Министерство образования и науки Республики Беларусь

Гомельский государственный университет
им. Ф. Скорины

Кафедра математических проблем управления

ИЗБРАННЫЕ ГЛАВЫ ИНФОРМАТИКИ

Практическое руководство по выполнению лабораторных работ
для студентов специальности
1-31 03 03-01 «Прикладная математика
(научно-производственная деятельность)»

Гомель, 2014

Составители: Е.А. Ерофеева

Рецензенты:

Рекомендовано научно-методическим советом университета.

В практическое руководство по выполнению лабораторных работ включены первые пять работ: разработка простого консольного приложения, работа с простыми классами и объектами, применение наследования и полиморфизма, применение внутренних классов, реализация интерфейсов.

Каждая из работ содержит краткие теоретические сведения из своего раздела, постановку задачи и иллюстрированный пример.

СОДЕРЖАНИЕ

1 Введение 4
1.1 Инструкция по установке инструментария разработки 4
1.2 Требования к оформлению кода 10
2 Рекомендации по выполнению лабораторных работ 11
2.1 Лабораторная работа №1. Простое консольное приложение 11
2.2 Лабораторная работа №2. Простые классы 16
2.3 Лабораторная работа №3. Наследование и полиморфизм 22
2.4 Лабораторная работа №4. Применение внутренних классов 30
2.5 Лабораторная работа №5. Реализация интерфейсов 34
Список литературы 40

1 Введение

1.1 Инструкция по установке инструментария разработки

Java 2 Standart Edition (J2SE) – стандартная редакция языка Java, используемая для разработки простых Java приложений.

Java-программа Состоит из одного или нескольких определений классов, размещенных в одном или нескольких текстовых файлах с расширением «.java». В *одном java-файле* может быть только *один public-класс*. Имя java-файла должно совпадать с именем public-класса. В результате компиляции для каждого класса из исходного файла создается class-файл (файл с расширением «.class»), содержащий байт-коды класса. Имя class-файла совпадает с именем класса.

Точкой входа в программу является метод:

```
public static void main(String[] argv)
```

одного из public-классов проекта.

Оператор «package имя», помещаемый в начале исходного программного файла, определяет именованный пакет, т.е. область в пространстве имен классов, где определяются имена классов, содержащихся в этом файле.

Любой класс Java относится к определенному пакету, который может быть неименованным (unnamed или default package), если оператор package отсутствует.

Действие оператора package указывает на месторасположение файла относительно корневого каталога проекта.

Например:

```
package example06;
package by.gsu.pm41.example06;
```

При создании пакета всегда следует руководствоваться простым правилом: называть его именем простым, но отражающим смысл, логику поведения и функциональность объединенных в нем классов.

Для получения доступа к другим пакетам используется инструкция **import**, помещаемая после объявления пакета.

```
import task02.arsenal.Arsenal; // только один класс
import task02.arsenal.*; // все классы пакета
```

Для каждого java-файла компилятор Java всегда автоматически импортирует (т.е. без указания оператора import) два пакета: `java.lang` пакет и текущий пакет.

Для разработки программ необходим Java Development Kit (JDK) – бесплатно распространяемый корпорацией Sun комплект разработчика приложений на языке Java, включающий в себя:

- компилятор Java (javac),
- стандартные библиотеки классов Java,
- примеры,
- документацию,
- различные утилиты,
- исполнительную систему Java (JRE).

В состав JDK *НЕ входит* интегрированная среда разработки на Java (IDE – Integrated Development Environment), поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.

Популярные интегрированные среды разработки на Java: Eclipse; NetBeans; IntelliJ IDEA; Sun Java Studio Creator; Borland JBuilder.

Важные моменты при настройке Eclipse:

1. Скачать и установить с программу с официального сайта (https://www.eclipse.org/downloads). В установку также входит и JDK, поэтому отдельно устанавливать его не надо.

2. Создать каталог для разработки приложений (workspace) и указать его при запуске Eclipse (рисунок 1).

