Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра экономической информатики

ЯЗЫКИ ПРОГРАММИРОВАНИЯ ДЛЯ РАЗРАБОТКИ СЕТЕВЫХ ПРИЛОЖЕНИЙ: ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

ЛАБОРАТОРНЫЙ ПРАКТИКУМ
для студентов специальностей
I-27 01 01 «Экономика и организация производства»,
I-26 02 03 «Маркетинг»
дневной формы обучения

В 2-х частях

Часть 1

Минск 2007

Библиотека БГУИР

УДК 681.3.061(075.8)
ББК 32.973.26-018.1 я 73
Я 41

Р е ц е н з е н т
зав. кафедрой интеллектуальных информационных технологий Белорусского государственного университета информатики и радиоэлектроники, д-р техн. наук, проф. В. В. Голенков

А в т о р ы:
Т. М. Унучек, В. Н. Комличенко,
Д. С. Марудов, Д. А. Сторожев, Е. Н. Унучек

Языки программирования для разработки сетевых приложний: Язык программирования JAVA : лаб. практикум для студ. спец. I-27 01 01 «Экономика и организация производства», I-26 02 03 «Маркетинг». В 2 ч. Ч. 1 / Т. М. Унучек [и др.]. – Минск : БГУИР, 2007. – 60 с.
Я 41
ISBN 978-985-488-079-2 (ч.1)

В лабораторном практикуме излагаются основы платформно-независимого объектно-ориентированного языка программирования Java 2, приведено описание основных его библиотек и классов.

Первая часть практикума состоит из четырех лабораторных работ, нацеленных на создание консольных приложений. Каждая лабораторная работа содержит основной теоретический материал по тематике работы, сопровождается большим числом примеров и законченных программ.

УДК 681.3.061(075.8)
ББК 32.973.26-018.1 я 73

ISBN 978-985-488-079-2 (ч.1)
ISBN 978-985-488-081-5

© УО «Белорусский государственный университет информатики и радиоэлектроники», 2007

Библиотека БГУИР

СОДЕРЖАНИЕ

Библиотека БГУИР

ВВЕДЕНИЕ

Java – объектно-ориентированная платформо-независимая многопоточная среда программирования. Первая версия *Java* была задумана Дж. Гослингом, П. Ноутоном, К. Вартом, Э. Франком, М. Шериданом. Спецификация была разработана компанией *Sun Microsystems* и выпущена в 1991 г. Сначала язык назывался *Oak* («Дуб») и был задуман как независимый от платформы язык с целью внедрения в электронные устройства различных производителей. Позднее проявилась другая его особенность – пригодность для использования в *WWW*. Непосредственно *Java* его стали называть в 1995 г.

Клиент-серверные возможности языка Java. Язык *Java* разработан для распределенной среды, предоставляет специальные классы и широкие возможности организации работы в сети. Он поддерживает протоколы *TCP/IP* и фактически снижает сложность доступа к сетевому ресурсу до уровня сложности доступа к файлу, а также обеспечивает ряд технологий для разработки сетевых приложений. Кроме того, *Java* позволяет создавать интерактивные сетевые программы за счет поддержки многопоточного программирования и эффективных решений распараллеливания и синхронизации процессов.

Java − интерпретируемый язык. Сначала программист компилирует исходный текст утилитой *javac* из набора инструментов *JDK* в *Java* в байт-коды. Байт-коды являются двоичными и не зависят от архитектуры компьютера (или от платформы). Байт-коды – незаконченное приложение, они не выполняются в операционной среде выполнения программ (*Java runtime environment*). Обычно в роли среды выступает браузер или *JVM* (*Java Virtual Machine* – виртуальная машина *Java*). Поскольку каждая среда выполнения создается для конкретной платформы, законченный программный продукт будет работать на этой платформе.

Java – объектно-ориентированный язык. Язык *Java* является объектно-ориентированным и, следовательно, относится к группе языков, рассматривающих данные как объекты и методы, используемые для этих объектов. *Java* − язык со строгой типизацией, что помогает выявлять возможные скрытые ошибки. В *Java* отсутствует множественное наследование и указатели, что с одной стороны, повышает читаемость, надежность программного кода, упрощает программирование и предохраняет от множества трудно диагностируемых ошибок, а с другой, несколько ограничивает в ряде ключевых возможностей. *Java* обеспечивает очень развитую объектно-ориентированную технологию поддержки обработки особых «исключительных» ситуаций. В *Java* можно создавать совершенно «абстрактные классы», называемые интерфейсами (*interface*). Интерфейсы

Библиотека БГУИР

позволяют описывать методы, разделяемые и реализуемые между несколькими классами, не учитывая при этом то, как другие классы используют данные методы.

Виртуальная Java-машина. Основой языка *Java* является виртуальная *Java*-машина. *JVM* – это виртуальный компьютер (модель компьютера), располагающийся только в оперативной памяти. *JVM* позволяет выполнять *Java*-приложения на множестве платформ, а не только в той системе, для которой скомпилирован код. Возможность компиляции *Java*-программ для *JVM* обеспечивает уникальность языка. Но для того чтобы приложения *Java* выполнялись на конкретной платформе, необходимо реализовать *JVM* для данной платформы (для каждой платформы своя *JVM*, что и обеспечивает мобильность *Java*).

Установка и основные утилиты Java. Программа установки *Java* (пакет *Java SDK*, известный также под названием *JDK*) представляет собой самораспаковывающийся архив, при разархивировании которого выдаются инструкции по установке. *Java SDK* обычно включает шесть подкаталогов:

- *bin* – содержит выполняемые модули и утилиты *JDK*;
- *demo* – включает множество апплетов, а также примеры текстов программ на *Java*;
- *docs* – содержит документацию по *Java*;
- *include* – включает заголовочные файлы C и C++, используемые для построения среды *Java*;
- *lib* – библиотеки и архивы, используемые в *Java*;
- *src* – исходные коды библиотек, созданных компанией *Sun*.

Наиболее важными в *Java* являются следующие утилиты:

- *javac* – компилятор *Java*, компилирует разработанный файл исходного текста программы в файл (байт-код) с таким же именем и расширением *.class*;
- *java* – утилита-интерпретатор запуска приложений *Java*;
- *appletviewer* – утилита просмотра апплетов;
- *jdb* – утилита тестирования приложений, написанных на *Java*;
- *javadoc* – утилита для создания документации.

В данном издании представлены четыре лабораторные работы, нацеленные на создание консольных приложений. Каждая лабораторная работа включает основной теоретический материал, сопровождается большим числом примеров и законченных программ. В конце каждой лабораторной работы содержатся задания для самостоятельного выполнения студентами. В приложении размещены полезные классы и методы, часто применяемые для написания приложений.

Библиотека БГУИР

ЛАБОРАТОРНАЯ РАБОТА №1

РАЗРАБОТКА КОНСОЛЬНЫХ ПРИЛОЖЕНИЙ

Цель: создание консольных приложений.

Понятие и определение класса

Класс – это шаблон для объекта. Объект – это экземпляр класса. Данные класса называются переменными экземпляра. Каждый объект содержит собственную копию этих переменных. Метод предназначен для обработки данных. Определение класса:

```
class classname  {
        type instance-variable1;
        type instance-variable2;
        // …
        type instance-variableN;
        type methodname1 (parameter-list)   {
        // тело метода
        }
        type methodname2 (parameter-list)   {
        // тело метода
        }
        //…
        type methodnameN(parameter-list)   {
        //  тело метода
        }
}
```

Рассмотрим пример класса, с помощью которого определяется минимальное и максимальное число.

Пример 1.1

```
class MinMax  {
        int x;
        int y;
        public int max(){
                if(x>y) return x;
                else return y;}
        public int min(){
```


Библиотека БГУИР

```
        if(x<y) return x;
        else return y;}
    }
```

Для определения экземпляра класса используется синтаксис:

ИмяКласса имяпеременной;
имяпеременной = new ИмяКласса (параметры инициализации);
или
ИмяКласса имяпеременной = new ИмяКласса (параметры инициализации);

Членами класса могут быть:
- поля;
- методы;
- внутренние типы (классов и интерфейсов);
- конструкторы;
- инициализаторы;
- статические инициализаторы.

Поля и методы могут иметь одинаковые имена.

Пример 1.2

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

}
}
```

Результат программы: 3, 3
3

Библиотека БГУИР

Модификаторы класса

public – класс доступен для использования, наследования вне зависимости от пакета или от каталога; *public*-классы должны храниться в файлах с именем *имя_класса.java*.

friendly – модификатор класса по умолчанию, если модификатор не определен явно для класса. Такой класс доступен только для объектов, находящихся в том же пакете. Вне пакета он выступает как *private*.

final – класс не может быть наследован, т.е. иметь подклассов.

abstract – класс, в котором объявлен хотя бы один абстрактный метод. Для таких классов нельзя создавать объекты. Такие классы используются для производных классов, а также для организации стандартизированных интерфейсов.

Пример 1.3

```
abstract class Myclass {
        abstract void actMet();
}
```

Объявление методов

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

- модификаторы (доступа в том числе);
- тип возвращаемого значения или ключевого слова *void*;
- имя метода;
- список аргументов в круглых скобках (аргументов может не быть);
- специальное throws-выражение.

Заголовок начинается с перечисления модификаторов. Для методов доступен любой из трех возможных модификаторов доступа. Также допускается использование доступа по умолчанию. Кроме этого, существует модификатор *final*, который говорит о том, что такой метод нельзя переопределять в наследниках. Можно считать, что все методы *final*-класса, а также все *private*-методы любого класса являются *final*.

Также поддерживается модификатор *native*. Метод, объявленный с таким модификатором, не имеет реализации на *Java*. Он должен быть написан на другом языке (*C/C++*, *Fortran* и т.д.) и добавлен в систему в виде загружаемой динамической библиотеки (например *DLL* для *Windows*). Наконец, существует еще один специальный модификатор *synchronized*, который будет рассмотрен в теме, описывающей потоки.

Библиотека БГУИР

Объявление объектов и операция *new*

Получение объектов класса − это двухшаговый процесс. Во-первых, нужно объявить переменную типа «класс». Она не определяет объект. Это просто переменная, которая может ссылаться на объект. Во-вторых, нужно получить актуальную, физическую копию объекта и назначить ее этой переменной. Это можно сделать с помощью операции *new*. Операция *new* распределяет динамически (т.е. во время выполнения) память для объекта и возвращает ссылку на нее. Данная ссылка является адресом ячейки памяти, выделенной объекту вышеуказанной операцией. Затем эта ссылка сохраняется в переменной. Таким образом, в *Java* все объекты класса должны быть распределены динамически.

Ниже приведен пример, в котором создается объект типа *Box* двумя способами.

Пример 1.4

```
// первый способ
Box mybox = new Box();
// второй способ
Box mybox;   // объявить ссылку на объект
mybox = new Box();     // распределить память для Box-объекта
```

Первый вариант комбинирует два шага, как это было вначале описано, второй пошагово создает объект типа *Box*.

Метод *main*

Итак, виртуальная машина реализуется приложением операционной системы и запускается по обычным правилам. Программа, написанная на *Java*, является набором классов. Требуется некая входная точка, с которой должно начинаться выполнение приложения. Такой входной точкой, по аналогии с языками *C/C++*, является метод *main()*. Для объявления метода *main()* используется следующий синтаксис:

```
public static void main(String[] args) {}
```

Модификатор *static* позволяет вызвать метод *main()*, не создавая объектов. Метод не возвращает никакого значения, хотя в *C* есть возможность указать код возврата из программы. В *Java* для этой цели есть метод *System.exit()*, который закрывает виртуальную машину и имеет аргумент типа *int*.

Библиотека БГУИР

Аргументом метода *main()* является массив строк. Он может заполняться дополнительными параметрами, которые указываются при вызове метода.

Пример 1.5

Пример демонстрирует вывод в консоль параметров, переданных в метод *main*. Для запуска примера необходимо передать сами параметры, например *java test.first.Test HelloWorld*

