ПРЕДИСЛОВИЕ

Пособие расширяет и включает переработанную и обновленную версию предыдущей книги авторов «Java 2. Практическое руководство», изданную в 2005 г. Рассмотренный материал относится к программированию на Java SE 6 и J2EE.

Книга написана на основе учебных материалов, использующихся в процессе обучения студентов механико-математического факультета и факультета прикладной математики и информатики Белгосуниверситета, а также слушателей курсов повышения квалификации и преподавательских тренингов EPAM Systems, Sun Microsystems и других учебных центров по ряду направлений технологий Java. При изучении Java знание других языков необязательно, книгу можно использовать для обучения программированию на языке Java «с нуля».

Интересы авторов, направленные на обучение, определили структуру этой книги. Она предназначена как для начинающих изучение Java-технологий, так и для продолжающих обучение на среднем уровне. Авторы считают, что «профессионалов» обучить нельзя, ими становятся только после участия в разработке нескольких серьезных Java-проектов. В то же время данный курс может служить ступенькой к мастерству. Прошедшие обучение по этому курсу успешно сдают различные экзамены, получают международные сертификаты и в состоянии участвовать в командной разработке промышленных программных проектов.

Книга разбита на три логических части. В первой части даны фундаментальные основы языка Java и концепции объектно-ориентированного программирования. Во второй части изложены наиболее важные аспекты применения языка, в частности коллекции, многопоточность и взаимодействие с XML. В третьей части приведены основы программировании распределенных информационных систем с применением сервлетов, JSP и баз данных, а также сформулированы основные принципы создания собственных библиотек тегов.

В конце каждой главы даются тестовые вопросы по материалам данной главы и задания для выполнения по рассмотренной теме. Ответы и пояснения к тестовым вопросам сгруппированы в отдельный блок.

В приложениях приведены дополнительные материалы, относящиеся к использованию HTML в информационных системах, основанных на применении Java-технологий, краткое описание портальных приложений и популярных технологий Struts и Hibernate для разработки информационных систем, а также Apache Ant для сборки этих приложений.

В создании некоторых приложений участвовали сотрудники EPAM Systems: приложение «UML» написано совместно с Валерием Масловым; приложение «Базы данных и язык SQL» написано Тимофеем Савичем; корректировка главы «XML&Java» и приложений «Struts» и «Hibernate» выполнена Сергеем Волчком; корректировка приложения «Apache Ant» и раздел «Основные понятия ООП» выполнены Евгением Пешкуром; приложение «JavaScript» создано при участии Александра Чеушева.

В разработке примеров принимали участие студенты механико-математического факультета и факультета прикладной математики и информатики БГУ. Авторы благодарны всем, принимавшим участие в подготовке этой книги.

Часть 1.
ОСНОВЫ ЯЗЫКА JAVA

В первой части книги излагаются вопросы, относящиеся к основам языка Java и технологии объектно-ориентированного программирования.

Глава 1
ВВЕДЕНИЕ В КЛАССЫ И ОБЪЕКТЫ

Основные понятия ООП

Возможности программирования всегда были ограничены либо возможностями компьютера, либо возможностями человека. В прошлом веке главным ограничением были низкие производительные способности компьютера. В настоящее время физические ограничения отошли на второй план. Со всё более глубоким проникновением компьютеров во все сферы человеческой деятельности, программные системы становятся всё более простыми для пользователя и сложными по внутренней архитектуре. Программирование стало делом команды и на смену алгоритмическим идеологиям программирования пришли эвристические, позволяющие достичь положительного результата различными путями.

Базовым способом борьбы со сложностью программных продуктов стало объектно-ориентированное программирование (ООП), являющееся в настоящее время наиболее популярной парадигмой. ООП предлагает способы мышления и структурирования кода.

ООП − методология программирования, основанная на представлении программного продукта в виде совокупности объектов, каждый из которых является экземпляром конкретного класса. ООП использует в качестве базовых элементов эвристическое взаимодействие объектов.

Объект − реальная именованная сущность, обладающая свойствами и проявляющая свое поведение.

