<Integer> treeSet = new
TreeSet<Integer>();
     static LinkedHashSet<Integer> linkedHashSet = new
LinkedHashSet<Integer>();
     static int array[] = new int[count];
     public static void fillSets() {
          Random rand = new Random();
          for (int i = 0; i < count; i++) {
               Integer element = rand.nextInt(100);
               array[i] = element;
               hashSet.add(element);
               treeSet.add(element);
               linkedHashSet.add(element);
          }
     }
     public static void print() {
          System.out.print("Array: \t\t" + "[");
          for (int i : array)
```

```
                    System.out.print(i + ",");
               System.out.println("\b\b" + "] (" + ar-
ray.length + ")");
               System.out.println("HashSet: \t" + hashSet +
" (" + hashSet.size() + ")");
               System.out.println("TreeSet: \t" +
treeSet.toString() + " (" + treeSet.size() + ")");
               System.out.println("LinkedHashSet: \t" +
linkedHashSet.toString() + " (" + linkedHashSet.size()
+ ")");
               System.out.println("Circle hashSet by itera-
tor");
               Iterator<Integer> iterator =
hashSet.iterator();
               while (iterator.hasNext()) {
                    System.out.print(iterator.next()+" ");
               }
          }
          public static void main(String[] argc) {
               fillSets();
               print();
          }
     }
```

Вывод выше приведенного класса представлен ниже:

```
Array:        [61,31,1,68,93,97,36,98,90,3,27,31,41,14,61,70,70,24,76,3]
(20)
HashSet:      [1,97,98,3,35,68,36,70,41,76,14 24,90,27,61,93,31] (17)
TreeSet:      [1,3,14,24,27,31,35,36,41,61,68,70,76,90,93,97,98] (17)
LinkedHashSet: [61,31,1,68,93,97,36,98,90,3,27,41,14,70,24,76,35] (17)
Circle hashSet by iterator
1 97 98 3 35 68 36 70 41 76 14 24 90 27 61 93 31
```

Отличительной особенностью `Set` является обеспечение уникальности своих элементов, поэтому из 20 произведенных и помещенных в массив `Array` случайных чисел, принимается 17 оригинальных элементов набора. (При повторном запуске данного примера получится другой состав случайных величин). Если внимательно посмотреть на значения массива `Array`, то можно заметить, что значения 61, 31 и 70 повторяются в массиве и поэтому повторы не входят в каждый из типов набора. Сравнительный анализ состава сформированных из случайных числовых наборов данных подтверждает ранее указанные характеристики наборов.

Итак, в классе `SetsSamples` определяются статическая переменные `count` (количество случайных величин), инициируются статические переменные `hashSet`, `treeSet`, `linkedHashSet`, соответственно определяющие три типа наборов - `HashSet`, `TreeSet`,

LinkedHashSet; также инициируется статический массив Array. Кроме того в классе SetsSamples определено три метода:

- void fillSets() - для генерации случайных величин и заполнения ими ранее определенных наборов и массива. Здесь для генерации случайных величин используется переменная rand - объект класса java.util.Random, который представляет собой генератор псевдослучайных чисел. Конструктор Random() создаёт генератор чисел, использующий уникальное начальное число. Для производства следующего случайного значения типа int используется метод nextInt(int n), который возвращает число в диапазоне от 0 до n;
- void print() - для оформления вывода элементов набора и количества элементов к каждом объекте. Следует обратить внимание на то, что в составе методов интерфейса Set отсутствует какой-либо метод, который возвращает конкретный элемент набора, и единственный способ обрабатвать набор - использовать итератор. Различные способы обхода элементов коллекции используются в данном методе. Кроме того, при выводе значений используется явное (необязательное) обращение к методу toString() наборов, который выполняет для нас полезную работу по выводу всех элементов без организации цикла;
- в методе main() выполняется обращение последовательно к методу fillSets() и print().

Map - Отображение

Интерфейс Map соотносит уникальные ключи со значениями. Ключ - это объект, который используется для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект отображения. После того как это значение сохранено, вы можете получить его по ключу.

Интерфейс отображения описывется interface Map<K, V>. В параметре K указывается тип ключей, в V - тип хранимых значений.

Отображение (Map) – это удобная структура, которая позволяет связать один объект (*ключ*), с другим объектом (*значение*). Естественно, ключ Map должен быть уникальным, и он используется для последующего извлечения значений. Коллекция Map Java может содержать только объекты и строго регламентирует их поведение.

Поскольку Map является интерфейсом, его экземпляр нельзя создать непосредственно, и необходимо использовать одну из реализаций интерфейса Map: HashMap, LinkedHashMap или TreeMap. HashMap обеспечивает максимальную скорость выборки, а порядок хранения его элементов не очевиден. TreeMap хранит ключи отсортированными по возрастанию, а LinkedHashMap хранит ключи в порядке вставки, но не обеспечивает скорость поиска HashMap. Далее рассматривается реализации HashMap, которая проста в использовании.

Ниже некоторые методы, которые можно выполнять с Map:

- put (String key, value) - поместить объект в Map;
- variable get(String key) - извлечь объект из Map;
- int size() - возвращает количество пар "ключ-значение" в отображении;
- Set<K> keySet() - возвращает набор, содержащий ключи вызывающего отображения. Метод предоставляет ключи вызывающего отображения в виде набора и позаоляет организовать перебор значений отображения;
- toString() - вывести содержимое в виде фигурных скобок, где ключи и значения разделяются знаком равенства. Ключи слева, значения справа;
- boolean containsKey(Object k) - возвращает значение true, если вызывающее отображение содержит ключ k, false - в противном случае;
- boolean containsValue(Object v) - возвращает значение true, если вызывающее отображение содержит значение v, false - в противном случае;
- boolean equals(Object o) - возвращает значение true, если параметр o - это отображение, содержащее одинаковые значения, false - в противном случае;
- V get(Object k) - возвращает значение, ассоциированное с ключом k. Возвращает значение null, если ключ не найден;
- boolean isEmpty() - возвращает значение true, если вызывающее отображение пустое, false - в противном случае;
- V put(K k, V v) - помещает элемент в вызывающее отображение, переписывая любое предшествующее значение, ассоциированное с ключом. Возвращает null, если

ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом;

- `V remove(Object k)` - удаляет элемент, ключ которого равен `k`;
- `Collection<V> values()` - возвращает коллекцию, содержащую значения отображения.

Использование Map

Чтобы поместить объект в отображение `Map`, необходимо определить ключ объекта и значение объекта. Обычно в качестве ключа применяется объект типа `String`. Рассмотрим пример, в котором используется метод `createMapOfIntegers`, который возвращает сформированное отображение `Map<String, Integer>`:

```
public static Map<String, Integer> createMapOfIntegers() {
    Map<String, Integer> mapOfIntegers = new HashMap<String, Integer>();
        mapOfIntegers.put("january", Integer.valueOf(31));
        mapOfIntegers.put("february", Integer.valueOf(28));
        mapOfIntegers.put("march", Integer.valueOf(31));
        mapOfIntegers.put("april", Integer.valueOf(30));
        // . . .
        mapOfIntegers.put("december", Integer.valueOf(31));
        return mapOfIntegers;
}
```

В этом примере `Map` содержит объекты типа `Integer` с ключом типа `String`, который служит символьным представлением месяца. Метод определен как статический, для того, чтобы обращаться к нему без создания экземпляра класса. Чтобы получить определенное числовое значение количества дней в месяце из отображения (типа `Integer`), указывается его символьное наименование:

```
mapOfIntegers Map <String, Integer> mapOfIntegers = createMapOfIntegers();
System.out.println (mapOfIntegers.get("december"));
```

Метод `get` возвращает значение элемента отображения `Map`, соответствующее названию месяца, если такое название определено в качестве ключа в отображении. В данном случае выводится число 31. Если значение ключа не обнаружено в отображении, то возвращается значение `null`.

Использование Set совместно с Map

В ряде случаев необходимо перебирать весь набор элементов Map, но интерфейс Map не поддерживает реализацию интерфейса Iterable. Поскольку к элементам отображения Map позволяет обращаться исключительно по ключу, для перебора понадобится набор ключей Set, соответствующий данному отображению Map, который содержит уникальные значения и может быть получен с использованием метода keySet(). Код класса ниже демонстрирует перебор всех элементов отображения и полученный вывод:

```
import java.util.*;
public class Code {
     public static void main(String[] args) {
          Map <String, Integer> mapOfIntegers = create-
MapOfIntegers();
          Set<String> keys = mapOfIntegers.keySet();
          for (String key : keys) {
            Integer value = mapOfIntegers.get(key);
            System.out.println ("Month " + key + " ко-
личество дней = " + value);
          }
     }
     public static Map <String, Integer> createMapOfIn-
tegers() {
            Map<String, Integer> mapOfIntegers = new
HashMap<String, Integer>();
            mapOfIntegers.put("january", Inte-
ger.valueOf(31));
            mapOfIntegers.put("february", Inte-
ger.valueOf(28));
            mapOfIntegers.put("march", Inte-
ger.valueOf(31));
            mapOfIntegers.put("april", Inte-
ger.valueOf(30));
            mapOfIntegers.put("may", Inte-
ger.valueOf(31));
            mapOfIntegers.put("june", Inte-
ger.valueOf(30));
            mapOfIntegers.put("july", Inte-
ger.valueOf(31));
            mapOfIntegers.put("august", Inte-
ger.valueOf(31));
            mapOfIntegers.put("september", Inte-
ger.valueOf(30));
            mapOfIntegers.put("october", Inte-
ger.valueOf(31));
            mapOfIntegers.put("november", Inte-
ger.valueOf(30));
```

```
                mapOfIntegers.put("december", Inte-
ger.valueOf(31));
            return mapOfIntegers;
        }
}
```

Полученный вывод:

```
Month november количество дней = 30
Month june количество дней = 30
Month september количество дней = 30
Month may количество дней = 31
Month august количество дней = 31
Month january количество дней = 31
Month february количество дней = 28
Month july количество дней = 31
Month december количество дней = 31
Month october количество дней = 31
Month april количество дней = 30
Month march количество дней = 31
```

Обратите внимание, что при выполнении вызова `System.out.println ("Month " + key + " количество дней = " + value);` вызывается автоматически метод `toString()` класса `Integer`, для переменной `value`, извлекаемой из отображения. Кроме того, из представленного вывода видно, что он не упорядочен - это еще раз доказывает такое свойство набора `HashSet`. Отображение `Map` не возвращает списка `List` своих ключей (отсутствует соответствующий метод), поскольку каждый ключ уникален, а уникальность является характеристикой набора `Set`.

Для практического применения коллекций выполним ниже следующее упражнение.

Упражнение 6 Задача состоит в том, чтобы создать объект `HashMap` с именем `restaurantMenu`, который хранит ключи типа `String` (названия блюд) и значение типа `Integer` (стоимость блюд).

Код ниже создает объект `HashMap`:

```
import java.util.HashMap;
public class Restaurant {

    public static void main(String[] args) {
        HashMap<String, Integer> restaurantMenu = new
HashMap<String, Integer>();

    }
}
```

Добавьте в этот код несколько блюд с их стоимостью, используя метод `add`, например, `restaurantMenu.put("Пицца Хат",`

130); Затем, выведите стоимость двух из введенных блюд, используя метод get, например, System.out.println(restaurantMenu.get("Пицца Хат"));. Затем выведите полный список блюд, содержащихся в объекте restaurantMenu, вставив в класс Restaurant завершенный фрагмент следующего кода:

```
System.out.println();
for (<Сюда вставить код>) {
    System.out.println("Блюдо " + item + " сто-
ит " + restaurantMenu.get(item) + " рублей(я).");
}
```

Вспомним, что список ключей для отображения может быть получен с помощью метода keySet(), например, restaurant-Menu.keySet();

Архивирование Java-кода

Теперь, когда мы достаточно много узнали о написании Java-приложений, вам может понадобиться упаковать их так, чтобы другие программисты могли их использовать, или импортировать код других разработчиков в свои приложения. В этом разделе показано, как это сделать.

Файлы JAR

JDK содержит утилиту jar, которая создает архив приложений Java - Java Archive. Если упаковать код в файл JAR, то другие разработчики могут просто загрузить этот файл в свои проекты и настроить их на использование вашего кода.

Создать файл JAR в Eclipse совсем не сложно. В Packsge Explorer щелкните правой кнопкой мыши на пакете ru.ifmo.intro и в контекстном меню выберите **Export**. Откроется диалоговое окно **Export**, в котором разверните и выберите **Java > JAR file**, и нажмите кнопку **Next**, как показано на Рис. 47.

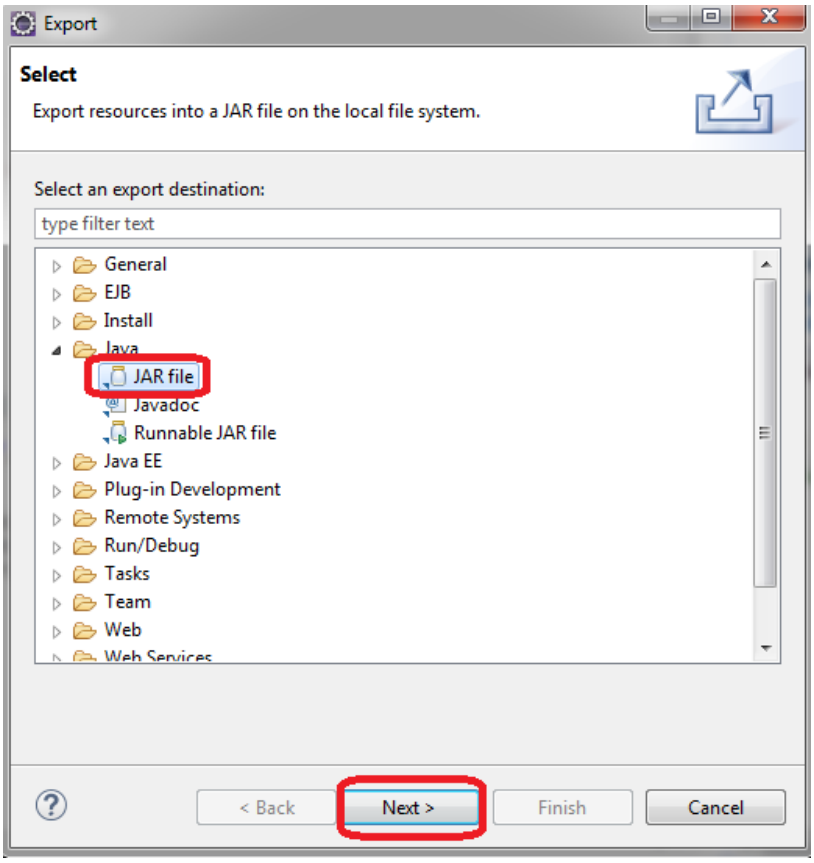


Рис. 47. Диалоговое окно Select.

В результате открывается следующее диалоговое окно **JAR File Specification**, показанное на Рис. 48. В этом диалоговом окне предоставляется возможность отметить экспортируемые ресурсы и указать место в файловой системе (поле **`Select the export destination`**:), где следует сохранить свой файл JAR, указав имя файла с расширением `.jar` – это расширение присваивается по умолчанию. Место можно выбрать в файловой системе, нажав на кнопку **`Browse`**. После заполнения диалогового окна нажмите на кнопку кнопку **`Finish`**.

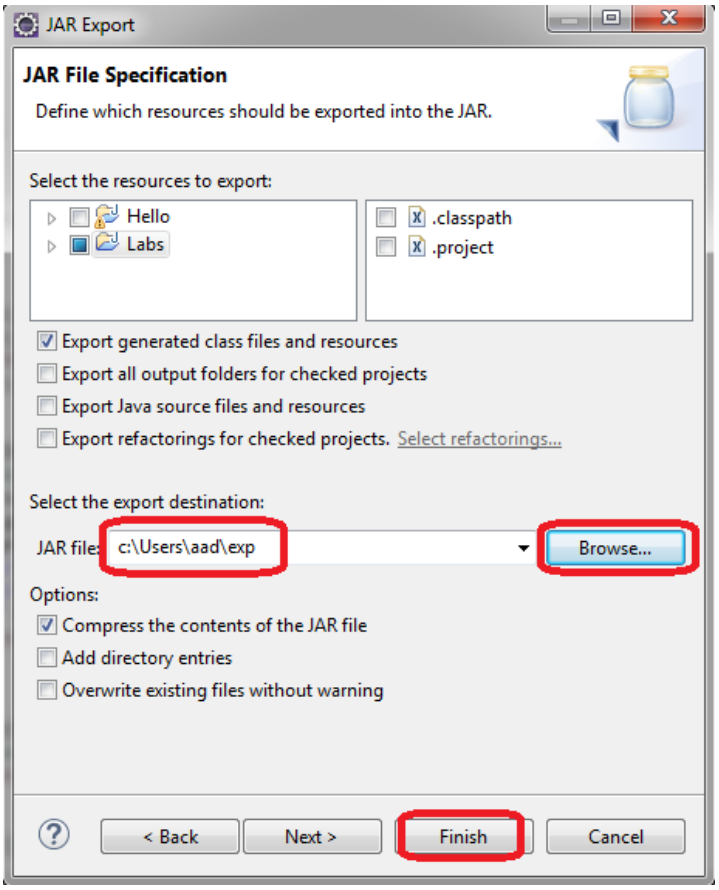


Рис. 48. Диалоговое окно экспорта JAR File Specification

После выполнения описанной процедуры это файл JAR можно будет обнаружить файл в выбранном месте. Если вы поместите этот файл в свою папку сборки в Eclipse, то можете использовать классы из собственного кода. Это также совсем не сложно, как вы увидите далее.

Использование сторонних приложений

Когда вы освоите Java-программирование, вам придется использовать большое количество приложений сторонних разработчиков для выполнения своих задач. Например, допустим, что требуется использовать `joda-time.jar`, библиотеку, заменяющую JDK, для управления, манипуляций и расчетов, связанных с датами/временем.

Допустим, что вы уже загрузили `joda-time` в виде файла JAR. Чтобы использовать классы этого файла, можно создать в своем проекте каталог `lib` (или с другим именем) и поместить туда файл JAR. Для создания каталога щелкните кнопкой мыши на корневой папке **Labs** в окне Project Explorer и, нажав правую кнопку, выберите в контекстном меню **New > Folder**, введите имя папки **lib** и нажмите кнопку **Finish**.

Новый подкаталог появится на том же уровне, что и `src`. Теперь средствами операционной системы скопируйте файл `joda-time`.jar в новый каталог `lib`. В нашем случае файл называется `joda-time-2.7.jar`. (Как правило имя JAR-файла содержит номер версии). Необходимо конфигурировать Eclipse, чтобы он включил классы из файла `joda-time-2.7.jar` в проект. Щелкните правой кнопкой мыши на проекте **Labs** в Project Explorer, а затем выберите **Properties** и в левой части окна выберите **Java Build Path**. В правой части быберите закладку **Libraries**, как показано на Рис. 49.

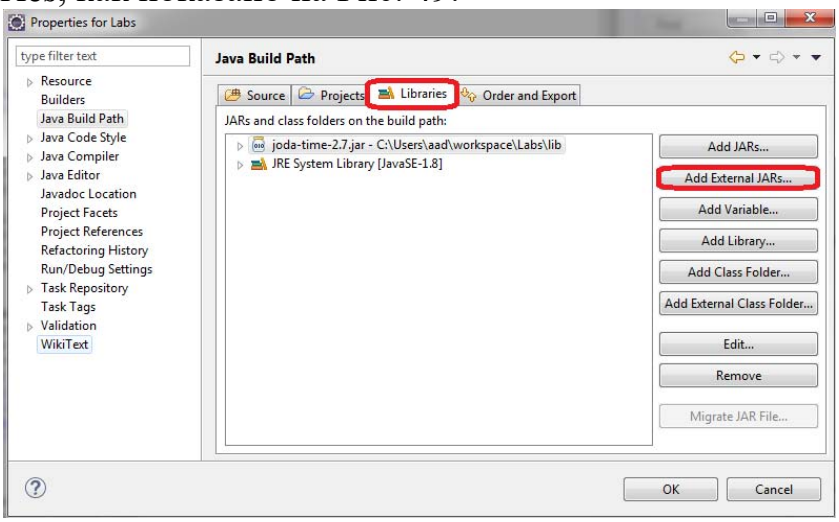


Рис 49. Диалоговое окно Java Build Path

Нажмите кнопку **Add External JARs**, а затем укажите в каталог `lib` проекта, выберите файл `joda-time-2.7.jar` и нажмите кнопку **OK**.

Как только завершится операция конфигурирования, файл JAR станет доступным для ссылки (импорта) в коде проекта Java. Обратите внимание, что в Project Explorer появилась новая папка с именем **Referenced Libraries**, которая содержит файл `joda-time-2.7.jar` как показано на Рис. 50.

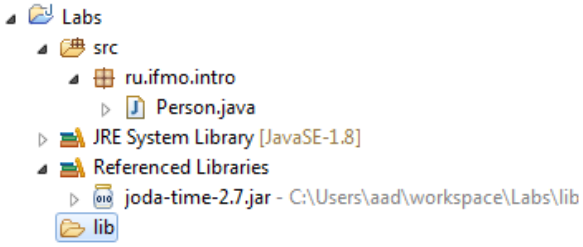


Рис. 50. Результат подключения внешней библиотеки классов

Теперь можно смело использовать классы подключенной библиотеки наряду со стандартными фрейворками.

Расширенные представления ОО подхода

В пособии уже рассмотрено достаточно средств Java для разработки простых приложений, и настало время более детально познакомиться с преимуществами применения объектно-ориентированного подхода для создания сложных сценариев программирования, и, в этой связи, более глубоко рассмотреть темы и примеры применения более сложного взаимодействия классов.

Приступим к рассмотрению способов совершенствования класса `Person`, в том числе:

- перегрузка методов;
- переопределение методов;
- сравнение одного объекта с другим;

Перегружаемые методы

Два или несколько методов с одинаковыми именами, но с разными списками аргументов (сигнатурами) называются *перегружаемыми (overloaded)* методами. Перегружаемые методы всегда располагаются в одном и том же классе, и во время выполнения программы среда исполнения Java (Java Runtime Environment - JRE, или Java Runtime) принимает решение, какой вариант перегружаемого метода вызывается, в зависимости от передаваемых ему аргументов.

Рис. 51 показывает пример применения метода `add` с различными сигнатурами, которые выбираются в процессе выполнения, в зависимости от используемых параметров, передаваемых при обращении к этому методу.

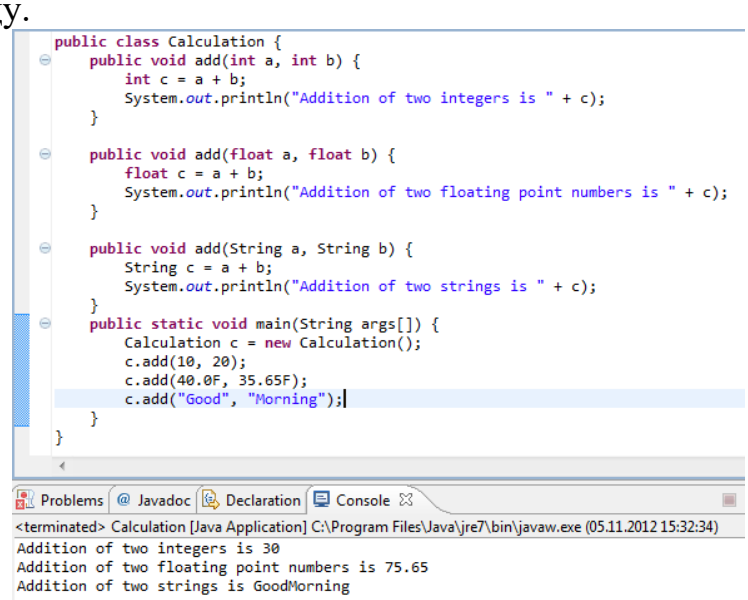


Рис. 51. Пример применения перегружаемых методов

Теперь допустим, что объекту `Person` требуется несколько методов распечатки отчета о своем текущем состоянии. Назовем эти методы `printAudit()`. Поместите ниже представленные перегружаемые методы в классе `Person` в редакторе Eclipse в удобное место.