```
package test.first;
public class Test {
	public static void main(String[] args) {
		for (int i=0; i<args.length; i++) {
			System.out.print(args[i]+" ");
		}
		System.out.println();
	}
}
```

Результат работы программы: HelloWorld

Если вышеприведенный модуль компиляции сохранен в файле *Test.java*, который лежит в директории *test\first*, то вызов компилятора записывается следующим образом:

javac test\first\Test.java

А вызов виртуальной машины:

java test.first.Test

Конструкторы

Конструктор – это тот же метод класса, обладающий некоторыми особенностями:

- имеет такое же название, как и класс;
- вызывается сам;
- не возвращает никакого значения.

Ниже продемонстрирован пример перегрузки конструктора.

Пример 1.6

```
// перегрузка конструктора
 class worker
 {	private int Age;
	public String Name;
	public worker()		// конструктор без параметров
```


Библиотека БГУИР

```
        { Age = 20;}
    public worker(int newAge, String newName)
        { Age = newAge;
          Name = newName;        }
    public int getAge()
        {return Age;}
};
class worker_pub
{  public static void main(String args[])
     { worker wrk1 = new worker( );
       worker wrk2 = new worker(40, "Petrov");
       System.out.println(wrk1. getAge()+wrk1.Name);
       System.out.println(wrk2. getAge()+wrk2.Name);      }  }
```

Результат выполнения программы:

20null

40Petrov

Если конструктор не определяется явно, то *Java* создает его сам, без параметров.

Использование атрибутов доступа

Рассмотрим классы в том же пакете. Внутри данного пакета любой класс имеет прямой доступ к имени любого другого класса, например для объявления переменных или типов параметров методов. Но переменные и методы, которые являются членами этого другого класса, не обязательно доступны. Их доступность управляется атрибутами доступа. Существуют четыре возможности при определении атрибута доступа для члена класса (включая неопределение ничего), и все они дают различные результаты. В табл. 1.1 показано, как атрибуты доступа, заданные для члена класса, определяют части среды *Java*, откуда к ним можно получить доступ.

Таблица 1.1

Атрибуты доступа к элементам класса

Атрибут	Разрешенный доступ
Отсутствие атрибута доступа	Из любого класса в том же пакете
public	Из любого класса откуда угодно
private	Никакого доступа вне класса
protected	Из любого класса в том же пакете и из любого подкласса где угодно

Библиотека БГУИР

На рис. 1.1 и 1.2 приведена графическая интерпретация табл. 1.1.

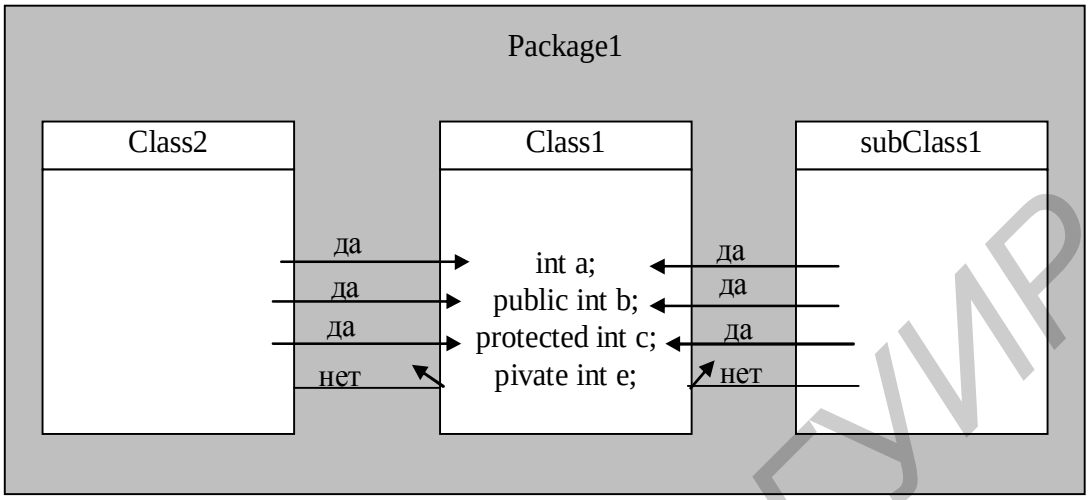

Рис. 1.1. Атрибуты доступа в рамках пакета

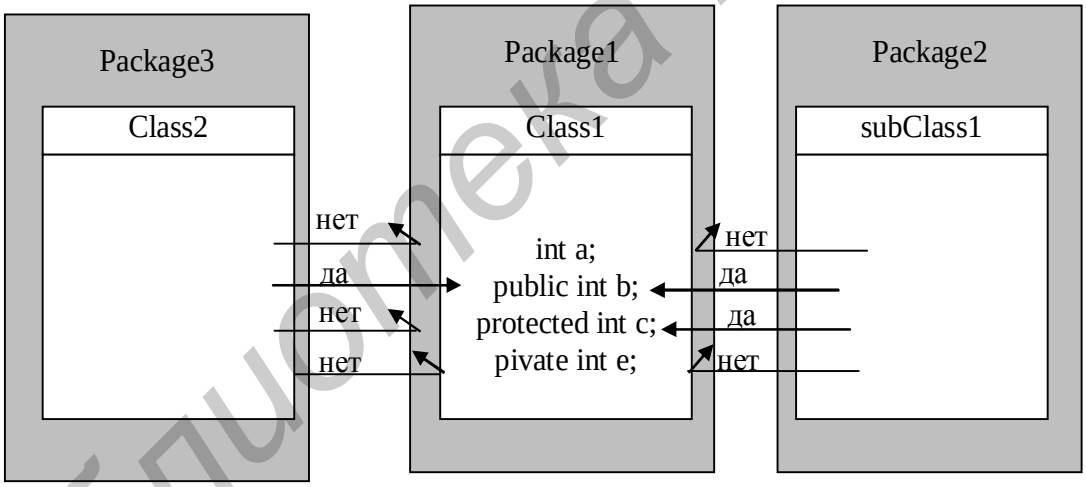

Рис. 1.2. Атрибуты доступа для нескольких пакетов

Статические переменные и методы

Объявляются при помощи ключевого слова *static*.

Переменные и методы, объявленные как *static*, являются разделяемыми свойствами всех классов, а не отдельных экземпляров объекта. То есть одна копия переменной разделяется всеми экземплярами объектов класса, поэтому к ним можно обращаться без ссылки на конкретный объект и до или без создания каких-либо объектов.

Методы, объявленные как *static*, могут работать только с переменными и методами, объявленными как *static*. Такие методы можно вызвать даже если объект не создан. При этом нестатические методы могут работать как с

Библиотека БГУИР

обычными, так и со статическими переменными. В статических методах нельзя ссылаться на *this* и на *super()* (конструктор базового класса).

Метод *main()* объявляется как *static*, потому что должен вызваться прежде, чем будут созданы какие-либо объекты.

Статические переменные создаются один раз, как глобальные, то есть нет дублирования, как при нестатических переменных. Статические переменные можно вызывать, не создав объекта класса.

Пример 1.7

```
class Myclass
{
static boolean ab_switch = true;
   static int a = 1;
   static int b;
   public static void f1(int x)
   {
   ab_switch =! ab_switch;
   System.out.println("x= "+x);
   System.out.println("a= "+a);
   }
   static{ //статический блок инициализации
        b=a*4;}
        public static void main(String args[])
        { f1(25);
        }
 }
```

ab_switch – изменяемая переменная, и вызов метода *f1* из каждого экземпляра объекта или любого метода меняет состояние *switch* для всех экземпляров объектов, в которых эта переменная определена.

Последовательность выполнения операторов:

a=1;

b=4;

Вызов метода *main( )* и, следовательно, метода *f1(25)*.

Результат выполнения программы:

```
x= 25
a= 1
```

Библиотека БГУИР

Статические переменные объявляют, когда нужна общая переменная на все экземпляры класса. Это может быть, например, счетчик количества экземпляров класса.

Пример 1.8

```
class someclass{
        static public int number;        //счетчик
        public  someclass(){
                number++;
        }
}
class Test{
public static void main(String args[]){
                someclass z1 = new someclass();
                someclass z2 = new someclass();
                System.out.println(someclass. number);
}
}
```

Результат выполнения программы: 2

main() – главная функция, с неё начинается выполнение любой программы. Она вызывается кодом, который находится вне программы (поэтому *main( )* объявляется как *public*).

Потоки

Java-программы выполняют ввод/вывод через потоки. Поток является абстракцией, которая или производит, или потребляет информацию. Поток связывается с физическим устройством с помощью системы ввода/вывода *Java* (*Java I/O system*). Все потоки ведут себя одинаковым образом, хотя фактические физические устройства, с которыми они связаны, могут сильно различаться. Таким образом, одни и те же классы и методы ввода/вывода можно применять к устройствам любого типа. Это означает, что поток ввода может извлекать много различных видов входных данных: из дискового файла, с клавиатуры или сетевого разъема. Аналогично, поток вывода может обратиться к консоли, дисковому файлу или сетевому соединению (сокету).

Благодаря потокам, ваша программа выполняет ввод/вывод, не понимая различий между клавиатурой и сетью. *Java* реализует потоки с помощью иерархии классов, определенных в пакете *java.io*.

Библиотека БГУИР

Байтовые и символьные потоки

Java 2 определяет два типа потоков: байтовый и символьный. Байтовые потоки предоставляют удобные средства для обработки ввода и вывода байт. Байтовые потоки используются, например, при чтении или записи данных в двоичном коде. Символьные потоки предоставляют удобные средства для обработки ввода и вывода символов. Они используют *Unicode* и поэтому могут быть интернационализированы. Кроме того, в некоторых случаях символьные потоки более эффективны, чем байтовые. Первоначальная версия *Java* (*Java 1.0*) не включала символьные потоки, и, таким образом, весь ввод/вывод был байтовым. Символьные потоки были добавлены в *Java 1.1*, а некоторые байтовые классы и методы были исключены.

Классы байтовых потоков

Байтовые потоки определяются в двух иерархиях классов. Наверху этой иерархии – два абстрактных класса: *InputStream* и *OutputStream*. Каждый из этих абстрактных классов имеет несколько конкретных подклассов (табл. 1.2), которые обрабатывают различия между разными устройствами, такими, как дисковые файлы, сетевые соединения и даже буферы памяти.

Абстрактные классы *InputStream* и *OutputStream* определяют несколько ключевых методов, которые реализуются другими поточными классами. Два наиболее важных – *read()* и *write()*, которые соответственно читают и записывают байты данных. Оба метода объявлены как абстрактные внутри классов *InputStream* и *OutputStream* и переопределяются производными поточными классами.

Таблица 1.2

Классы байтовых потоков

Поточный класс	Назначение
InputStream OutputStream	Абстрактные классы, которые описывают поточный ввод и вывод
BufferedInputStream BufferedOutputStream	Буферизированные потоки ввода и вывода
ByteArrayInputStream ByteArrayOutputStream	Поток ввода, который читает из байт-массива Поток вывода, который записывает в байт-массив
FileInputStream FileOutputStream	Поток ввода, который читает из файла Поток вывода, который записывает в файл
RandomAccessFile	Поддерживает ввод/вывод файла произвольного доступа

Библиотека БГУИР

Классы символьных потоков

Символьные потоки определены в двух иерархиях классов. Наверху этой иерархии два абстрактных класса: *Reader* и *Writer*. Они обрабатывают потоки символов *Unicode*. В *Java* существуют несколько конкретных подклассов каждого из них. Классы *Reader* и *Writer* – наследники *InputStream* и *OutputStream*. Если с их помощью записывать или считывать текст, то сначала необходимо сопоставить каждому символу его числовой код. Такое соответствие называется кодировкой. Классы символьных потоков показаны в табл. 1.3.

Абстрактные классы *Reader* и *Writer* определяют несколько ключевых методов, которые реализуются другими поточными классами. Два самых важных метода – *read()* и *write()*, которые читают и записывают символы данных соответственно. Они переопределяются производными поточными классами.

Таблица 1.3

Классы символьных потоков

Символьный класс	Назначение
Reader Writer	Абстрактные классы символьного потока ввода и вывода
BufferedReader BufferedWriter	Буферизированные символьные потоки ввода и вывода
FileReader FileWriter	Поток ввода, который читает поток символов из файла Выходной поток, который записывает символы в файл
StringReader StringWriter	Поток ввода, который читает из строки Поток вывода, который записывает в строку
PrintWriter	Поток вывода, который поддерживает методы print() и println()

Известно, что *Java* использует кодировку *Unicode*, в которой символы представляются двухбайтным кодом. Байтовые потоки зачастую работают с текстом упрощенно – они просто отбрасывают старший байт каждого символа. В реальных же приложениях они могут использовать различные кодировки (даже для русского языка их существует несколько). Поэтому в версии *Java 1.1* появился дополнительный набор классов, основывающийся на типах *Reader* и *Writer*.

Библиотека БГУИР

Эта иерархия очень схожа с аналогичной для байтовых потоков *InputStream* и *OutputStream*. Главное отличие между ними – *Reader* и *Writer* работают с потоком символов (char). Только чтение массива символов в Reader описывается методом *read(char[])*, а запись в *Writer – write(char[])*.

В табл. 1.4 приведены соответствия классов для байтовых и символьных потоков.

Таблица 1.4

Соответствие классов для байтовых и символьных потоков

Байтовый поток	Символьный поток
InputStream	Reader
OutputStream	Writer
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
Нет аналога	InputStreamReader
Нет аналога	OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
PrintStream	PrintWriter
DataInputStream	Нет аналога
DataOutputStream	Нет аналога
ObjectInputStream	Нет аналога
ObjectOutputStream	Нет аналога
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter
StringBufferInputStream	StringReader
Нет аналога	StringWriter
LineNumberInputStream	LineNumberReader
PushBackInputStream	PushBackReader
SequenceInputStream	Нет аналога

Как видно из табл. 1.4, различия крайне незначительны и предсказуемы.

Например, отсутствует преобразование в символьное представление примитивных типов *Java* и объектов (*DataInput/Output*, *ObjectInput/Output*). Добавлены классы-мосты, преобразующие символьные потоки в байтовые:

Библиотека БГУИР

InputStreamReader и *OutputStreamWriter*. Именно на их основе реализованы *FileReader* и *FileWriter*. Метод *available()* класса *InputStream* в классе *Reader* отсутствует, он заменен методом *ready()*, возвращающим булевое значение, – готов ли поток к считыванию (то есть будет ли считывание произведено без блокирования).

В остальном же использование символьных потоков идентично работе с байтовыми потоками.

Классы-мосты *InputStreamReader* и *OutputStreamWriter* при преобразовании символов также используют некоторую кодировку. Ее можно задать, передав в конструктор в качестве аргумента ее название. Если оно не будет соответствовать никакой из известных кодировок, будет брошено исключение *UnsupportedEncodingException*. Вот некоторые из корректных значений этого аргумента (чувствительного к регистру!) для распространенных кодировок: «*Cp1251*», «*UTF-8*», «*8859_1*» и т.д.

В следующем примере приведена программа, которая демонстрирует *read()*, читая символы с консоли, пока пользователь не напечатает «q»:

Пример 1.9