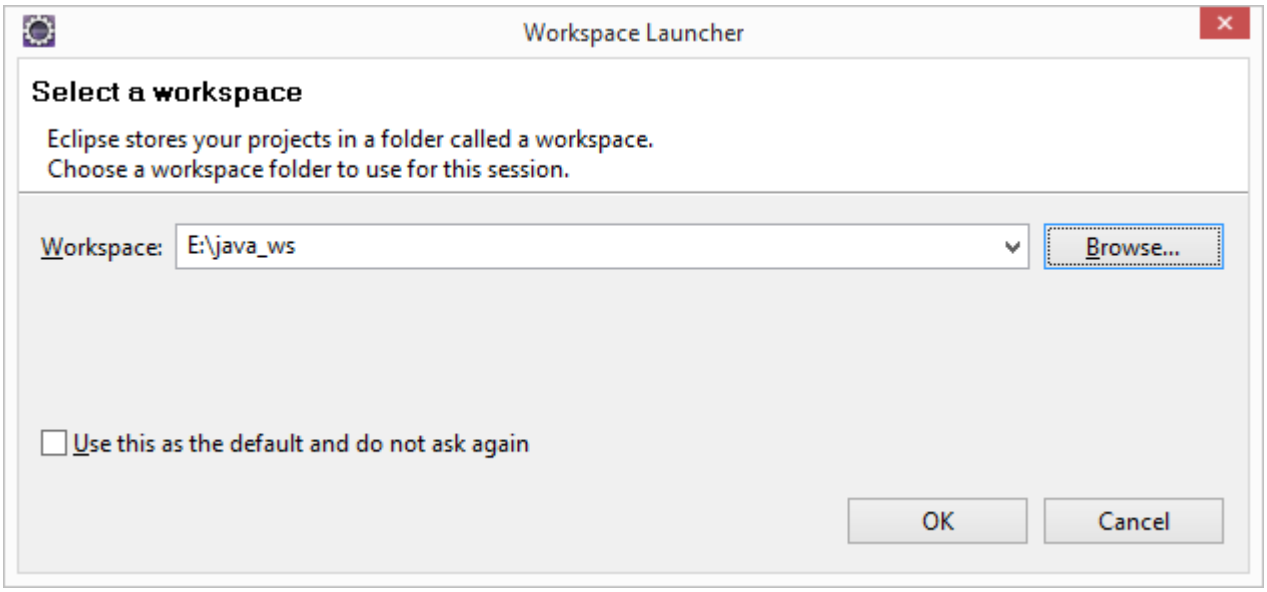

Рисунок 1 – Выбор рабочего каталога

После выбора рабочего каталога откроется основное окно разработки приложений (рисунок 2).

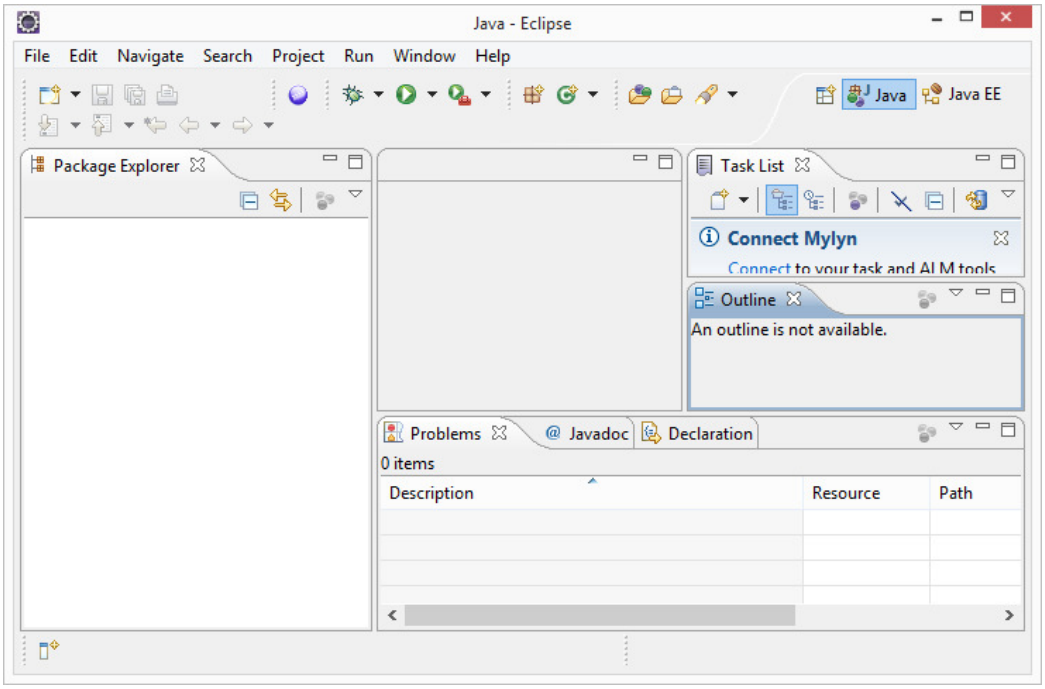

Рисунок 2 – Основное окно разработки приложений

Для создания нового проекта необходимо в главном меню выбрать **File** (Файл) – **New** (Новый) – **Java Project** (Java проект). В окне создания проекта (рисунок 3) вводится название проекта, также можно выбрать версию JRE и настроить дополнительные параметры.

Пример нового проекта представлен на рисунке 4.

Для того чтобы открыть уже существующий код простого проекта есть два пути:

1) скопировать весь проект в рабочий каталог и произвести импорт (**File** (Файл) – **Import** (Импорт) – **General** (Основное) – **Existing Projects into Workspace** (Существующий проект в рабочее пространство));
2) создать новый проект и скопировать исходные java-файлы в каталог «src» нового проекта.

При импорте существующего проекта (первый вариант) возможны конфликты в настройках JRE импортируемого проекта и используемого Eclipse, поэтому для простых проектов рекомендуется использовать второй вариант.

Для создания нового пакета в выпадающем меню на папке «src» (или уже имеющегося пакета), выбирается пункт меню **New** (Новый) – **Package** (Пакет). Аналогично пакету создаем новый класс. Диалоговое окно создания нового класса приведено на рисунке 5.