В применении к объектно-ориентированным языкам программирования понятие объекта и класса конкретизируется, а именно:

Объект − обладающий именем набор данных (полей объекта), физически находящихся в памяти компьютера, и методов, имеющих доступ к ним. Имя используется для доступа к полям и методам, составляющим объект. В предельных случаях объект может не содержать полей или методов, а также не иметь имени.

Любой объект относится к определенному классу.

Класс содержит описание данных и операций над ними.

В классе дается обобщенное описание некоторого набора родственных, реально существующих объектов. Объект − конкретный экземпляр класса.

В качестве примера можно привести чертеж танка или его описание (класс) и реальный танк (экземпляр класса, или объект).

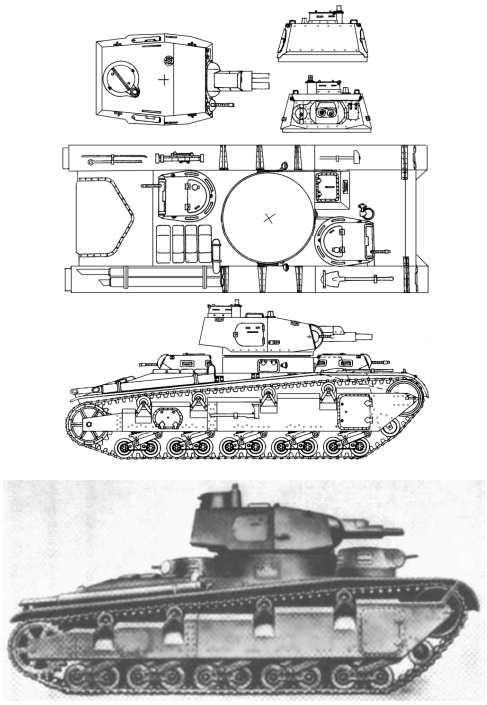

Рис. 1.1. Описание класса (чертеж) и реальный объект

Класс принято обозначать в виде прямоугольника, разделенного на три части:

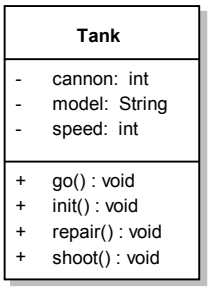

Рис. 1.2. Графическое изображение класса

Объектно-ориентированное программирование основано на принципах:

- абстрагирования данных;
- инкапсуляции;
- наследования;
- полиморфизма;
- «позднего связывания».

Инкапсуляция (encapsulation) − принцип, объединяющий данные и код, манипулирующий этими данными, а также защищающий в первую очередь данные от прямого внешнего доступа и неправильного использования. Другими словами, доступ к данным класса возможен только посредством методов этого же класса.

Наследование (inheritance) − это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним свойства и методы, характерные только для него.

Наследование бывает двух видов:

одиночное − класс (он же подкласс) имеет один и только один суперкласс (предок);

множественное − класс может иметь любое количество предков (в Java запрещено).

Графически наследование часто изображается в виде диаграмм UML:

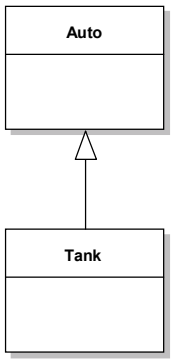

Рис. 1.3. Графическое изображение наследования

Класс «Auto» называется суперклассом, а «Tank» − подклассом.

Полиморфизм (polymorphism) − механизм, использующий одно и то же имя метода для решения двух или более похожих, но несколько отличающихся задач.

Целью полиморфизма применительно к ООП является использование одного имени для задания общих для класса действий. В более общем смысле концепцией полиморфизма является идея «один интерфейс, множество методов».

Механизм «позднего связывания» в процессе выполнения программы определяет принадлежность объекта конкретному классу и производит вызов метода, относящегося к классу, объект которого был использован.

Механизм «позднего связывания» позволяет определять версию полиморфного метода во время выполнения программы. Другими словами, иногда невозможно на этапе компиляции определить, какая версия переопределенного метода будет вызвана на том или ином шаге программы.

Краеугольным камнем наследования и полиморфизма предстает следующая парадигма: «***объект подкласса может использоваться всюду, где используется объект суперкласса***».

При вызове метода класса он ищется в самом классе. Если метод существует, то он вызывается. Если же метод в текущем классе отсутствует, то обращение происходит к родительскому классу и вызываемый метод ищется у него. Если поиск неудачен, то он продолжается вверх по иерархическому дереву вплоть до корня (верхнего класса) иерархии.