```
import java.io.*;
class BRRead {
public static void main(String args [ ]) throws IOException
        {
                char c;
                BufferedReader br = new
                BufferedReader(new InputStreamReader(System.in));
                System.out.println("Enter symbols or 'q' to exit");
                // чтение символов
                do {
                        c= (char)br.read();
                        System.out.println(c);
                        }
                while(c != 'q');
        }
}
```

Результат выполнения программы:

```
Enter symbols or 'q' to exit
123abcq
1
```

Библиотека БГУИР

```
2
3
a
b
c
q
```

Следующая программа демонстрирует *BufferedReader* и метод *readLine()*. Она читает и отображает строки текста, пока не будет введено слово «stop».

Пример 1.10

```
import java.io.*;
class BRReadLines {
public static void main(String args[])throws IOException
{
        // создать BufferedReader, используя System.in
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        String str;
        System.out.println("Enter text");
                        System.out.println("Enter 'stop' to exit");
        do {
                str = br.readLine();
                System.out.println(str);
                } while(!str.equals("stop"));
        }
}
```

Программный код для записи символьных данных в файл приведен в следующем примере.

Пример 1.11

```
String fileName = "d:\\file.txt";
//строка, которая будет записана в файл
String data = "Some data to be written and read.\n";
try{
  FileWriter fw = new FileWriter(fileName);
  BufferedWriter bw = new BufferedWriter(fw);
```

Библиотека БГУИР

```
    System.out.println("Write some data to file: " + fileName);
    // несколько раз записать строку
    for(int i=(int)(Math.random()*10);--i>=0;)
      bw.write(data);
    bw.close();
    // считываем результат
    FileReader fr = new FileReader(fileName);
    BufferedReader br = new BufferedReader(fr);
    String s = null;
    int count = 0;
    System.out.println("Read data from file: " + fileName);
    // считывать данные, отображая на экран
    while((s=br.readLine())!=null)
      System.out.println("row " + ++count + " read:" + s);
    br.close();
  } catch(Exception e) {e.printStackTrace();}
```

Следующий пример демонстрирует извлечение из строки последовательности подстрок, которые разделены простыми пробелами.

Пример 1.12

```
public class ExtractSubstring
{
public static void main(String[] args)
{
String text = "To be or not to be"; //строка для расчленения
int count = 0;                  //число подстрок
   char separator = ' ';              //разделитель подстрок
  // определить число подстрок
  int index = 0;
  do  {
     ++count;                 //увеличить счетчик подстрок
    ++index;                 //перейти за последнюю позицию
     index = text.indexOf(separator, index);
   }
   while (index != -1);
   //поместить подстроку в массив
     String[] subStr = new String[count];   //выделить для подстрок
     index = 0;                      //индекс начала подстроки
```

Библиотека БГУИР

```
    int endIndex = 0;              //индекс окончания подстроки
   for(int i = 0; i < count; i++)
   {
   endIndex = text.indexOf(separator,index); //найти
                                                //следующий разделитель
    if(endIndex == -1)             //если он не найден
    subStr[i] = text.substring(index); //извлечь до конца
    else                           //иначе
    subStr[i] = text.substring(index, endIndex);
                                //до конечного индекса
   index = endIndex + 1;           //задать начало для
                                //следующего цикла
    }
    //вывести подстроки
    for(int i = 0; i < subStr.length; i++)
      System.out.println(subStr[i]);
   }
}
```

Результат выпонения программы:

To
be
or
not
to
be

В следующем примере создается последовательность точек и линии, соединяющие каждую пару последовательных точек, вычисляется общая длина линии. Пример включает три класса: *TryPackage*, *Point*, *Line*.

Пример 1.13

Код класса TryPackage

```
import Geometry.*;    // импортировать классы Point и Line
public class TryPackage
{
  public static void main(String[] args)
  {
    double[][] coords = { {1.0, 0.0}, {6.0, 0.0},
```

Библиотека БГУИР

```
                    {6.0, 10.0},{10.0,10.0},
                    {10.0, -14.0}, {8.0, -14.0}};

   //создать массив точек и заполнить его объектами Point
    Point[] points = new Point[coords.length];
    for(int i = 0; i < coords.length; i++)
      points[i] = new Point(coords[i][0],coords[i][1]);
    //создать массив линий и заполнить его с помощью пар
    //объектов Point
    Line[] lines = new Line[points.length - 1];
    double totalLength = 0.0; //сохранить общую длину линии
    for(int i = 0; i < points.length - 1; i++)
    {
      //создать Line
      lines[i] = new Line(points[i], points[i+1]);
      //добавить ее длину
      totalLength += lines[i].length();
      System.out.println("\nLine "+(i+1)+' ' +lines[i] +
                    "  Length is " + lines[i].length());
    }
   //вывести общую длину
    System.out.println("\n\nTotal line length = " + totalLength);
  }
}
```

Код класса Point

```
package Geometry;
public class Point
{
//создать точку (Point) по ее координатам
  public Point(double xVal, double yVal)
  {
    x = xVal;
    y = yVal;
  }
  //создать точку (Point) из существующего объекта (Point)
  public Point(final Point aPoint)
  {
    x = aPoint.x;
```

Библиотека БГУИР

```
    y = aPoint.y;
  }
    //переместить точку
  public void move(double xDelta, double yDelta)
  {
  //значения параметров являются приращениями для текущих координат
    x += xDelta;
    y += yDelta;
  }
  //вычислить расстояние до другой точки public double distance (final
    Point aPoint)
  {
    return Math.sqrt(
          (x - aPoint.x)*(x - aPoint.x) +
          (y - aPoint.y)*(y - aPoint.y) );
  }
  //преобразовать точку (Point) в строку
  public String toString()
  {
    return Double.toString(x) + ", " + y;    //Как "x, y"
  }
  //извлечь координату x
  public double getX()
  { return x;  }
  //извлечь координату y
  public double getY()
  { return y;  }
 //задать координату x
  public void setX(double inputX)
  { x = inputX;  }
  //задать координату y
  public void setY(double inputY)
  { y = inputY;  }
  //координаты точки
  private double x;
  private double y;
}
```

Код класса Line

Библиотека БГУИР

```
package Geometry;
public class Line
{
  //создать прямую (Line) по двум точкам
  public Line(final Point start, final Point end)
  {
    this.start = new Point(start);
    this.end = new Point(end);
  }
  //создать прямую (Line) из двух пар координат
  public Line(double xStart, double yStart, double xEnd, double yEnd)
  {
    start = new Point(xStart, yStart); //создать начальную точку
    end = new Point(xEnd, yEnd);       //создать конечную точку
  }
  //вычислить длину прямой
  public double length()
  {
    return start.distance(end); //использовать метод из класса Point
  }
  //вернуть точку как пересечение двух прямых
  public Point intersects(final Line line1)
  {
    Point localPoint = new Point(0, 0);
    double num =
      (this.end.getY() - this.start.getY())*(this.start.getX() - line1.start.getX()) -
     (this.end.getX() - this.start.getX())*(this.start.getY() - line1.start.getY());
    double denom =
      (this.end.getY() - this.start.getY())*(line1.end.getX() - line1.start.getX()) -
      (this.end.getX() - this.start.getX())*(line1.end.getY() - line1.start.getY());
        localPoint.setX(line1.start.getX() + (line1.end.getX() -
          line1.start.getX())*num/denom);
    localPoint.setY(line1.start.getY() + (line1.end.getY() -
line1.start.getY())*num/denom);
    return localPoint;
  }
  //преобразовать прямую (Line) в строку
  public String toString()   {
    return "(" + start+ "):(" + end + ")"; //Как "(start):(end)"
```

Библиотека БГУИР

```
  }                          //т.е."(x1, y1):(x2, y2)"
  // Data members
  Point start;               //начальная точка прямой
  Point end;                 //конечная точка прямой
}
```

Откомпилировав файлы и запустив программу, получим следующий результат работы программы:

```
Line 1 (1.0, 0.0) : (6.0, 0.0)   Length is 5.0
Line 2 (6.0, 0.0) : (6.0, 10.0)   Length is 10.0
Line 3 (6.0, 10.0) : (10.0, 10.0)   Length is 4.0
Line 4 (10.0, 10.0) : (10.0, -14.0)   Length is 24.0
Line 5 (10.0, -14.0) : (8.0, -14.0)   Length is 2.0
Total line length = 45.0
```

Задания для самостоятельного выполнения

1. Создать массив переменных *String* и инициализировать массив названиями месяцев от января до декабря. Создать массив, содержащий 12 случайных десятичных значений между 0.0 и 100.0. Вывести название каждого месяца вместе с соответствующим десятичным значением. Вычислить и вывести среднее значение 12 значений.

2. Написать программу, которая задает переменную *String*, содержащую параграф текста на выбор. Извлечь слова из текста и отсортировать их в алфавитном порядке. Вывести отсортированный список слов.

3. Создать массив из десяти переменных *String*, каждая из которых содержит произвольную строку – месяц/день/год, например 01/10/06. Проанализировать каждый элемент в массиве и вывести представление даты в форме 10 января 2006.

4. Написать программу для создания случайной последовательности прописных букв, которая не включает гласные буквы.

5. Написать программу для создания случайной последовательности строчных букв, которая не включает гласные буквы.

6. Создать объект типа *String* и проинициализировать его текстовой строкой. Определить количество гласных, пробелов и общее количество букв.

7. Создать массив объектов типа *String* и проинициализировать его следующими текстовыми строками: «To be or not to be that is the question», «I am the student of economical department», «My name is John», «Hello world». Воспользоваться методом *indexOf()*, чтобы определить в массиве подстроки

Библиотека БГУИР

«be», «is»,«am»,«department», «hello». Вывести номер строки и для нее номер указанного элемента.

8. Создать массив объектов типа String и проинициализировать его следующими строками: «To;be:or*not;to:be*that;is:the*question», «I;am:the*student;of:economical*department», «My;name:is*John», «Hello;world». Использовать метод *indexOf()* совместно с методом *substring()* для извлечения из исходного массива строк последовательности подстрок, которые разделены символами «;», «:»,«*».

9. Написать программу, с помощью которой создается последовательность точек и линия, соединяющая каждую пару последовательных точек. Затем вычисляется общая длина линии. Найти на этой линии отрезок минимальной длины.

10. Написать программу для создания прямоугольного массива, содержащего таблицу умножения от 1×1 до 12×12. Вывести таблицу как 13 столбцов с числовыми значениями, выровненными справа в столбцах (рис. 1.3). Первая строка вывода – это заголовки столбцов без заголовка для первого столбца, затем числа от 1 до 12 для остальных столбцов. Первый элемент в каждой из последующих строк является заголовком строки, изменяющимся от 1 до 12.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	1	2	3	4	5	6	7	8	9	10	11	12
2	2	4	6	8	10	12	14	16	18	20	22	24
3	3	6	9	12	15	18	21	24	27	30	33	36
4	4	8	12	16	20	24	28	32	36	40	44	48
5	5	10	15	20	25	30	35	40	45	50	55	60
6	6	12	18	24	30	36	42	48	54	60	66	72
7	7	14	21	28	35	42	49	56	63	70	77	84
8	8	16	24	32	40	48	56	64	72	80	88	96
9	9	18	27	36	45	54	63	72	81	90	99	108
10	10	20	30	40	50	60	70	80	90	100	110	120
11	11	22	33	44	55	66	77	88	99	110	121	132
12	12	24	36	48	60	72	84	96	108	120	132	144

Рис. 1.3. Результат вывода программы

11. Написать программу, с помощью которой создается последовательность точек и линия, соединяющая каждую пару

Библиотека БГУИР

последовательных точек. Затем вычисляется средняя длина линии. Найти на этой линии отрезок максимальной длины.

12. Диаметр Солнца равен приблизительно 865 000 милям, а диаметр Земли – 7600 милям. Вычислить с помощью методов класса *Math*:

– объем Земли в кубических милях;

– объем Солнца в кубических милях;

– отношение объема Солнца к объему Земли.

Затем вывести эти значения. Считать, что Земля и Солнце являются шарами. Объем шара задается формулой $4\pi r^3/3$, где r – радиус шара.

13. Написать программу, которая по трем точкам определит вид треугольника: прямоугольный, равнобедренный, равносторонний или разносторонний.

14. Написать программу, которая в матрице произвольного порядка определит индекс строки с минимальным элементом и индекс столбца с максимальным элементом этой матрицы.

15. Написать программу, которая в матрице произвольного порядка определит отношение среднего значения элементов, расположенных на главной диагонали, к среднему значению элементов, расположенных на побочной диагонали этой матрицы.

Библиотека БГУИР

ЛАБОРАТОРНАЯ РАБОТА №2

РАБОТА С ФАЙЛАМИ

Цель: разработка консольных приложений с возможностью записи и чтения в/из файла.

Классы *FileInputStream* и *FileOutputStream*

Класс *FileInputStream* используется для чтения данных из файла. Конструктор такого класса в качестве параметра принимает название файла, из которого будет производиться считывание. При указании строки имени файла нужно учитывать, что она будет напрямую передана операционной системе, поэтому формат имени файла и пути к нему может различаться на разных платформах. Если при вызове этого конструктора передать строку, указывающую на несуществующий файл или каталог, то будет брошено *java.io.FileNotFoundException*. Если же объект успешно создан, то при вызове его методом *read()* возвращаемые значения будут считываться из указанного файла.

Для записи байт в файл используется класс *FileOutputStream*. При создании объектов этого класса, то есть при вызовах его конструкторов, кроме имени файла также можно указать, будут ли данные дописываться в конец файла либо файл будет перезаписан. Если указанный файл не существует, то сразу после создания *FileOutputStream* он будет создан. При вызовах методов *write()* передаваемые значения будут записываться в этот файл. По окончании работы необходимо вызвать метод *close()*, чтобы сообщить системе, что работа по записи файла закончена.

Пример 2.1

В примере реализован класс, который записывает информацию в файл *d:\\test.txt* .

```
import java.io.*;
class FileWriter{
  public static void main(String args[]){
  byte[] bytesToWrite = {1, 2, 3};
  byte[] bytesReaded = new byte[10];
  String fileName = "d:\\test.txt";
  try {
       // создать выходной поток
  FileOutputStream outFile = new FileOutputStream(fileName);
  System.out.println("File is open to write");
```


Библиотека БГУИР

```
  // записать массив
  outFile.write(bytesToWrite);
  System.out.println("Written: " + bytesToWrite.length + " byte");
  // по окончании использования должен быть закрыт
  outFile.close();
  System.out.println("Output stream is closed");
  // создать входной поток
  FileInputStream inFile = new FileInputStream(fileName);
  System.out.println("File is open to read");
  // узнать, сколько байт готово к считыванию
  int bytesAvailable = inFile.available();
  System.out.println("Ready to read: " + bytesAvailable + " byte");
  // считать в массив
  int count = inFile.read(bytesReaded,0,bytesAvailable);
  System.out.println("Read: " + count + " byte");
  for (int i=0;i<count;i++)
  System.out.print(bytesReaded[i]+",");
  System.out.println();
  inFile.close();
  System.out.println("Input stream is closed");
  } catch (FileNotFoundException e) {
  System.out.println("Its impossible to write to file: " + fileName);
  } catch (IOException e) {
  System.out.println("Input/output error: " + e.toString());
  }
  }
}
```

Результат выполнения программы:

```
File is open to write
Written 3 byte
Output stream is closed
File is open to read
Ready to read 3 byte
Read 3 byte
1,2,3,
Input stream is closed
```

Библиотека БГУИР

При работе с *FileInputStream* метод *available()* практически наверняка вернет длину файла, то есть число байт, сколько вообще из него можно считать. Но нежелательно использовать его при написании программ, которые должны устойчиво работать на различных платформах: метод *available()* возвращает число байт, которое может быть на данный момент считано без блокирования. Тот факт, что, скорее всего, это число и будет длиной файла, является лишь частным случаем работы на некоторых платформах.

В приведенном примере для наглядности закрытие потоков производилось сразу же после окончания их использования в основном блоке. Однако лучше закрывать потоки в *finally*-блоке.

Пример 2.2

```
...
} finally {
try{inFile.close();}catch(IOException e){};
}
```

Такой подход гарантирует, что поток будет закрыт и будут освобождены все связанные с ним системные ресурсы.

BufferedInputStream* и *BufferedOutputStream

На практике при считывании с внешних устройств ввод данных почти всегда необходимо буферизировать. Для буферизации данных служат классы *BufferedInputStream* и *BufferedOutputStream*.

BufferedInputStream содержит массив байт, который служит буфером для считываемых данных. То есть когда байты из потока считываются либо пропускаются (метод *skip()*), сначала заполняется буферный массив, причем из надстраиваемого потока загружается сразу много байт, чтобы не требовалось обращаться к нему при каждой операции *read* или *skip*. Также класс *BufferedInputStream* добавляет поддержку методов *mark()* и *reset()*. Эти методы определены еще в классе *InputStream*, но там их реализация по умолчанию бросает исключение *IOException*. Метод *mark()* запоминает точку во входном потоке, а вызов метода *reset()* приводит к тому, что все байты, полученные после последнего вызова *mark()*, будут считываться повторно, прежде чем новые байты начнут поступать из надстроенного входного потока.

BufferedOutputStream предоставляет возможность производить многократную запись небольших блоков данных без обращения к устройству вывода при записи каждого из них. Сначала данные записываются во внутренний буфер. Непосредственное обращение к устройству вывода и

Библиотека БГУИР

соответственно запись в него произойдут, когда буфер заполнится. Инициировать передачу содержимого буфера на устройство вывода можно и явным образом, вызвав метод *flush()*. Также буфер освобождается перед закрытием потока. При этом будет закрыт и надстраиваемый поток (так же поступает *BufferedInputStream*).

Следующий пример наглядно демонстрирует повышение скорости считывания данных из файла с использованием буфера.

Пример 2.3

```
import java.util.*;
import java.io.*;
class DemoBufferedIOStream
{public static void main(String args[]){
  try {
    String fileName = "d:\\file1";
    InputStream inStream = null;
    OutputStream outStream = null;
     //Записать в файл некоторое количество байт
    long timeStart = System.currentTimeMillis();
    outStream = new FileOutputStream(fileName);
    outStream = new BufferedOutputStream(outStream);
    for(int i=1000000; --i>=0;) {
      outStream.write(i);
    }
    long time = System.currentTimeMillis() - timeStart;
    System.out.println("Writing time: " + time + " millisec");
    outStream.close();
     // Определить время считывания без буферизации
    timeStart = System.currentTimeMillis();
    inStream = new FileInputStream(fileName);
    while(inStream.read()!=-1){
    }
    time = System.currentTimeMillis() - timeStart;
    inStream.close();
    System.out.println("Direct read time: " + (time) + " millisec");
    // Теперь применим буферизацию
    timeStart = System.currentTimeMillis();
    inStream = new FileInputStream(fileName);
    inStream = new BufferedInputStream(inStream);
```

Библиотека БГУИР

```
        while(inStream.read()!=-1){
        }
        time = System.currentTimeMillis() - timeStart;
        inStream.close();
        System.out.println("Buffered read time: " + (time) + " millisec");
      } catch (IOException e) {
        System.out.println("IOException: " + e.toString());
        e.printStackTrace();
      }
      }
}
```

Результат выполнения программы:

```
Writing time: 359 millisec
Direct read time: 6546 millisec
Buffered read time: 250 millisec
```

В данном случае не производилось никаких дополнительных вычислений, занимающих процессорное время, только запись и считывание из файла. При этом считывание с использованием буфера заняло в 10 (!) раз меньше времени, чем аналогичное без буферизации. Для более быстрого выполнения программы запись в файл производилась с буферизацией, однако ее влияние на скорость записи нетрудно проверить, убрав из программы строку, создающую *BufferedOutputStream*.

DataInputStream* и *DataOutputStream

До сих пор речь шла только о считывании и записи в поток данных в виде *byte*. Для работы с другими примитивными типами данных *Java* определены интерфейсы *DataInput* и *DataOutput* и их реализации – классы-фильтры *DataInputStream* и *DataOutputStream*.

Интерфейсы *DataInput* и *DataOutput* определяют, а классы *DataInputStream* и *DataOutputStream* соответственно реализуют методы считывания и записи значений всех примитивных типов. При этом происходит конвертация этих данных в набор *byte* и обратно. Чтение необходимо организовать так, чтобы данные запрашивались в виде тех же типов, в той же последовательности, как и производилась запись. Если записать, например, *int* и *long*, а потом считывать их как *short*, чтение будет выполнено корректно, без исключительных ситуаций, но числа будут получены совсем другие.

Библиотека БГУИР

Пример 2.4