```
public void printAudit(StringBuilder buffer) {
     buffer.append("Name=");
     buffer.append(getName());
     buffer.append(",");
     buffer.append("Age=");
     buffer.append(getAge());
     buffer.append(",");
     buffer.append("Height=");
     buffer.append(getHeight());
     buffer.append(",");
     buffer.append("Weight=");
     buffer.append(getWeight());
     buffer.append(",");
     buffer.append("EyeColor=");
     buffer.append(getEyeColor());
     buffer.append(",");
     buffer.append("Gender=");
     buffer.append(getGender());
}
public void printAudit() {
     StringBuilder sb = new StringBuilder();
     printAudit(sb);
     print(sb.toString());
}
```

Здесь представлены две перегружаемые версии `printAudit()`, причем одна из них, фактически, использует другую. Предлагая две версии, вы предоставляете вызывающему методу возможность выбора способа вывода данных об объекте класса `Person`. Java Runtime вызовет нужный метод в зависимости от переданных параметров.

Существует два простых правила перегрузки методов, при нарушении которых компилятор выдает сообщение об ошибке:

- нельзя перегружать метод, изменив тип возвращаемого им значения;
- не должно быть двух методов с одной сигнатурой.

Переопределяемые методы

Классы в Java-коде существуют в определенной иерархии. Классы, находящиеся выше данного класса в иерархии, являются суперклассами данного класса. Любой конкретный класс является подклассом каждого класса, находящегося выше в иерархии. Подкласс наследует

свойства и поведение своих суперклассов. На вершине иерархии каждого класса находится объект `Object`. Другими словами, каждый класс является подклассом (и наследует из него) класса `Object`.

Если подкласс некоторого класса создает свою собственную реализацию метода, определенного в родительском классе (суперклассе), то используется *переопределение метода (Overriding*). Мы уже рассматривали такую ситуацию при описании наследования и полиморфизма. Чтобы проанализировать возможности переопределения методов, введем в рассмотрение класс `Employee`, который является подклассом (или дочерним классом) суперкласса `Person`, наследующим его атрибуты и поведение, и добавляет специфические особенности. На Рис.52 изображается диаграмма наследования классов UML.

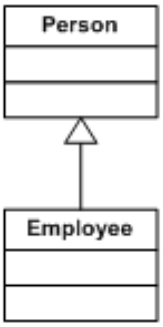


Рис. 52. Класс `Employee` наследует класс `Person`

Пусть класс `Employee` содержит следующие дополнительные атрибуты: код налогоплательщика, идентификатор; дата приема на работу, заработная плата.

Для создания такого класса в файле с именем `Employee.java`, щелкните правой кнопкой мыши на пакете `ru.ifmo.intro` в проекте **Labs** Package Explorer и выберите **New > Class.** Откроется диалоговое окно **New Java Class** для создания нового класса Java, как показано на Рис. 53.

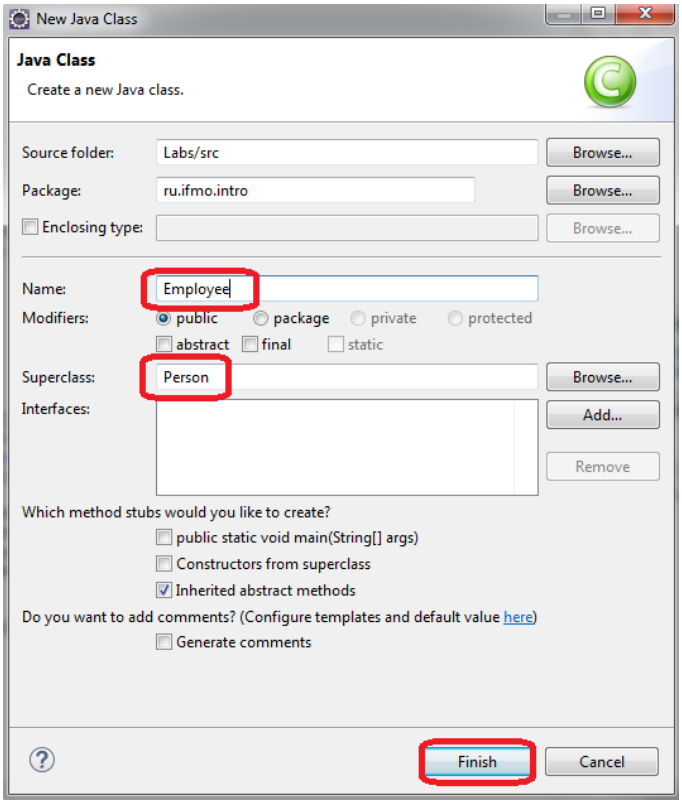


Рис. 53. Диалоговое окно New Java Class

Введите `Employee` в поле Name и `Person` вместо `java.lang.Object` в поле Superclass, затем нажмите кнопку **Finish**. После завершения создания код класса `Employee` появится в окне редактирования. Следует иметь в виду, что конструкторы являются специальными методами, не являются полноправными объектно-ориентированными членами и не наследуются от суперкласса. Таким образом, неявно конструктор суперкласса `Person()` не определяется по умолчанию для конструктора `Employee()` и требуется определить явный конструктор. Для этого убедитесь, что указатель мыши находится в окне редактирования класса `Employee`, и, нажав правую кнопку из контекстного меню последовательно выберите **Source > Generate Constructors from Superclass.** Откроется диалоговое окно **Generate Constructors from Superclass**, показанное на Рис. 54.

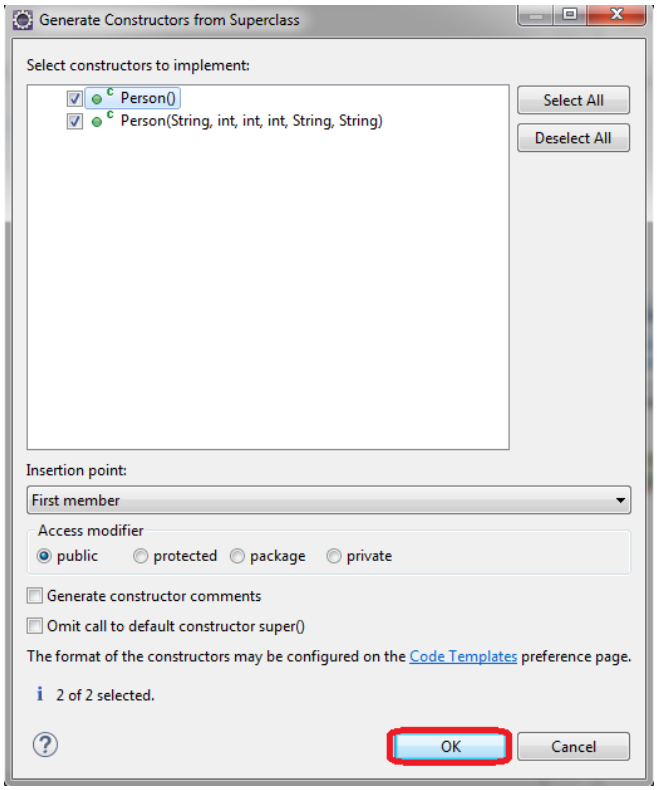


Рис. 54. Создание конструкторов в диалоговом окне Superclass

Отметьте оба конструктора и нажмите кнопку **OK.** Eclipse сгенерирует конструкторы и класс `Employee` будет выглядеть следующим образом:

```
package ru.ifmo.intro;
public class Employee extends Person {
     public Employee() {
          super();
     }
     public Employee(String name, int age, int height,
int weight, String eyeColor, String gender) {
     super(name, age, height, weight, eyeColor, gen-
der);
     }
}
```

Как видно из представленного кода, класс `Employee` (и все его объекты) наследует атрибуты и поведение своего суперкласса `Person`, и кроме того он должен иметь некоторые собственные свойства, такие как:

```
private String taxIDNumber; // код налогоплательщика
private String empID;  // идентификатор
private Date empHireDate;   // дата приема на работу
private BigDecimal empSalary;  // заработная плата
```

Добавьте перечисленные атрибуты в класс `Employee` и сгенерируйте с помощью Eclipse (**Source->Generate Setters and Getters...**) му-

таторы (setter и getter). После этого код класса `Employee` будет иметь следующий вид:

```
package ru.ifmo.intro;
import java.math.BigDecimal;
import java.util.Date;
public class Employee extends Person {
     private String taxIDNumber; // код налогоплатель-
щика
     private String empID;  // идентификатор
     private Date empHireDate; // дата приема на работу
     private BigDecimal empSalary;  // заработная плата
     public Employee() {
          super();
          // TODO Auto-generated constructor stub
     }
     public Employee(String name, int age, int height,
int weight,
               String eyeColor, String gender) {
          super(name, age, height, weight, eyeColor,
gender);
          // TODO Auto-generated constructor stub
     }
     public String getTaxIDNumber() {
          return taxIDNumber;
     }
     public void setTaxIDNumber(String taxIDNumber) {
          this.taxIDNumber = taxIDNumber;
     }
     public String getEmpID() {
          return empID;
     }
     public void setEmpID(String empID) {
          this.empID = empID;
     }
     public Date getEmpHireDate() {
          return empHireDate;
     }
     public void setEmpHireDate(Date empHireDate) {
          this.empHireDate = empHireDate;
     }
     public BigDecimal getEmpSalary() {
          return empSalary;
     }

     public void setEmpSalary(BigDecimal empSalary) {
          this.empSalary = empSalary;
     }
}
```

Переопределение метода: `printAudit()`

Теперь, можно приступить к переопределению методов, в частности, метода `printAudit()`, который использовался для форматирования текущего состояния экземпляра класса `Person`. `Employee` наследует поведение `Person`, и если создать экземпляр `Employee`, установить его атрибуты и вызвать один из перегружаемых методов `printAudit()`, то вызов отработает, но полученный результат не будет в полной мере отражать класс `Employee`. Проблема в том, что метод `printAudit()` не может выводить атрибуты, специфические для `Employee`, поскольку `Person` не запрограммирован делать это, и он даже не знает о классе `Employee`.

Решение проблемы состоит в том, чтобы переопределить метод `printAudit()`, который в качестве параметра принимает `StringBuilder`, и добавить код для вывода атрибутов, специфичных для `Employee`.

Для этого в IDE Eclipse следует выбрать последовательно **Source > Override/Implement Methods...**, и откроется диалоговое окно, представленное на Рис. 55.

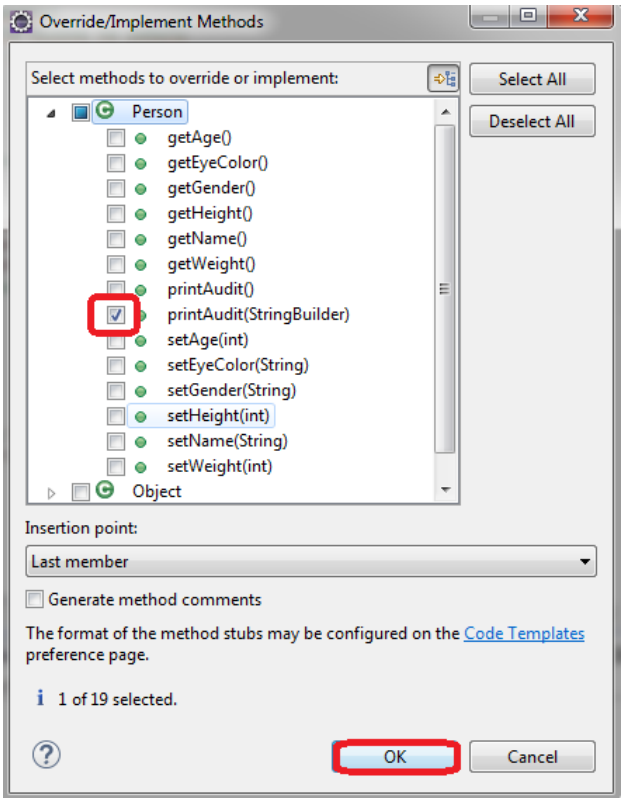


Рис. 55. Диалоговое окно Override/Implement Methods

Выберите из предложенного списка методов printAudit (StringBuilder) и нажмите кнопку **OK**. Eclipse сгенерирует в составе класса Employee шаблон метода следующего вида:

```
@Override
public void printAudit(StringBuilder buffer) {
      // TODO Auto-generated method stub
      super.printAudit(buffer);
}
```

@Override известна как аннотация (annotation) (введена в JDK 1.5), которая предлагает компилятору проверить, имеется ли такой метод в суперклассе для переопределения. Это существенно помогает, если вы ошиблись с именем printAudit(). Если @Override не используется и printAudit() является ошибочным PrintAudit(), это приведет к тому, что метод будет рассматриваться как новый в подклассе, вместо переопределения такового в подклассе. Если используется @Override, компилятор выдаст ошибку. Аннотация @Override не обязательна, но применять ее является хорошим стилем.

Наша задача дополнить вывод данными из объекта класса Employee, например, следующим образом:

```
buffer.append("taxIDNumber =");
buffer.append(getTaxIDNumber());
buffer.append(",");
buffer.append("EmpIDr=");
buffer.append(getEmpID());
buffer.append(",");
buffer.append("EmpSalary=");
buffer.append                                    (getEmp-
Salary().setScale(2).toPlainString());
```

Обратите внимание на вызов метода super.printAudit(), в котором суперкласс (Person) запрашивается выполнить его метод printAudit(), а затем добавляется к нему выполнение метода printAudit() для Employee.

Вызов super.printAudit() не обязательно должен быть выполнен вначале, естественно вывести первоначально общие атрибуты. На самом деле, вообще не нужно вызывать super.printAudit(). Если вообще не обращаться к методу суперклассу, то можно либо вывести все или некоторые атрибуты из Person самостоятельно (в методе Employee.printAudit ()), либо не выводить их вовсе. Выполнение вызова super.printAudit() в данном случае рационально.

Члены класса

Переменные и методы, определенные в классах `Person` и `Employee`, являются переменными и методами *экземпляров класса*. Чтобы получить доступ к ним, необходимо иметь ссылку на экземпляр класса. Каждый экземпляр класса (объект) имеет переменные и методы, и для каждого из объектов конкретное поведение будет отличаться, поскольку оно основано на состоянии экземпляра класса, т.е. на значении атрибутов объекта.

Классы, как таковые, также могут иметь переменные и методы, которые называются *членами класса*. Члены класса объявляются с помощью ключевого слова `static`, о чем ранее уже упоминалось. Еще раз подчеркнем различия между членами класса и членами экземпляров:

- каждый экземпляр класса - объект - совместно использует общую копию переменной класса;
- методы класса вызываются в самом классе без создания экземпляра класса;
- методы экземпляра *имеют* доступ к переменным класса, а методы класса *не имеют* доступа к переменным экземпляра;
- методы класса могут обращаться только к переменным класса.

Определение переменных и методов класса

Использование переменных и методов класса полезно в ряде случаев, а именно:

- для объявления констант, чтобы любой экземпляр класса получал к ним доступ;
- для отслеживания "счетчиков" созданных экземпляров класса;
- для класса с методами утилитами, когда экземпляр класса не понадобится (например, ранее применяемый метод `print()` класса `Person`).

Переменные класса

Для объявлении переменной класса используется ключевое слово `static`:

<модификатор доступа> `static` *<тип данных> <имя переменной>* `[=` *<начальное значение>*`];`

(Здесь квадратные скобки указывают на необязательность того, что в них заключено и не являются частью синтаксиса объявления.)

JVM создает область памяти для хранения всех переменных экземпляра класса для каждого экземпляра этого класса при начальной за-

грузке, а для каждой переменной класса создается единственная копия, независимо от количества созданных экземпляров. Все экземпляры класса могут получать доступ к одной и той же копии переменной.

Например, в классе `Person` объявлен атрибут `Gender` как переменная типа `String`, но не определено для его значений никаких ограничений. Для придания определенности значений атрибута `gender` можно ввести в класс `Person` следующий фрагмент кода, определяющий константы:

```
public static final String GENDER_MALE = "MALE";
public static final String GENDER_FEMALE= "FEMALE";
```

Это очевидно потребует использования этих констант в методах, связанных с присвоением значений атрибута `gender`, в частности, при создании экземпляра класса:

```
Person p = new Person("Joe Q Author", 42, 173, 82, "Brown", GENDER_MALE);
```

В Java используются следующие правила объявления констант:

- имена записываются прописными буквами;
- имена состоят из нескольких слов, разделенных символами подчеркивания (_);
- используется ключевое слово `final` (значения окончательны, и не могут быть изменены);
- объявляются с модификатором доступа `public` (доступны для других классов, которым необходимо ссылаться на их значения по имени).

Если в классе `Person` для использования константы `MALE` при вызове конструктора `Person`, достаточно просто указать ее имя, то для использования константы вне класса нужно использовать имя класса, в котором она объявлена: `String genderValue = Person.GENDER_MALE;`

Методы класса

Для объявлении метода класса используется ключевое слово `static`:

<модификатор доступа> static <тип возвращаемого значения> <имя метода>([*<список аргументов>*])

(Здесь квадратные скобки указывают на необязательность того, что в них заключено и не являются частью синтаксиса объявления.)

Также методы класса иногда еще называют *статическими методами*.

JVM создает область памяти для хранения всех методов экземпляра класса для каждого экземпляра этого класса при начальной за-

грузке, а для каждого метода класса создается единственная копия независимо от количества созданных экземпляров класса. Все экземпляры класса могут получать доступ к одной и той же копии метода.

Использование методов класса

В классе `Person` метод `print` объявлен как статический, и по этой причине является методом класса. В этой связи для использования данного метода отсутствует необходимость создавать экземпляр класса `Person` при использовании данного метода. Если в классе `Person` для вызова метода `print` достаточно просто указать его имя, то для использования метода вне класса нужно использовать имя класса, в котором метод объявлен: `Person.print("");`

Сравнение объектов

Язык Java предоставляет два способа сравнения объектов:

- с использованием оператора "==";
- с использованием метода `equals()`.

Сравнение объектов с помощью оператора "=="

Оператор "==" сравнивает объекты на равенство, так что выражение `a == b` возвращает значение истина `(true)`, если `a` и `b` имеют одно и то же значение. Для объектов это означает, что оба ссылаются на один и тот же экземпляр объекта. Для примитивных типов это означает, что значения идентичны. Рассмотрим следующий пример использования оператора == для сравнения:

```
int int1 = 1;
int int2 = 1;
Person.print("1. int1==int2? "+(int1 == int2));// true
Integer integer1 = Integer.valueOf(int1);
Integer integer2 = Integer.valueOf(int2);
Person.print("2. Integer1==Integer2? "+(integer1 == integer2)); // true
integer1 = new Integer(int1);
integer2 = new Integer(int2);
Person.print("3. Integer1 == Integer2?  " + (integer1 == integer2)); // false
Employee employee1 = new Employee();
Employee employee2 = new Employee();
Person.print("4. Employee1 == Employee2?  " + (employee1 == employee2)); // false
```

В результате его выполнения создается следующий вывод:

```
1. int1 == int2?  true
2. Integer1 == Integer2? true
3. Integer1 == Integer2?  false
```

```
4. Employee1 == Employee2?  false
```

В первом случае значения примитивов одни и те же и результат сравнения `true`. Во втором случае объекты типа `Integer` ссылаются на один и тот же экземпляр, так что == вновь возвращает значение `true`. В третьем случае, хотя объекты `Integer` содержат одно и то же значение, == возвращает результат `false`, потому что `integer1` и `integer2` относятся к разным объектам. Аналогично предыдущему объясняется, почему `employee1 == employee2` возвращает `false`.

Сравнение объектов с помощью метода `equals()`

`equals()` - это метод, который каждый объект языка Java наследует от метода экземпляра `java.lang.Object`. Вызывается метод `equals()` точно так же, как любой другой метод - `a.equals(b);` – оператор вызывает метод `equals()` объекта `a`, передавая ему ссылку на объект `b`. При использовании данного метода в Java-программе по умолчанию при проверке на эквивалентность выполняется с помощью синтаксиса ==. Однако поскольку `equals()` - это метод, его можно переопределить. Рассмотрим следующий пример для сравнения двух объектов с помощью метода `equals()`.

```
Integer integer1 = Integer.valueOf(1);
Integer integer2 = Integer.valueOf(1);
Person.print("1: integer1 == integer2?  " + (integer1
== integer2)); // true
Person.print("2: integer1.equals(integer2)? " +
intger1.equals(integer2)); // true
integer1 = new Integer(integer1);
integer2 = new Integer(integer2);
Person.print("3: integer1 == integer2? A: " + (integer1
== intger2)); // false
Person.print("4: integer1.equals(integer2)?  " +
intger1.equals(integer2)); // true
Employee employee1 = new Employee();
Employee employee2 = new Employee();
Person.print("5: employee1 == employee2?  " + (employ-
ee1 == eployee2)); // false
Person.print("6:employee1.equals(employee2)?"+  employ-
ee1.equals(employee2)?); // false
```

В результате выполнения фрагмента создается следующий вывод:

```
1: integer1 == integer2?  true
2: integer1.equals(integer2)? true
3: integer1 == integer2? A: false
4: integer1.equals(integer2)?  true
```

```
5: employee1 == employee2?  false
6: employee1.equals(employee2)? false
```

Примечание о сравнении значений Integer

В первых строках метод equals() по отношению к Integer возвращает true и операция "==" возвращает true - equals() и должен работать как "==". При этом обратите внимание на то, что происходит в строках вывода 3 и 4, когда создаются отдельные объекты, содержащие одно и то же значение 1: == возвращает false, потому что integer1 и integer2 относятся к разным объектам, но equals() возвращает true.

Так происходит, поскольку разработчики JDK решили, что для Integer значение equals() должно отличаться от стандартного (когда сравниваются ссылки на объект, чтобы определить, относятся ли они к одному и тому же объекту), возвращая значение true, когда внутренние значения int совпадают. Если взглянуть внутрь метода equals класса Integer, то наблюдаем следующий метод:

```
public boolean equals(Object obj) {
    if (obj instanceof Integer) {
        return value == ((Integer)obj).intValue();
    }
    return false;
}
```

Очевидно, что здесь не используется сравнение ссылок, а сравниваются int значения. Это нужно просто иметь в виду.

Для строк 5 и 6 метод equals() класса Employee не переопределен, поэтому поведение по умолчанию (как использование ==) возвращает false, учитывая, что employee1 и employee2 действительно относятся к разным объектам.

Как правило, разработчкики определяют метод equals() в соответствии с создаваемым приложением.

Переопределение equals()

Переопределяя поведение метода Object.equals(), можно определять, что именно equals() будет означать для данных классов в приложении. Например, в приложении присутствует класс Dog с полями "кличка, возраст, порода, награды", то в зависимости от логики работы приложения, метод equals() может проверять только возраст, породу и кличку или какую либо комбинацию значений этих полей. В более сложном приложении, имеющем дело с классами, содер-

жащими большое количество полей, часть из которых не примитивы также возникает потребность реализовать адекватный принцип сравнения.

Для переопределения `equals()` можно просто ввести код, определяющий новое содержание метода, но удобно для этого использовать IDE Eclipse. Рассмотрим это на примере. Убедитесь, что в окне редактирования Eclipse IDE исходный текст `Employee` находится в фокусе, и после нажатия правой кнопки в контекстном меню выберите **Source > Override/Implement Methods**. Появится диалоговое окно, представленное на Рис.56.

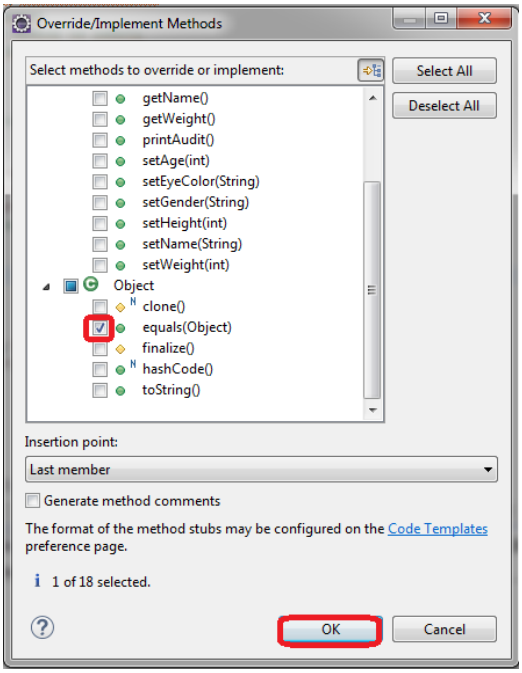


Рис. 56. Диалоговое окно Override/Implement Methods

Мы уже использовали аналогичный диалог, но в данном случае требуется реализовать метод суперкласса `Object.equals()`. Найдите в списке объект `Object`, отметьте метод `equals(Object)` и нажмите кнопку **OK**. Eclipse сгенерирует следующий код и разместит его в исходном файле:

```
@Override
public boolean equals(Object obj) {
	// TODO Auto-generated method stub
	return super.equals(obj);
}
```

Здесь в соответствии с командой `return` по-прежнему происходит обращение к методу `equals(obj)` суперкласса `Person`. В дальнейшем в случае необходимости можно закодировать в составе метода

equals класса Employee соответствующую процедуру формирования возвращаемого значения типа boolean.

Однако в данном случае логично, что два объекта Employee будут эквивалентны между собой, если состояние этих объектов одинаково. То есть, они эквивалентны, если их атрибуты - фамилия, имя, возраст, принадлежащие суперклассу Person, совпадают, и поэтому следует предусмотреть соответствующую модификацию поведения класса Person, затрагивающую метод equals, которая рассматривается в следующем разделе.

Автоматическое создание метода equals()

Eclipse позволяет генерировать метод equals() на основе переменных экземпляра (атрибутов), определенных для класса. Поскольку класс Employee является подклассом класса Person, то можно сгенерировать метод equals() для Person. В окне Project Explorer Eclipse щелкните правой кнопкой мыши Person и из контекстного меню выберите **Generate hashCode() and equals()**, после чего откроется диалоговое окно, показанное на Рис. 57.

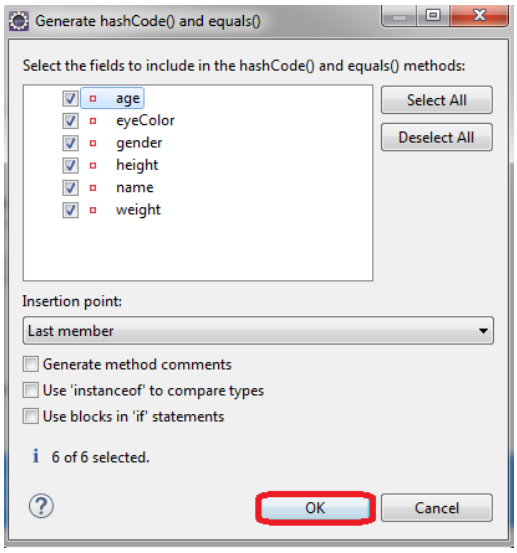


Рис. 57. Диалоговое окно Generate hashCode() and equals() methods

Выберите все атрибуты и нажмите кнопку **OK**. Eclipse сгенерирует метод equals(), который имеет следующий вид:

```
@Override
public boolean equals(Object obj) {
     if (this == obj)
          return true;
     if (obj == null)
          return false;
     if (getClass() != obj.getClass())
```