Язык Java

Объектно-ориентированный язык Java, разработанный в Sun Microsystems, предназначен для создания переносимых на различные платформы и операционные системы программ. Язык Java нашел широкое применение в Интернет-приложениях, добавив на статические и клиентские Web-страницы динамическую графику, улучшив интерфейсы и реализовав вычислительные возможности. Но объектно-ориентированная парадигма и кроссплатформенность привели к тому, что уже буквально через несколько лет после своего создания язык практически покинул клиентские страницы и перебрался на сервера. На стороне клиента его место занял язык JavaScript.

При создании язык Java предполагался более простым, чем его синтаксический предок C++. На сегодняшний день с появлением версий J2SE 1.5.0 (Тигр) и Java SE 6 (Мустанг) возможности языка Java существенно расширились и во многом перекрывают функциональность C/C++/C#. Отсутствие указателей (наиболее опасного средства языка C++) нельзя считать сужением возможностей, а тем более недостатком, это просто требование безопасности. Возможность работы с произвольными адресами памяти через бестиповые указатели позволяет игнорировать защиту памяти. Отсутствие в Java множественного наследования легко заменяется на более понятные конструкции с применением, например, интерфейсов.

Системная библиотека классов языка Java содержит классы и пакеты, реализующие и расширяющие базовые возможности языка, а также сетевые возможности, взаимодействие с базами данных, графические интерфейсы и многое другое. Методы классов, включенных в эти библиотеки, вызываются из JVM (Java Virtual Machine) во время интерпретации программы.

В Java все объекты программы расположены в динамической памяти – куче (heap) и доступны по объектным ссылкам, которые, в свою очередь, хранятся в стеке (stack). Это решение исключило непосредственный доступ к памяти, но усложнило работу с элементами массивов и сделало ее менее эффективной по сравнению с программами на C++. В свою очередь, в Java предложен усовершенствованный механизм работы с коллекциями, реализующими основные динамические структуры данных и списки. Необходимо отметить, что объектные ссылки языка Java содержат информацию о классе объектов, на которые они ссылаются, так что объектные ссылки – это не указатели, а дескрипторы объектов. Наличие дескрипторов позволяет JVM выполнять проверку совместимости типов на фазе интерпретации кода, генерируя исключение в случае ошибки. В Java изменена концепция организации динамического распределения памяти: отсутствуют способы программного освобождения динамически выделенной памяти. Вместо этого реализована система автоматического освобождения памяти (сборщик мусора), выделенной с помощью оператора **`new`**.

Стремление разработчиков упростить Java-программы и сделать их более понятными привело к необходимости удаления из языка файлов-заголовков (h-файлов) и препроцессорной обработки. Файлы-заголовки C++, содержащие прототипы классов и распространяемые отдельно от двоичного кода этих классов, усложняют управление версиями, что дает возможность несанкционированного доступа к частным данным. В Java-программах спецификация класса и его реализация всегда содержатся в одном и том же файле.

Java не поддерживает структуры и объединения, являющиеся частными случаями классов в C++. Язык Java не поддерживает перегрузку операторов и typedef, беззнаковые целые (если не считать таковым **char**), а также использование методами аргументов по умолчанию. В Java существуют конструкторы, но отсутствуют деструкторы (применяется автоматическая сборка мусора), не используется оператор **goto** и слово **const**, хотя они являются зарезервированными словами языка.

Ключевые и зарезервированные слова языка Java:

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

Кроме ключевых слов, в Java существуют три литерала: **null**, **true**, **false**, не относящиеся к ключевым и зарезервированным словам. Зарезервированные слова: **const**, **goto**.

Нововведения версий 5.0 и 6.0

В версии языка J2SE 5.0 внесены некоторые изменения и усовершенствования:

- введена возможность параметризации класса;
- поддерживается перечисляемый тип;
- упрощен обмен информацией между примитивными типами данных и их классами-оболочками;
- разрешено определение метода с переменным количеством параметров;
- возможен статический импорт констант и методов;
- улучшен механизм формирования коллекций;
- добавлен форматированный консольный ввод/вывод;
- увеличено число математических методов;
- введены новые способы управления потоками;
- используется поддержка стандарта Unicode 4.0;
- добавлены аннотации, новые возможности в ядре и др.