```
try {
 ByteArrayOutputStream out = new ByteArrayOutputStream();
  DataOutputStream outData = new DataOutputStream(out);
    outData.writeByte(128);
    // этот метод принимает аргумент int, но записывает
    // лишь младший байт
    outData.writeInt(128);
    outData.writeLong(128);
    outData.writeDouble(128);
    outData.close();
    byte[] bytes = out.toByteArray();
    InputStream in = new ByteArrayInputStream(bytes);
    DataInputStream inData = new DataInputStream(in);
  System.out.println("Correct sequence reading:");
    System.out.println("readByte: " + inData.readByte());
    System.out.println("readInt: " + inData.readInt());
    System.out.println("readLong: " + inData.readLong());
    System.out.println("readDouble: " + inData.readDouble());
    inData.close();
    System.out.println("Inverted sequence reading:");
    in = new ByteArrayInputStream(bytes);
    inData = new DataInputStream(in);
    System.out.println("readInt: " + inData.readInt());
    System.out.println("readDouble: " + inData.readDouble());
    System.out.println("readLong: " + inData.readLong());
    inData.close();
    } catch (Exception e) {
    System.out.println("Impossible IOException occurs: " +
    e.toString());
    e.printStackTrace();
}
```

Результат выполнения программы:

```
Correct sequence reading:
readByte: -128
readInt: 128
readLong: 128
```

Библиотека БГУИР

readDouble: 128.0
Inverted sequence reading:
readInt: -2147483648
readDouble: -0.0
readLong: -9205252085229027328

Класс *File*

Если классы потоков осуществляют реальную запись и чтение данных, то класс *File* – это вспомогательный инструмент, призванный обеспечить работу с файлами и каталогами.

Объект класса *File* является абстрактным представлением файла и пути к нему. Он устанавливает только соответствие с ним, при этом для создания объекта неважно, существует ли такой файл на диске. После создания можно выполнить проверку, вызвав метод *exists*, который возвращает значение *true*, если файл существует. Создание или удаление объекта класса *File* никоим образом не отображается на реальных файлах. Для работы с содержимым файла можно получить экземпляры *FileI/OStream*.

Объект *File* может указывать на каталог (узнать это можно путем вызова метода *isDirectory()*). Метод *list* возвращает список имен (массив *String*) содержащихся в нем файлов (если объект *File* не указывает на каталог, будет возвращен *null*).

Следующий пример демонстрирует использование объектов класса *File*.

Пример 2.5

```
import java.io.*;
public class FileDemo {
  public static void findFiles(File file, FileFilter filter,
    PrintStream output) throws IOException{
    if (file.isDirectory()) {
    File[] list = file.listFiles();
    for (int i=list.length; --i>=0;) {
    findFiles(list[i], filter, output);
    }
    } else {
    if (filter.accept(file))
    output.println("\t" + file.getCanonicalPath());
    }
 }
  public static void main(String[] args) {
 class NameFilter implements FileFilter {
```

Библиотека БГУИР

```
    private String mask;
    NameFilter(String mask) {
      this.mask = mask;
    }
    public boolean accept(File file){
      return (file.getName().indexOf(mask)!=-1)?true:false;
    }
  }
  File pathFile = new File(".");
  String filterString = ".java";
  try {
    FileFilter filter = new NameFilter(filterString);
    findFiles(pathFile, filter, System.out);
  } catch(Exception e) {
    e.printStackTrace();
  }
  System.out.println("work finished");
 }
}
```

При выполнении этой программы на экран будут выведены названия (в каноническом виде) всех файлов с расширением *.java*, содержащихся в текущем каталоге и всех его подкаталогах.

Для определения того, что файл имеет расширение *.java*, использовался интерфейс *FileFilter* с реализацией в виде внутреннего класса *NameFilter*. Интерфейс *FileFilter* определяет только один метод *accept*, возвращающий значение, определяющее, попадает ли переданный файл в условия фильтрации. Помимо этого интерфейса, существует еще одна разновидность интерфейса фильтра – *FilenameFilter*, где метод *accept* определен несколько иначе: он принимает не объект файла к проверке, а объект *File*, указывающий на каталог, где находится файл для проверки, и строку его названия. Для проверки совпадения с учетом регулярных выражений нужно соответствующим образом реализовать метод *accept*. В конкретном приведенном примере можно было обойтись и без использования интерфейсов *FileFilter* или *FilenameFilter*. На практике их можно использовать для вызова методов *list* объектов *File* – в этих случаях будут возвращены файлы с учетом фильтра.

Также класс *File* предоставляет возможность получения некоторой информации о файле:

Библиотека БГУИР

- методы *canRead* и *canWrite* – возвращается *boolean*-значение, можно ли будет приложению производить чтение и изменение содержимого из файла соответственно;
- *getName* – возвращает строку – имя файла (или каталога);
- *getParent, getParentName* – возвращают каталог, где файл находится в виде строки названия и объекта *File* соответственно;
- *getPath* – возвращает путь к файлу (при этом в строку преобразуется абстрактный путь, на который указывает объект *File*);
- *isAbsolutely* – возвращает *boolean* значение, является ли абсолютным путь, которым указан файл. Определение, является ли путь абсолютным, зависит от системы, где запущена *Java*-машина. Так, для *Windows* абсолютный путь начинается с указания диска либо символом '\'. Для *Unix* абсолютный путь начинается символом '/' ;
- *isDirectory, isFile* – возвращает *boolean* значение, указывает ли объект на каталог либо файл соответственно;
- *isHidden* – возвращает *boolean* значение, указывает ли объект на скрытый файл;
- *lastModified* – дата последнего изменения;
- *length* – длина файла в байтах.

Также можно изменить некоторые свойства файла – методы *setReadOnly*, *setLastModified*, назначение которых очевидно из названия. Если нужно создать файл на диске, это позволяют сделать методы *createNewFile*, *mkDir*, *mkDirs*. Соответственно *createNewFile* создает пустой файл (если таковой еще не существует), *mkDir* создает каталог, если для него все родительские уже существуют, а *mkDirs* создает каталог вместе со всеми необходимыми родительскими.

Файл можно и удалить – для этого предназначены методы *delete* и *deleteOnExit*. При вызове метода *delete* файл будет удален сразу же, а при вызове *deleteOnExit* по окончании работы *Java*-машины (только при корректном завершении работы) отменить запрос уже невозможно.

Таким образом, класс *File* дает возможность достаточно полного управления файловой системой.

Класс *RandomAccessFile*

Этот класс реализует сразу два интерфейса – *DataInput* и *DataOutput* – следовательно, может производить запись и чтение всех примитивных типов *Java*. Эти операции, как следует из названия, производятся с файлом. При этом их можно производить поочередно, произвольным образом перемещаясь по файлу с помощью вызова метода *seek(long)* (переводит на указанную позицию в

Библиотека БГУИР

файле). Узнать текущее положение указателя в файле можно вызовом метода *getFilePointer*.

При создании объекта этого класса конструктору в качестве параметров нужно передать два параметра: файл и режим работы. Файл, с которым будет проводиться работа, указывается либо с помощью *String* – название файла, либо объектом *File*, ему соответствующим. Режим работы (*mode*) – представляет собой строку либо «*r*» (только чтение), либо «*rw*» (чтение и запись). Попытка открыть несуществующий файл только на чтение приведет к исключению *FileNotFoundException*. При открытии на чтение и запись он будет незамедлительно создан (или же будет брошено исключение *FileNotFoundException*, если это невозможно осуществить).

После создания объекта *RandomAccessFile* можно воспользоваться методами интерфейсов *DataInput* и *DataOutput* для проведения с файлом операций считывания и записи. По окончании работы с файлом его следует закрыть, вызвав метод *close*.

В следующем примере создается объект типа *String*, инициализируется текстовой строкой, определяется количество гласных, пробелов и общее количество букв. Реализован также механизм записи информации в файл и ее считывания из файла.

Пример 2.6

```
import java.io.*;
public class Lab1
{
private String data;
private String filename;
private String choice;
private RandomAccessFile fio;
private BufferedReader in=
new BufferedReader(new InputStreamReader(System.in));
  public void runConsol(){
    while(true){
          try{
          System.out.println("\nEnter your choice:");
          System.out.println("1.Read text from file");
          System.out.println("2.Type text");
          System.out.println("3.Exit");
          //чтение выбора пользователя
                choice=in.readLine();
```

Библиотека БГУИР

```
                if (choice.compareTo("1")==0){
                System.out.println("Type your filename");
                filename=in.readLine();
                fio = new RandomAccessFile(new File(filename), "r");
                data=fio.readLine();
                fio.close();
                System.out.println("\nFile Input:\n"+data);
                        int spaces=0, glas=0, lett=0;
char ch;
for(int i=0;i<data.length();i++)
{
        ch=Character.toLowerCase(data.charAt(i));
        if(Character.isWhitespace(ch))
        spaces++;
        if((ch=='a')||(ch=='e')||(ch=='i')||(ch=='o')||(ch=='u')||(ch=='y'))
        glas++;
        lett++;
}
System.out.println("\nspaces - "+spaces+"\nvowels - "+glas+"\nletters -
 "+(lett-spaces));
                }
        else if (choice.compareTo("2")==0){
                System.out.println("Type your text");
                data=in.readLine();
                System.out.println("Type your filename");
                filename=in.readLine();
                fio = new RandomAccessFile(new File(filename), "rw");
                fio.writeBytes(data);
                fio.close();
                System.out.println("Your text was saved");
        }
        else if (choice.compareTo("3")==0){
                return;
        }
}
catch(FileNotFoundException e){
        System.out.println("File not found");
}
catch(IOException e){
```

Библиотека БГУИР

```
                System.out.println("Error1");
            }
            catch(Exception e){
                System.out.println("Error2");
            }
        }
    }
}
```

Результат выполнения программы:

```
Enter your choice:
1.Read text from file
2.Type test
3.Exit
```

Выбрав 2-й вариант, введем строку и имя файла, где необходимо сохранить строку. После этого будет предложено снова осуществить выбор.

Теперь выберем 1-й вариант. Введем имя нашего файла. В результате выведется исходная строка и итоги ее анализа: количество пробелов, гласных и общее количество букв. И снова будет предложен выбор.

Теперь выберем 3-й вариант. После этого программа завершит свою работу.

Задания для самостоятельного выполнения

Используя варианты заданий к лабораторной работе №1, разработать программу, которая позволяла бы работать с тремя файлами. Исходная информация должна храниться в файле file1.txt, программа ее считывает и осуществляет с ней преобразования, затем преобразованная информация записывается в файл file2.txt, а в файл file3.txt программа должна записать исходную и преобразованную информацию.

Библиотека БГУИР

ЛАБОРАТОРНАЯ РАБОТА №3

РАЗРАБОТКА КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ TCP СОЕДИНЕНИЙ

Цель: создание клиент-серверных приложений с использованием протокола TCP/IP.

Серверы и клиенты

В контексте работы в сети используются такие термины, как клиент и сервер. Сервер – это все, что имеет некоторый разделяемый (коллективно используемый) ресурс. Существуют вычислительные серверы, которые обеспечивают вычислительную мощность; серверы печати, которые управляют совокупностью принтеров; дисковые серверы, которые предоставляют работающее в сети дисковое пространство, и *Web*-серверы, которые хранят *Web*-приложения. Клиент – любой другой объект, который хочет получить доступ к серверу. Сервер – это постоянно доступный ресурс, в то время как клиент может «отключиться» после того, как он был обслужен.

Различие между сервером и клиентом существенно только, когда клиент пытается подключиться к серверу. Как только они соединятся, происходит процесс двухстороннего общения, и не важно, что один является сервером, а другой – клиентом.

Работа сервера – слушать соединение, которое выполняется с помощью специального создаваемого серверного объекта (сокета), содержащего *IP*-адрес и номер порта. Работа клиента – попытаться создать соединение с сервером, которое выполняется с помощью клиентского сокета. Как только соединение установлено, соединение превращается в потоковый объект ввода/вывода. С этого момента можно рассматривать соединение как файл, который можно читать и в который можно записывать данные. Единственная особенность – файл может обладать определенным интеллектом и обрабатывать передаваемые команды.

Эти функции обеспечиваются расширением программы сетевой библиотеки *java.net.*;*.

Сокеты

Передача данных по сети – сложный процесс, включающий в себя определение пути доставки данных, организацию взаимодействия, алгоритмы синхронизации, обработки сбойных ситуаций и т.п. Программное обслуживание такого процесса сложное. Для упрощения введено понятие сокета (гнезда) как конечной точки коммуникации. Сокет (гнездо, разъем) – это

Библиотека БГУИР

программная абстракция, используемая для представления «терминалов» соединений между двумя машинами.

Каждый из сокетов определяется типом и ассоциированным с ним процессом. Реально для передачи организуются определенные дескрипторы *TCP*-соединения, так называемые гнезда (*socket*): гнездо сервера и гнездо клиента, которые в *Internet*-домене включают в себя *IP*-адреса сервера и клиента и номера портов, через которые они взаимодействуют. Сервер обычно имеет закрепленный и постоянный во взаимодействии номер порта, а клиенту, обращающемуся по этому номеру для связи к серверу, назначается некоторый другой (эфемерный) номер порта после установления соединения с сервером на сеанс их взаимодействия. Таким образом основной порт освобождается для установления последующих связей (номер порта выбирается сервером из числа не занятых в диапазоне от 1024 до 65 535). Эта комбинация (*IP*-адрес и номера портов) однозначно определяет отдельные сетевые процессы в сети *Internet* (номера портов до 1024, как правило, резервируются для широко известных приложений, например 80 – для связывания с серверами *Web* по протоколу *HTTP*).

Сокеты для работы в сети можно создать двух типов:

1) потоковые для *TCP*-соединения. *TCP* могут передавать данные только между двумя приложениями, т.к. они предполагают наличие канала между этими приложениями;

2) дейтаграммные. Для дейтаграмм не нужно создавать канал, данные посылаются приложению с использованием адреса, состоящего из сокета и номера порта (в дейтаграммах не гарантируются доставка и корректность последовательности передачи пакетов). Для передачи дейтаграмм не нужны ни механизмы подтверждения связи, ни механизмы управления потоком данных.

Тип соединения с использованием дейтаграмм будет рассмотрен в лабораторной работе №4. В данной лабораторной работе рассмотрим *TCP*-соединения.

Для упрощения представления такого соединения представим себе сокет, размещенный на некоторой машине, и виртуальный «кабель», соединяющий две машины, каждый конец которого вставлен в сокет. Для *TCP*-соединений в *Java* используется два класса сокетов: *ServerSocket* – класс, используемый сервером, чтобы «слушать» входящие соединения, и *Socket* – используемый клиентом для инициирования соединения.

Сокеты TCP/IP серверов

Как было указано выше, для создания сокетов серверов используется класс *ServerSocket*. Указанный класс используется для создания серверов,

Библиотека БГУИР

которые прослушивают либо локальные, либо удаленные программы клиента, чтобы соединяться с ними на опубликованных портах.

Конструкторы класса *ServerSocket*:

ServerSocket(int port) – создает сокет сервера на указанном порте с длиной очереди по умолчанию 50.

ServerSocket(int port, int maxQueue) – создает сокет сервера на указанном порте с максимальной длиной очереди *maxQueue*.

ServerSocket(int port, int maxQueue, InetAddress localAddress) – создает сокет сервера на указанном порте с максимальной длиной очереди *maxQueue*.

Класс *ServerSocket* имеет метод *accept()*, который является блокирующим вызовом: сервер будет ждать клиента, чтобы инициализировать связь, и затем вернет нормальный *Socket*-объект, который будет использоваться для связи с клиентом.

Как только клиент создает соединение по сокету, *ServerSocket* возвращает с помощью метода *accept()* соответствующий клиенту объект *Socket* на сервере, по которому будет происходить связь со стороны сервера. Начиная с этого момента, появляется соединение сокет–сокет, и можно считать эти соединения одинаковыми, т.к. они действительно одинаковые.

Пример 3.1

```
ServerSocket httpServer=new ServerSocket(port);//создание сокета
//сервера
Socket reg=httpServer.accept();  //прослушивание (ожидание
 // запроса на соединения)
// прием содержания соединения клиента
….
//отправка клиенту сообщения
…
//разрыв соединения
```

В представленном коде после установки соединения с клиентом метод *accept()* возвращает объект класса *Socket* (в данном случае *reg*), с помощью которого можно создавать и использовать байтовые и символьные потоки для обмена данными с клиентами. Для этого с гнездом связываются входной и выходной потоки, которые реализуются с помощью классов *InputStream* и *OutputStream*:

Пример 3.2

```
//  получение входного и выходного потоков
```

Библиотека БГУИР

InputStream inputstream = reg.getInputStream();

OutputStream outputstream = reg.getOutputStream();

Получив объекты, реализующие потоки, можно воспользоваться предоставляемыми ими методами, чтобы организовать взаимодействие по сети. Например, организовать чтение байта из входного потока можно при помощи метода *read*, а запись байта в выходной поток – с использованием метода *write*:

Пример 3.3

```
int c= inputstream.read(); // чтение байта из входного потока
outputstream. write(c);    //запись байта в выходной поток
```

Сокеты TCP/IP клиентов

Для создания сокета клиента используется конструктор:

Socket(String hostname, int port) – создает сокет, соединяющий локальную хост-машину с именованной хост-машиной и портом; может выбрасывать исключение *UnknownHostException* или *IOException*.

Socket(InetAddress ipAddress, int port) – создает сокет, аналогичный предыдущему, но используется уже существующий объект класса *InetAddres* и порт; может выбрасывать исключение *IOException*.

Сокет может в любое время просматривать связанную с ним адресную и портовую информацию при помощи методов, представленных в табл. 3.1.

Таблица 3.1

Методы просмотра адресной и портовой информации

Метод	Описание
InetAddress getInetAddress()	Возвращает *InetAddress*-объект, связанный с *Socket*-объектом
int getPort()	Возвращает удаленный порт, с которым соединен данный *Socket*-объект
int getLocalPort()	Возвращает локальный порт, с которым соединен данный *Socket*-объект

После создания *Socket*-объект можно применять для получения доступа к связанным с ним потокам ввода/вывода. Данные потоки используются точно так же, как потоки ввода/вывода, описанные в лабораторной работе №1.

Рассмотрим пример, в котором необходимо создать приложение клиент-сервер. Клиент считывает строку с клавиатуры, отображает ее на экране, передает ее серверу, сервер отображает ее на экране, переводит в верхний

Библиотека БГУИР

регистр и передает клиенту, который, в свою очередь, снова отображает ее на экране.

Пример 3.4

Программа сервера:

```
import java.io.*;//импорт пакета, содержащего классы для
//ввода/вывода
import java.net.*;//импорт пакета, содержащего классы для работы в
//сети Internet
public class server
{public static void main(String[] arg)
{//объявление объекта класса ServerSocket
ServerSocket serverSocket = null;
Socket clientAccepted     = null;//объявление объекта класса Socket
ObjectInputStream  sois   = null;//объявление байтового потока ввода
ObjectOutputStream soos   = null;//объявление байтового потока вывода
try {
System.out.println("server starting....");
serverSocket = new ServerSocket(2525);//создание сокета сервера для
//заданного порта
clientAccepted = serverSocket.accept();//выполнение метода, который
//обеспечивает реальное подключение сервера к клиенту
System.out.println("connection established....");
//создание потока ввода soos = new
sois = new ObjectInputStream(clientAccepted.getInputStream());
ObjectOutputStream(clientAccepted.getOutputStream());//создание потока
//вывода
String clientMessageRecieved = (String)sois.readObject();//объявление
//строки и присваивание ей данных потока ввода, представленных
//в виде строки (передано клиентом)
while(!clientMessageRecieved.equals("quite"))//выполнение цикла: пока
//строка не будет равна «quite»
{
System.out.println("message recieved: '"+clientMessageRecieved+"'");
clientMessageRecieved = clientMessageRecieved.toUpperCase();//приведение
//символов строки к
//верхнему регистру
soos.writeObject(clientMessageRecieved);//потоку вывода
//присваивается значение строковой переменной (передается клиенту)
```

Библиотека БГУИР

```
        clientMessageRecieved = (String)sois.readObject();//строке
//присваиваются данные потока ввода, представленные в виде строки
//(передано клиентом)
        }   }catch(Exception e)    {
        } finally {
        try {
sois.close();//закрытие потока ввода
soos.close();//закрытие потока вывода
clientAccepted.close();//закрытие сокета, выделенного для клиента
serverSocket.close();//закрытие сокета сервера
} catch(Exception e) {
        e.printStackTrace();//вызывается метод исключения e
}
}
}
}
```

Программа клиента:

```
import java.io.*;//импорт пакета, содержащего классы для
// ввода/вывода
import java.net.*;//импорт пакета, содержащего классы для
// работы в сети
public class client {
public static void main(String[] arg) {
try {
System.out.println("server connecting....");
Socket clientSocket = new Socket("127.0.0.1",2525);//установление
//соединения между локальной машиной и указанным портом узла сети
System.out.println("connection established....");
BufferedReader stdin =
new BufferedReader(new InputStreamReader(System.in));//создание
//буферизированного символьного потока ввода
ObjectOutputStream coos =
new ObjectOutputStream(clientSocket.getOutputStream());//создание
//потока вывода
ObjectInputStream  cois =
new ObjectInputStream(clientSocket.getInputStream());//создание
//потока ввода
```

Библиотека БГУИР