```
            return false;
        Person other = (Person) obj;
        if (age != other.age)
            return false;
        if (eyeColor == null) {
            if (other.eyeColor != null)
                return false;
        } else if (!eyeColor.equals(other.eyeColor))
            return false;
        if (gender == null) {
            if (other.gender != null)
                return false;
        } else if (!gender.equals(other.gender))
            return false;
        if (height != other.height)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        if (weight != other.weight)
            return false;
        return true;
    }
```

Можно удалить сгенерированный метод `hashCode()` за ненадобностью в данный момент.

Метод `equals()`, сгенерированный Eclipse, кажется сложным, но делает он довольно простую вещь: если переданный объект – тот же, что и текущий, то `equals()` возвратит значение `true`. Если переданный объект – это `null`, он возвратит значение `false`.

Затем метод проверяет на совпадение объекты `Class` (что означает, что переданный объект должен быть объектом класса `Person`). Если они совпадают, то проверяется каждое значение атрибута переданного объекта на совпадение со значением экземпляра `Person`. Если атрибуты имеют значение `null` (то есть отсутствуют), `equals()` проверит все атрибуты, и если они совпадают, объекты будут считаться равными. Такое поведение подойдет не для каждого приложения, но для большинства задач это работает.

Упражнение 7: Создание метода `equals()` для объекта `Employee`

Создайте метод `equals()` для класса `Employee`, и следуя пунктам раздела Автоматическое создание метода, сгенерируйте метод

equals() для класса Person. После генерации метода equals(), добавьте метод main() в класс Employee следующего вида:

```
    public static void main(String[] args) {
        Employee employee1 = new Employee();
        employee1.setName("James Bond");
        Employee employee2 = new Employee();
        employee2.setName("James Bond");
        Person.print("1: employee1 == employee2? " + (em-
ployee1 == employee2));
        Person.print("2: employee1.equals(employee2)? "
                + employee1.equals(employee2));
        }
```

Выполните класс Employee. После выполнения кода класса Employee, вы должны получить следующий результат:

```
1: employee1 == employee2? false
2: employee1.equals(employee2)? true
```

В данном случае одного только совпадения значений атрибутов Name достаточно, чтобы equals() рассматривал объекты эквивалентными.

Добавьте к объектам другие атрибуты, используя соответствующие методы Set, и проанализируйте результат выполнения класса Employee.

Упражнение 8: Переопределение метода toString()

Метод printAudit(), который был использован для формирования и вывода состояния объекта в формате String типичен для большинства классов, и разработчики Java встроили такую операцию в класс Object, в виде метода с именем toString(). Реализация метода toString() по умолчанию не очень полезна, но она доступна для каждого объекта. Переопределив метод toString(), можно сделать это унаследованное поведение более полезным, и применение IDE Eclipse способствует переопределению.

Для выполнения переопределения в Project Explorer щелкните правой кнопкой мыши на классе Employee и последовательно выберите **Source > Generate toString()...**. Откроется диалоговое окно, представленное на Рис. 58.

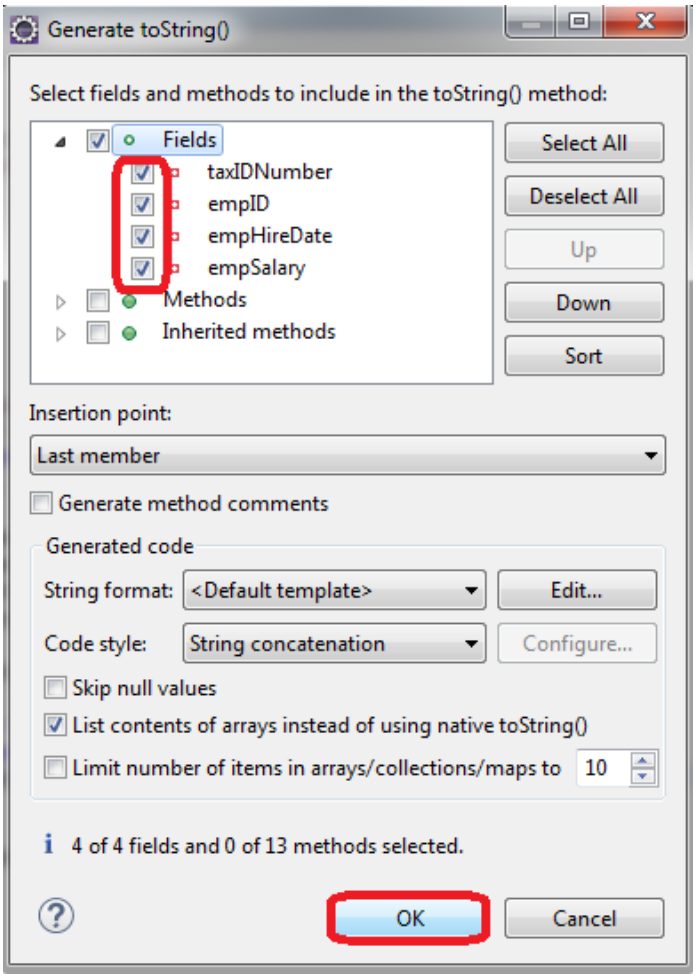


Рис. 58. диалоговое окно Generate toString()

Выберите все атрибуты и нажмите кнопку **OK**. Ниже представлен код, сгенерированный Eclipse для `Employee`:

```
@Override
public String toString() {
	return "Employee [taxIDNumber=" + taxIDNumber + ",
empID=" + empID	+ ", empHireDate=" + empHireDate + ",
empSalary=" + empSalary+ "]";
}
```

Код, сгенерированный Eclipse для метода toString, не содержит обращения к методу `toString()` суперкласса `Person`, что можно исправить, применив приведенную ниже корректировку метода:

```
@Override
public String toString() {
	return super.toString() +"Employee [taxIDNumber="
+	taxIDNumber + ", empID=" + empID	+ ", empHire-
Date=" + empHireDate + ", empSalary="	+ empSalary + "]";
}
```

Теперь, если модифицировать метод `printAudit()` в классе `Employee` с использованием `toString()`, код значительно упрощается:

```
@Override
public void printAudit() {
     StringBuilder buffer = new StringBuilder();
     super.printAudit(buffer);
     buffer.append(toString());
     print(buffer.toString());
}
```

Теперь `toString()` выполняет всю тяжелую работу по форматированию текущего состояния объекта, и вы просто добавляете в `StringBuilder` то, что он возвращает метод `toString()`.

Сделайте соответствующие изменения в классе `Employee` и проверьте работу метода `printAudit()`.

Хорошим стилем программирования считается переопределение в своих классах метода `toString()`, поскольку в процессе отладки обязательно возникнет необходимость в анализе состояния объекта, которое удобно выводить методом `toString()`.

Рассмотрим наследование и полиморфизм применительно к другой предметной области, основанной на геометрических фигурах. Это поможет выполнить ряд заданий в последующем упражнении. В рамках предлагаемой предметной области рассматриваются классы, образующие иерархию, представленную на Рис. 59.

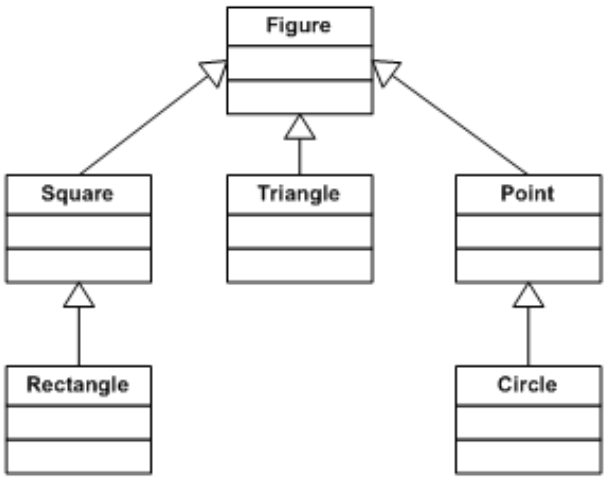


Рис. 59 Диаграмма классов геометрических фигур

На представленной диаграмме классов базовым классом является `Figure`, от которого наследуются `Square` - "квадрат", `Triangle` - "треугольник" и `Point` - "точка". От класса `Square` наследуется `Rectangle` - "прямоугольник". От класса `Point`, свою очередь, насле-

дуется класс `Circle` - "окружность". Отметим, что в соответствии с принципами UML на диаграмме принято изображать стрелки в направлении от подкласса к суперклассу.

Часто понятие суперкласс подталкивает начинающих программистов к неверной логике: суперкласс пытаются наделить сложным поведением, а его подклассы неверно воспринимаются как "упрощенные классы" и наделяются упрощенным по сравнению с суперклассом поведением. На самом деле потомки должны обладать более сложным устройством и поведением по сравнению с суперклассами.

Чем ближе к вершине иерархии расположен класс, тем более общим и универсальным (general) он является, но одновременно - более простым. Базовому классу, который лежит в основе иерархии, всегда присваивают имя, которое характеризует все объекты - экземпляры подклассов, и которое выражает наиболее общую абстракцию, применимую к таким объектам. Поэтому на вершите диаграммы классов расположен класс с именем `Figure`, и все остальные классы также представляют собой геометрические фигуры того или иного типа.

Любая фигура будет иметь поля данных: `x` и `y` - координаты фигуры на экране, а также `color` - цвет фигуры. Значения координат `x` и y должно устанавливаться в конструкторе с соответствующими параметрами. Как мы условились, для каждого класса целесообразно переопределить метод `toString()`. Далее в этом классе могут быть помещены setter/getter для этих полей. Правда методы доступа и установки могут потребоваться в создаваемом приложении не все, и их можно было бы распределить по конкретным классам, исходя из функциональности приложения. Кроме того, в классе `Figure` определяются методы `show()` - показать фигуру на экране, и `hide()` - скрыть ее и `getArea()` - вычислить площадь фигуры. Для образованной переменной `figure` типа `Figure` вызовы `figure.show()` и `figure.hide()` будут соответственно показывать или скрывать объект, на который ссылается эта переменная, а метод `figure.getArea()`- возвращать площадь фигуры. При этом класс `Figure` может обозначить поведение только на уровне абстракций путем объявления соответствующих методов без их реализации, а конкретные классы будут обеспечивать реализацию методов для показа, скрытия и вычисления площади. В этом как раз и состоит основное преимущество объектно-ориентированного программирования по сравнению с процедурным - в возможности написания полиморфного кода.

Класс `Figure` может иметь следующий вид:

```
public class Figure {
```

```
        double x;
        double y;
        String color = "Red";

        Figure(double x,double y) {
             this.x = x;
             this.y=y;
        }
        public void show() {
             System.out.println ("appearing");
        }
        public void hide() {
             System.out.println ("disappearing");
        }
        public double getArea() {
             return 0.0;
        }

        public double getX() {
             return x;
        }
        public void setX(double x) {
             this.x = x;
        }
        public double getY() {
             return y;
        }
        public void setY(double y) {
             this.y = y;
        }
        public String getColor() {
             return color;
        }
        public void setColor(String color) {
             this.color = color;
        }
        @Override
        public String toString() {
             return "Figure [x=" + x + ", y=" + y + ",
    color=" + color + "]";
        }
    }
```

Класс `Point` является наследником `Figure`, поэтому он будет иметь поля данных `x` и `y`, наследуемые от `Figure`. Естественно, в классе `Point` эти поля дополнительно определять не следует и они определяются при вызове конструктора `Figure()`. Класс `Point` может выглядеть следующим образом:

```
class Point extends Figure{
      Point(double x, double y){
            super(x,y);
      }
}
```

В классе `Point` реализован конструктор, который обращается к конструктору класса `Figure` для определения переменных `x` и `y`.

Класс `Circle` наследуется от класса `Point` в соответствии с диаграммой (моделью классов), поэтому в нем также имеются поля `x` и `y`, наследуемые от `Figure` (как центр окружности), но возникает потребность определить дополнительное поля данных, определяющее радиус – `radius`. Кроме того, для окружности возможна операция изменения радиуса, поэтому в ней может появиться новый метод, обеспечивающий это действие – `setRadius` и данный класс в зависимости от значения радиуса может предоставлять вычисляемое ненулевое значение площади с помощью метода переопределенного метода `getArea()`. В состав класса может входить один или больше конструкторов, перелпределение метода `toString()`, и, возможно, некоторые другие методы, необходимые для поддержки функциональности приложения . Например, метод может выглядеть следующим образом:

```
class Circle extends Point{
      private double radius;
      Circle(double x, double y, double radius){
            super(x,y);
            this.radius=radius;
      }
      Circle(Point point, double radius){
            super(point.x,point.y);
            this.radius=radius;
      }
      public double getRadius() {
            return radius;
      }

      public void setRadius(double radius) {
            this.radius = radius;
      }
      @Override
      public double getArea() {
            return Math.PI*radius*radius;
      }
      public double getLine() {
            return 2*Math.PI*radius;
      }
      @Override
```

```
        public String toString() {
            return super.toString()+"Circle [radius=" +
    radius + "]";
        }
    }
```

Как видно из представленного кода класса, в его составе, кроме поля radius, присутствуют два конструктора с различными сигнатурами, переопределен метод `toString()`, в котором закодировано обращение к методу `toString()` суперкласса и методы `getArea()` и `getLine()`, которые возвращают площадь и длину окружности соответственно. Обратите внимание, что при обращении к методу `toString()` суперкласса, обращение выполняется к этому методу класса `Figure`, в котором этот метод реализован.

Для проверки работоспособности кода создадим класс `Test`, в котором создаются объекты с использованием различных конструкторов и выводится сведения об этих объектах. Класс может иметь следующий вид:

```
public class Test {
    public static void main(String[] args) {
        Circle circle = new Circle(1,1,10);
        circle.show();
        System.out.println(circle);
        System.out.println("Имеет
площадь="+circle.getArea());
        System.out.println("Имеет
длину="+circle.getLine());
        circle.hide();
        Point point = new Point(2,2);
        Circle circle1 = new Circle(point,20);
        circle1.show();
        System.out.println(circle1);
        System.out.println("Имеет площадь="
+circle1.getArea());
        System.out.println("Имеет длину="
+circle1.getLine());
        circle1.hide();
    }
}
```

Выполнение класса Test произведет ниже следующий вывод, анализ которого подтверждает предположение о правильности разработанного кода:

```
appearing
Figure [x=1.0, y=1.0, color=Red]Circle [radius=10.0]
Имеет площадь=314.1592653589793
Имеет длину=62.83185307179586
```

```
disappearing
appearing
Figure [x=2.0, y=2.0, color=Red]Circle [radius=20.0]
Имеет площадь=1256.6370614359173
Имеет длину=125.66370614359172
disappearing
```

Класс `Circle` в диаграмме классов можно считать описанием "усложненной класса" `Point`. При этом важно, что любой объект такого типа рассматривается как "точка, которую усложнили". Можно, например, считать, что класс `Circle` - это такая "расширенная точка".

Аналогично, класс `Square` наследует поля `x` и `y`, но в нем добавляется поле, соответствующее стороне квадрата – `side`. Естественно, могут добавляться setter/getter для этого поля, а также переопределяется метод `getArea()`, предназначенный для вычисления площади квадрата.

В классе `Triangle` в качестве новых, не унаследованных полей данных, могут выступать координаты вершин треугольника; либо координаты одной из вершин, длины прилегающих к ней сторон и угол между ними, и так далее. Также могут быть добавлены setter/getter для введенных полей и другие методы, востребованные функциональностью приложения.

Каждый объект подкласса при любых значениях полей должен рассматриваться как экземпляр суперкласса, и с тем же поведением на уровне обобщенных действий, но с некоторыми изменениями на уровне реализации этих действий. В концепции наследования основное внимание уделяется поведению объектов. Объекты с разным поведением имеют другой тип. При этом значения полей данных характеризуют состояние объекта, но не его тип.

По своему поведению любой объект типа `Circle` вполне может рассматриваться как экземпляр типа `Point` и даже вести себя в точности как точка. Но не наоборот - объекты типа `Point` не обладает поведением `Circle`, поскольку для окружности можно изменить значение радиуса `radius` (вызвать метод `setRadius`), а для точки такая операция не имеет смысла или запрещена.

Представленная на Рис. 59 диаграмма классов является не единственным возможным вариантом, и возможно, не самым эффективным с точки зрения использования наследования и полиморфизма. В качестве альтернативы можно рассмотреть диаграмму классов, представленную на Рис. 60.

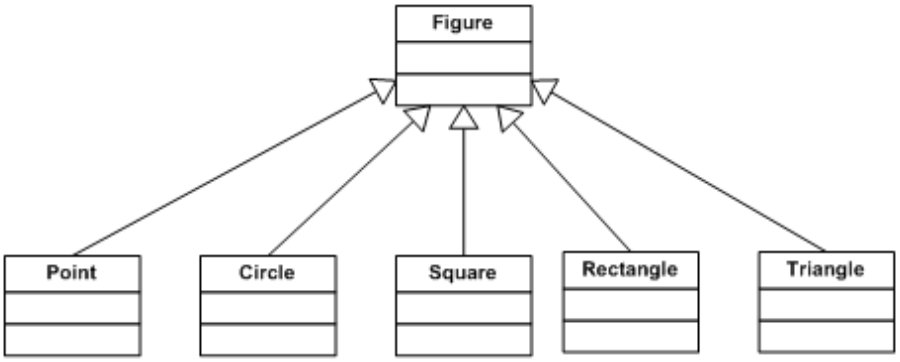


Рис. 60 Альтернативная диаграмма графических классов

По всей вероятности последний вариант диаграммы (Рис. 60), когда все классы наследуются непосредственно от `Figure`, во многих случаях будет предпочтительнее. Более того, никакого выигрыша при написании программного кода увеличение числа поколений наследования не дает: код, написанный для класса `Point`, вряд ли будет использоваться для объектов классов `Circle`. Исходить требуется из того, что наследование само по себе не является самоцелью – это инструмент для написания более экономного полиморфного кода. Более того, увеличение числа поколений приводит к снижению надежности кода.

Реорганизация (реинжиниринг) классов обычно выполняется на завершающих этапах разработки приложений и, в основном, предназначена для увеличения эффективности поддержки приложений в процессе функционирования, концентрируя изменяющийся функционал в минимальном количестве классов. На начальных этапах разработки этим не следует злоупотреблять.

В последующем упражнении 9 рассматриваются практические аспекты работы с графическими классами разной степени отношений.

Упражнение 9. Рассмотрим примеры разработки классов из предметной области, связанной с графическими фигурами.

Первый вариант - Класс `Circle`:

Класс с именем `Circle` спроектирован, как показано на диаграмме классов (Рис. 61).

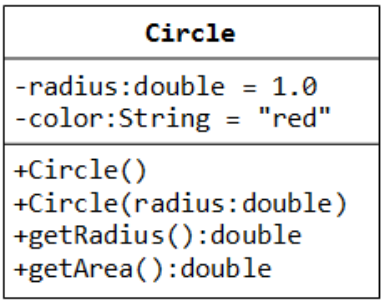


Рис. 61. Диаграмма класса Circle

Класс содержит:

- Две `private` переменных экземпляра: `radius` (тип `double`) и `color` (тип `String`), со значениями по умолчанию `1.0` и `"red"`, соответственно;
- Два перегружаемых конструктора;
- Два `public` метода: `getRadius()` и `getArea()`.

Исходный код для `Circle` следующий:

```
public class Circle {
   // private переменные реализации, не доступны за пределами класса
   private double radius;
   private String color;
   // Первый конструктор, который присваивает для radius color значения по умолчанию
   public Circle() {
      radius = 1.0;
      color = "red";
   }
   // второй конструктор устанавливает radius с заданным значением, но color по умолчанию
   public Circle(double r) {
      radius = r;
      color = "red";
   }
   // public метод для извлечения radius
   public double getRadius() {
     return radius;
   }
  // public метод для вычисления площади круга
   public double getArea() {
      return radius*radius*Math.PI;
   }
}
```

Можно ли выполнить класс `Circle`? Почему нет? Очевидно, что класс `Circle` не содержит метода `main()` . Следовательно, он не может выполниться непосредственно. Этот класс `Circle` является строительным блоком, что означает, что он должен быть использован другой программой (классом).

Напишите тестовую программу в отдельном файле с именем `TestCircle`, которая использует класс `Circle`, как показано ниже:

```
public class TestCircle {// сохранить как "TestCircle.java"
public static void main(String[] args) {
//Объявление и размещение экземпляра класса Circle с именем c1
```

```
//  Со значениями radius и color по умолчанию
      Circle c1 = new Circle();
// Оператор точка (.)используется для вызова методов экземпляра  c1.
      System.out.println("Окружность имеет радиус =" + c1.getRadius() + " и площадь = " + c1.getArea());
// Объявление и размещение экземпляра класса Circle с именем c2
//  с заданным значением radius и color по умолчанию
      Circle c2 = new Circle(2.0);
// Используется оператор точка для вызова методов экземпляра c2.
      System.out.println("Окружность имеет радиус ="+c2.getRadius() + " и площадь = " + c2.getArea());
   }
}
```

Выполните программу `TestCircle` и проанализируйте результаты.

Модифицируйте класс `Circle` следующим образом:

1. Добавьте конструктор. Добавить третий конструктор в класс `Circle` для включения аргументов для задания значений `radius` и `color`.

```
// Конструктор для создания нового экземпляра Circle с заданными значениями radius и color
public Circle (double r, String c) {......}
```

Модифицируйте тестовую программу `TestCircle` для создания экземпляра класса `Circle`, использующего данный конструктор.

2. Добавьте `Getter`. Добавить `getter` для переменной `color` для извлечения значения переменной `color` экземпляра класса `Circle`.

```
// Getter для извлечения значения переменной color
public String getColor() {......}
```

Модифицировать тестовую программу для проверки этого метода.

3. public vs. private. Может ли быть получен в классе `TestCircle` доступ к переменной экземпляра `radius` непосредственно (например., `System.out.println(c1.radius));` или для присвоения нового значения `radius` (например, `c1.radius=10.0`)? Попробуйте и объяснить сообщение об ошибке.

4. Добавьте `Setter`. Есть ли необходимость изменять значения переменных `radius` и `color` экземпляра класса Circle после создания? Если да, добавьте два `public` метода `setter` для изменения `radius` и `color` экземпляра класса `Circle` следующим образом:

```
// Setter для экземпляра переменной radius
public void setRadius(double r) {
   radius = r;
}
// Setter для экземпляра переменной color
public void setColor(String c) { ..... }
```

Измените класс `TestCircle` для проверки методов, например,

```
Circle c3 = new Circle();     // создание экземпляра класса Circle
c3.setRadius(5.0);            // изменить radius
c3.setColor(...);             // изменить color
```

5. Ключевое слово "`this`". Вместо того, чтобы использовать имена переменных, такие как `r` (для `radius`) и `c` (для `color`) в аргументах методов обычно используют имена `radius` (для `radius`) и `color` (для `color`) и ключевое слово "`this`" для разрешения конфликта между переменными экземпляра и аргументами метода, например:

```
// Переменная экземпляра
private double radius;
// Setter для radius
public void setRadius(double radius) {
   this.radius = radius;    // "this.radius" ссылается на переменную экземпляра
                            //  "radius" ссылается на аргумент метода
}
```

Модифицируйте все конструкторы и `setter` в классе `Circle` с использованием ключевого слова "`this`".

6. Метод `toString()`. Каждый хорошо спроектированный класс Java обычно содержит `public` метод `toString()`, который возвращает полезное описание экземпляра класса (в операторе `return` типа `String`). Метод `toString()` может быть вызван явно (через использование оператора `instanceName.toString()`) как любой другой метод; или неявно через `System.out.println(<параметр>)`. Если экземпляр передается методу `System.out.println(anInstance)`, то метод `toString()` экземпляра вызывается неявно. Например, введите следующий метод `toString()` в класс `Circle`:

```
public String toString() {
   return "Circle: radius= " + radius + " color= " + color;
}
```

Попытайтесь вызвать метод `toString()` явно, также как любой другой `метод`:

```
Circle c1 = new Circle(5.0);
System.out.println(c1.toString());   // явный вызов
```

Метод `toString()` вызывается неявно, когда экземпляр передается в метод `println()`, например:

```
Circle c2 = new Circle(1.2);
System.out.println(c2.toString());  // явный вызов
System.out.println(c2);             // println() вызывает toString() неявно, точно также как ранее
System.out.println("Оператор '+' вызывает toString() тоже: " + c2);  // '+' тоже вызывет toString()
```

Второй вариант - Класс `MyPoint` и `MyCircle`:

Класс с именем `MyPoint`, который моделирует 2D точки с координатами `x` и `y`, спроектирован как показано на Рис. 62.

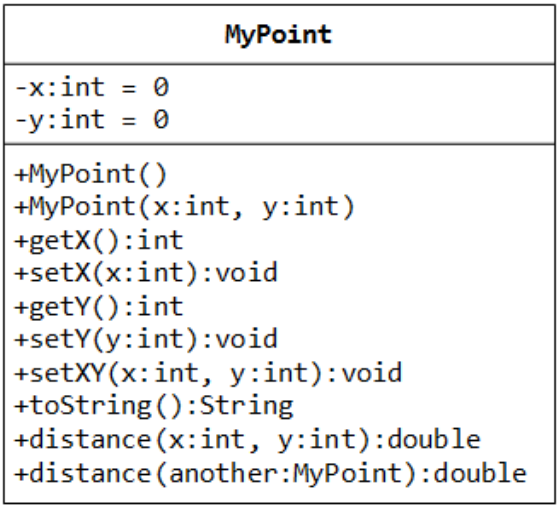


Рис. 62. Диаграмма класса `MyPoint`

Класс содержит:

- Две переменных экземпляра `x (int)` и `y (int)`.
- Безаргументный конструктор, который создает точку с координатами в (0, 0).
- Конструктор, который создает точку с заданными координатами `x` и `y`.
- Getter и setter для переменных экземпляра `x` и `y`.
- Метод `setXY()` для установки значений `x` и `y`.
- Метод `toString()`, который возвращает `String` значение в виде "(x, y)".

- Метод с именем `distance(int x, int y)`, который возвращает расстояние от `this` точки до другой точки с заданными координатами `(x, y)`.
- Перегружаемый метод с сигнатурой `distance(MyPoint another)`, который возвращает расстояние от `this` точки до заданного экземпляра класса `MyPoint` с именем `another`.

Требуется:

1. Написать код класса `MyPoint`. Также написать тестовую программу (с именем `TestMyPoint`) для проверки работоспособности всех методов, определенных в классе:

```
// Перегружаемый метод distance((int x, int y) имеет следующий вид:
public double distance(int x, int y) {
// эта версия принимает два значения int в качестве аргументов
   int xDiff = this.x - x;
   int yDiff = ......
   return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
}

public double distance(MyPoint another) {
// Эта версия принимает экземпляр MyPoint в качестве аргумента
   int xDiff = this.x - another.x;
   .......
}
// Тестовая программа
MyPoint p1 = new MyPoint(3, 0);
MyPoint p2 = new MyPoint(0, 4);
......
// Проверка метода distance()
System.out.println(p1.distance(p2));
//какой вариант?
System.out.println(p1.distance(5, 6));
// какой вариант?
.....
```

2. В тестовой программе написать код, который размещает 10 точек в массиве `MyPoint`, и инициализирует их с координатами: (1, 1), (2, 2), ... (10, 10).

Совет: Необходимо инициировать массив, а также каждый из десяти экземпляров `MyPoint` .