Для версии Java SE 6 характерны высокая скорость, стабильность и оптимальное потребление памяти.

Изменения и усовершенствования:

- новый механизм исполнения сценариев Scripting API;

- поддержка Java-XML Web Service (JAX-WS) для создания приложений поколения Web 2.0;
- улучшены возможности интернационализации ПО, в том числе использования различных региональных форматов и методов преобразования данных;
- новый набор java.awt.Desktop API;
- поддержка области состояния: два новых класса, SystemTray и TrayIcon;
- модернизация в Java Foundation Classes (JFC) и Swing;
- Java-XML Binding (JAXB 2.0);
- JDBC 4.0.

Простое приложение

Изучение любого языка программирования удобно начинать с программы вывода обычного сообщения.

```
// пример # 1 : простое линейное  приложение: First.java
package chapt01;

public class First {
      public static void main(String[] args) {
       // вывод строк
           System.out.print("Мустанг ");
           System.out.println("уже здесь!");
      }
   }
```

В следующем примере то же самое будет сделано с использованием метода класса, реализованного на основе простейшего применения объектно-ориентированного программирования:

```
/* пример # 2 : простое объектно-ориентированное приложение :
FirstProgram.java */
package chapt01;

public class FirstProgram {
      public static void main(String[] args) {
      //объявление и создание объекта firstObject
            MustangLogic firstObject = new MustangLogic();
      //вызов метода, содержащего вывод строки
            firstObject.jumpMustang();
      }
}
// пример # 3 : простой класс: MustangLogic
class MustangLogic {
      public void jumpMustang() {// определение метода
      // вывод строки
            System.out.println("Мустанг уже здесь!");
      }
}
```

Здесь класс **FirstProgram** используется для того, чтобы определить метод **main()**, который запускается автоматически интерпретатором Java и может называться контроллером этого простейшего приложения. Метод **main()** содержит аргументы-параметры командной строки **String[] args**, представляющие массив строк, и является открытым (**public**) членом класса. Это означает, что метод **main()** виден и доступен любому классу. Ключевое слово **static** объявляет методы и переменные класса, используемые при работе с классом в целом, а не только с объектом класса. Символы верхнего и нижнего регистров здесь различаются, как и в C++. Тело метода **main()** содержит объявление объекта

```
MustangLogic firstObject = new MustangLogic();
```

и вызов его метода

```
firstObject.jumpMustang();
```

Вывод строки "Мустанг уже здесь!" в примере осуществляет метод **println()** (**ln** – переход к новой строке после вывода) свойства **out** класса **System**, который подключается к приложению автоматически вместе с пакетом **java.lang**. Приведенную программу необходимо поместить в файл **FirstProgram.java** (расширение **.java** обязательно), имя которого совпадает с именем класса.

Объявление классов предваряет строка

```
package chapt01;
```

указывающая на принадлежность классов пакету с именем **chapt01**, который является на самом деле каталогом на диске. Для приложения, состоящего из двух классов, наличие пакетов не является необходимостью. При отсутствии слова **package** классы будут отнесены к пакету по умолчанию, размещенному в корне проекта. Если же приложение состоит из нескольких сотен классов (вполне обычная ситуация), то размещение классов по пакетам является жизненной необходимостью.

Классы из примеров 2 и 3 сохраняются в файлах **FirstProgram.java**. На практике рекомендуется хранить классы в отдельных файлах.

Простейший способ компиляции – вызов строчного компилятора из корневого каталога (в нем находится каталог **chapt01**):

```
javac chapt01/FirstProgram.java
```

При успешной компиляции создаются файлы **FirstProgram.class** и **Mustang.class**. Запустить этот виртуальный код можно с помощью интерпретатора Java:

```
java chapt01.FirstProgram
```

Здесь к имени приложения **FirstProgram.class** добавляется имя пакета **chapt01**, в котором он расположен.