```
System.out.println("Enter any string to send to server \n\t('quite' − programme
terminate)");
String clientMessage = stdin.readLine();
System.out.println("you've entered: "+clientMessage);
while(!clientMessage.equals("quite")) {//выполнение цикла, пока строка
//не будет равна «quite»
coos.writeObject(clientMessage);//потоку вывода присваивается
//значение строковой переменной (передается серверу)
System.out.println("~server~: "+cois.readObject());//выводится на
//экран содержимое потока ввода (переданное сервером)
System.out.println("--------------------------");
clientMessage = stdin.readLine();//ввод текста с клавиатуры
System.out.println("you've entered: "+clientMessage);//вывод в
//консоль строки и значения строковой переменной
}
coos.close();//закрытие потока вывода
cois.close();//закрытие потока ввода
clientSocket.close();//закрытие сокета
}catch(Exception e)       {
e.printStackTrace();//выполнение метода исключения
}
}
}
```

Для запуска приведенного кода сервера и клиента сначала необходимо запустить приложение сервера, потом – приложение клиента.

После запуска сервера на экране появится консольное окно сервера со строкой:

server starting ….

После запуска клиента и установления его соединения с сервером на экране появится консоль с текстом:

server connecting….

connection established ….

Enter any string to send to server

<”quite”– program will terminate>

После установления соединения на сервере появляется еще одна строка, свидетельствующая об этом:

connection established….

Библиотека БГУИР

Теперь введем в окне клиента какую-либо строку, например Hello BSUIR. Пошлем ее серверу.

В окне сервера появятся следующие строки:

message recieved: 'Hello BSUIR'

В окне клиента появятся дополнительные строки:

you've entered: Hello BSUIR

~server~: HELLO BSUIR

Теперь введем слово *«quite»* на клиенте. Приложения завершат свою работу. Сначала закрывается окно клиента, затем – окно сервера.

Задания для самостоятельного выполнения

Разработать приложение на основе *TCP*-соединения, позволяющее осуществлять взаимодействие клиента и сервера по совместному решению задач обработки информации. Приложение должно располагать возможностью передачи и модифицирования получаемых (передаваемых) данных.

1. Разработать приложение-калькулятор для совершения простейших арифметических операций. Исходные параметры и тип операции (+, -, /, ·) вводятся на клиентской части и передаются серверу. Сервер возвращает клиенту результат операции.
2. Разработать приложение-чат. На сервере и клиенте отображаются передаваемые сообщения и время их отправления.
3. Разработать приложение-генератор случайных чисел. На клиентской части вводится целое положительное число N и передается серверу, а тот в свою очередь возвращает клиенту массив случайных чисел от 1 до N.
4. Разработать приложение-поисковик слов. На сервере хранится определенный текст. На клиентской части вводится слово для поиска и передается серверу, а тот в свою очередь осуществляет поиск этого слова в тексте и возвращает клиенту все предложения, в которых встречается это слово.
5. Разработать приложение-счетчик букв. На клиентской части вводится строка и передается серверу, а тот в свою очередь осуществляет подсчет гласных и согласных букв и возвращает этот результат клиенту.
6. Разработать приложение-определитель матрицы. На клиентской части вводится исходная матрица произвольного порядка и передается серверу, а тот в свою очередь вычисляет определитель этой матрицы и возвращает результат клиенту.

Библиотека БГУИР

7. Разработать приложение для нахождения обратной матрицы размером 3х3. Исходная матрица вводится на клиентской части и передается серверу, а тот в свою очередь возвращает клиенту обратную матрицу.

8. Разработать приложение для определения победителя лотереи. На сервере хранятся номера билетов. На каждом билете имеются 10 случайных чисел от 1 до 100. На клиентской части вводятся 10 чисел от 1 до 100, и сервер должен определить номер билета, в котором имеется больше всего совпадений с введенными числами.

9. Разработать приложение для определения призовых мест на соревнования по прыжкам в длину. На сервере хранятся фамилии участников соревнований, их идентификационные номера. На клиентской части вводятся результаты прыжков по каждому идентификационному номеру, а сервер возвращает фамилии спортсменов, занявших 1, 2 и 3 места.

10. Разработать приложение для определения суммы подоходного налога. На клиентской части вводятся заработные платы сотрудников предприятия и передаются северу, а тот в свою очередь возвращает суммы налога. Причем для з/п меньше 100 000 руб. применяется ставка налога 5 %, для з/п от 100 000 до 500 000 – ставка 10 %, для з/п больше 500 000 – ставка 15 %.

11. Разработать приложение по поиску квартиры для покупки. Стоимости квартир и их адреса хранятся на сервере. На клиентской части вводится предельная сумма для покупки квартиры, а сервер возвращает клиенту адреса всех квартир с такой или меньшей стоимостью.

12. Разработать приложение, серверная часть которого в матрице произвольного порядка определяла бы индекс строки с минимальным элементом и индекс столбца с максимальным элементом этой матрицы и возвращала этот результат клиенту.

13. Разработать приложение, серверная часть которого в матрице произвольного порядка определяла бы отношение среднего значения элементов, расположенных на главной диагонали, к среднему значению элементов, расположенных на побочной диагонали этой матрицы, и возвращала результат клиенту.

14. Разработать приложение, в котором серверная часть хранит информацию о расписании занятий студентов. Клиентская часть имеет возможность просматривать, редактировать и удалять необходимую информацию.

15. Разработать приложение, в котором серверная часть осуществляет расчет себестоимости продукции. При этом пользователь на клиенте вводит необходимую информацию, например: основная заработная плата, дополнительная заработная плата, материалы, прочие затраты и т.д., посылает

Библиотека БГУИР

ее на сервер. Сервер производит расчет и высылает назад клиенту рассчитанную полную себестоимость.

Библиотека БГУИР

ЛАБОРАТОРНАЯ РАБОТА №4

РАЗРАБОТКА КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ *UDP* СОЕДИНЕНИЙ

Цель: разработка клиент-серверных приложений с использованием протокола *UDP*.

Протокол *UDP*

UDP представляет собой альтернативу *TCP*, требующую меньших накладных расходов. В отличие от *TCP*, *UDP* имеет следующие характеристики:

- ненадежный сетевой протокол. *UDP* не имеет ни встроенного механизма обнаружения ошибок, ни средств повторной пересылки поврежденных или потерянных данных;
- отсутствие установления логического соединения. Перед пересылкой данных *UDP* не устанавливает логического соединения. Информация пересылается в предположении, что принимающая сторона ее ожидает;
- основан на сообщениях. Позволяет приложениям пересылать информацию в виде сообщений, передаваемых посредством дейтаграмм (*datagram*), которые являются единицами передачи данных в *UDP*.

Как и в *TCP*, в *UDP* применяется схема адресации с использованием портов, позволяющая нескольким приложениям параллельно принимать и посылать данные. В то же время порты *UDP* отличаются от портов *TCP*. Например, одно приложение может отзываться на номер 512 порта *UDP*, а при этом другой независимый сервис может обрабатывать порт 512, относящийся к *TCP*.

Характеристика сокетов *UDP*

UDP существенно отличается от *TCP*. Наиболее подходящая для *UDP* аналогия – связь посредством почтовых открыток.

В протоколе *UDP* диалог должен быть разделен на небольшие сообщения, которые умещаются в небольшой пакет определенного размера. Когда посылается сообщение, нельзя быть уверенным, что ответ будет получен: сообщение могло быть потеряно по пути, мог потеряться ответ получателя, получатель также мог игнорировать сообщение.

Почтовые открытки, которыми обмениваются сетевые программы, называются дейтаграммами (*datagrams*). Дейтаграмма содержит массив байт.

Библиотека БГУИР

Принимающая программа может извлечь этот массив и декодировать информацию, а затем, возможно, послать ответную дейтаграмму.

Как и для протокола *TCP*, программирование для *UDP* будет использовать абстракцию сокета, но сокеты *UDP* сильно отличаются от сокетов *TCP*. Если продолжить почтовую аналогию, то сокет *UDP* соответствует почтовому ящику.

Почтовый ящик идентифицируется адресом владельца, но нет необходимости заводить новый ящик для каждого, кому нужно посылать сообщения (можно, однако, создать отдельный ящик для газет, чтобы они не попадали в ящик для писем). Для посылки сообщения достаточно написать на открытке адрес, по которому она должна быть доставлена. Затем она помещается в почтовый ящик и (раньше или позже) уходит по назначению.

Можно, в принципе, бесконечно долго ожидать, пока сообщение дойдет до почтового ящика. Когда сообщение получено, его можно прочесть. На открытке содержится также метаинформация, позволяющая по обратному адресу получить сведения об отправителе сообщения.

Итак, программирование с использованием *UDP* требует решить следующие задачи: создание правильно адресованной дейтаграммы, создание сокета для рассылки и получения дейтаграмм данным приложением, помещение дейтаграмм в сокет для передачи по назначению, ожидание получения дейтаграмм из сокета, декодирование дейтаграмм для выделения самого сообщения, адреса отправителя и другой метаинформации.

Классы *UDP*

Необходимые средства поддержки протокола *UDP* находятся в пакете *java.net*. Для создания дейтаграмм в *Java* существует класс *DatagramPacket*. При получении дейтаграммы по протоколу *UDP* класс *DatagramPacket* используется также для чтения данных, адреса отправителя и метаинформации.

Чтобы создать дейтаграмму для отправки на удаленную машину, используется следующий конструктор:

public DatagramPacket(byte[] ibuf, int length, InetAddress iaddr, int iport);

Здесь *ibuf* − массив байт, содержащий кодированное сообщение; *length* − количество байт, которое должно быть помещено в пакет, что определяет размер дейтаграммы; *iaddr* − это экземпляр класса *InetAddress*, который хранит *IP*-адрес получателя; *iport* указывает номер порта, на который посылается дейтаграмма.

Библиотека БГУИР

Чтобы получить дейтаграмму, необходимо использовать другой конструктор для объекта *DatagramPacket*, в котором будут находиться принятые данные. Прототип конструктора имеет вид

public DatagramPacket(byte[] ibuf, int length);

Здесь *ibuf* − массив байт, куда должны быть скопированы данные из дейтаграммы, а *length* − количество байт, которое должно быть скопировано.

Дейтаграммы не ограничены определенной длиной; можно создавать как очень длинные, так и очень короткие дейтаграммы. Заметим, однако, что между клиентом и сервером должно существовать соглашение о длине дейтаграмм, поскольку они оба должны создать массив байт нужного размера перед созданием объекта *DatagramPacket* для посылки или получения дейтаграммы.

Когда дейтаграмма получена, как будет продемонстрировано ниже, можно прочитать ее данные. Другие методы позволяют получить метаинформацию, относящуюся к сообщению:

- *public int getLength()*; – возвращает количество байт, из которых состоят данные дейтаграммы;
- *public byte[] getData()*; – позволяет получить массив, содержащий эти данные;
- *public InetAddress getAddress()*; – возвращает адрес отправителя;
- *public int getPort()*; – возвращает номер порта *UDP*, используемый отправителем.

Отправка и получение дейтаграмм осуществляется при помощи класса *DatagramSocket*, который создает сокет *UDP*. У него есть два конструктора, один из которых позволяет системе назначить любой из свободных портов *UDP*. Другой дает возможность задать конкретный порт, что полезно при разработке сервера. Как и для портов *TCP*, в большинстве операционных систем порты с номерами меньше 1024 доступны только процессам с привилегиями суперпользователя.

public DatagramSocket() throws SocketException;
public DatagramSocket(int port) throws SocketException;

Сокет, созданный первым конструктором, можно использовать для отправки правильно адресованных дейтаграмм при помощи следующего метода класса *DatagramSocket*:

public void send(DatagramPacket p) throws IOException;

Библиотека БГУИР

Если *DatagramSocket* создан вторым конструктором, можно получить дейтаграмму:

public syncronized void receive(DatagramPacket p) throws IOException;

Заметим, что метод *receive()* блокируется до момента получения дейтаграммы. Поскольку *UDP* является ненадежным протоколом, нельзя быть уверенным, что возврат из *receive()* вообще произойдет.

Когда закончен обмен через сокет *UDP*, его следует закрыть методом

public syncronized void close();

Рассмотрим пример, реализующий приложение клиент/сервер с использованием сокетов *UDP*.

Пример 4.1

Листинг приложения сервера:

```
import java.net.*;
import java.io.*;
public class UDPServer {
public final static int DEFAULT_PORT = 8001;//определение порта
//сервера
public final String VERSION_CMD = "VERS";//определение версии
//команды
public final String QUIT_CMD = "QUIT";//определение
//команды «выход»
public final byte[] VERSION = { 'V', '2', '.', '0' };//создание массива
//для определения версии сервера
public final byte[] UNKNOWN_CMD = { 'U', 'n', 'k', 'n', 'o', 'w', 'n', ' ',
                                    'c', 'o', 'm', 'm', 'a', 'n', 'd' };//неизвестная команда
public void runServer() throws IOException {//метод сервера runServer
DatagramSocket s = null;//создание объекта DatagramSocket
try {
boolean stopFlag = false;//создание флага stopFlag и его инициализация
//значением false
byte[] buf = new byte[512];//буфер для приема/передачи дейтаграммы
s = new DatagramSocket(DEFAULT_PORT);//привязка сокета к
```

Библиотека БГУИР

```
//реальному объекту с портом DEFAULT_PORT
System.out.println("UDPServer: Started on " + s.getLocalAddress() + ":"
                        + s.getLocalPort());//вывод в консоль сообщения
while(!stopFlag) {//цикл до тех пор, пока флаг не примет значение true
DatagramPacket recvPacket = new DatagramPacket(buf,
buf.length);//создание объекта дейтаграммы для получения данных
s.receive(recvPacket);//помещение полученного содержимого в
//объект дейтаграммы
String cmd = new String(recvPacket.getData()).trim();//извлечение
//команды из пакета
System.out.println("UDPServer: Command: " + cmd);
DatagramPacket sendPacket = new DatagramPacket(buf, 0,
recvPacket.getAddress(), recvPacket.getPort());//формирование объекта
                                        // дейтаграммы для отсылки данных
int n = 0;//количество байт в ответе
if (cmd.equals(VERSION_CMD)) {//проверка версии команды
n = VERSION.length;
System.arraycopy(VERSION, 0, buf, 0, n);
}
else if (cmd.equals(QUIT_CMD)) {
stopFlag = true;//остановка сервера
continue;
}
else {
n = UNKNOWN_CMD.length;
System.arraycopy(UNKNOWN_CMD, 0, buf, 0, n);
}
sendPacket.setData(buf);//установить массив посылаемых данных
sendPacket.setLength(n);//установить длину посылаемых данных
s.send(sendPacket);//послать сами данные
} // while(server is not stopped)
System.out.println("UDPServer: Stopped");
}
finally {
 if (s != null) {
s.close();//закрытие сокета сервера
}
}
}
```

Библиотека БГУИР