```
MyPoint[] points = new MyPoint[10]; // Объявляется и размещается массив
for (......) {
```

```
        points[i] = new MyPoint(...);     // Размещаются все
экземпляры MyPoint
    }
```

Класс `MyCircle`, моделирует окружность с центром `center` с координатами `(x, y)` и радиусом `radius`, спроектирован как показано на Рис. 63.

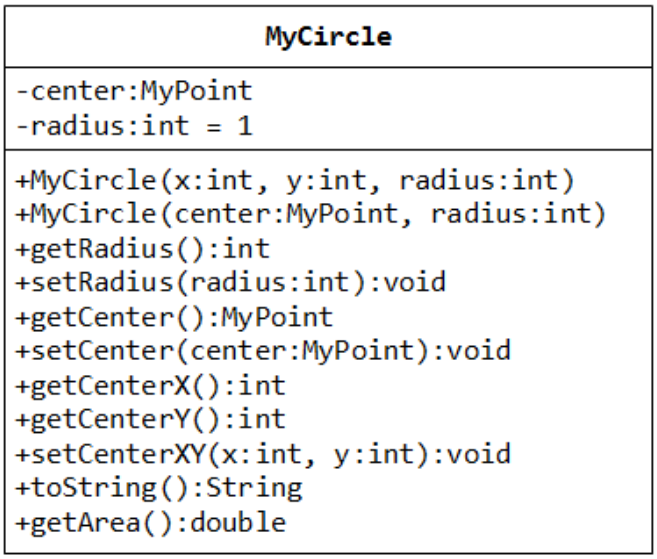


Рис. 63. Диаграмма класса `MyCircle`

Класс `MyCircle` использует экземпляр класса `MyPoint` для определения переменной `center`.

Класс с именем `MyPoint` определен выше.

Класс `MyCircle` содержит:

- Две `private` переменных экземпляра: `center` (экземпляр `MyPoint`) и `radius (int)`.
- Конструктор, который создает экземпляр `MyCircle` с заданными координатами центра `(x, y)` и значением `radius`.
- Перегружаемый конструктор, который создает экземпляр `MyCircle` с заданным экземпляром `MyPoint` в качестве значения `center` и значением `radius`.
- Различные `getter` и `setter`.
- Метод `toString()`, который возвращает `String` описание экземпляра в формате "Circle @ (x, y) radius=r".
- Метод `getArea()`, который возвращает значение площади круга типа `double`.

Написать класс MyCircle дополнительно к классу MyPoint. Также написать программу с именем TestMyCircle для проверки всех методов, определенных в классе MyCircle.

Третий вариант: Классы Author и Book

Класс Author спроектирован, как показано на Рис. 64.

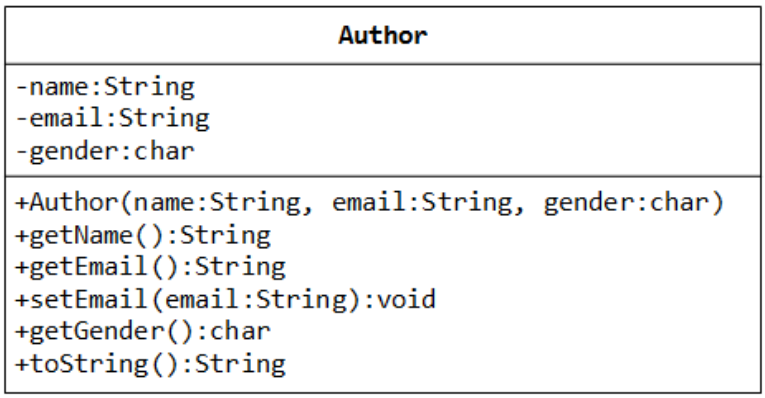


Рис. 64. Диаграмма класса Author

Класс Author содержит:

- Три private переменных реализации: name типа String, email типа String) и gender типа char (значения либо 'm' либо 'f');
- Один конструктор класса для инициализации name, email и gender с заданными значениями;

```
public Author (String name, String email, char gender) {......}
```

(Отсутствует конструктор по умолчанию для Author, поскольку нет экземпляра класса по умолчанию для name, email и gender.)

- public методы getters/setters: getName(), getEmail(), setEmail()и getGender();

(Отсутствуют setter для name и gender, поскольку атрибуты не могут быть изменены.)

- Метод toString() , который возвращает "имя-автора (gender) at email", например, "Vincent Willem van Gogh (m) at Vincent @somewhere.com".

Написать класс Author . Также написать тестовую программу с именем TestAuthor для тестирования конструктора и методов public . Попытайтесь изменить атрибут email автора, например,

```
Author anAuthor = new Author("Vincent Willem van Gogh",
"Vincent@somewhere.com", 'm');
System.out.println(anAuthor);   // вызов toString()
anAuthor.setEmail("paul@anywhere.com")
```

```
System.out.println(anAuthor);
```

Класс с именем `Book` спроектирован как показано на Рис. 65.

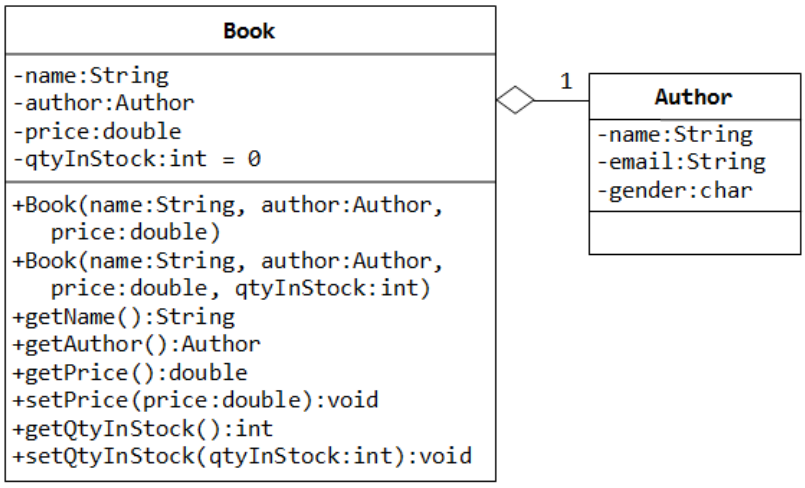


Рис. 65. Диаграмма класса `Book` и `Author`

На представленной диаграмме символ  обозначает, что объект класса `Author` входит в состав экземпляра класса `Book` в единственном экземпляре, т.е. у экземпляра класса `Book` имеется единственный автор.

Класс содержит:

- Четыре `private` переменных экземпляра класса: `name (String)`, `author` (класса `Author`, который только что создан, предположим, что каждая книга имеет одного, и только одного автора), `price (double)`, и `qtyInStock (int)`;
- Два конструктора:

```
public Book (String name, Author author,
double price) {...}
public Book (String name, Author author,
double price, int qtyInStock) {...}
```

- `public` методы `getName()`, `getAuthor()`, `getPrice()`, `setPrice()`, `getQtyInStock()`, `setQtyInStock()`.
- метод `toString()`, который возвращает `"'book-name' автора author-name (gender) at email"`.

(Имейте в виду, что метод `toString()` класса `Author` возвращает `"author-name (gender) at email".`)

Закодируйте класс `Book`, который использует класс `Author`, написанный ранее. Также напишите класс с именем `TestBook` для проверки работы конструктора и методов `public` в классе `Book`.

Имейте в виду, что необходимо создать экземпляр Author перед созданием экземпляра Book. Например, так:

```
Author anAuthor = new Author(......);
Book aBook = new Book("Java for dummy", anAuthor, 19.95, 1000);
// Ниже используется анонимный экземпляр класса Author
Book anotherBook = new Book("Thinking in Java", new Author(......), 29.95, 666);
```

Обратите внимание, что оба класса Book и Author содержат переменные с именем name, которые должны различаться ссылкой на экземпляр класса. Для экземпляра Book, например, aBook, точечная нотация aBook.name ссылается на поле name в классе Book; в то время как для экземпляра Author, например, anAuthor, точечная нотация anAuthor.name ссылается на поле name в классе Author. Нет необходимости (и не рекомендуется) называть переменные bookName и authorName.

В приложении выполнить:

1. Вывод на экран name и email автора из экземпляра класса Book. (Совет: aBook.getAuthor().getName(), aBook.getAuthor().getEmail()).

2. Включить новые методы с именами getAuthorName(), getAuthorEmail(), getAuthorGender() в класс Book , которые возвращают name, email и gender автора книги. Например,

```
public String getAuthorName() { ..... }
```

Исключения- Exception

Не существует сложной программы, которая выполняется правильно во всех случаях, и язык Java поддерживает механизм обработки ситуаций, в которых код может работать не совсем так, как предусматривает алгоритм вычислений.

Exception (Исключение) - это событие, которое происходит во время выполнения программы, нарушая нормальный ход выполнения ее команд. Обработка исключений - важная часть Java-программирования, которая позволяет помещать потенциально опасный код в блок try (что означает: "попробуем так и посмотрим, вызовет ли это исключение") и добавлять средства для обнаружения исключений (ошибок) различного типа.

Прежде чем приступить к обработке исключений, взгляните на код, приведенный ниже:

```
Scanner in = new Scanner(System.in);
int input;
```

```
System.out.print("Введите целое число: ");
input = Integer.parseInt(in.next());//для ввода int
```

Код выше предполагает, что действия пользователя будут корректными и, в ответ на приглашение ввести целое число, с клавиатуры будут введены цифровые символы, которые могут быть преобразованы в целое число. Однако нет возможности контролировать действия пользователя и он, по невнимательности, может ввести набор символов, который не может быть преобразован в целое число. В этом случае программа завершится аварийно и пользователю будет выведено сообщение об ошибке и т.н. стек вызовов, содержащий список методов, вызванных данной программой, начиная с первого вызванного метода и заканчивая с текущим методом. В нашем случае сообщение может иметь следующий вид:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "4r"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at ru.ifmo.practicejava.Except.main(Except.java:12)
```

В сообщении говорится о том, что произошло исключение (NumberFormatException), связанное с преобразованием в числовое значение введенного пользователем набора символов с клавиатуры. Конечно же, уж не пользователю следует разбираться в этом, достаточно информативном для разработчика, сообщении об ошибке, и разработчику программы следует не допускать таких ситуаций.

В процессе выполнения программы на языке Java, как правило, имеется связь с внешней средой (пользователями, ресурсами файловых систем, связностью с сетевыми ресурсами и т.п.), которая не является предсказуемой и, в этой связи, несет потенциальную угрозу успешному выполнению программы.

Java предоставляет возможность использования блоков `try-catch-finally`, чтобы пытаться выявлять и программно обрабатывать ошибки времени исполнения.

Блоки try, catch и finally

В ниже следующем классе реализован цикл ввода данных от пользователя, до тех пор, пока не будет введено правильное значение. Здесь используются стандартные блоков для обработки исключений `try`, `catch` и `finally`.

```
package ru.ifmo.practicejava;
import java.util.Scanner;
```

```
public class Except {
public static void main(String[] args) {
          // TODO Auto-generated method stub
          Scanner in = new Scanner(System.in);
          int input = 0;
          Boolean er = true;
          while (er) {
               System.out.print("Введите целое: ");
               try {
                    input = Inte-
ger.parseInt(in.next());// для ввода int
                    er=false;
               } catch (NumberFormatException nfe) {
                    System.out.println("Придется повто-
рить - неверный формат данных"+nfe.getMessage());
                    in.nextLine(); // Очищаем поток
                    continue;
               } finally {
                    if (er) System.out.println("Пока еще
значение равно = " + input);
               }
          }
          System.out.println("Вы ввели целое число = "
+ input);
          in.close();
     }
}
```

Блоки `try`, `catch` и `finally` вместе образуют "капкан для ловли исключений". Вначале код, который является потенциально опасным и может вызвать исключение, заключается в состав оператора `try`. Если предполагаемое исключение происходит, управление сразу передается соответствующему блоку `catch`, который обрабатывает исключение. Когда один или оба оператора выполнены, управление передается блоку `finally`, который выполняется всегда, вне зависимости от того, имело место исключение или нет. "Поймав" исключение, можно попытаться аккуратно обойти его или же выйти из программы (или метода).

В представленном выше классе обнаруживается неверный формат введенных данных, выводится соответствующее сообщение пользователю и предлагается повторить ввод данных. После введения правильного формата данных цикл ввода завершается и выводится введенное значение.

Иерархия исключений

Язык Java поддерживает иерархию исключений, состоящую из большого количества типов исключений, сгруппированных в две основные категории:

Контролируемые исключения проверяются компилятором (то есть компилятор проверяет, что ваш код где-то обрабатывает их).

Неконтролируемые исключения (или *исключения времени выполнения*) не проверяются компилятором.

Контролируемые исключения представляют собой ошибки, которые можно и нужно обрабатывать в программе, к этому типу относятся все потомки класса `Exception`. Они проверяются компилятором (то есть компилятор проверяет, что ваш код где-то обрабатывает их, а Eclipse будет этому способствовать).

На Рис. 66 представлена упрощенная иерархия классов исключений.

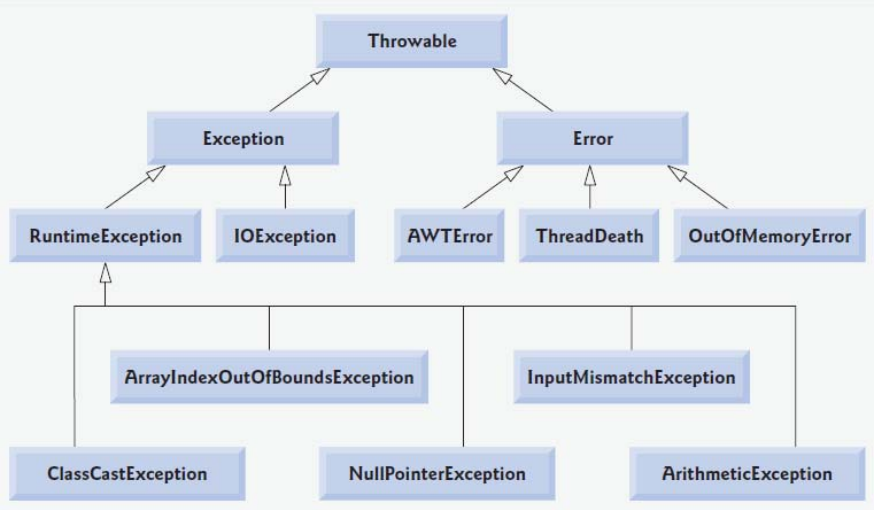


Рис. 66. Иерархия классов исключений.

Исключения делятся на несколько классов, но все они имеют общего предка — класс `java.lang.Throwable`. Только объекты этого класса или его наследники могут вызваны JVM при возникновении какой-нибудь исключительной ситуации, а также только эти объекты могут вызывать исключения во время выполнения программы с помощью ключевого слова `throw`. Потомками этого класса являются подклассы `Exception` и `Error`.

Error - это подкласс, который представляет серьезные проблемы, возникающие во время выполнения приложения. Большинство из этих ошибок сигнализируют о ненормальном ходе выполнения программы, т.е. о наступлении критических проблемах и эти ошибки не рекоменду-

ется отмечать в методах посредством throws-объявления, и поэтому они называются не проверяемые (unchecked).

Exception - это подкласс в иерархии, на который при программировании на Java следует уделять основное внимание. Этот уровень иерархии также разделяется на две ветви: исключения, производные от класса `RuntimeException`, и `IOExeption`. Исключения типа `RuntimeException` возникают вследствие ошибок программирования. Исключения типа `IOExeption` являются следствием непредвиденного стечения обстоятельств, возникающих, как правило, из-за ошибок ввода-вывода, при выполнении вполне корректных программ.

Ниже представлены некоторые, в том числе, и не изображенные на Рис. 66 типы исключений, наследованные от **RuntimeException:**

- **`IndexOutOfBoundsException`** - выбрасывается (thrown), когда указанный индекс некоторого элемента в структуре данных (массив/коллекция) не попадает в диапазон существующих индексов.
- **`NullPointerException`** - выбрасывается, когда ссылка на объект, к которому выполняется обращение не определен и содержит значение `null`.
- **`ClassCastException`** – выбрасывается, когда происходит ошибка приведения типов. Всякий раз при приведении типов выполняется проверка на возможность приведения (проверка осуществляется с помощью оператора **`instanceof`**.
- **`ArithmeticException`** - выбрасывается, когда выполняются недопустимые арифметические операции, например деление на ноль.

Типы исключений, наследованные от класса **Error:**

- **`ThreadDeath`** - вызывается при неожиданной остановке потока посредством метода `Thread.stop()`.
- **`StackOverflowError`** - ошибка переполнение стека. Часто возникает в рекурсивных функциях из-за неправильного условия выхода.
- **`OutOfMemoryError`** - ошибка переполнения памяти.

Когда программа вызывает исключение, говорят, что она *выбрасывает* (throws) его. Если метод способен выбрасывать исключения, которые он сам не обрабатывает, он должен объявить об этом, чтобы вызывающие его методы могли обеспечить обработку этих исключений. Для определения списка исключений, которые могут выбрасываться методом, используется оператор `throws`. Если метод в яв-

ном виде (т.е. с помощью оператора `throw`) выбрасывает исключение соответствующего класса, тип класса исключений должен быть указан в операторе `throws` в объявлении этого метода. В ниже следующем примере выполняется такое описание:

```
public class ThrowsDemo {
     static void procedure() throws Throwable
     {
          System.out.println("внутри процедуры");
          throw new IllegalAccessException("пример вы-
брасывания");
     }
     public static void main(String args[]) throws
Throwable {
          procedure();
     }
}
```

Контролируемое исключение объявляется для компилятора в любом методе с помощью ключевого слова `throws` в сигнатуре этого метода, после чего следует список исключений через запятую, которые метод может потенциально выдать в ходе своего исполнения. Если код вызывает метод, который указывает, что он выдает исключение одного или нескольких типов, их надо определенным образом обрабатывать или добавить `throws` в сигнатуру этого метода, чтобы передать обработку исключения этого типа другим методам.

Альтернативой выполненного описания является самостоятельная обработка исключений, как показано ниже:

```
public class ThrowsDemo {
     static void procedure() throws Throwable
     {
          System.out.println("внутри процедуры");
          throw new IllegalAccessException("пример вы-
брасывания");
     }
     public static void main(String args[])  {
          try {
               procedure();
          } catch (Throwable e) {
               e.printStackTrace();
          }
     }
}
```

Оператор `throw` используется для выбрасывания исключения «вручную». Для этого нужно иметь объект подкласса класса `Throwable`, который может быть получен как параметр оператора

catch, либо создать с помощью оператора new. Общая форма оператора throw: throw <объект тип Throwable >;

При достижении этого оператора нормальное выполнение кода немедленно прекращается, так что следующий за ним оператор не выполняется. Ближайший окружающий блок try проверяется на наличие соответствующего выброшенного исключению обработчика catch. Если такоой находится, то ему передается управление. Если нет, то проверяется следующий из вложенных операторов try до тех пор, пока либо не будет найден подходящий раздел catch, либо обработчик исключений исполняющей системы Java не остановит программу, выведя при этом состояние стека вызовов. Ниже приводится пример, в котором сначала создается объект-исключение, затем оператор throw выбрасывает исключение, после чего это же исключение выбрасывается повторно - на этот раз уже кодом перехватившего его в первый раз раздела catch.

```
public class ThrowDemo {
     static void demoproc() {
          try {
                    throw new
NullPointerException("Демонстрация throw");
          }
          catch (NullPointerException e) {
                    System.out.println("Перехвачена
внутри demoproc:" + e.getMessage());
                     throw e;
          // Повтороно выбрасывает исключение
          }
          }
     public static void main(String args[]) {
          try {
               demoproc();
          } catch (NullPointerException e) {
               System.out.println("повторно
перехвачена:");
          }
     }
}
```

В приведенном примере обработка исключения выполняется в два этапа. Метод main() создает контекст для исключения и вызывает demoproc(). Этот метод также устанавливает контекст для обработки исключения, создает новый объект класса NullPointerException и с помощью оператора throw выбрасывает это исключение. Исключение перехватывается внутри метода demoproc(), причем объект-

исключение доступен коду обработчика через параметр e. Код обработчика выводит сообщение о том, что возбуждено исключение, а затем снова возбуждает его с помощью оператора throw, в результате чего оно передается обработчику исключений в методе main(). Ниже приведен результат, полученный при запуске класса:

```
Перехвачена внутри demoproc:Демонстрация throw
повторно перехвачена:
```

В общем случае при возникновении исключения виртуальная машина Java ищет установленный обработчик исключений ранее в цепочке вызовов методов и, если он достигнет вершины стека, не обнаружив соответствующего обработчика, останавливает выполнение метода класса и выводит сообщение об ошибке в виде стека.

Блоков catch , перехватывающих различные типы ошибок, может быть несколько, но они должны быть определенным образом структурированы. Если какие-то исключения являются подклассами других исключений, то дочерние классы размещаются перед родительскими классами в порядке следования блоков catch. Рассмотрим следующий фрагмент кода:

```
try {
 // Код, потециально опасных операций...
} catch (NullPointerException e) {
 // Обработка NPE...
} catch (Exception e) {
 // Обработка других общих исключений...
}
```

В этом примере NullPointerException - это класс, наследующий класс исключения Exception, так что его следует поместить перед более общим catch блоком Exception.

Вообще говоря, в Java имеется пять ключевых операторов для работы с исключениями:

- try - данное ключевое слово используется для отметки начала блока кода, который потенциально может привести к ошибке.
- catch - ключевое слово для отметки начала блока кода, предназначенного для перехвата и обработки исключений.
- finally - ключевое слово для отметки начала блока кода, которое является дополнительным. Этот блок помещается после последнего блока catch. Управление обычно передаётся в блок finally в любом случае.
- throw - служит для генерации исключений. Оператор throw используется для возбуждения исключения «вручную».

- `throws` - ключевое слово, которое прописывается в сигнатуре метода и обозначающее, что метод потенциально может выбросить исключение с указанным типом.

Общий вид конструкции для "захвата" исключительной ситуации выглядит следующим образом:

```
try {
      // блок кода }
      throw(e) // возбуждение исключения
 catch (ТипИсключения1 e) {
    // обработчик исключений типа ТипИсключения1
     }
 catch (ТипИсключения2 e) {
      // обработчик исключений типа ТипИсключения2
     throw(e) // повторное возбуждение исключения
     }
 finally { }
```

Оператор `finally` предназначен для того, чтобы обеспечить гарантированное выполнение какого-либо фрагмента кода, вне зависимости от того, возникла ли исключительная ситуация. Если блок `finally` завершается ненормально, то весь `try` завершится ненормально по той же причине. Даже в тех случаях, когда в методе нет соответствующего возбужденному исключению раздела `catch`, блок `finally` будет выполнен до того, как управление перейдет к операторам, следующим за разделом `try`. У каждого раздела `try` должен быть, по крайней мере, один раздел `catch` или блок `finally`. Блок `finally` очень удобен для закрытия файлов и освобождения любых других ресурсов, захваченных для временного использования в начале выполнения метода.

Пример с обработкой нескольких типов исключений приведен ниже:

```
import java.util.Scanner;
public class Exept {
     public static void main(String[] args) {
          int[] m = { 0, 0, 0 };
          System.out.println("Вводите число от 0 до
3");
          Scanner sc = new Scanner(System.in);
          try {
               int a = sc.nextInt();
               m[a] = 4 / a;
               System.out.println("Результат вычисле-
ний=" + m[a]);
          } catch (ArithmeticException e) {
```

```
                System.out.println("Произошла недопусти-
мая арифметическая операция");
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Обращение по недопу-
стимому   индексу массива");
            }
            finally {sc.close();}
        }
}
```

При выполнении представленной программы пользователю предлагается ввести чиска от 0 до 3. Если вводится с клавиатуры 1 или 2, то программа отработает без создания каких-либо исключений и выводит полученный результат вычислений. Если пользователь вводит 0, то возникнет исключение класса `ArithmeticException`, и оно будет обработано первым блоком `catch`. Если пользователь вводит 3, то возникнет исключение класса `ArrayIndexOutOfBoundsException` (выход за приделы массива), и оно будет обработано вторым блоком `catch`. Если пользователь вводит нецелое число, то возникнет исключение класса `InputMismatchException` (несоответствие типа вводимого значение), и оно будет выброшено в формате стандартной ошибки, поскольку это исключение никак не обрабатывется в классе.

Блок `finally` отработает в любом случае и закроет объект `sc`.

Здесь мы всего лишь затронули тему обработки исключений Java и, на самом деле, для профессиональной разработки приложений этому следует уделять значительное внимание.

Создание Java-приложений

Продолжим совершенствовать класс `Person` для использования его в Java-приложении. В реультате такого развития будет формироваться представление о том, как класс или набор классов преобразуются в приложение.

