**Лабораторная работа 1**

**РАЗРАБОТКА КОНСОЛЬНЫХ ПРИЛОЖЕНИЙ**

**Цель работы:**

Создать консольные приложения для выполнения поставленных задач.

**Задание:**

1. Реализовать программу для создания массива переменных String и инициализировать массив с 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. Реализовать программу для создания массива из десяти переменных String, каждая из которых содержит произвольную строку - месяц/день/год, например 01/09/2022. Проанализировать каждый элемент в массиве и вывести представление даты в формате 01 сентября 2022.
3. Написать программу для создания случайной последовательности N прописных букв, которая не включает гласные буквы.
4. Создать объект типа String и проинициализировать его текстовой строкой. Определить количество гласных, пробелов и общее количество букв.
5. Создать массив объектов типа String и проинициализировать его текстовой строкой. Извлечь из массива строки по введенному разделителю.

**Рекомендации по выполнению работы:**

Понятие н определенно 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-variable 1;

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 х;

int у;

public int max(){

if(x>y) return x;

else return y;}

public int min(){

if(x<y) return х;

else return у;}

}

Для определения экземпляра класса используется синтаксис:

*ИмяКласса ИмяПеременной;*

*ИмяПеременной = new ИмяКласса (параметры инициализации);*

или

*ИмяКласса ИмяПеременной = new ИмяКласса ( параметры инициализации);*

*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 х=3;

int х() {return х;}

int у=х();

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

Модификаторы класса

*public –* класс доступен для использования, наследования вне зависимости 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 интерфейсов.

Пример 1.3

abstract class МуСlass {

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 3-х 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++ и т.д.) и добавлен в систему в виде загружаемой динамической библиотеки (например *DLL* для *Windows).* Cуществует еще один специальный модификатор *synchronized,* который будет рассмотрен в теме, описывающей потоки.

Объявление объектов и операция *new*

Получение объектов класса - это двух шаговый 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 Вох();

*//* второй способ

Box mybox; // объявить ссылку на объект

mybox = new Вох(); // распределить память для Вох-объекта

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

**Метод** *main*

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 *mainf)* 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 С 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.*

gs[])

{ 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 не определение 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 |

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 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 МуClass

{

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=а\*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:

а=1;

b=4;

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

Результат выполнения программы:

x= 25

a= 1Статические 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);

}

}

*main()* – главная функция, с нее 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 сети. 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.*

Байтовые и символьные потоки

*Java*  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 1.0* не включала символьные потоки, и. таким образом, весь ввод/вывод был байтовым.

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 |
| BufferedlnputStream  BufferedOutputStream | Буферизированные потоки ввода и вывода |
| ByteArrayInputStream ByteArrayOutputStream | Поток ввода, который читает из байт-массива  Поток вывода, который записывает в байт-  массив |
| FilelnputStream  FileOutputStream | Поток ввода, который читает из файла  Поток вывода, который записывает в файл |
| RandomAccessFile | Поддерживает ввод/вывод файла произвольного доступа |

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.

Таблица 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 поточными классами.

|  |  |
| --- | --- |
| Символьный класс | Назначение |
| Reader  Writer | Абстракные классы потока ввода и вывода |
| BufferedReader  BufferedWriter | Буферизированные символьные потоки ввода и вывода |
| FileReader  FileWriter | Поток ввода/вывода, который читает/записывает поток символов из/в файла |
| StringReader  StringWriter | Поток ввода/вывода, который читает/записывает из/в строку |
| PrintWriter | Поток вывода, который поддерживает методы print() и println() |

*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).

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 |
| Input Stream | Reader |
| OutputStream | Writer |
| ByteArrayInputStream | CharArrayReader |
| ByteArrayOutputStream | CharArrayWriter |
| Нет аналога | InputStreamReader |
| Нет аналога | OutputStreamWriter |
| FilelnputStream | FileReader |
| FileOutputStream | FileWriter |
| FilterlnputStream  FilterOutputStream  BufferedInputtStream  BufferedOutputStream | FilterReader  FilterWriter  BufferedReader  BufferedWriter |
| PrintStream | PrintWriter |
| DatalnputStream | Нет аналога |
| DataOutputStream | Нет аналога |
| ObjectInputStream  ObjectOutputStream | Нет аналога  Нет аналога |
| PipedlnputStream | PipedReader |
| PipedOutputStream | PipedWriter |
| StringBufferlnutStream | StringReader |
| Нет аналога | StringWriter |
| LineNumberlnputStream | LineNumberReader |
| PushBacklnputStream | PushBackReader |
| SequencelnputStream | Нет аналога |

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

Например, отсутствует преобразование в символьное представление примитивных типов *Java* и объектов *(Datalnput/Output, ObjectInput/Output).* Добавлены классы-мосты, преобразующие символьные потоки в байтовые:*InputStrearn Reader* и *OutputStreamWriter.* Именно на их основе реализованы *FileReader* и *FileWriter.* Метод *availablef)* 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: *«Ср1251»у «UTF-8»* 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 lOException

{

char c;

BufferedReader br = new

BufteredReader(new lnputStreamReader(System.in));

System.out.printlnf'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

2

3

а

b

с

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 lOException

{

// 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 буде! записана и файл

String data = "Some data to be written and read.\n";

try{

FileWriter fw = new FileWriter(fileName);

BufferedWriter bw = new BufferedWriter(fw);

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 разделения

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

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 = end Index + 1; //задать начало для

//следующего цикла

}

//вывести подстроки

for(int i = 0; i < subStr.length; i++)

System.out.println(subStr[i]);

}

}

Результат выполнения программы:

To

be

or

not

to

be