```
public static void main(String[] args) {//метод main
try {
UDPServer udpSvr = new UDPServer();//создание объекта udpSvr
udpSvr.runServer();//вызов метода объекта runServer
}
catch(IOException ex) {
ex.printStackTrace();
}
}
}
```

Листинг приложения клиента:

```
import java.net.*;
import java.io.*;
public class UDPClient {//описание класса клиента
public void runClient() throws IOException {//метод клиента runClient
DatagramSocket s = null;//создание дейтаграммы
try {
byte[] buf = new byte[512]; //буфер для приема/передачи дейтаграммы
s = new DatagramSocket();//привязка сокета к реальному объету
System.out.println("UDPClient: Started");
byte[] verCmd = { 'V', 'E', 'R', 'S' };
DatagramPacket sendPacket = new DatagramPacket(verCmd, verCmd.length,
InetAddress.getByName("127.0.0.1"), 8001);//создание
//дейтаграммы для отсылки данных
s.send(sendPacket);//посылка дейтаграммы
DatagramPacket recvPacket = new DatagramPacket(buf,
buf.length);//создание дейтаграммы для получения данных
s.receive(recvPacket);//получение дейтаграммы
String version = new String(recvPacket.getData()).trim();//извлечение
//данных (версии сервера)
System.out.println("UDPClient: Server Version: " + version);
byte[] quitCmd = { 'Q', 'U', 'I', 'T' };
sendPacket.setData(quitCmd);//установить массив посылаемых данных
sendPacket.setLength(quitCmd.length);//установить длину посылаемых
// данных
s.send(sendPacket); //послать данные серверу
System.out.println("UDPClient: Ended");
}
```

Библиотека БГУИР

```
finally {
if (s != null) {
s.close();//закрытие сокета клиента
} } }
public static void main(String[] args) {//метод main
try {
UDPClient client = new UDPClient();//создание объекта client
client.runClient();//вызов метода объекта client
}
catch(IOException ex) {
ex.printStackTrace();
}
}
}
```

Вначале запустите сервер, затем клиент. В результате на сервере появятся строки:

```
UDPServer: Started on 0.0.0.0/0.0.0.0:8001
UDPServer: Command: VERS
UDPServer: Command: QUIT
UDPServer: Stopped
```

На клиенте появятся строки:

```
UDPClient: Started
UDPClient: Server Version: V2.0
UDPClient: Ended
```

Задания для самостоятельного выполнения

Разработать приложение на основе *UDP*-соединения, позволяющее осуществлять взаимодействие клиента и сервера по совместному решению задач обработки информации. Приложение должно располагать возможностью передачи и модифицирования получаемых (передаваемых) данных. Возможности клиента: передать серверу исходные параметры (вводятся с клавиатуры) для расчета значения функции, а также получить расчетное значение функции. Возможности сервера: по полученным от клиента исходным параметрам рассчитать значение функции, передать клиенту расчетное значение функции, а также сохранить исходные параметры и значение функции в файл.

1. $a = \ln(y^{-\sqrt{|x|}})(x - \frac{y}{2}) + \sin^2 arctgz + e^{x+y}$.

Библиотека БГУИР

2. $b = \sqrt{10(\sqrt{x} + x^{yz})(\sin^2 z - |x+y|)}e^z$.

3. $c = 5arctgx - \frac{1}{4}\cos\frac{x + 3|x-y| + x^2}{|x+y^2|^3 + x^3}$.

4. $j = \frac{e^{|x-y|}|x-y|^{x+y}}{arctgx + tgz} + \sqrt{x^6 + \ln^2 y}$.

5. $f = \left| x^{\frac{y}{x}} - \sqrt{\frac{y}{x}} \right| + (y-x)\frac{\cos y - e^{\frac{z}{(y-x)}}}{1+(y-x)^2}$.

6. $d = y^x + \sqrt{|x| + e^y} - \frac{z^3 \sin^2 y}{y + \frac{z^3}{y - z^3}}$.

7. $e = |\cos x - e^y|^{1+2\ln^2 y}(1 + z + \frac{z^2}{2} + \frac{z^3}{3})$.

8. $l = \frac{1 + \sin^2(x+y)}{\left| e^x - \frac{2y}{1 + x^2 y^3} \right|} x^{|y|} + \cos^2(arctg\frac{1}{z})$.

9. $h = \frac{\sqrt{8 + |x+y|^2 + z}}{x^2 + y^2 + z^2} - e^{|x-y|}(tg^2 z + \sqrt[5]{|z|})$.

10. $s = \frac{2\cos(x - \frac{p}{6})}{e^{0.5} + \sin^2 y}(1 + \frac{z^2}{3 - z^5/5})$.

11. $c = y + \frac{e^{x-y}}{y + \frac{x^2}{y + \frac{x^3}{y}}}(1 + tg^2\frac{z}{3})^{\sqrt{|y|+7}}$.

12. $q = \lg(\sqrt{e^{x-y} + x^{|y|} + z})(x - \frac{x^3}{3} - \frac{x^7}{7})$.

13. $v = 6 + \frac{e^{x - \sin y}}{y + \frac{tg(x^2)}{(y + \frac{x^7}{z})}}(1 + ctg^7\frac{z}{100})^{\sqrt{|y|+3}}$.

Библиотека БГУИР

14. $t = \dfrac{2x(1+x^2)^2}{x+\sqrt[3]{|1+x^5|}}\left(\dfrac{\sqrt{x}}{2y+10}\right).$

15. $x = \dfrac{x+\dfrac{y}{5+\sqrt{x}}}{|y-x|+\sqrt[3]{x}}e^{z+1}+\sin^3 z.$

Библиотека БГУИР

ЛИТЕРАТУРА

1. Ноутон, П. Java 2 / П. Ноутон, Г. Шилдт; пер. с англ. – СПб. : BHV – Санкт-Петербург, 2000.

2. Хортон, А. Java 2– JDK 1.3. В 2 т. Т. 1. / А. Хортон; пер. с англ. – М. : Издательство «Лори», 2002.

3. Хортон, А. Java 2– JDK 1.3. В 2 т. Т. 2. / А. Хортон; пер. с англ. – М. : Издательство «Лори», 2002.

4. Блинов, И. Н. Java 2 : практическое руководство / И. Н. Блинов, В. С. Романчик. – Минск : УниверсалПресс, 2005.

5. Дейтел, Х. М. Технологии программирования на Java 2. Кн. 1: Графика, JavaBeans, интерфейс пользователя / Х. М. Дейтел, П. Дж. Дейтел, С. И. Сантри. – СПб. : BHV – Санкт-Петербург, 2000.

6. Дейтел, Х. М. Технологии программирования на Java 2. Кн. 3: Корпоративные системы, сервлеты, JSP, Web-сервисы / Х. М. Дейтел, П. Дж. Дейтел, С. И. Сантри. – СПб. : BHV – Санкт-Петербург, 2000.

7. Флэнаган, Д. Java : справочник / Д. Флэнаган; пер. с англ. – М. : Символ, 2004.

8. Вязовик, В. С. Программирование на Java : курс лекций для вузов по спец. 351400 «Прикладная информатика» / В. С. Вязовик. – М. : Интернет-университет информ. технологий, 2003.

Библиотека БГУИР

ПРИЛОЖЕНИЕ

Обработка строк

Как и в большинстве других языков программирования, строка в *Java* – это последовательность символов. Но в отличие от многих языков, которые реализуют строки как символьные массивы, в *Java* строки реализуются как объекты типа *String*.

Каждая создаваемая строка в действительности является объектом типа *String*. Даже строчные константы – это фактически *String*-объекты. Например, в утверждении

System.out.println(“This is a String, too”);

строка “This is a String, too” является *String*-константой.

Для работы со строками определен класс *String* в стандартной библиотеке *Java* в пакете *java.lang* (этот пакет импортируется по умолчанию).

Конструкторы:

public String(); //создает пустую строку
public String(char value[ ]); //создает строку из массива символов
public String(byte bytes[ ]); //создает строку из массива байт

Пример
String str = new String(“ Какая-то строка”);
Можно также записать
String str = “Какая-то строка”;

Для строк определена операция сложения, которая означает конкатенацию строк. Определена операция сложения с числом: сначала число преобразуется в строку, а потом соединяются строка с числом. Определена операция сложения строки с любым объектом. Для объекта вызывается метод toString, затем – конкатенация строк.

Пример
System.out.println(“Ошибка “+e.toString());

Методы для работы со строками:
public int compareTo(String anotherString); //этот метод
// сравнивает 2 строки
public int indexOf(int ch); //ищет индекс в строке
public int indexOf(String str); //ищет указанную строку в строке

Библиотека БГУИР

```
public int length();                    //возвращает длину строки
public String substring(int beginIndex, int endIndex);  //выделяет
//подстроку из строки
public String trim(); //удаляет из строки начальный и конечный пробелы
public char charAt(int index);    //выбирает из строки символ с
// индексом index
public boolean equals(Object str);        //проверяет равенство строк
```

Преобразование строки в число

Одной из наиболее обычных работ в программировании является преобразование строкового представления числа в его внутренний, двоичный формат. *Java* обеспечивает простой способ ее выполнения. В классах *Byte*, *Short*, *Integer* и *Long* определены методы *parseByte*, *parseShort ()*, *parseInt ()* и *parseLong ()*. Эти методы возвращают byte-, short-, int- или long-эквивалент числовой строки, с которой они вызываются. Подобные методы также существуют для классов *Double* и *Float*.

Программа, которая суммирует список чисел, введенных пользователем, преобразует строчное представление каждого числа в int-значение, используя метод *parseInt()*.

Пример

```
import  java.io.*;
class ParseDemo {
public static void main(String args[])
throws IOException {
// создать объект BufferedReader, используя System.in
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
String str;
int i;
int sum=0;
System.out.println("Вводите числа от 0 до quit.");
do {
str = br.readLine();
try {
i = Integer.parseInt(str) ;
} catch(NumberFormatException e) {
System.out.println(“Неправильный формат”);
i =0;
```

Библиотека БГУИР

```
}
sum+=1;
System.out.println(“Текущая сумма” + sum);
}while (i!=0);
   }
}
```

Конвертация чисел в строки

Чтобы конвертировать полное число в десятичную строку, используйте версии метода *toString()*, определенные в классах *Byte*, *Short*, *Integer* или *Long*. В классах *Integer* и *Long* определяются также методы *toBinaryString()*, *toHexString()* и *toOctalString()*, которые переводят внутримашинное значение в двоичную, шестнадцатеричную или восьмеричную строку соответственно.

Следующая программа демонстрирует двоичное, шестнадцатеричное и восьмеричное преобразование.

Пример

```
class StringConversions {
public static void main(String args[]) { int num = 19648;
System.out.println(num + " в двоичной форме: " +
Integer.toBinaryString(num));
System.out.println(num + " в восьмеричной форме: " +
Integer.toOctalString(num));
System.out.println(num + " в шестнадцатеричной форме: " +
Integer.toHexString(num)); } }
```

Вывод этой программы:

```
19648 в двоичной форме: 100110011000000
19648 в восьмеричной форме: 46300
19648 в шестнадцатеричной форме: 4cc0
```

Библиотека БГУИР

Св. план 2007, поз. 143

Учебное издание

Унучек Татьяна Михайловна
Комличенко Виталий Николаевич
Марудов Дмитрий Сергеевич и др.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ ДЛЯ РАЗРАБОТКИ СЕТЕВЫХ ПРИЛОЖЕНИЙ: ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

ЛАБОРАТОРНЫЙ ПРАКТИКУМ
для студентов специальностей
I-27 01 01 «Экономика и организация производства»,
I-26 02 03 «Маркетинг»
дневной формы обучения

В 2-х частях

Часть 1

Редактор Н. В. Гриневич
Корректор М. В. Тезина

Подписано в печать 14.09.2007.	Формат 60x84 1/16.	Бумага офсетная.
Гарнитура «Таймс»	Печать ризографическая.	Усл.печ.л. 3,72.
Уч.-изд.л. 3,2.	Тираж 250 экз.	Заказ 336.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0056964 от 01.04.2004. ЛП №02330/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6

Библиотека БГУИР