Элементы Java-приложения

Каждому Java-приложению требуется точка входа, чтобы виртуальная машина Java знала, откуда начинается выполнение кода. Как мы уже неоднократно отмечали, такой точкой входа служит метод `main()`. Разработанные классы приложения обычно не содержат метода `main()`, но, по крайней мере, один класс в приложении, должен иметь метод с таким именем.

Ранее мы имели дело с классом `Person` и его подклассом `Employee`, которые имели отношение к задаче персонального учета кадров. Теперь давайте введем в рассмотрение новый класс, который придает свойство приложения отдельным классам предметной области и содержит метод `main()`, который является точкой входа создаваемого приложения по учету кадров.

Создадим в Eclipse для запуска приложения по учету кадров в том же пакете класс, используя известную процедуру, которую мы применяли для создания классов `Person` и `Employee`. Присвоим этому классу имя `HumanResourcesApp` и отметим флажок для добавления к классу метода `main()`. Eclipse сгенерирует класс, в котором к методу `main()` добавим несколько строк кода, предназначенных для создания объекта класса `Employee` и присвоения этому объекту значений атрибутов, чтобы он выглядел следующим образом:

```
package ru.ifmo.intro;
public class HumanResourcesApp {
     public static void main(String[] args) {
          Employee e = new Employee();
           e.setName("Petroff");
           e.setEmpID("0001");
           e.setTaxIDNumber("123-45-6789");
           e.printAudit();
     }
}
```

Теперь выполним класс `HumanResourcesApp` и рассмотрим полученный вывод, который будет иметь следующий вид:

```
Name=Petroff,Age=0,Height=0,Weight=0,EyeColor=null,Gender=nu
llru.ifmo.intro.Employee@56864185Employee [taxIDNumber=123-
45-6789, empID=0001, empHireDate=null, empSalary=null]
```

Таким образом, мы создали простое Java-приложение, и в дальнейшем будем рассматривать возможности Java для разработки более сложных приложений.

Наследование

В настоящем пособии неоднократно встречались с примерами наследования. Здесь повторяется и более подробно объясняется, как работает наследование, включая иерархию наследования, конструкторы и наследование, а также абстракцию наследования.

Известно, что классы в Java-коде образуют иерархию. Классы, которые по иерархии находятся выше данного класса, называются *суперклассами* этого класса. Каждый конкретный класс является *подклассом* каждого класса, расположенного в иерархии выше его.

При этом подкласс наследует свойства своего суперкласса. Класс `java.lang.Object` находится на вершине иерархии классов, и это означает, каждый класс Java является подклассом класса `Object` и наследует его свойства.

Например, рассмотрим класс `Person`, который выглядит следующим образом:

```
package ru.ifmo.intro;
public class Person {
     public static final String GENDER_MALE = "MALE";
     public static final String GENDER_FEMALE= "FE-
MALE";
     private String name;
     private int age;
     private int height;
     private int weight;
     private String eyeColor;
     private String gender;
     public Person() {
          super();
     }
// .....
}
```

Класс `Person` неявно наследует свойства `Object`. Поскольку это выполняется для каждого класса, нет необходимости вводить утверждение `extends Object` для каждого определяемого класса. Это наследование означает, что класс наследует свойства своего суперкласса, а именно, класс `Person` имеет доступ к представленным переменным и методам своего суперкласса, т.е. `Person` может "видеть" и использовать общедоступные методы и переменные объекта `Object`, а также его защищенные методы и переменные.

Теперь рассмотрим класс `Employee`, который наследует свойства `Person`. Определение класса будет выглядеть следующим образом:

```
package ru.ifmo.intro;

import java.math.BigDecimal;
import java.util.Date;

public class Employee extends Person {
     private String taxIDNumber; // код
налогоплательщика
     private String empID; // идентификатор
     private Date empHireDate; // дата приема на работу
     private BigDecimal empSalary; // заработная плата
```

```
// .....
}
```

Такие языки, как C++, поддерживают концепцию *множественного наследования*, и это означает, в любой точке иерархии класс может наследовать свойства одного или нескольких классов. Язык Java поддерживает только одиночное *наследование*, то есть ключевое слово `extends` можно использовать только применительно к одному классу. Таким образом, иерархия классов для любого заданного класса Java всегда состоит из пути, проходящего от текущего класса до `java.lang.Object`.

Как мы увидим в дальнейшем, в языке Java имеется обходной путь ограничений одиночного наследования, который поддерживается возможностью реализации нескольких интерфейсов в одном классе.

Описание наследования для `Employee` указывает на то, что `Employee` имеет доступ ко всем общедоступным и защищенным переменным и методам `Person` (он непосредственно расширяет его), а также класса `Object` (поскольку он фактически расширяет и этот класс, хотя и опосредованно). Однако поскольку `Employee` и `Person` находятся в одном и том же пакете, `Employee` имеет также доступ к переменным и методам package-private (их еще иногда называют *дружественными (friendly)*) класса `Person`.

Чтобы расширить иерархию классов еще на один уровень, можно создать третий класс, который расширяет `Employee`:

```
package ru.ifmo.intro;
public class Manager extends Employee {
// . . .
}
```

В языке Java любой класс может иметь не более одного суперкласса, но любое количество подклассов. Это самая важная особенность иерархии наследования языка Java, о которой следует помнить.

Конструкторы и наследование

Конструкторы не являются полноценными элементами классов, поскольку они не наследуются, и поэтому конструкторы необходимо явно реализовывать в подклассах. Прежде чем продолжить рассмотрение концепции наследования, вспомним некоторые из правил определения и вызова конструкторов.

Во-первых, конструктор всегда имеет то же имя, что и класс, в котором он используется, и в его сигнатуре отсутствует описание возвращаемого типа. Например:

```
public class Person {
 public Person() {
 }
}
```

Во-вторых, у каждого класса должен быть, по крайней мере, один конструктор, и если явно не определяется конструктор для класса, то компилятор сгенерирует так называемый *конструктор по умолчанию*.

Чтобы применять конструктор суперкласса, отличный от конструктора по умолчанию, нужно его явно определить. Например, предположим, что у класса `Person` должен применяться конструктор, который принимает имя создаваемого объекта `Person`. Конструктор `Person` можно вызвать в процессе инициализации экземпляра `Employee` из конструктора `Employee()` по умолчанию, как показано в коде ниже:

```
public class Person {
 private String name;
 public Person() {
 }
 public Person(String name) {
 this.name = name;
 }
}
// При этом, в Employee.java
public class Employee extends Person {
 public Employee() {
 super("Petroff");
 }
}
```

Имейте в виду, конструкторы суперклассов, если они вам понадобятся, лучше реализовывать в подклассах и вызывать одинаково. Ниже определяется конструктор класса `Employee`, сигнатура которого выглядит аналогично, классу `Person`, так что их вызов совпадает:

```
public class Person {
 private String name;
 public Person(String name) {
 this.name = name;
 }
}
// При этом, в Employee.java
public class Employee extends Person {
 public Employee(String name) {
 super(name);
 }
}
```

С точки зрения обслуживания такой подход является более предпочтительным.

Объявление конструктора

Первое, что обычно делает конструктор, он вызывает конструктор по умолчанию своего непосредственного суперкласса, если только в первой строке кода конструктора не вызывали другой конструктор. Например, следующие два варианта объявления функционально идентичны, так что можно выбирать любой из них:

Вариант 1:

```
public class Person {
     public Person() {
     }
}
// При этом, в Employee.java
public class Employee extends Person {
     public Employee() {
     }
}
```

Вариант 2:

```
public class Person {
     public Person() {
     }
}
// Между тем, в Employee.java
public class Employee extends Person {
     public Employee() {
          super();
     }
}
```

Отличаются только тем, что во втором случае явно вызывается конструктор по умолчанию.

При создании конструктора с параметрами, необходимо побеспокоиться о создании конструктора по умолчанию (без параметров), в противном случае он будет недоступен, и, в этой связи, ниже следующий код вызовет ошибку компиляции, поскольку конструктор класса `Person` не предусмотрен:

```
public class Person {
          private String name;
          public Person(String name) {
               this.name = name;
          }
}
// Между тем, в Employee.java
public class Employee extends Person {
```

```
        public Employee() {
        }
}
```

В представленном выше коде отсутствует конструктор по умолчанию, поскольку класс содержит альтернативный конструктор с аргументами, и явно не определен конструктор по умолчанию для суперкласса. Конструктор по умолчанию иногда называют конструктором без аргументов (no-arg), поскольку существуют условия, при которых он не генерируется компилятором, и поэтому не всегда может рассматриваться как конструктор по умолчанию.

Конструктор может быть вызван другим конструктором изнутри класса с использованием ключевого слова `this` и соответствующего списка параметров. Так же как вызов `super()`, вызов `this()` должен быть первой строкой кода конструктора. Например:

```
public class Person {
          private String name;
          public Person() {
               this("Guest");
          }
          public Person(String name) {
               this.name = name;
          }
}
```

Такое обращение часто встречается, когда один конструктор делегирует свои полномочия другому, передавая ему некоторое значение по умолчанию при вызове этого конструктора. Это также хороший способ добавить новый конструктор к классу при минимальном влиянии на код, который уже использует прежний конструктор.

Уровни доступа к конструкторам

Конструкторы могут иметь любой уровень доступа; при этом применяются определенные правила видимости, которые приведены в Таблице 7.

Таблица 7. Правила доступа к конструкторам

Модификатор доступа к конструктору	Описание
`public`	Конструктор может быть вызван любым классом.
`protected`	Конструктор может быть вызван классом из того же пакета или любым его подклассом.

Без модификатора (*package-private*)	Конструктор может быть вызван любым классом из того же пакета.
`private`	Конструктор может быть вызван только из того класса, в котором он определен.

Существуют ситуации, когда конструкторы объявляются с доступом `protected` или `private`. Конструкторы `private` используются, когда по тем или иным причинам не допускается прямое создание объекта класса с помощью ключевого слова `new`, например, с использованием модели `Factory`. В модели `Factory` для создания экземпляров класса обычно используется статический метод, который может вызвать конструктор с модификаторм доступа `private`.

Если суперкласс переопределяет метод или переменную суперкласса (другими словами, если подкласс реализует член с таким же именем), то переопределенные члены скрываются (не видны) в суперклассе. Если быть более точным, переопределение переменной скрывает ее, а переопределение метода просто переопределяет его, но эффект одинаков: переопределенный член суперкласса, по существу, не виден из производного класса. Однако при этом можно обратиться к членам суперкласса при помощи ключевого слова `super`:

```
super.hiddenMemberName
```

Абстрактные классы и методы

В контексте ООП *абстрагирование* означает обобщение данных и поведения до типа, более высокого по иерархии наследования, чем текущий класс. При перемещении переменных или методов из подкласса в суперкласс говорят, что эти члены *абстрагируются*. Основной причиной этого является возможность многократного использования общего кода путем размещения его как можно выше по иерархии. Когда общий код собирается в одном месте, облегчается его обслуживание.

Существуют ситуации ООП, когда требуется создавать классы, которые служат только как абстракции, и создавать их экземпляры, быть может, никогда не придется. Такие классы называются абстрактными классами. Например, вы можете создать класс `Animal` (животное). Нет смысла создавать экземпляр этого класса: на практике вам нужно будет создавать экземпляры конкретных классов, например, `Dog` (собака). Но все классы `Animal` имеют некоторые общие действия, например, способность издавать звуки и перемещаться. То, что `Animal`

может издавать звуки, еще ни о чем не говорит и издаваемый звук зависит от вида животного. Это моделируется с помощью определения общего поведения в абстрактном классе и обязанности подклассов реализовывать конкретное поведение, зависящее от их типа. Такая модель представлена на Рис. 67 в терминах диаграммы классов UML.

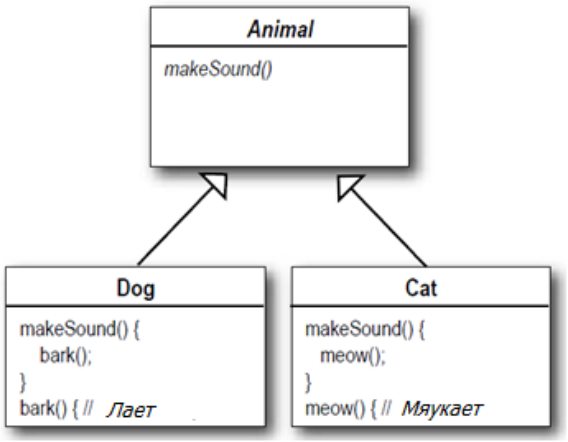


Рис. 67 Диаграммы классов UML с абстрактным классом

Связь между подклассом и суперклассом на диаграмме изображается стрелкой вида  от подкласса к суперклассу. Могут быть образованы подклассы, реализующие все методы абстрактного класса. Если какие-либо методы в подклассе, расширяющем абстрактный класс, остаются нереализованными, то подкласс остается абстрактным.

Реализация модели, изображенной на Рис.67 на языке Java может быть выполнена следующим образом:

```
abstract public class Animal {
   abstract public void makeSound();
}

public class Cat extends Animal {
@Override
public void makeSound() {
     meow();
}

public void meow() {
     System.out.println("Meow!");
}
}

public class Dog extends Animal {
     @Override
     public void makeSound() {
          bark();
```

```
    }

    public void bark() {
        System.out.println("Bark!");
    }
}
```

В общем случае, в иерархии классов могут одновременно находиться как абстрактные, так и конкретные классы. При этом, как видно из кода, определенные методы должны быть реализованы по-разному для каждого подкласса, реализующего суперкласс. Абстрактные классы формально подчиняются следующим основным правилам:

- любой класс может быть объявлен абстрактным;
- абстрактные классы не допускают создания своих экземпляров;
- абстрактный метод не может содержать тела метода;
- класс, содержащий абстрактный метод, должен объявляться как `abstract`.

Использование абстрагирования

Допустим, что в приложении требуется не допускать непосредственное создание экземпляров класса `Employee` с использованием ключевого слова `new`. Для удовлетворения этого требования достаточно объявить этот класс с ключевым словом `abstract`:

```
public abstract class Employee extends Person {
// и т.д.
}
```

Это приведет к тому, что при попытке использовать ниже следующий код диагностируется ошибка компиляции:

```
public void someMethodSomwhere() {
 Employee p = new Employee();// ошибка компиляции!!
}
```

Компилятор выведет сообщение, поскольку `Employee` является абстрактным классом, и экземпляр этого класса не может быть создан.

Предположим, что требуется метод для оценки активности объекта `Employee` для вычисления полученного бонуса. Это требование может оказаться общим для всех объектов `Employee`, но между всеми потенциальными подклассами наследниками данного класса, вычисление данной величины может существенно различаться и не обеспечивает возможности для повторного использования кода. В этом случае метод `float bonus()` объявляется как `abstract`, заставляя все подклассы реализовывать его:

```
public abstract class Employee extends Person {
```

```
        // …
    public abstract float bonus();
        // …
    }
```

В дальнейшем каждый прямой подкласс `Employee` (как, например, `Manager`) должен реализовать метод `bonus()`. IDE Eclipse, например, при указании класса `Person` в качестве суперкласса предложит в составе сгенерированного класса предусмотреть метод `bonus()`, который должен быть реализован для последующей возможности создавать экземпляр класса `Manager`. Фрагмент код ниже сгенерирован IDE Eclipse.

```
    public class Manager extends Employee {
        @Override
        public float bonus() {
            return 0; // Здесь следует определить код
        }
        … // Другие методы
    }
```

Для завершения описания класса `Manager` необходимо определить действующую процедуру начисления величины бонуса для объекта данного класса. При этом, как только подкласс некоторого уровня реализовал абстрактный метод `bonus()`, всем последующим его подклассам нет необходимости реализовывать этот метод, но остается возможность переопределить процедуру выполнения. Например, если имеется класс `Executive`, который расширяет класс `Manager`, то определение, представленное ниже вполне допускается:

```
    public class Executive extends Manager {
        public Executive() {
        }
        … // Другие методы
     }
```

Механизм применения абстрактных классов является мощным средством разработки повторного использования кода и нуждается в некоторых комментариях.

Во-первых, не следует применять абстракции в начальной стадии разработки приложения, даже если у вас имеется достаточный опыт программирования. Основное внимание в процессе разработки следует уделять обеспечению работоспособности классов приложения для удовлетворения требованиям пользователей приложения. Использование абстрактных классов на ранней стадии проектирования вынуждает следовать определенному способу формирования архитектуры приложения и, таким образом, навязывает ограничения в выборе средств разработки. Следует иметь в виду, что общее поведение (в котором заключается

суть абстрактных классов) всегда можно организовать выше по графу наследования и лучше делать только в случае возникновения такой необходимости.

Во-вторых, какими бы мощными классы не были, по мере возможности избегайте использования абстрактных классов, пока в поведении суперклассов не будет много общего. Глубокие графы наследования могут затруднить поддержку кода. Ищите компромисс между слишком большими классами, и кодом, удобным для поддержки.

Присвоение ссылки из одного класса переменной типа, принадлежащего другому классу, допускается, но существуют определенные правила. Рассмотрим следующий пример:

```
Manager manager = new Manager();
Employee employee;
Executive executive = new Executive();
Person person = manager;      // проходит компиляцию
manager = employee;           // не проходит компиляцию!
employee= manager;            // проходит компиляцию
Employee employee2 = employee;// проходит компиляцию
manager = executive ;         // проходит компиляцию
executive = manager;          // не проходит компиляцию!
employee=manager;             // проходит компиляцию
person = employee ;           // проходит компиляцию
```

При выполнении операции присвоения переменная слева должна быть супертипом выражения, указанного с правой стороны, иначе компилятор выдает сообщение об ошибке следующего вида: `Type mismatch: cannot convert from Employee to Manager`. Вообще говоря, выражение справа от операции присвоения должно быть либо подклассом, либо тем же классом, переменной, указанной справа. Так, `Employee` является наследником класса `Person`, но не является наследником класса `Manager`, и компилятор отслеживает соотношения между классами в графе наследования.

Интерфейсы

В ряде случаев, в процессе разработки программного обеспечения, возникают обстоятельства, когда разным коллективам разработчиков надо «договориться» о том, на основании каких соглашений разрабатывать взаимодействующие программы. При этом каждая команда должна иметь возможность писать свой код независимо от другой команды разработчиков. Интерфейсы в ООП как раз являются таким соглашением о взаимодействии или контрактом. Интерфейс определяет границу взаимодействия между классами или компонентами, специфицируя определенную абстракцию, которую применяет реализующая

сторона. Вообще говоря, интерфейс определяет поведенческую активность класса, использующего интерфейс, но оставляет за классом право определять, каким именно образом выполняется реализация этого поведения.

В качестве примера рассмотрим ситуацию, в которой разрабатывается программное обеспечение управления автомобилями без участия человека. Производители автомобилей разрабатывают программное обеспечение, которое выполняет управление движением, выдавая команды на выполнение маневров: остановиться, поехать, увеличить скорость, повернуть и т.д. Другие коллективы разработчиков создают системы машинного видения, распознающие объекты окружающей среды, и используют полученные данные для принятии решений по изменению параметров движения: скорости, направления и т.п.. Производители автомобилей публикуют интерфейс-стандарт, который определяет методы для управления машиной. Таким образом, сторонние разработчики могут знать какие методы вызывать, чтобы управлять движением автомобиля, а производители автомобилей могут изменять внутреннюю реализацию своего продукта в любое время. Ни одна из групп разработчиков не знает, как написаны программы, реализующие объявленные методы интерфейса.

Интерфейс (**interface***)*- это именованный набор моделей поведения (и/или постоянных элементов данных), который должен обеспечить класс, реализующий данный интерфейс. Интерфейс представляет собой семантическую и синтаксическую конструкцию в коде программы, используемую для специфицирования услуг, предоставляемых классом или компонентом. В ООП интерфейс является строго формализованным элементом объектно-ориентированного языка и в качестве семантической конструкции широко используется в языках программирования.

Объявление интерфейсов очень похоже на упрощённое объявление классов. Оно начинается с заголовка, и сначала указываются модификаторы. Интерфейс может быть объявлен как `public`, и в этом случае он будет доступен для общего использования, либо модификатор доступа может не указываться, тогда интерфейс доступен только для типов своего пакета. Модификатор `abstract` для интерфейса указывать не требуется, поскольку все интерфейсы являются абстрактными классами. Синтаксис допускает такое указание, но делать этого не рекомендуется, чтобы не загромождать код. Далее записывается ключевое слово `interface` и имя интерфейса. После этого может следовать ключевое слово `extends` и список интерфейсов, от которых будет наследоваться объявляемый интерфейс. Родительских типов (классов

и/или интерфейсов) может быть много - главное, чтобы не было повторений, и чтобы отношение наследования не образовывало циклической зависимости. Ниже представлен формат объявления интерфейса:

```
public interface  interfaceName extends <Список интерфейсов>{
  [returnType] methodName([argumentList]);
}
```

Во многом объявление интерфейса выглядит аналогично объявлению класса, за исключением того, что при этом используется ключевое слово `interface`. Имя интерфейсу присваивается произвольным образом (в соответствии с правилами языка), и по действующим в языке соглашениям, имена интерфейсам присваиваются так же, как имена классов.

Тело интерфейса состоит из объявления элементов, то есть полей-констант и абстрактных методов. Все поля интерфейса автоматически объявляются `public final static`, так что эти модификаторы указывать необязательно и даже нежелательно, чтобы не загромождать код. Поскольку поля являются финальными, необходимо их сразу инициализировать. Методы, определенные в интерфейсе, объявляют лишь сигнатуру и не имеют тела - за создание тела метода отвечает класс, реализующий интерфейс (как в случае с абстрактными методами). Для того чтобы использовать интерфейс, его нужно реализовать. Это значит обеспечить поведение для методов, определенных в интерфейсе. Каждый класс может реализовывать любое количество доступных интерфейсов. При этом в классе должны быть реализованы все абстрактные методы, появившиеся при наследовании от интерфейсов или родительского класса, чтобы новый класс мог быть объявлен неабстрактным.

Иерархии интерфейсов определяются аналогично классам, за исключением того, что один класс может реализовать сколько угодно интерфейсов. (Класс, как известно, может расширять только один класс). Если один класс наследует другой и реализует интерфейс(ы), то в определении класса сначала указывается единственный наследуемый класс после ключевого слова `extends`, а затем перечисляются реализуемые интерфейсы после ключевого слова `implements`.

Реализация интерфейсов

В качестве примера применения интерфейса рассмотрим следующую ситуацию. Допустим, в организации существует два типа работников: штатные сотрудники и работники по контракту. Вы планируете создавать классы `Employee` и `Contractor` для выполнения функций, которые отражают специфику для этих различных групп. Каждый человек

имеет право на повышение заработной платы, хотя для работников это означает, повышение зарплаты, а для подрядчиков это увеличение почасовой или дневной ставки. Вместо того чтобы создавать два различных метода в этих классах, лучше определить интерфейс, который называется, допустим, `Payable`, и содержит объявление метода `increasePay()`, и оба класса должны будут реализовать его. Код интерфейса `Payable` представлен ниже:

```
public interface Payable {
     int INCREASE_CAP = 20;
     boolean increasePay(int percent);
}
```

Здесь объявлена переменная с атрибутом `final`, которая определяет максимальный процент увеличения зарплаты (все переменные, объявленные в интерфейсе, автоматически становятся `public static final`). Если в будущем изменится величина `INCREASE_CAP`, то необходимо будет изменить значение только в одном место - интерфейсе `Payable`.

Чтобы использовать интерфейс, его нужно *реализовать*, что означает простое предоставление тела метода, которое, в свою очередь, обеспечивает поведение для решения задачи интерфейса. При реализации интерфейса обеспечивается поведение метода (методов) этого интерфейса. Необходимо реализовать методы с сигнатурами, соответствующими сигнатурам интерфейса, с добавлением модификатора доступа `public`.

Ниже представлены коды классов `Employee` и `Contractor`:

```
public class Employee extends Person implements Payable
{
     public boolean increasePay(int percent) {
          // здесь реализация увеличения зарплаты
     }
}
public class Contractor extends Person implements Paya-
ble {
     public boolean increasePay(int percent) {
          // здесь реализация увеличения зарплаты
     }
}
```

Классы `Employee` и `Contractor` наследуют класс `Person` и поэтому содержат утверждение `extends`, и кроме того они содержат утверждение `implements`, поскольку должны реализовать метод `increasePay()` в каждом из классов, в противном случае код не будет компилироваться. Создание классов с общими интерфейсами при-

водит к более прозрачному дизайну приложения и делает код более читаемым.

Интерфейсы-маркеры

Интерфейс может вообще не объявлять методов для последующей реализации. Такие интерфейсы называются *маркерными интерфейсами (marker interfaces)*, потому что они маркируют класс как реализацию этого интерфейса, но не предлагают особого явного поведения. Отсутствует необходимость писать какой-либо код реализации этих интерфейсов; компилятор Java сделает необходимые добавления за вас. Объекты класса, реализующие маркерный интерфейс будет поддерживать определенную функциональность. Одним из примеров такого интерфейса является `Serializable`. Например, если класс реализует `Serializable`, JVM будет иметь возможность сериализовать этот класс - превращать в последовательность байтов (в JVM сервера) таким образом, что последовательность может быть отправлена на другую JVM (на машине клиента), которая сможет воссоздать экземпляр объекта, или десериализовать его.

Кроме того, Java имеет оператор `instanceof`, который позволяет проверить во время выполнения, является ли объект определенным типом. Вы можете использовать оператор `instanceof`, как показано ниже, для проверки реализации объектом маркерного интерфейса, или любого другого типа интерфейса или класса следующим образом:

```
if (receivedFromServerObj instancef Serializable) {
    // что-либо выполняется
}
```

Применение интерфейсов

Существуют три главные причины для использования интерфейсов:

- Создание удобных или описательных пространств имен.
- Установление связи между классами, находящимися в различных иерархиях.
- Сокрытие деталей реализации типа от вашего кода.