Чтобы выполнить приложение, необходимо загрузить и установить последнюю версию пакета, например с сайта **java.sun.com**. При инсталляции рекомендуется указывать для размещения корневой каталог. Если JDK установлена в директории (для Windows) **c:\jdk1.6.0**, то каталог, который компилятор Java будет рассматривать как корневой для иерархии пакетов, можно вручную задавать с помощью переменной среды окружения **CLASSPATH** в виде:

```
CLASSPATH=.;c:\jdk1.6.0\
```

Переменной задано еще одно значение '**.**' для использования текущей директории, например **c:\temp**, в качестве рабочей для хранения своих собственных приложений.

Чтобы можно было вызывать сам компилятор и другие исполняемые программы, переменную **PATH** нужно проинициализировать в виде

```
PATH=c:\jdk1.6.0\bin
```

Этот путь указывает на месторасположение файлов **javac.exe** и **java.exe**. В различных версиях операционных систем путь к JDK может указываться различными способами.

Однако при одновременном использовании нескольких различных версий компилятора и различных библиотек применение переменных среды окружения начинает мешать эффективной работе, так как при выполнении приложения поиск класса осуществляется независимо от версии. Когда виртуальная машина обнаруживает класс с подходящим именем, она его и подгружает. Такая ситуация предрасполагает к ошибкам, порой трудноопределимым. Поэтому переменные окружения лучше не определять вовсе.

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

```
/* пример # 4 : вывод аргументов командной строки: OutArgs.java */
package chapt01;

public class OutArgs {
      public static void main(String[] args) {
            for (String str : args)
                  System.out.printf("Apr-> %s%n", str);
      }
}
```

В данном примере используется новый вид цикла версии Java 5.0 **for** языка Java и метод форматированного вывода **printf()**. Тот же результат был бы получен при использовании традиционного цикла

```
for (int i = 0; i < args.length; i++)
   System.out.println("Apr-> " + args[i]);
```

Запуск этого приложения осуществляется с помощью следующей командной строки вида:

```
java chapt01.OutArgs 2007 Mustang "Java SE 6"
```

что приведет к выводу на консоль следующей информации:

```
Apr-> 2007
Apr-> Mustang
Apr-> Java SE 6
```

Приложение, запускаемое с аргументами командной строки, может быть использовано как один из способов ввода строковых данных.

Классы и объекты

Классы в языке Java объединяют поля класса, методы, конструкторы и логические блоки. Основные отличия от классов C++: все функции определяются внутри классов и называются методами; невозможно создать метод, не являющийся методом класса, или объявить метод вне класса; ключевое слово inline, как в C++, не поддерживается; спецификаторы доступа **public**, **private**, **protected** воздействуют только на те объявления полей, методов и классов, перед которыми они стоят, а не на участок от одного до другого спецификатора, как в C++; элементы по умолчанию не устанавливаются в **private**, а доступны для классов из данного пакета. Объявление класса имеет вид:

```
[спецификаторы] class ИмяКласса [extends СуперКласс]
    [implements список_интерфейсов]{/*определение класса*/}
```

Спецификатор доступа к классу может быть **public** (класс доступен в данном пакета и вне пакета), **final** (класс не может иметь подклассов), **abstract** (класс может содержать абстрактные методы, объект такого класса создать нельзя). По умолчанию спецификатор устанавливается в дружественный (friendly). Такой класс доступен только в текущем пакете. Спецификатор friendly при объявлении вообще не используется и не является ключевым словом языка. Это слово используется, чтобы как-то определить значение по умолчанию.

Любой класс может наследовать свойства и методы суперкласса, указанного после ключевого слова **extends**, и включать множество интерфейсов, перечисленных через запятую после ключевого слова **implements**. Интерфейсы представляют собой абстрактные классы, содержащие только нереализованные методы.

```
// пример # 5 : простой пример Java Beans класса: User.java
package chapt01;

public class User {
      public int numericCode;//нарушение инкапсуляции
      private String password;

      public void setNumericCode(int value) {
            if(value > 0) numericCode = value;
            else numericCode = 1;
      }
      public int getNumericCode() {
            return numericCode;
      }
      public void setPassword(String pass) {
            if (pass != null) password = pass;
            else password = "11111";
      }
      public String getPassword() {
      //public String getPass() {//некорректно – неполное имя метода
            return password;
      }
}
```