При использовании интерфейса для хранения родственных констант этот интерфейс предоставляет наглядное название для ссылки на эти константы. Например, если создан интерфейс с именем `Language` для хранения строковых констант для названия языков, то могли бы об-

ратиться к названиям языков так: `Language.ENGLISH`. Такая возможность облегчает чтение вашего кода.

Как известно, язык Java поддерживает только одиночное наследование. Другими словами, класс может быть подклассом только одного суперкласса, что иногда является ограничением. При использовании интерфейсов можно совместно использовать классы из различных иерархий. Это мощная возможность языка, позволяющая интерфейсу определять набор поведений, который должны поддерживать все классы, реализующие интерфейс. Возможно, единственной общностью классов, реализующих интерфейс, будет являться то, что они совместно используют поведения, определенные в этом интерфейсе.

И последнее, но не менее важное. Использование интерфейсов дает вам возможность при желании игнорировать детали конкретных типов. Использование интерфейсов — один из вариантов обеспечения полиморфизма в объектных языках и средах. Все классы, реализующие один и тот же интерфейс, с точки зрения определяемого ими поведения, ведут себя внешне одинаково. Это позволяет писать обобщённые алгоритмы обработки данных, использующие в качестве типов параметры интерфейсов, и применять их к объектам различных типов, всякий раз получая требуемый результат. Например, интерфейс `Cloneable` может описать абстракцию клонирования (создания точных копий) объектов, специфицировав метод `Clone`, который должен выполнять копирование содержимого объекта в другой объект того же типа. Тогда любой класс, объекты которого может потребоваться копировать, должен реализовать интерфейс `Cloneable` и предоставить метод `Clone`, а в любом месте программы, где требуется клонирование объектов, для этой цели у объекта вызывается метод `Clone`. Причём, использующему этот метод коду достаточно иметь только описание интерфейса, он может ничего не знать о фактическом классе, объекты которого копируются. Таким образом, интерфейсы позволяют разбить программную систему на модули без взаимной зависимости кода.

Создание интерфейсов в Eclipse

В Eclipse имеется удобная поддержка для использования рефакиторинга. Рефакторинг или реорганизация кода - процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью оптимизировать архитектуру класса и облегчить понимание её работы. В общем случае, в основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование ма-

ленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и улучшения поддержки.

Таким образом, если при создании некоторого класса, выясняется абстрактная сущность (некоторая общность, которая должна быть определена на уровне суперкласса) реализованных методов, то имеется возможность на основании соданного класса, применяя рефакторинг, создать суперкласс для данного класса, в котором ряд методов будут иметь описание `abstact`, а текущий класс получит описание `extends`. Например, пусть имеется класс, описанный ниже:

```
public class Process {
     public int add(int i, int j) {
          return (i + j);
     }

     public int sub(int i, int j) {
          return (i - j);
     }
}
```

Предположим, что методы `add` и `sub` в различных классах могут выполняться по-разному, в зависимости от типа решаемой классом задачи. Для того, чтобы преобразовать этот класс в архитектуру с интерфейсом следует поместить указатель мыши в поле редактирования класса и после нажатия на правую выбрать из контекстного меню **Refactor** (или нажать **<alt> <shift>T** и далее выбрать Extract Interface. Откроется диалоговое окно, в котором предлагается сделать установку элементов управления, как показано на Рис. 68.

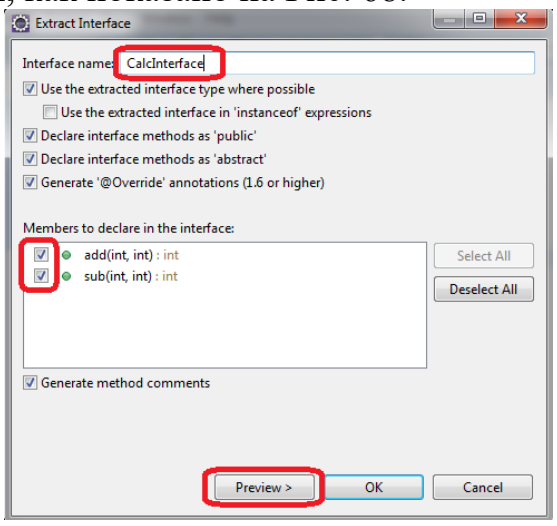


Рис. 68 Диалоговое окно Extract Interface

При нажатии на кнопку Preview откроется окно с предварительным просмотром планируемого рефактоиринга, который представлен на Рис. 69.

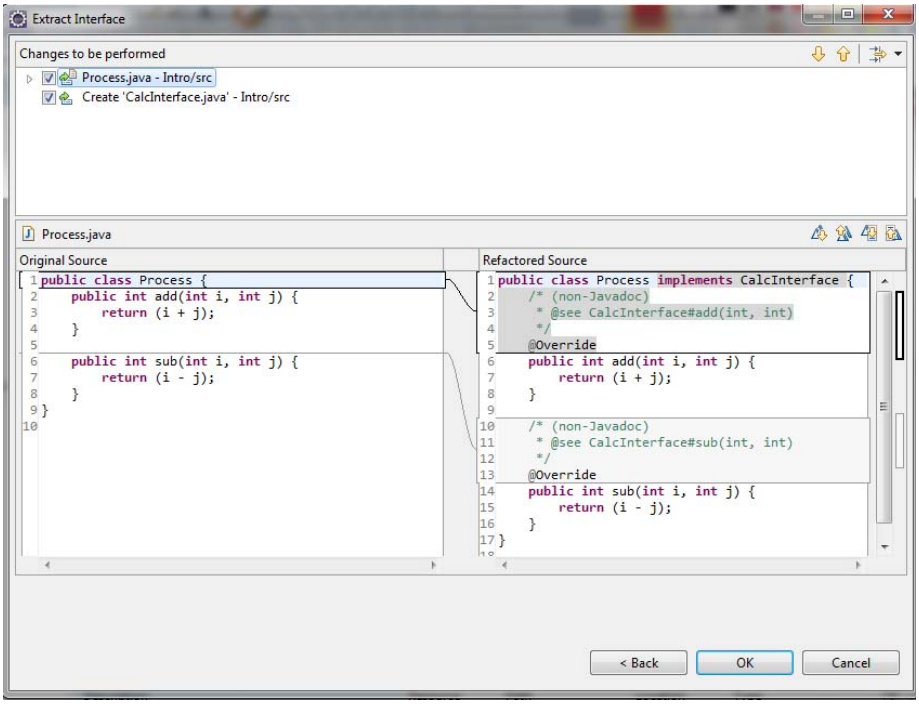


Рис. 69. Предварительный просмотр рефакторинга

При нажатии на кнопку OK выполняется рефакторинг, в результате которого содается интерфейс `CalcInterface` следующегго содержания:

```
public interface CalcInterface {
          public abstract int add(int i, int j);
          public abstract int sub(int i, int j);
}
```

А также изменяется текущий класс `Procees`, который реализует абстрактные методы, объявленные в интерфейсе `CalcInterface`, который представлен ниже:

```
public class Process implements CalcInterface {
     /* (non-Javadoc)
      * @see CalcInterface#add(int, int)
      */
     @Override
     public int add(int i, int j) {
          return (i + j);
     }
     /* (non-Javadoc)
      * @see CalcInterface#sub(int, int)
      */
     @Override
     public int sub(int i, int j) {
```

```
            return (i - j);
        }
}
```

Если вы решите, что один из классов должен реализовать интерфейс, Eclipse легко сгенерирует правильную сигнатуру метода. Для этого при создании класса следует добавить имя интерфейса, который должен реализовать класс, как показано на Рис. 70.

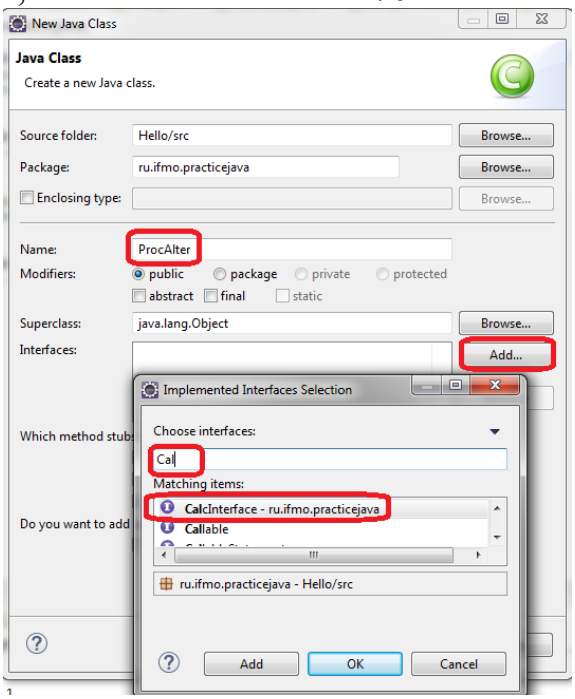


Рис. 70. Диалоговое окно создания класса, реализующего интерфейс

На Рис. 71 определяется новый класс с именем `ProcAlter`, и после нажатия на кнопку **Add** в диалоговом окне выбора интерфейса вводится имя интерфейса с последующим выбором из списка совпадающих имен. Класс может реализовать несколько интерфейсов, и список добавленных интерфейсов может содержать несколько строк. После создания класса с указанными интерфейсами Eclipse сгенерирует заглушки для каждого из методов, которые впоследствии необходимо доопределить в соответствии с конкретной реализацией данного класса, как показано на Рис. 71.

```
1  package ru.ifmo.practicejava;
2
3  public class ProcAlter implements CalcInterface {
4
5      @Override
6      public int add(int i, int j) {
7          // TODO Auto-generated method stub
8          return 0;
9      }
10
11     @Override
12     public int sub(int i, int j) {
13         // TODO Auto-generated method stub
14         return 0;
15     }
16
17
18
19 }
```

Рис. 71. Шаблоны методов интерфейса для реализации

Иначе, при кодировании класса, реализующего интерфейс(ы), можно изменить сигнатуру класса введением ключевого слово `implements` с названием интерфейса или интерфейсов. Eclipse подчеркивает утверждение определения класса красной волнистой линией и отмечает его как ошибочное, потому что класс не содержит метода (методов) интерфейса, как показано на Рис. 72.

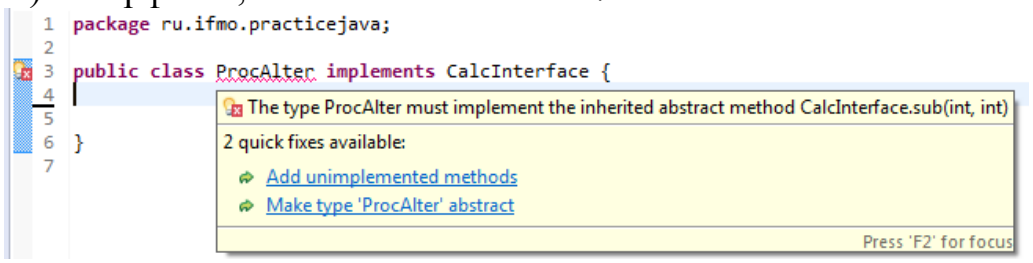


Рис. 72. Диагностика отсутствия методов реализации интерфейса

Подведите указатель мыши к подчеркнутой строке кода, либо выберите строку и нажмите **Ctrl + 1**. Eclipse предложит подсказки как на Рис. 70 и выберите из них **Add Unimplemented Methods** (Добавить нереализованные методы), и будут сгенерированы методы аналогично Рис. 70 в составе кода класса.

Можно объявить класс абстрактным в случае, если он реализует определенный интерфейс, но он не реализует все методы этого интерфейса, поскольку абстрактные классы не обязаны предоставлять реализации всех методов, которые они заявляют к реализации. При этом, первый конкретный класс (то есть, первый класс, который может быть создан) должен реализовать все методы, не реализованные в иерархии.

Применение абстрактных классов и интерфейсов

Если два или более классов имеют много общих функций, и некоторые методы классов должны быть реализованы по-разному, можно создать общего абстрактного предка и столько подклассов, наследующих это общее поведение, сколько необходимо. Объявите в суперклассе абстрактными те методы, которые подклассы должны реализовать поразному, и реализуйте эти методы в подклассах.

Если несколько классов не имеют общей функциональности, но должны проявлять некоторое согласованное поведение, не следует создавать общего предка, но дайте им реализовать интерфейс, который объявляет требуемое поведение.

Интерфейсы и абстрактные классы схожи в том, что они гарантируют, что необходимые методы будут реализованы в соответствии с требуемыми сигнатурами методов. Но они отличаются в том, как программа спроектирована. В то время как абстрактные классы требуют, чтобы был обеспечен общий предок для классов, интерфейсы этого не требуют.

Вложенные классы

Вложенный класс - это класс, определенный внутри другого класса. Экземпляр обычного класса может существовать сам по себе, но в отличие от него, экземпляр внутреннего класса не может существовать без привязки к включающему его классу верхнего уровня. Ниже представлен вложенный класс:

```
public class OuterClass {
. . .
 public class NestedClass {
  . . .

 }
}
```

Переменные и методы вложенного Java-классы можно определить с любым уровнем доступа: `public`, `private` или `protected`. Вложенные классы могут быть полезны, когда нужно управлять обработкой внутри класса по объектно-ориентированной модели, но эта функциональность ограничена классом, в котором она нужна.

Как правило, вложенный класс используется для случаев, когда нужен класс, тесно связанный с классом, в котором он определен. Например, если в классе нужно использовать другой класс и этот класс используется только в этом классе и более ни к какому другому классу отношение не имеет, то имеет смысл этот класс сделать вложенным. Вложенный класс имеет доступ к данным `private` в пределах его

включающего класса, но это сопряжено с некоторыми побочными эффектами, которые не очевидны в начале работы с вложенными (или внутренними) классами.

Вложенный класс может быть помещен как на уровень описания охватывающего класса, внутрь некоторого метода охватывающего класса, а также внутрь некоторого блока.

Область видимости вложенных классов

В соответствии с правилами видимости к переменной-члену можно обращаться только через экземпляр класса (объект). Такие же правила применяются и к вложенным классам.

Предположим, что установлено соотношение между менеджером `Manager` и вложенным классом `DirectReports`, который определяет группу сотрудников `Employee`, которые подчиняются руководителю `Manager`:

```
public class Manager extends Employee {
 private DirectReports directReports;
 public Manager() {
 this.directReports = new DirectReports();
 }
. . .
 private class DirectReports {
 . . .
 }
}
```

Каждый экземпляр класса `Manager` соответствует отдельному человеку, а объект `DirectReports` соответствует группе реальных людей (сотрудников), которые ему непосредственно подчиняются. У разных руководителей будут различные подчиненные сотрудники `DirectReports`. В этом случае на вложенный класс `DirectReports` имеет смысл ссылаться только в контексте экземпляра класса `Manager`, содержащего данный объект, и поэтому он описан с атрибутом `private`.

Внутренние классы с атрибутом public

Поскольку вложенный класс имеет атрибут доступа `private`, экземпляр объекта `DirectReports` может создать только класс `Manager`. Что произойдет, если необходимо предоставить возможность создания экземпляров `DirectReports` внешнему объекту? Казалось бы, в этом случае можно присвоить классу `DirectReports` область

видимости public, и тогда любой внешний код сможет создавать экземпляры DirectReports, как показано на рисунке ниже:

```
public class Manager extends Employee {
 public Manager() {
 }
. . .
 public class DirectReports {
 . . .
 }
}
//
public static void main(String[] args) {
 Manager.DirectReports dr = new Manag-
er.DirectReports();
// Оператор выше не будет работать!
}
```

Код, приведенный выше не будет компилироваться, в связи со способом описания класса DirectReports в классе Manager и правилами для действующей области видимости.

В соответствии с правилами видимости, если объявляется любая нестатическая переменная в классе Manager, то компилятор будет требовать сначала, сослаться на объект Manager и лишь затем на переменную класса. Это правило распространяется и на класс DirectReports. Чтобы создать экземпляр вложенного класса public, можно использовать специальную версию оператора new, которая применяется в сочетании со ссылкой на некоторый содержащий вложенный класс экземпляр. Ниже приведенный код удовлетворяет сформулированным условиям области видимости и позволит создать вложенный класс DirectReports:

```
public class Manager extends Employee {
 public Manager() {
 }
. . .
 private class DirectReports {
 . . .
 }
}
// Между тем, где-нибудь в другом методе...
public static void main(String[] args) {
 Manager manager = new Manager();
 Manager.DirectReports dr = manager.new Direc-
tReports();
}
```

Обратите внимание, что синтаксис использует ссылку на содержащий экземпляр, затем стоит оператор точка (.) и ключевое сло-

во new, после которого следует конструктор создаваемого внутреннего класса.

Статические внутренние классы

Иногда необходимо создать класс, тесно связанный (концептуально) с данным классом, но в котором правила видимости несколько ослаблены и не требуют ссылки на содержащий объект данного класса. В такой ситуации действуют правила ссылок на статические внутренние классы. В этом случае содержащий экземпляр не нужен. Статические вложенные классы, не имеют доступа к нестатическим полям и методам обрамляющего класса, что в некотором роде аналогично статическим методам, объявленным внутри класса. Доступ к нестатическим полям и методам может осуществляться только через ссылку на экземпляр обрамляющего класса. В этом плане статические внутренние классы очень похожи на любые другие классы верхнего уровня. Статические внутренние классы ведут себя как обычные классы Java и, в действительности, должны использоваться только тогда, когда нужно жестко связать класс с его определением. Ниже представлен пример использования статического вложенного класса:

```
public class Bar {
  void foo {
  //создание объекта вложенного статического класса In-
ner
    new Inner();
  }

  static final int CONSTANT = 0;

  // вложенный статический класс
  private static class Inner {
    // поля и методы класса
  }
}
```

Свойства вложенного класса:

- Вложенный статический класс определяется с ключевым словом static, что делает его аналогом статических полей и методов класса. Иным словами, такой класс не связан ни с одним экземпляром внешнего класса.
- Вложенный статический класс имеет доступ ко всем static членам внешнего класса. И, наоборот, все методы внешнего класса имеют доступ ко всем членам вложенного статического класса. Например, в примере к static полю CONSTANT

класса `Bar` можно обратиться из вложенного статического класса `Inner` просто по имени константы.

- В коде, расположенном за пределами внешнего класса на вложенный статический класс можно ссылаться по имени внешнего класса с добавлением после точки имени вложенного класса. В примере к вложенному классу Inner можно обратиться по `Bar.Inner`.
- Во вложенном статическом классе нельзя по имени обратиться к нестатическим членам внешего класса. Но можно создать экземпляр внешнего класса и обратиться после этого к его полю, даже если это поле объявлено с модификатором доступа `private`. Например, так, как показано в примере ниже:

```
public class Outer {
  private int x;
  static class Inner {
    // создаем объект окружающего класса
    Outer o = new Outer();
    // обращаемся к полю x
    o.x;
  }
}
```

Анонимные внутренние классы

Анонимный класс (anonymous class) - это локальный класс без имени. Анонимный класс определяется и инициализируется в едином выражении с помощью ключевого слова `new`. Несмотря на то, что определение локального класса в Java представляет собой оператор в блоке, определение анонимного класса является выражением. Это означает, что его можно записать как часть большого выражения, например, метода. Язык Java позволяет объявлять классы почти везде, даже в середине метода, если это необходимо, и даже без присвоения классу имени. По сути, это просто особенность компилятора, но бывают моменты, когда анонимные внутренние классы чрезвычайно удобны. Поскольку анонимный класс является локальным классом, он имеет все те же ограничения, что и локальный класс. Можно объявить анонимный класс, который будет расширять другой класс или реализовывать интерфейс при объявлении одного, единственного объекта, когда остальным объектам этого класса будет соответствовать реализация метода, определенная в самом классе. В качестве примера применения анонимного класса рассмотрим следующее определение класса:

```
public class Foo {
  // определено множество различных методов
```

```
    public void foo1() { }
    public void foo2() { }
    …
    public void foo100() { }
}
```

Этот класс содержит несколько методов, но необходимо создать всего лишь один экземпляр точно такого же класса, как `Foo`, но с новым методом `fooN()`. В этом случае может использоваться анонимный класс:

```
Foo f = new Foo {
    public void fooN() {
      // код метода
    }
}
```

В примере выше мы получили экземпляр класса наследника класса `Foo` с новым методом `fooN()`. И более того, в этом выражении можно сразу выполнить объявленный метод следующим образом:

```
new Foo {
    public void fooN() {
      // код метода
    }
 }.fooN();
```

Использование анонимных классов целесообразно во многих случаях, в том числе, когда:

- тело класса является очень коротким;
- нужен всего один экземпляр класса;
- класс используется в месте его создания или сразу после него;
- имя класса не важно и не облегчает понимание кода.

Тип Generic

Введение обобщенного типа (`generic`) в JDK 5 стало существенным шагом в развитии языка Java. Обобщения - это параметризованные типы. С помощью параметризованных типов можно объявлять классы, интерфейсы и методы, в которых тип данных указан в виде параметра. Обобщения добавили в язык безопасность типов.

Основным мотивом введения generic-ов в Java было неудобство работы с нетипизированными коллекциями. Использование нетипизированных коллекций приводит к большим трудозатратам, нежели типизированных, причем может порождать ошибки. Эти ошибки обнаруживаются только во время исполнения. Можно всякий раз, когда требуется коллекция, создавать ее типизированный вариант, что избавляет от про-

блем, связанных с применением коллекций, но приводит к большому объему дублированию кода. Generic-и позволяют избежать подобных проблем за счет введения специальных параметров типов и обобщения реализации классов и методов. Они позволяют не создавать отдельную копию типизированной коллекции для измененного типа элемента, а создавать единую обобщенную реализацию, в которой тип элемента заменен параметром типа, и впоследствии возложить работу по созданию специализированных коллекций на компилятор.

Обобщенные типы - это механизм компилятора, посредством которого можно некоторым стандартным образом создавать (и использовать) типы (классов, интерфейсов и т.п.), получая единый код и *параметризуя* (или *обобщая*) все остальное.

Рассмотрим пример класса, который присутствовал в JDK в течение длительного времени: `java.util.ArrayList`, который представляет собой `список объектов`, поддерживаемых массивом.

Фрагмент кода ниже показывает, как создается экземпляр `java.util.ArrayList`.

```
ArrayList arrayList = new ArrayList();
arrayList.add("Hello, world!");
arrayList.add(new Integer(10));
arrayList.add("Congratulate you!");
```

Как видно, `ArrayList` неоднороден и содержит два элемента типа `String` и один тип `Integer`. До выпуска версии JDK 5 в языке Java не средств, которые могли ограничивать такой код, что приводило к многочисленным ошибкам программирования.

Рассмотрим дальнейшие действия, связанные с извлечением данных, помещенных в список. Поскольку список не является однородным, необходимо при извлечении элемента из списка использовать приведение типов. Приведение безопасно, поскольку будет проверено во время исполнения, но при приведении к типу, отличного от извлекаемого, и не являющегося его супертипом, будет сгенерировано исключение `ClassCastException`. В приведенном ниже примере создается коллекция, содержащая две строки и одно целое число, после чего содержимое коллекции выводится на консоль. При этом делается предположение, что коллекция содержит только элементы типа `String`. Для организации цикла в коде метода `printCollection` используется объект класса `Iterator`, созданный на основе объекта класса `Collection` и позволяющий осуществлять перебор элементов коллекции, используя, в частности, метод `hasNext()`, который возвращает `true`, если объект содержит следующий элемент. Кроме этого метода,

класс `iterator` поддерживает также метод `next()`, осуществляющий переход к следующему элементу и метод `remove()`, удаляющий элемент их коллекции (здесь не применяется).

```
package ru.ifmo.practicejava;
import java.util.*;
public class Example1 {
     private void testCollection() {
          List list = new ArrayList();
          list.add(new String("Hello world!"));
          list.add(new String("Good bye!"));
          list.add(new Integer(95));
          printCollection(list);
     }
     private void printCollection(Collection c) {
          Iterator i = c.iterator();
          while (i.hasNext()) {
               String item = (String) i.next();
               System.out.println("Item: " + item);
          }
     }
     public static void main(String argv[]) {
          Example1 e = new Example1();
          e.testCollection();
     }
}
```

Обратите внимание, что используется явное приведение типов – в методе `printCollection`. Этот класс компилируется, но во время исполнения генерируется `ClassCastException`, так как происходит попытка привести значение типа `Integer`, хранящееся в одном из элементов коллекции, к типу `String`:

```
Item: Hello world!
Item: Good bye!
Exception in thread "main" java.lang.ClassCastException: ja-
va.lang.Integer cannot be cast to java.lang.String
at ru.ifmo.practicejava.Example1.printCollection(Example1.java:19)
at ru.ifmo.practicejava.Example1.testCollection(Example1.java:12)
at ru.ifmo.practicejava.Example1.main(Example1.java:26)
```

Не зная, что находится в `ArrayList`, необходимо проверять элемент, к которому происходит обращение, чтобы определить, можно ли выполнять операции с таким типом, в противном случае получите возможность исключения `ClassCastException`.

С помощью обобщенного типа можно указать тип элемента, который содержится в `ArrayList`, и фрагмент кода ниже показывает реализацию такой возможности:

```
import java.util.*;

public class Example2 {
     private void testCollection() {
          List<String> list = new ArrayList<String>();
          list.add(new String("Hello world!"));
          list.add(new String("Good bye!"));
          list.add(new Integer(95));
          printCollection(list);
     }
     private void printCollection(Collection<String> c)
{
          Iterator<String> i = c.iterator();
          while (i.hasNext()) {
               System.out.println("Item: " + i.next());
          }
     }
     public static void main(String argv[]) {
          Example2 e = new Example2();
          e.testCollection();
     }
}
```

При попытке скомпилировать этот код будет диагностирована ошибка, информирующая о том, что не допускается добавлять элемент тира `Integer` в коллекцию элементов типа `String`. Таким образом, generic-и улучшают проверку типов при компиляции и, следовательно, такие ошибки приведения типов выявляются на этапе компиляции, а не во время исполнения.