Класс **User** содержит два поля **numericCode** и **password**, помеченные как **public** и **private**. Значение поля **password** можно изменять только при помощи методов, например **setPassword()**. Поле **numericCode** доступно непосредственно через объект класса **User**. Доступ к методам и **public**-полям данного класса осуществляется только после создания объекта данного класса.

```
/*пример # 6 : создание объекта, доступ к полям
 и методам объекта: UserView.java : Runner.java */
package chapt01;

class UserView {
      public static void welcome(User obj) {
  System.out.printf("Привет! Введен код: %d, пароль: %s",
        obj.getNumericCode(), obj.getPassword());
      }
}
public class Runner {
       public static void main(String[] args) {
             User user = new User();
             user.numericCode = 71;//некорректно - прямой доступ
             // user.password = null; // поле недоступно
             user.setPassword("pass"); //корректно
             UserView.welcome(user);
     }
}
```

Компиляция и выполнение данного кода приведут к выводу на консоль следующей информации:

```
Привет! Введен код: 71, пароль: pass
```

Объект класса создается за два шага. Сначала объявляется ссылка на объект класса. Затем с помощью оператора **new** создается экземпляр объекта, например:

```
User user; //объявление ссылки
user = new User(); //создание объекта
```

Однако эти два действия обычно объединяют в одно:

```
User user = new User();/*объявление ссылки и создание объекта*/
```

Оператор **new** вызывает конструктор, поэтому в круглых скобках могут стоять аргументы, передаваемые конструктору. Операция присваивания для объектов означает, что две ссылки будут указывать на один и тот же участок памяти.

Сравнение объектов

Операции сравнения ссылок на объекты не имеют особого смысла, так как при этом сравниваются адреса. Для сравнения значений объектов необходимо использовать соответствующие методы, например, **equals()**. Этот метод наследуется в каждый класс из суперкласса **Object**, который лежит в корне дерева иерархии всех классов и переопределяется в произвольном классе для определения эквивалентности содержимого двух объектов этого класса.

```
/* пример # 7 : сравнение строк и объектов : ComparingStrings.java */
package chapt01;

public class ComparingStrings {
     public static void main(String[] args) {
          String s1, s2;
          s1 = "Java";
          s2 = s1; // переменная ссылается на ту же строку
          System.out.println("сравнение ссылок "
             + (s1 == s2)); // результат true

          // создание нового объекта добавлением символа
          s1 += '2';
          // s1-="a"; //ошибка, вычитать строки нельзя
          // создание нового объекта копированием
          s2 = new String(s1);
          System.out.println("сравнение ссылок "
             + (s1 == s2)); // результат false

          System.out.println("сравнение значений "
             + s1.equals(s2)); // результат true
     }
}
```

Консоль

Взаимодействие с консолью с помощью потока **System.in** представляет собой один из простейших способов передачи информации в приложение. В следующем примере рассматривается ввод информации в виде символа из потока ввода, связанного с консолью, и последующего вывода на консоль символа и его числового кода.

```
// пример # 8 : чтение символа из потока System.in : DemoSystemIn.java
package chapt01;

public class ReadCharRunner {

  public static void main(String[] args) {
     int x;
     try {
        x = System.in.read();
        char c = (char)x;
        System.out.println("Код символа: " + c + " =" + x);
     } catch (java.io.IOException e) {
        e.printStackTrace();
     }
   }
}
```

Обработка исключительной ситуации **IOException**, которая возникает в операциях ввода/вывода и в любых других взаимодействиях с внешними устройствами, осуществляется в методе **main()** с помощью реализации блока **try-catch**.

Ввод блока информации осуществляется с помощью чтения строки из консоли. Далее строка может быть использована в исходном виде или преобразована к требуемому виду.

```
// пример # 9 : чтение строки из консоли : ReadCharRunner.java
package chapt01;
import java.io.*;//подключение пакета классов

public class ReadCharRunner {

      public static void main(String[] args) {
      /* байтовый поток ввода System.in передается конструктору потока
            чтения при создании объекта класса InputStreamReader  */
            InputStreamReader is =
                  new InputStreamReader(System.in);
      /* производится буферизация данных, исключающая необходимость
       обращения к источнику данных при выполнении операции чтения */
            BufferedReader bis = new BufferedReader(is);
            try {
                  System.out.println(
                  "Введите Ваше имя и нажмите <Enter>:");
      /* чтение строки из буфера; метод readLine() требует обработки
            возможной ошибки при вводе из консоли в блоке try */
                  String name = bis.readLine();
                  System.out.println("Привет, " + name);
            } catch (IOException e) {
                  System.err.print("ошибка ввода " + e);
            }
      }
}
```