Обратите внимание на новый синтаксис, использующий для создания экземпляра `ArrayList` параметризованный тип. Параметризованный тип состоит из имени класса или интерфейса `X` и секции параметров `<T1, T2, ..., Tn>`, которая должна соответствовать числу объявленных параметров `X`, и каждый аргумент должен удовлетворять требованиям, предъявляемым к соответствующим параметрам типа. Следующий фрагмент кода показывает сигнатуру определения класса для `ArrayList`:

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
{
  // ...
}
```

Здесь `E` – параметр типа, который действует как заглушка для типа, определяемого при использовании списка.

Обобщенный тип дополнил язык Java специальным синтаксисом для работы с такими объектами, как списки, в которых как правило перебирается элемент за элементом. Например, если перебирается `ArrayList`, код из `printCollection` можно переписать следующим образом:

```
private void printCollection(Collection c)
    {
     for (String s : c) {
     System.out.println("Item: "+s);
     }
}
```

Этот синтаксис работает для объектов любого типа, которые итерабельны (то есть реализуют интерфейс `Iterable`).

Параметризованные классы и методы

Параметризованные классы очень удобны для работы с коллекциями, и в качестве примера рассмотрим использование интерфейса `List`. Как известно, этот интерфейс представляет собой упорядоченную коллекцию объектов. В наиболее распространенных вариантах использования в `List` добавляются элементы, а затем к ним обращаются по индексу или с помощью перебора элементов `List`.

Если предполагается параметризация класса, нужно посмотреть, применимы ли следующие критерии:

- *базовый класс* находится в центре некоторой обертки: то есть "некоторая сущность", находящаяся в центре класса может применяться широко, и окружающие эту сущность элементы (например, атрибуты), идентичны;
- *общее поведение*: независимо от этого " некоторая сущность", которая находится в центре класса, в значительной мере выполняются одни и те же операции.

Коллекция отвечает следующим требованиям, поскольку оба эти критерия выполняются, в частности:

- "некоторая сущность" представляет собой класс в составе `Collection`;
- операции (`add`, `remove`, `size`, `clear` и т.п.) в значительной степени одинаковы, независимо от типа объекта, реализующего коллекцию.

Параметризованный `List`

Код создания `List` в синтаксисе типов выглядит следующим образом:

```
List<E> listRef = new concreteListClass<E>();
```

Здесь символ E, который означает Element, и есть как раз та "некоторая сущность", о чем упоминалось ранее, а concreteListClass - это инициализированный класс JDK. JDK включает в себя несколько реализаций List<E>, но мы будем использовать ArrayList<E>. Другой способ определить обсуждаемый родовой класс – использовать конструкцию Class<T>, где T означает Type (тип). Как правило, E в коде Java означает ссылку на ту или иную коллекцию, а T - на параметризованный класс.

Таким образом, чтобы создать ArrayList, скажем, из элементов java.lang.Integer, нужно сделать следующее:

```
List<Integer> listOfInt = new ArrayList<Integer>();
```

В качестве примера использования параметризованного класс рассмотрим класс SimpleList, который выполняет три метода:

add() - добавляет элемент в конец списка SimpleList;

size() - возвращает текущее количество элементов SimpleList;

clear() - полностью очищает содержимое SimpleList.

Код ниже показывает содержание параметризованного класса SimpleList:

```
import java.util.ArrayList;
import java.util.List;

public class SimpleList<E> {
     private List<E> list;

     public SimpleList() {
          list = new ArrayList<E>();
     }
     public E add(E e) {
          if (list.add(e))
               return e;
          else
               return null;
     }
     public int size() {
          return list.size();
     }
     public void clear() {
          list.clear();
     }
}
```

SimpleList можно параметризовать с помощью любого подкласса Object. Чтобы создать и использовать список SimpleList, допустим, объектов java.math.BigDecimal, нужно сделать следующее:

```
        public static void main(String[] args) {
         SimpleList<BigDecimal> sl = new SimpleList <BigDec-
imal>();
         sl.add(BigDecimal.ONE);
         System.out.println("SimpleList size is : " +
sl.size());
         sl.add(BigDecimal.ZERO);
        System.out.println ("SimpleList size is : " +
sl.size());
         sl.clear();
        System.out.println ("SimpleList size is : " +
sl.size());
        }
```

В результате выполнения метода main() выводятся следующая информация:

```
INFO: SimpleList size is : 1
INFO: SimpleList size is : 2
INFO: SimpleList size is : 0
```

Программирование ввода/вывода

В данном разделе представлен обзор пакета java.io и использование некоторых инструментов для сбора и обработки данными из различных источников. Для работы с физическим файлами и каталогами (директориями), расположенными на внешних носителях, в приложениях Java используются классы из пакета java.io, предоставляющего пользователю большое число классов и методов для реализации ввода-вывода.

Файлы

Из всех источников данных, доступных для Java-приложений, файлы - наиболее распространенный и зачастую наиболее удобный способ хранения данных во внешней памяти. Если Java-приложение должно читать файл, необходимо использовать потоки ввода-вывода (*streams*), которые преобразуют входящие байты из файлов в типы языка Java. Потоки ввода/вывода используются для передачи данных в файлы, на консоль или в сетевые соединения. Потоки представляют собой объекты соответствующих классов. Класс java.io.File служит для хранения и обработки в качестве объектов каталогов и имен файлов. Этот класс не поддерживает методов для работы с содержимым файла,

но позволяет манипулировать такими свойствами файла, как права доступа, дата и время создания, путь в иерархии каталогов, создание, удаление файла, изменение его имени и каталога и т.д. Объект класса `File` создается одним из нижеприведенных способов:

```
File myFile = new File("\\com\\temp.txt ");
File myDir = new File("c:\\jdk1.6.0\\src\\java\\io");
File myFile = new File(myDir, "File.java");
File myFile = new File("c:\\com", "myfile.txt");
File myFile = new File(new URI("Интернет-адрес"));
```

Конструктор класса `File` принимает данные об имени и местоположении создаваемого файла в качестве параметров. При первом вызове создается файл с именем `temp.txt` в каталоге \com. При втором - создается объект, соответствующий подкаталогу. Третий и четвертый случаи аналогичны: для создания объекта указывается каталог и имя файла. В последнем случае создается объект, соответствующий соединению с адресом в Интернете. Конструктору `File` можно передавать любую переменную `String`, если ее значение представляет допустимое имя файла для ОС, независимо от того, существует или нет файл, на который она ссылается.

Существует разница между разделителями, применяемыми при записи пути к файлу: для системы Unix – "/", а для Windows – "\\". Для случаев, когда неизвестно, в какой операционной системе будет выполняться программа, предусмотрены специальные константы в классе `File`:

```
public static final String separator;
public static final char separatorChar;
```

С помощью этих полей можно задать путь, универсальный для любой системы:

```
 File myFile = new File(File.separator + "com"
+ File.separator + "temp.txt" );
```

Также предусмотрен еще один тип разделителей – для директорий:

```
public static final String pathSeparator;
public static final char pathSeparatorChar;
```

Класс `File` поддерживает более тридцати методов, и наиболее используемые из них рассмотрены в следующем примере:.

```
package ifmo.ru;

import java.io.File;
import java.io.IOException;
import java.util.Date;
```

```
public class FileTest {
     public static void main(String[] args) {
          // объект типа File связывается с файлом на диске
testFile.txt
          File fp = new File("file" + File.separator + "test-
File.txt");
          if (fp.exists()) {
               System.out.println(fp.getName() + " существует");
               if (fp.isFile()) {// если объект – файл
                    System.out.println("Путь к файлу: " +
fp.getPath());
                    System.out.println("Абсолютный путь: " +
                              fp.getAbsolutePath());
                    System.out.println("Размер файла: " +
fp.length());
                    System.out.println("Последняя модификация: "
                              + new Date(fp.lastModified()));
                    System.out.println("Файл доступен для чте-
ния: " +                              fp.canRead());
                    System.out.println("Файл доступен для запи-
си: " +                              fp.canWrite());
                    System.out.println("Файл удален: " +
fp.delete());
               }
          } else {
               System.out.println("файл с именем " +
fp.getName()+ " не                         существует");
               try {
                    if (fp.createNewFile())
                         System.out.println("Файл с именем " +
fp.getName()    +                         " создан");
               } catch (IOException e) {
                    System.err.println(e);
               }
          }
          // Далее в объект типа File помещается каталог
          // в корне проекта должен быть создан каталог
file.sample с несколькими
          // файлами
          File dir = new File("file" + File.separator + "sam-
ple");
          if (dir.exists() && dir.isDirectory()) {
          // если объект является каталогом и если этот
                         каталог существует
               System.out.println("каталог " + dir.getName() + "
существует");
               File[] files = dir.listFiles();
```

```
                for (int i = 0; i < files.length; i++) {
                     Date date = new
Date(files[i].lastModified());
                     System.out.print("\n" + files[i].getPath() +
"  | "
                                + files[i].length() + " | " +
                                          date.toString());
                }
           }
           File[] paths; // массив для доступных корневых путей
          try{
           // метод listRoots() возвращает доступные корневые ка-
талоги в массив
              paths = File.listRoots();

              // для каждого каталога в массиве
              for(File path:paths)
              {
                 // выводится путь и доступное и свободное про-
странство
                 System.out.printf("\n%s %,d из %,d
свободно.",path.getPath(), path.getUsableSpace(),
path.getTotalSpace());
              }
           }catch(Exception e){
              // если происходит любая ошибка
              e.printStackTrace();
           }
      }
}
```

В результате выполнения программы файл `TestFile.txt` будет удален, а на консоль выводится следующая информация:

```
testFile.txt существует
Путь к файлу: file\testFile.txt
Абсолютный путь:
F:\eclipse\wroxspace\FileMan\file\testFile.txt
Размер файла: 0
Последняя модификация: Wed Jun 10 10:15:56 MSK 2015
Файл доступен для чтения: true
Файл доступен для записи: true
Файл удален: true
каталог sample существует
file\sample\CreateFile.java | 350 | Wed Jun 10 09:26:26 MSK
2015
file\sample\FileTest.java  | 2119 | Wed Jun 10 09:41:33 MSK
2015
C:\ 11 809 472 512 из 104 751 689 728 свободно.
```

```
D:\ 0 из 0 свободно.
F:\ 335 334 363 136 из 395 248 136 192 свободно.
G:\ 0 из 0 свободно.
```

На основании представленного выше примера видно, что класс `java.io.File` поддерживает множество полезных методов, которые можно использовать для создания и удаления файлов, получения атрибутов файла, создания каталогов (путем передачи имени каталога в качестве аргумента в конструктор `File`), определения, является ли ресурс файлом, каталогом или символической ссылкой, и т.п. У каталога как объекта класса `File` есть дополнительная возможность - представление списка имен файлов с помощью методов `list(), listFiles(), listRoots()`.

Реальные действия по вводу/выводу программы Java состоят в выводе полученных результатов на принтер, в файл, базу данных или передача по сети. Исходные данные также часто необходимо загружать из файла, базы данных или из сети. Основой для организации ввода-вывода являются потоки.

Байтовые и символьные потоки ввода/вывода Java

Для того чтобы отвлечься от особенностей конкретных устройств ввода/вывода, в Java употребляется понятие потока (stream). Считается, что в программу поступает входной поток (`input stream`) символов в кодировке `Unicode` или просто байтов, воспринимаемый в программе методами `read()`. Из программы выводится выходной поток (`output stream`) символов или байтов с использованием методов `write(), print () или println()`. При этом не важно, куда направлен поток: на консоль, на принтер, в файл или в сеть, методы `write ()` и `print ()` ничего об этом не знают.

Потоки ввода последовательности байтов являются подклассами абстрактного класса `InputStream`, потоки вывода – подклассами абстрактного класса `OutputStream`. Эти классы являются суперклассами классов, предназначенных для ввода данных из массивов байтов, строк символов, объектов, а также для выбора данных из файлов или сетевых соединений. При работе с файлами используются подклассы этих классов, соответственно, `FileInputStream` и `FileOutputStream`, конструкторы которых открывают поток и связывают его с соответствующим физическим файлом.

Для чтения байта или массива байтов используются абстрактные методы `read()` и `read(byte[] b)` класса `InputStream`. Метод возвращает значение `–1`, если достигается конец потока данных, и по-

этому возвращаемое значение имеет тип `int`, а не `byte`. При взаимодействии с информационными потоками возможны различные исключительные ситуации, поэтому обработка исключений вида `try-catch` при использовании методов чтения и записи является обязательной. В конкретных классах потоков ввода указанные выше методы реализованы в соответствии с предназначением класса. В классе `FileInputStream` данный метод читает один байт из файла, а поток `System.in` как встроенный объект подкласса `InputStream` позволяет вводить данные с консоли. Абстрактный метод `write(int b)` класса `OutputStream` записывает один байт в поток вывода. Оба этих метода блокируют поток до тех пор, пока байт не будет записан или прочитан. После окончания чтения или записи в поток его всегда следует закрыть с помощью метода `close()`, чтобы освободить ресурсы приложения. Поток ввода связывается с одним из источников данных, в качестве которых могут быть использованы массив байтов, строка, файл, «pipe»-канал, сетевые соединения и др. Иерархия классов байтовых потоков ввода приведена на Рис. 73.

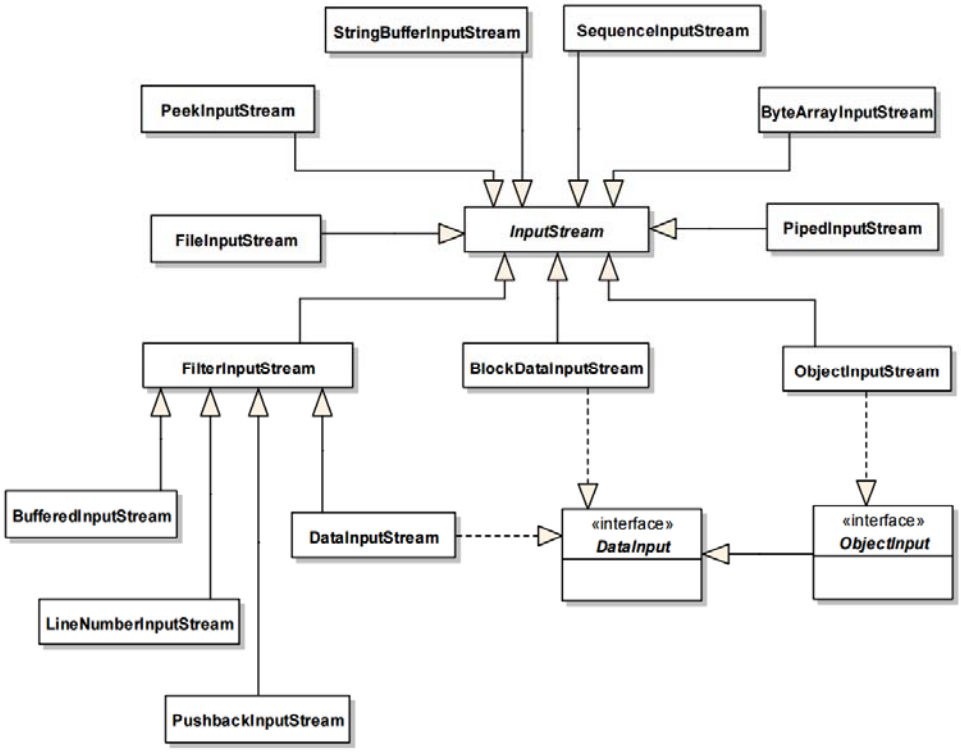


Рис. 73. Иерархия классов байтовых потоков ввода

Абстрактный класс `FilterInputStream` используется как шаблон для настройки классов ввода, наследуемых от класса `InputStream`. Класс `DataInputStream` предоставляет методы для чтения из потока данных значений базовых типов, но начиная с версии

JDK 1.2 класс был помечен как `deprecated` (устаревший), и не рекомендуется к использованию. Класс `BufferedInputStream` присоединяет к потоку буфер для ускорения последующего доступа.

Иерархия классов байтовых потоков вывода приведена на рис. 74.

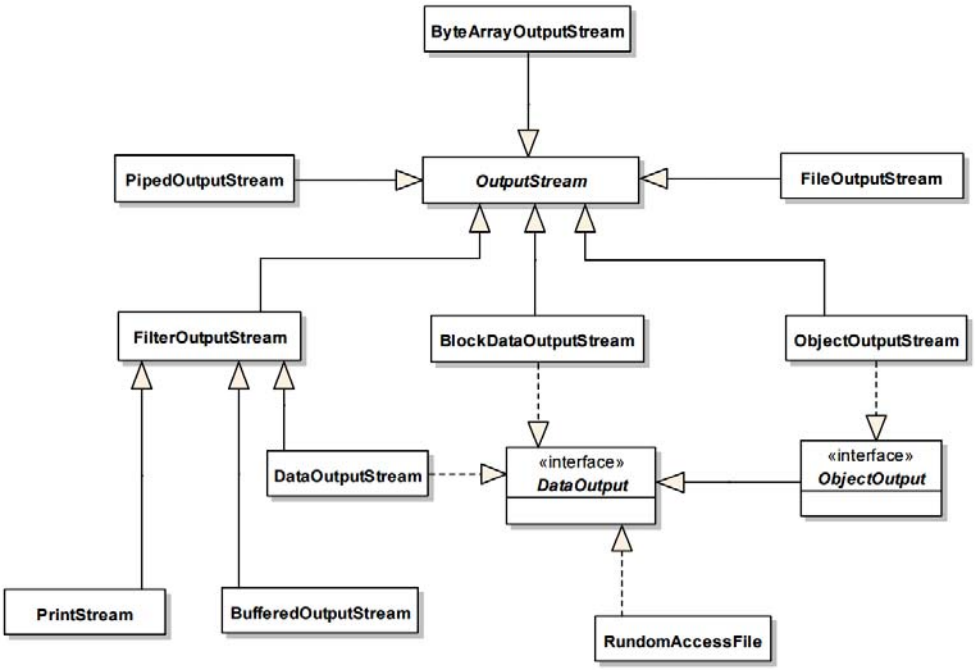


Рис. 74. Иерархия классов байтовых потоков вывода

Абстрактный класс `FilterOutputStream` используется как шаблон для настройки производных классов. Класс `BufferedOutputStream` присоединяет буфер к потоку для ускорения вывода и уменьшения количества обращений к внешним устройствам. Начиная с версии JDK 1.2 пакет `java.io` существенным образом модифицирован. В нем появились новые классы, которые производят скоростную обработку потоков, хотя и не полностью перекрывают возможности классов предыдущей версии.

Для обработки символьных потоков в формате `Unicode` применяется отдельная иерархия подклассов абстрактных классов `Reader` и `Writer`, которые почти полностью повторяют функциональность байтовых потоков, но являются более актуальными при передаче текстовой информации. Например, аналогом класса `FileInputStream` является класс `FileReader`. Такой широкий выбор потоков позволяет выбрать наилучший способ записи в каждом конкретном случае. На Рис. 75 представлены классы потоков ввода и вывода 16-битных символов в формате `Unicode`.

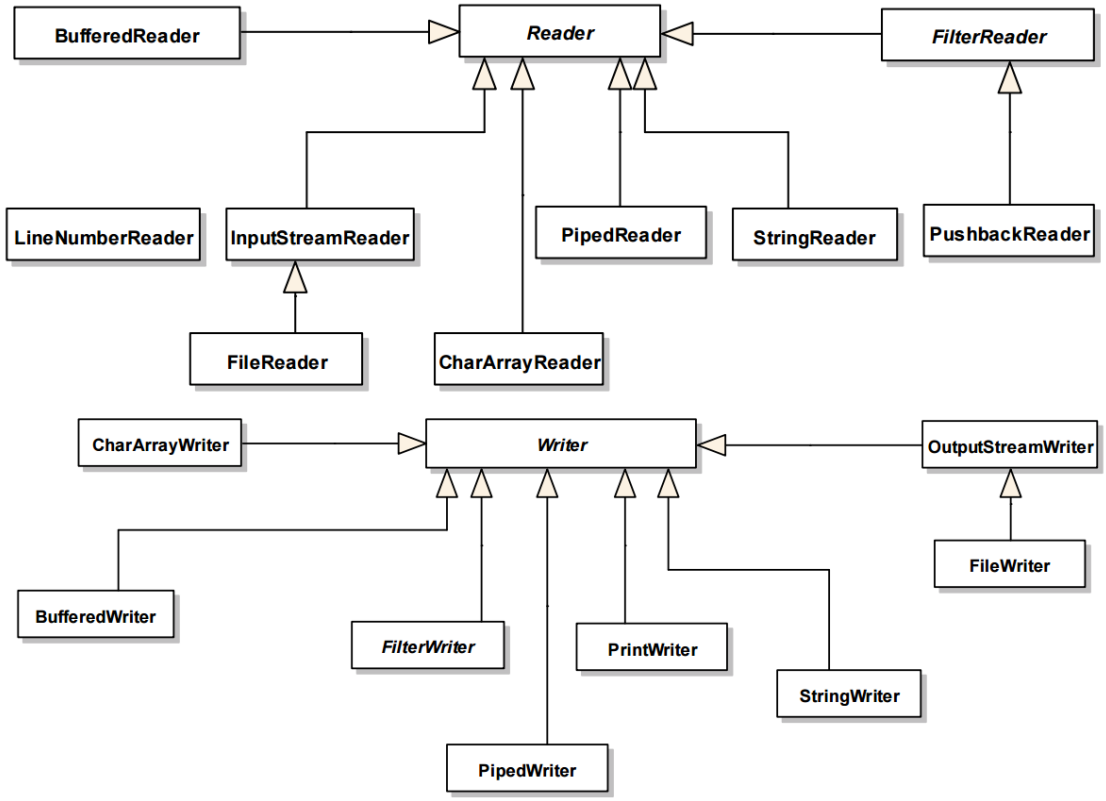


Рис. 75. Иерархия символьных потоков ввода/вывода

Потоки `Character` считывают (`Reader` и его подклассы) и записывают (`Writer` и его подклассы) 16-битные символы.

Вместо академического рассмотрения применения потоков во всей полноте возможностей, рассмотрим ряд примеров, демонстрирующих выполнение типичных задач ввода-вывода.

Чтение по одному символу из файла

Существует несколько способов чтения данных из файла. Возможно, самый простой подход состоит в следующем:

- создать объект `FileReader` для файла, из которого требуется считывать символы.
- вызывать в цикле метод `read()` и считывать по одному символу за раз до конца файла.

Пример чтения из файла приведен ниже:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class ReadDemo {
public static void main(String[] args) {
     File f = new File("testFile.txt");// должен
          существовать!
     int b, count = 0;
```

```
        try {
            FileReader is = new FileReader(f);
            /* FileInputStream is = new FileIn-
putStream(f);                  */// альтернатива
            while ((b = is.read()) != -1) {/* чтение */
                System.out.print((char) b);
                count++;
            }
            is.close(); // закрытие потока ввода
        } catch (IOException e) {
            System.err.println("ошибка файла: " + e);
        }
        System.out.print("\n число байт = " + count);
    }
```

В рассмотренном примере описывается объект `f` типа `File`, который ссылается на файл с именем `testFile.txt`, расположенный в текущем проекте и содержащий некоторую текстовую информацию. В блоке `try` используется один из конструкторов `FileReader(f)` или `FileInputStream(f)` (закомментирован в коде), который открывает поток `is` и связывает его с файлом `f`. Далее в цикле до конца файла вводится из потока символ (`is.read()`) и присваивается целочисленной переменной `b` и преобразованный в `char` выводится на экран. Также в цикле рассчитывается количество введенных символов (`count++;`). Для закрытия потока `is` используется метод `close()`. В случае обнаружения ошибки ввода/вывода, вместо традиционного стека выводится сообщение об ошибке. При чтении из потока имеется возможность пропустить `n` байт с помощью метода `long skip(long n)`.

Класс `FileReader` предназначен для чтения потока символов из файла. Конструкторы этого класса предполагают использование по умолчанию кодировки символов и размер буфера. Чтобы указать эти значения самостоятельно, должен быть вызван конструктор `InputStreamReader` на `FileInputStream`. Для чтения потока байтов можно использовать класс `FileInputStream`.