В результате запуска приложения будет выведено, например, следующее:

```
Введите Ваше имя и нажмите <Enter>:
Остап
Привет, Остап
```

Позже будут рассмотрены более удобные способы извлечения информации из потока ввода, в качестве которого может фигурировать не только консоль, но и дисковый файл, сокетное соединение и пр.

Кроме того, в шестой версии языка существует возможность поддержать национальный шрифт с помощью метода **printf()** определенного для класса **Console**.

```
/* пример # 10 : использование метода printf() класса Console: PrintDeutsch.java */
public class PrintDeutsch {
      public static void main(String[] args) {
            String str = "über";
```

```
            System.out.println(str);
            Console con = System.console();
            con.printf("%s", str);
      }
}
```

В результате будет выведено:

□**ber**
über

Простой апплет

Одной из целей создания языка Java было создание апплетов – небольших программ, запускаемых Web-браузером. Поскольку апплеты должны быть безопасными, они ограничены в своих возможностях, хотя остаются мощным инструментом поддержки Web-программирования на стороне клиента.

```
// пример # 11 : простой апплет: FirstApplet.java
import java.awt.Graphics;
import java.util.Calendar;

public class FirstApplet extends javax.swing.JApplet {
      private Calendar calendar;

      public void init() {
            calendar = Calendar.getInstance();
            setSize(250,80);
      }
      public void paint(Graphics g) {
            g.drawString("Апплет запущен:", 20, 15);
            g.drawString(
                  calendar.getTime().toString(), 20, 35);
      }
}
```

Для вывода текущего времени и даты в этом примере был использован объект **Calendar** из пакета **java.util**. Метод **toString()** используется для преобразования информации, содержащейся в объекте, в строку для последующего вывода в апплет с помощью метода **drawString()**. Цифровые параметры этого метода обозначают горизонтальную и вертикальную координаты начала рисования строки, считая от левого верхнего угла апплета.

Апплету не нужен метод **main()** – код его запуска помещается в метод **init()** или **paint()**. Для запуска апплета нужно поместить ссылку на его класс в HTML-документ и просмотреть этот документ Web-браузером, поддерживающим Java. При этом можно обойтись очень простым фрагментом (тегом) **<applet>** в HTML-документе **view.html**:

```
<html><body>
<applet code= FirstApplet.class width=300 height=300>
</applet></body></html>
```

Сам файл **FirstApplet.class** при таком к нему обращении должен находиться в той же директории, что и HTML-документ. Исполнителем HTML-документа является браузер Microsoft Internet Explorer или какой-либо другой, поддерживающий Java.

Результат выполнения документа **view.html** изображен на рис.1.4.

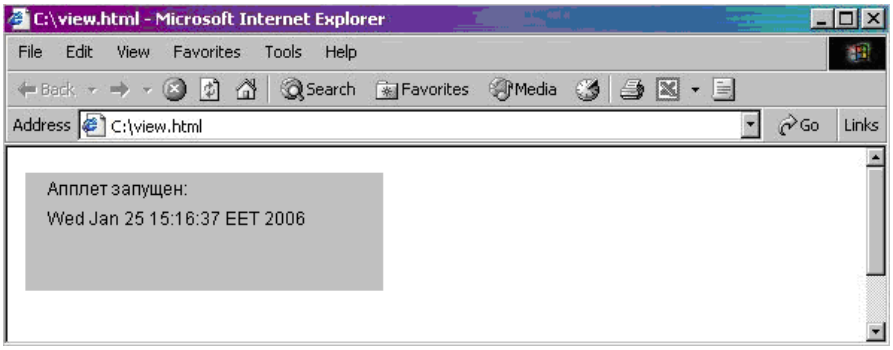

Рис. 1.4. Запуск и выполнение апплета

Для запуска апплетов можно использовать также входящую в JDK программу **appletviewer.exe**.

Задания к главе 1

Вариант А

1. Создать класс **Hello**, который будет приветствовать любого пользователя, используя командную строку.
2. Создать приложение, которое отображает в окне консоли аргументы командной строки метода **main()** в обратном порядке.
3. Создать приложение, выводящее **n** строк с переходом и без перехода на новую строку.
4. Создать приложение для ввода пароля из командной строки и сравнения его со строкой-образцом.
5. Создать программу ввода целых чисел как аргументов командной строки, подсчета их суммы (произведения) и вывода результата на консоль.
6. Создать приложение, выводящее фамилию разработчика, дату и время получения задания, а также дату и время сдачи задания. Для получения последней даты и времени использовать класс **Calendar** из пакета **java.util**.
7. Создать апплет на основе предыдущего задания и запустить его с помощью HTML-документа.

Вариант В

Ввести с консоли **n** целых чисел и поместить их в массив. На консоль вывести:

1. Четные и нечетные числа.
2. Наибольшее и наименьшее число.
3. Числа, которые делятся на 3 или на 9.
4. Числа, которые делятся на 5 и на 7.
5. Элементы, расположенные методом пузырька по убыванию модулей.

6. Все трехзначные числа, в десятичной записи которых нет одинаковых цифр.
7. Наибольший общий делитель и наименьшее общее кратное этих чисел.
8. Простые числа.
9. Отсортированные числа в порядке возрастания и убывания.
10. Числа в порядке убывания частоты встречаемости чисел.
11. "Счастливые" числа.
12. Числа Фибоначчи: f0 = f1 = 1, f (n) = f (n–1) + f (n–2).
13. Числа-палиндромы, значения которых в прямом и обратном порядке совпадают.
14. Элементы, которые равны полусумме соседних элементов.
15. Период десятичной дроби **p = m/n** для первых двух целых положительных чисел **n** и **m**, расположенных подряд.
16. Построить треугольник Паскаля для первого положительного числа.

Тестовые задания к главе 1

Вопрос 1.1.

Дан код:

```
class Quest1 {
     private static void main (String a) {
            System.out.println("Java 2");
     }  }
```

Какие исправления необходимо сделать, чтобы класс **Quest1** стал запускаемым приложением? (выберите 2 правильных варианта)

1) объявить класс **Quest1** как **public**;
2) заменить параметр метода **main()** на **String[] a**;
3) заменить модификатор доступа к методу **main()** на **public**;
4) убрать параметр из объявления метода **main()**.

Вопрос 1.2.

Выберите истинные утверждения о возможностях языка Java: (выберите 2)

1) возможно использование оператора **goto**;
2) возможно создание метода, не принадлежащего ни одному классу;
3) поддерживается множественное наследование классов;
4) запрещена перегрузка операторов;
5) поддерживается многопоточность.

Вопрос 1.3.

Дан код:

```
class Quest3 {
      public static void main(String s[ ]) {
            String args;
            System.out.print(args + s);
      }
}
```

Результатом компиляции кода будет:

1) ошибка компиляции: метод **main()** содержит неправильное имя параметра;

2) ошибка компиляции: переменная **args** используется до инициализации;
3) ошибка компиляции: несовпадение типов параметров при вызове метода **print()**;
4) компиляция без ошибок.

Вопрос 1.4.

Дан код:

```
public class Quest4 {
      public static void main(String[] args) {
            byte b[]=new byte[80];
            for (int i=0;i<b.length;i++)
                  b[i]=(byte)System.in.read();
            System.out.print("Ok");
} }
```

Результатом компиляции и запуска будет:

1) вывод: Ok;
2) ошибка компиляции, так как метод **read()** может порождать исключительную ситуацию типа **IOException**;
3) ошибка компиляции, так как длина массива **b** может не совпадать с длиной считываемых данных;
4) ошибка времени выполнения, так как массив уже проинициализирован.

Вопрос 1.5.

Дан код:

```
public class Quest5{
      public static void main(){
            System.out.print("A");  }
      public static void main(String args){
            System.out.print("Б");  }
      public static void main(String[] args){
            System.out.print("B");  }  }
```

Что будет выведено в результате компиляции и запуска:

1) ошибка компиляции;
2) Б;
3) ВБА;
4) В;
5) АБВ.