Пример чтения из файла с применением конструктора `InputStreamReader` на `FileInputStream` приведен ниже:

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
public class ReadDemo {
    public static void main(String[] args) throws
        FileNotFoundException {
```

```
                File f = new File("temp.txt");//должен быть!
                FileInputStream fis = new FileInputStream(f);
                int b, count = 0;
                try {
                     InputStreamReader is = new In-
putStreamReader(fis); // Конструктор на FileInputStream
                     /* FileInputStream is = new FileIn-
putStream(f); */// альтернатива
                     while ((b = is.read()) != -1) {/* чтение
*/
                          System.out.print((char) b);
                          count++;
                     }
                     is.close(); // закрытие потока ввода
                } catch (FileNotFoundException e) {
                     System.err.println("ошибка файла: " +
e);
                } catch (IOException e) {
                     System.err.println("ошибка файла: " +
e);
                }
                System.out.print("\n число байт = " + count);
           }
     }
```

Запись в файл

Как и в случае чтения из файла, существует несколько способов записи в файл. Рассмотрим наиболее простой подход, который состоит в следующем:

- Создать объект `FileOutputStream` для файла, в который нужно записывать данные.
- Вызывать метод `write()` для записи последовательности символов.

Для вывода символов (байтов) или массива символов (байтов) в поток используются потоки вывода - объекты подкласса `FileWriter` суперкласса `Writer` или подкласса `FileOutputStream` суперкласса `OutputStream`. Ниже представлен пример вывода массива в связанные с файлами потоки с использованием метода write() в виде символов (FileWriter) и байтов (FileOutputStream):

```
     import java.io.File;
     import java.io.FileOutputStream;
     import java.io.FileWriter;
     import java.io.IOException;
     public class Writer {
          public static void main(String[] args) {
```

```
            String pArray[] = { "2015 ", "Java SE 8" };
            File fbyte = new File("byte.txt");
            File fsymb = new File("symbol.txt");
            try {
                 FileOutputStream fos = new
                 FileOutputStream(fbyte);
                 FileWriter fw = new FileWriter(fsymb);
                 for (String a : pArray) {
                      fos.write(a.getBytes());
                      fw.write(a);
                 }
                 System.out.println("Данные записаны");
                 fos.close();
                 fw.close();
            } catch (IOException e) {
                 System.err.println("ошибка файла: "+ e);
            }
       }
  }
```

В результате выполнения программы будут получены два файла byte.txt и symbol.txt с идентичным набором данных, но созданные различными способами. В рассмотренном примере создается символьный массив pArray, содержащий 2 элемента.

В отличие от классов `FileInputStream` и `FileOutputStream` класс `RandomAccessFile` позволяет реализовать произвольный доступ к потокам, как ввода, так и вывода. Поток рассматривается при этом как массив байтов, доступ к элементам осуществляется с помощью метода `seek(long poz)`. Для создания потока можно использовать один из конструкторов:

```
RandomAccessFile(String name, String mode);
RandomAccessFile(File file, String mode);
```

Параметр mode равен "`r`" для чтения или "`rw`" для чтения и записи.

Класс `RandomAccess`, представленный ниже, демонстрирует запись элементов массива по порядку в файл с использованием потока `RandomAccessFile` и чтение из этого же потока в обратном порядке:

```
package ifmo.ru;

import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccess {

     public static void main(String[] args) {
          float data[] = { 10, 20, 40, 80, 160, 320 };
```

```
            try {
                RandomAccessFile rf = new RandomAccess-
File("temp.txt", "rw");
                for (float d : data)
                    rf.writeFloat(d); // запись в файл
                /* чтение в обратном порядке */
                for (int i = data.length - 1; i >= 0; i-
-) {
                    rf.seek(i * 4); // обращаемся к эле-
менту по номеру
                    // длина каждой переменной типа
double равна 4-и байтам
                    System.out.println(rf.readFloat());
                }
                rf.close();
            } catch (IOException e) {
                System.err.println(e);
            }
        }
}
```

Вывод класса `RandomAccess` представлен ниже:

```
320.0
160.0
80.0
40.0
20.0
10.0
```

Система ввода/вывода языка Java содержит стандартные потоки ввода, вывода и вывода ошибок. Класс `System` пакета `java.lang` содержит поле `in`, которое является ссылкой на объект класса `InputStream`, и поля `out`, `err` – ссылки на объекты класса `PrintStream`, объявленные со спецификаторами `public static` и являющиеся стандартными потоками ввода, вывода и вывода ошибок, соответственно. Эти потоки связаны с консолью, но могут быть переназначены на другое устройство.

Так, появляется возможность в отличие от привычного ввода данных в программе с клавиатуры (`System.in`) с использованием класса `Scanner`, переопределить стандартный поток данных на существующий файл, содержащий необходимые значения. Для этого необходимо вместо `System.in` указать объект типа `File`, который ссылается на существующий файл. Ниже представлен пример ввода данных из файла и их вывода на экран:

```
import java.io.File;
import java.util.Scanner;
class ScanDemo {
```

```
        public static void main(String[] args) {
            try {
                Scanner in = new Scanner(new
File("temp.txt"));
                StringBuffer data = new StringBuffer();
                while (in.hasNext())
                    da-
ta.append(in.nextLine()).append("\n");
                System.out.println(data.toString());
                in.close();
            } catch ( Exception ex ) {
                ex.printStackTrace();
            }
        }
    }
```

В результате выполнения данного класса `ScanDemo` будет выведено содержимое файла `temp.txt`. Для ввода данных из файла `temp.txt` в текстовом формате используется метод `nextLine()` объекта `in` класса `Scanner`, который связывает стандартный входной поток с файлом. К объекту `in` класса `Scanner` по-прежнему могут применяться все методы, которые позволяют организовать ввод данных и преобразование их соответствующему типу.

Аналогичным образом можно переопределить сами входные и выходные потоки, связывая их с файлами. В частности, для каждого потока в классе `System` существует следующие статические методы:

- `public static void setIn(InputStream in)` для переназначения стандртного входного потока. При этом `in` определяет новый стандартный входной поток;
- `public static void setOut(PrintStream out)` для переназначения стандартного выходного потока. При этом `out` определяет новый стандартный выходной поток.

Используя эти методы можно переопределять входной и выходной потоки, связывя их с файлами.

Рассмотрим переопределение на следующем примере, в котором поток FileOutputStream устанаввливается к полю `out` и `err` для того, чтобы результат вывода на консоль записывался в файл. Код, реализующий поставленную задачу имеет следующий вид:

```
    import java.io.FileInputStream;
    import java.io.FileOutputStream;
    import java.io.IOException;
    import java.io.PrintStream;
    public class SetOut {
        public static void main(String[] args) {
```

```
            try(FileInputStream fis = new FileIn-
putStream("temp.txt");
                FileOutputStream fos = new FileOut-
putStream("server.log",true);) {
          //устанавливаем inputStream
                System.setIn(fis);
                char c = (char) System.in.read();//ввод
символа
                System.out.print(c);//вывод считанного
символа
          //устанавливаем outputStream
                System.setOut(new PrintStream(fos));
                System.out.print("Привет, Java\n");
                //устанавливаем errorStream
                System.setErr(new PrintStream(fos));
                System.err.println("Сообщение об исключе-
нии\n");
            } catch (IOException e) {
                e.printStackTrace();
            }
      }
}
```

В начале метода `main()` класса `SetOut` выполняется попытка связать два потока с соответствующими файлами. При этом предполагается, что для входного потока `FileInputStream` указанный файл `temp.txt` должен существовать в подкаталоге текущего проекта Eclipse. В случае отсутствия указанного файла выбрасывается исключение `FileNotFoundException`. Выходной поток связывается с файлом `server.log` с указанием того, что данный файл открывается для добавления.

Далее в процессе выполнения выше приведенного кода вводится единственный символ из входного потока, который переопределен на файл `temp.txt` и, этот введенный символ выводится на консоль. Для ввода большего количества символов необходимо было бы организовать цикл. Далее переопределяется стандартный выходной поток `out`, и сообщение в этот поток направляются в файл `server.log`. Затем на этот же файл переопределяется поток вывода ошибок `err` и вывод в этот поток также перенаправляется в файл `server.log`. В результате выполнения класса файл `server.log` создается в подкаталоге проекта и содержит выведенные в него данные слндующего вида:

```
Привет, Java
Сообщение об исключении
```

При каждом запуске данного класса вывод добавляется к содержимому файла `server.log`.

Для назначения вывода текстовой информации в произвольный поток следует использовать класс `PrintWriter`, являющийся подклассом абстрактного класса `Writer`.

Для наиболее удобного вывода информации в файл (или в любой другой поток) следует организовать следующую последовательность инициализации потоков с помощью класса `PrintWriter`:

```
new PrintWriter(new BufferedWriter(
new FileWriter(new File("file.txt"))));
```

В результате класс `BufferedWriter` выступает классом-оберткой для класса `FileWriter` при выводе в файл, так же как и класс `BufferedReader` для `FileReader` при чтении из файла.

Приведенный ниже класс `WriteToFile` демонстрирует вывод в файл строк и чисел с плавающей точкой с использованием класса `PrintWriter`.

```
package ifmo.ru;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class WriteToFile {

     public static void main(String[] args) {
          // Объявляется и создается файл для вывода
          File f = new File("result.txt");
          FileWriter fw = null; // Объявляется поток
вывода
          try {
               // Поток FileWriter связывется с файлом
               fw = new FileWriter(f, true);
                         } catch (IOException e) {
               System.err.println("ошибка потока "+ e);
               System.exit(1); // Выход из программы
          }
          // Буфферируется потока для вывода
          BufferedWriter bw = new BufferedWriter(fw);
          // Создается и связывается объект PrintWriter
                         PrintWriter pw = new PrintWrit-
er(bw);
          double[] v={1.10, 1.2, 1.401, 5.01, 6.22,
7.34, 8.05};
          for (double version : v)
               pw.println("Java "+ version);
```

```
            pw.close();
            System.out
            .println("Программа WriteToFile завершилась
успешно\n"
            + "Проверьте содержимое файла result.txt");
      }
}
```

В результате выполнения программы `WriteToFile` содержимое файла `result.txt` будет иметь следующее содержание:

```
Java 1.10
Java 1.20
Java 1.40
Java 5.01
Java 6.22
Java 7.34
Java 8.05
```

Для вывода данных в файл в текстовом формате использовался поток вывода `PrintWriter` и метод `println()`. После соединения этого потока с файлом на диске посредством символьного потока `BufferedWriter` и удобного средства записи в файл `FileWriter` появляется возможность выполнять запись текстовой информации с помощью обычных методов `println()`, `print()`, `printf()`, `format()`, `write()`, `append()`.

Для ввода из файла удобно использовать не байтовый, а символьный поток. В этой ситуации для ввода используется подкласс `BufferedReader` абстрактного класса `Reader` и методы `read()` и `readLine()` для чтения символа и строки соответственно. Этот поток для организации чтения из файла обычно инициализируется объектом класса `FileReader` в виде:

```
new BufferedReader(new FileReader(new File("file.txt")));
```

Чтение файла в примере `ReadDemo` с использованием этой технологии можно выполнить следующим образом:

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
public class BuffferedReadDemo {
      public static void main(String[] args) {

            File f = new File("testFile.txt");// должен быть!
            int count = 0;
            try {
```

```
                    BufferedReader br =
                    new BufferedReader(new FileReader(f));
                    /* FileInputStream is = new FileIn-
putStream(f); */
                    // альтернатива
                    String tmp = "";
                    while ((tmp = br.readLine()) != null) {
                         /* чтение */
                         System.out.println(tmp);
                         count++;
                    }
                    br.close(); // закрытие потока ввода
               } catch (IOException e) {
                    System.err.println("ошибка файла: " + e);
               }
               System.out.print("\n число строк = " + count);
          }
     }
```

В процессе работы класса `BuffferedReadDemo` объявляется буферированный поток `FileReader` на основе объекта файла `f` типа `File`, связанного с существующим файлом `testFile.txt`. В цикле до конца файла считываются строки файла и присваиваются переменной `tmp` типа `String`. Введенные строки выводится в стандартный выходной поток, демонстрируя содержание считываемого файла.

В заключении следует отметить, что чтение и запись потоков символов по одному байту или символу не является эффективным решением, поэтому в большинстве случаев вместо этого, если возможно, следует использовать ввод/вывод с буферизацией. Здесь мы рассмотрели лишь достаточно простые возможности того, что можно делать с помощью этой важной библиотеки Java. Можно быть уверенным, что в составе этой библиотеки всегда можно найти эффективный вариант выполнения ввода-вывода с использованием соответствующих потоков и методов.

Сериализация Java

Кроме данных базовых типов, в поток можно отправлять объекты произвольных классов. Процесс преобразования объектов в потоки байтов для хранения называется сериализацией. Процесс извлечения объекта из потока байтов называется десериализацией. *Сериализация* - это процесс, когда состояние объекта и его метаданные (например, имя класса объекта и имена его атрибутов) сохраняются в особом двоичном формате. Преобразование объекта в этот формат - *сериализация* - позво-

ляет сохранить всю информацию, необходимую для последующего восстановления (или *десериализации*) объекта в нужное время.

Существует две основных причины сериализации объекта:

- *Сохранение объекта* - это запись состояния объекта в постоянном устройстве хранения, например, в базе данных.
- *Передача объекта* означает передачу объекта в другой компьютер или систему.

Для того, чтобы объекты класса могли выполнять процесс сериализации, этот класс должен расширять интерфейс `Serializable`. Все подклассы такого класса также будут сериализованы. Многие стандартные классы реализуют этот интерфейс. Этот процесс заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора `static` или `transient`, поскольку такие поля, не могут быть предметом сериализации, но существует различие в десериализации. Так, поле со спецификатором `transient` после десериализации получает значение по умолчанию, соответствующее его типу (объектный тип всегда инициализируется по умолчанию значением `null`), а поле со спецификатором `static` получает значение по умолчанию в случае отсутствия в области видимости объектов данного типа, а при их наличии получает значение, которое определено для существующего объекта.

java.io.Serializable

Первым шагом к выполнению работы по сериализации является предоставление объектам возможности использовать этот механизм. Каждый объект, который нужно сериализовать, должен реализовать интерфейс `java.io.Serializable`:

```
import java.io.Serializable;
public class Person implements Serializable {
// и т.д.
}
```

Интерфейс `Serializable` помечает объекты класса `Person` для среды исполнения как *сериализуемые*. При этом каждый подкласс `Person` также помечается как сериализуемый.

При попытке сериализовать объект любые несериализуемые атрибуты объекта вызовут выдачу из виртуальной машины Java исключения `NotSerializableException`. Этим обстоятельством можно управлять с помощью ключевого слова `transient`, которое указывает среде времени выполнения, что пытаться сериализовать те или иные атрибуты не нужно. В этом случае сам программист несет ответствен-

ность за восстановление этих атрибутов для правильного функционирования объекта.

Пример сериализации объекта

Теперь рассмотрим пример, в котором сочетается все, что вы только что узнали об операциях ввода/вывода Java и о сериализации.

Допустим, что перед нами стоит задача создать приложение на языке Java, предназначенное для сохранения сведений о пользователях в файле. Кроме того, приложение должно обеспечивать вывод из файла списка сохраненных пользователей. Это является типичной задачей по учету сведений с различными типами объектов. Для выполнения этой задачи, прежде всего, необходимо разработать сериализуемый класс, который хранит необходимые для регистрации сведения. Ниже представлен код класса `User.java` с минимально необходимым набором полей:

```
package business;
import java.io.Serializable;

public class User implements Serializable
{
     private String firstName;
     private String lastName;
     private String emailAddress;
     public User() {
          firstName = "";
          lastName = "";
          emailAddress = "";
     }
     public User(String firstName, String lastName,
String emailAddress) {
          super();
          this.firstName = firstName;
          this.lastName = lastName;
          this.emailAddress = emailAddress;
     }
     public String getFirstName() {
          return firstName;
     }
     public void setFirstName(String firstName) {
          this.firstName = firstName;
     }
     public String getLastName() {
          return lastName;
     }
     public void setLastName(String lastName) {
```

```
            this.lastName = lastName;
        }
        public String getEmailAddress() {
            return emailAddress;
        }
        public void setEmailAddress(String emailAddress) {
            this.emailAddress = emailAddress;
        }
    }
```

Класс `User` содержит два конструктора и методы `get` и `set` для получения и присвоения значений полей, соответственно.

Для реализации необходимых операций ввода-вывода создается класс `UserIO`, в котором разрабатываются два метода:

- метод `void add(User user, ObjectOutputStream oos)` предназначен для записи атрибутов объекта `user` в открытый ранее файл, связанный с выходным потоком `ObjectOutputStream`.
- метод `List<User> getUsers(String filepath)` предназначен открытия и чтения файла с именем `filepath` с последующим формировнием списка `List`, состоящего из введенных объектов `User`.

Ниже представлен код класса `UserIO`, реализующий эти методы:

```
    package data;
    import java.io.*;
    import java.util.ArrayList;
    import java.util.List;
    import business.User;
    public class UserIO {
        public static void add(User user, ObjectOut-
putStream oos) throws IOException {
            // Объект user записывается в поток oos
            oos.writeObject(user);
        }
        public static List<User> getUsers(String filepath)
{
            // Открывается файл с именем filepath
            File file  = new File(filepath);
            // Создается List users
            List <User> users = new ArrayList <User>();
            try {
                // Создается входной поток объектов
                FileInputStream fis = new FileIn-
putStream(file);
                ObjectInputStream ois = new ObjectIn-
putStream(fis);
```

```
                    while (fis.available()>0) {
                         try{
     // Выполняется чтение объекта и добавление к списку
                              User    unknown    =    (User)
ois.readObject();
                              users.add(unknown);
                         }
                         catch(EOFException    |    ClassNot-
FoundException e){
                System.out.println("Что-то пошло не так");
                              break;
                         }
                    }
                    ois.close(); // Закрывается поток
                    return users; // Возвращается список
               } catch (IOException e) {
                    e.printStackTrace();
                    return null;
               }
          }
     }
```

Здесь для записи объекта вызывается метод `writeObject()` для выходного потока. `writeObject ()` - это метод, который использует Java-сериализацию для преобразования объекта в поток. В методе `getUsers` считывается только что сериализованный файл и десериализуется его содержание, тем самым восстанавливая состояние объекта `User`. Метод `readObject()` вызывается для входного потока и использует Java-десериализацию для преобразования потока в объект.

Для проверки работоспособности методов созданного класса `UserIO` создается класс `TestIO`, содержащий метод `main`, в котором определяются несколько объектов класса `User`, и которые последовательно записываются в файл с помощью созданного потока `ObjectInputStream` и путем вызова методу `add` класса `UserIO`. Затем выполняется вызов метода `getUsers`, который создает и возвращает объект `List`, содержащий список ранее записанных во внешней памяти объектов `User`. В заключении сформированный список `list` выводится на консоль. Ниже представлен код класса `TestIO` с комментариями.

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.List;
```

```
     import data.UserIO;
     public class TestIO {
          public static void main(String[] args) throws IO-
Exception {
               String filename = "demo.dat";
               // Создается и открывается файл
               File file = new File(filename);
               // Выходной поток связывается с файлом
               FileOutputStream fos = new FileOut-
putStream(file);
          // Создается выходной поток для записи объектов
               ObjectOutputStream oos = new ObjectOut-
putStream(fos);
          // Создаются объекты для последующей зписи в файл
               User u1 = new User("Ivan", "Ivanov",
"Ivan.Ivanov@gmail.com");
               User u2 = new User("Petr", "Petrov",
"Petr.Petrov@gmail.com");
               User u3 = new User("Stepan", "Stepanov",
"Stepan.Stepanov@gmail.com");
               // Объекты записываются в файл
               UserIO.add(u1, oos);
               UserIO.add(u2, oos);
               UserIO.add(u3, oos);
               oos.close(); // Закрывается файл
               // Формируется список на основе файла
               List<User> list = UserIO.getUsers(filename);
               // Из списка выводятся значение атрибутов
               for (User lst : list) {
                    System.out.printf(
                    "FirstName = %s, LastName = %s,
EmailAddress = %s \n",lst.getFirstName(), lst.getLastName(),
          lst.getEmailAddress());
               }
          }
     }
```

В результате выполнения класса `TestIO` формируется следующий вывод на консоли:

```
FirstName =  Ivan, LastName = Ivanov, EmailAddress =
Ivan.Ivanov@gmail.com
FirstName =  Petr, LastName = Petrov, EmailAddress =
Petr.Petrov@gmail.com
FirstName =  Stepan, LastName = Stepanov, EmailAddress =
Stepan.Stepanov@gmail.com
```

Созданные объекты класса User в процессе записи и чтения из файла были сериализованы и десериализованы. В большинстве случаев маркировка объектов как `Serializable` - это все, о чем когда-нибудь

придется беспокоиться в связи с сериализацией. В тех случаях, когда нужно явно выполнять сериализацию и десериализацию объектов, можно использовать подход, показанный в выше приведенных примерах.

Свойство serialVersionUID

При сериализации объекта класса, реализующего интерфейс `Serializable`, учитывается порядок объявления полей в классе и поэтому, если измененился порядок полей в классе, то десериализация пройдет некорректно. Это обусловлено тем, что в каждый класс, реализующий интерфейс `Serializable`, на стадии компиляции добавляется поле `private static final long serialVersionUID`. Это поле содержит уникальный идентификатор версии сериализованного класса. Оно может вычисляться по содержимому класса – полям, их порядку объявления, методам, их порядку объявления.

Это поле записывается в поток при сериализации класса. Это единственный случай, когда `static`-поле сериализуется. При десериализации значение этого поля сравнивается с имеющимся у класса в виртуальной машине. Если значения не совпадают, инициируется исключение `java.io.InvalidClassException`. Соответственно, при любом изменении в классе это поле поменяет свое значение.

Если набор полей класса и их порядок жестко определены, то методы класса могут меняться. В этом случае сериализации ничего не угрожает, однако стандартный механизм не позволит десериализовать данные. Чтобы избежать этого, требуется самостоятельно в классе определить поле `private static final long serialVersionUID`:

```
import java.io.Serializable;
public class Person implements Serializable {
 private static final long serialVersionUID = 20150620L;
// ...
}
```

В принципе, можно указывать любое значение этой статической переменной и изменять его всякий раз при внесении изменений в класс. При этом можно использовать разработанную схему для номера версии `serialVersionUID` (в примере используется текущая дата), причем нужно объявить эту переменную как `private static final` с типом `long`.

Естественно, предполагается, что данный класс участвует в работе системы и передается между различными компонентами приложения. Таким образом, при каждом добавлении или удалении элементов класса

(то есть атрибутов и методов) следует изменять его `serialVersionUID`. При этом в результате некорректного распространения класса по различным компонентам системы выводится исключение `InvalidClassException`, и не допускается выполнение приложения с ошибкой из-за несовместимых изменений класса.

При использовании IDE Eclipse система уведомляет о том, что для сериализуемого класса не объявлена данная переменная с помощью сообщения "`The serializable class User does not declare a static final serialVersionUID field of type long`". При этом имеется два варианта объявления значения на основе порядкового номера версии и содержимого кода сериализуемого класса как показано на Рис. 74.

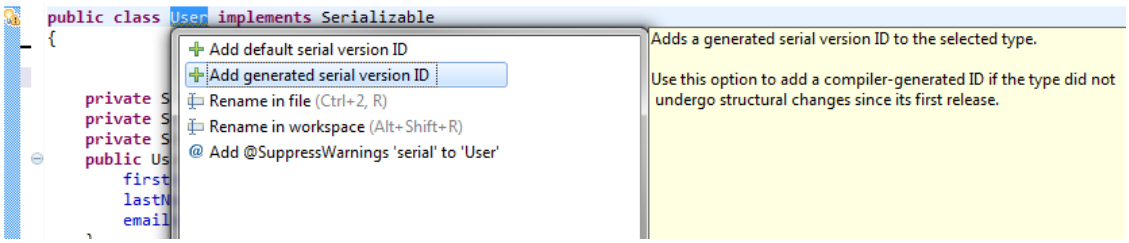


Рис. 74. Варианты генерации объявления переменной `serialVersionUID`

Заключение

В настоящем пособии представлен лишь базовый уровень языка Java, который в дальнейшем позволит изучать другие классы и интерфейсы, расширяющие возможности применения средств программирования Java. При дальнейшем изучении можно осваивать средства создания графического интерфейса пользователя, которые могут быть основаны на использовании библиотек SWING, SWT, GWT; взаимодействие с базами данных с использованием JDBC, Hibernate; создание распределенных приложений на основе применения программирования сокетов, RMI, Java Enterprise Edition и многое другое. Также следует иметь в виду, что на основе использования виртуальной машины Java в настоящее время создано большое количество языков, таких как, Scala, Groovy, Clojure, Xtend и др., которые рассчитаны на облегчение труда Java-кодировщиков.

Литература

1. Ноутон П., Шилдт Г. Java 2 в подлиннике.: Пер. с англ. – СПб.:БХВ-Петербург, 2000. – 1072 с.: ил.

2. Еккель Б. Философия Java. Библиотека программиста. 3-изд. – Спб: Питер, 2003.- 972 с.:ил.

4. Герберт Шилдт. Java. Руководство для начинающих. Вильямс. – М.: КУДИЦ-ОБРАЗ, 2012. – 624 с

4. Герберт Шилдт Java. 8-е издание. Исчерпывающее описание языка Java. Вильямс. – М.: КУДИЦ-ОБРАЗ, 2013. – 1104 с.

5. Кэти Сиерра, Берт Бейт Изучаем Java. Издательство: Эксмо, 2012, 720 с.

8. Вязовик Н.А. Программирование на Java. М., 2003, 592 с.

9. Дарвин Я.Ф. Java. Сборник рецептов для профессионалов. Питер, 2002, 792 с.

УНИВЕРСИТЕТ ИТМО

Миссия университета – генерация передовых знаний, внедрение инновационных разработок и подготовка элитных кадров, способных действовать в условиях быстро меняющегося мира и обеспечивать опережающее развитие науки, технологий и других областей для содействия решению актуальных задач.

КАФЕДРА СЕТЕВЫХ И ОБЛАЧНЫХ ТЕХНОЛОГИЙ

Кафедра Сетевых и облачных технологий образована путем объединения кафедры Телекоммуникационные системы и кафедры Сервисов и услуг в инфокоммуникационных системах в 2015 году.

Кафедра ведет подготовку бакалавров по профилям "Инфокоммуникационные технологии в сервисах и услугах связи" и "Облачные технологии" по направлению 11.03.02 "Инфокоммуникационные технологии и системы связи" и магистерскую подготовку по профилю " Информационные технологии и сервисы в телекоммуникациях" по направлению 11.04.02 "Инфокоммуникационные технологии и системы связи".

Кафедра выполняет большой объем научно-исследовательских работ, связанных развитием и применением сетевых и облачных технологий. В настоящее в рамках федеральной целевой программы проводятся работы по следующим темам:

- Разработка концепции комплексного решения централизованного управления наземным транспортом с учетом межрегионального характера движения на основе облачных и первазивных вычислений (2014-2016 гг.)
- Разработка технологии построения программно-конфигурируемых квантово-криптографических сетей (2015-2017 гг.)

По заказу компаний партнеров совместно с компанией EMC выполняется работа "Анализ функциональности системы OpenStack для использования в качестве универсальной облачной платформы (2015-2016 гг.)

На кафедре функционируют учебно-исследовательские центры компаний Huawei, EMC и National Instruments, оснащенные современным оборудованием и учебно-методическими материалами.

Анатолий Алексеевич Дубаков

Введение в объектно-ориентированное программирование на Java

Учебное пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж 100 экз.
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49