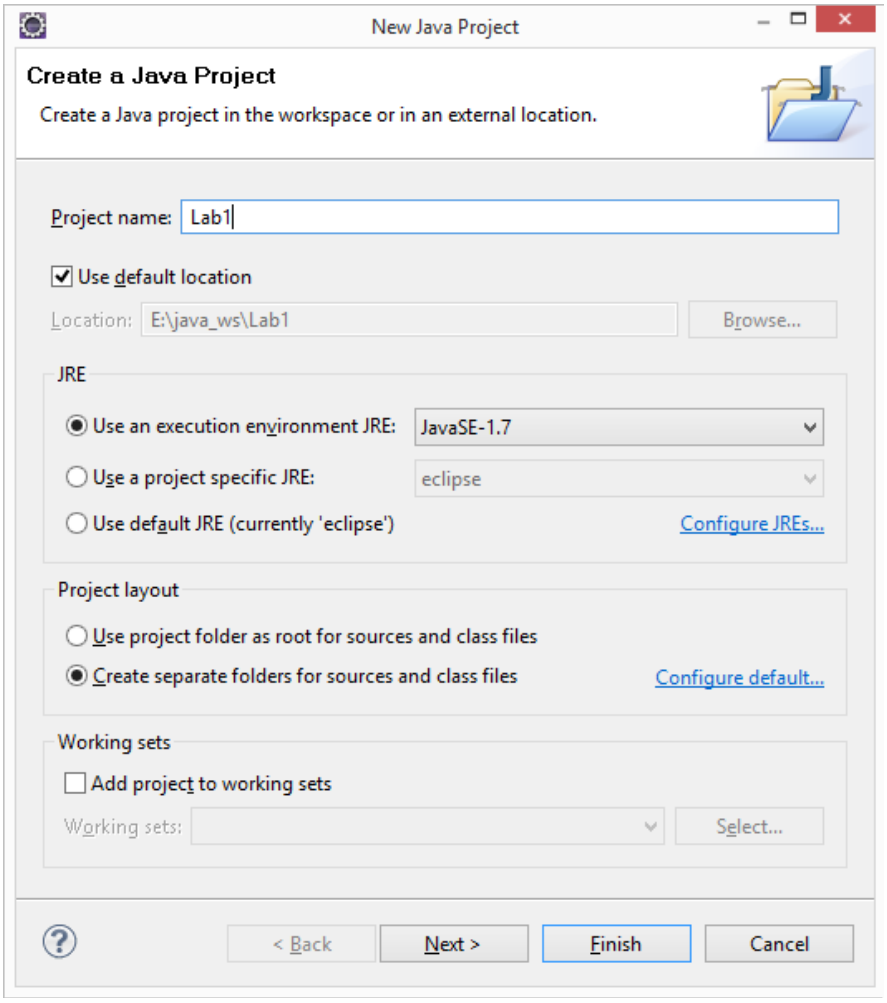

Рисунок 3 – Окно создания проекта

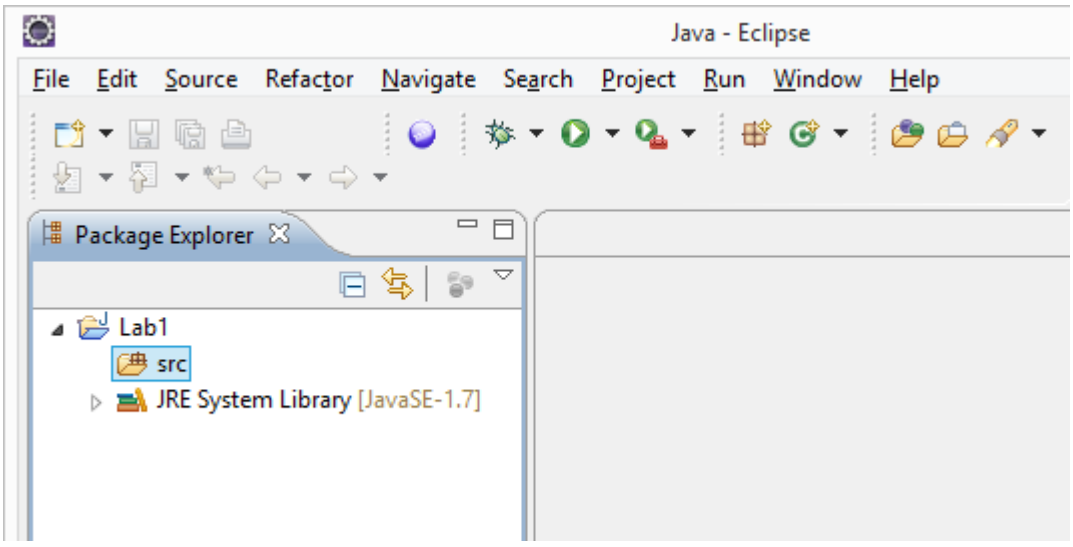

Рисунок 4 – Новый проект

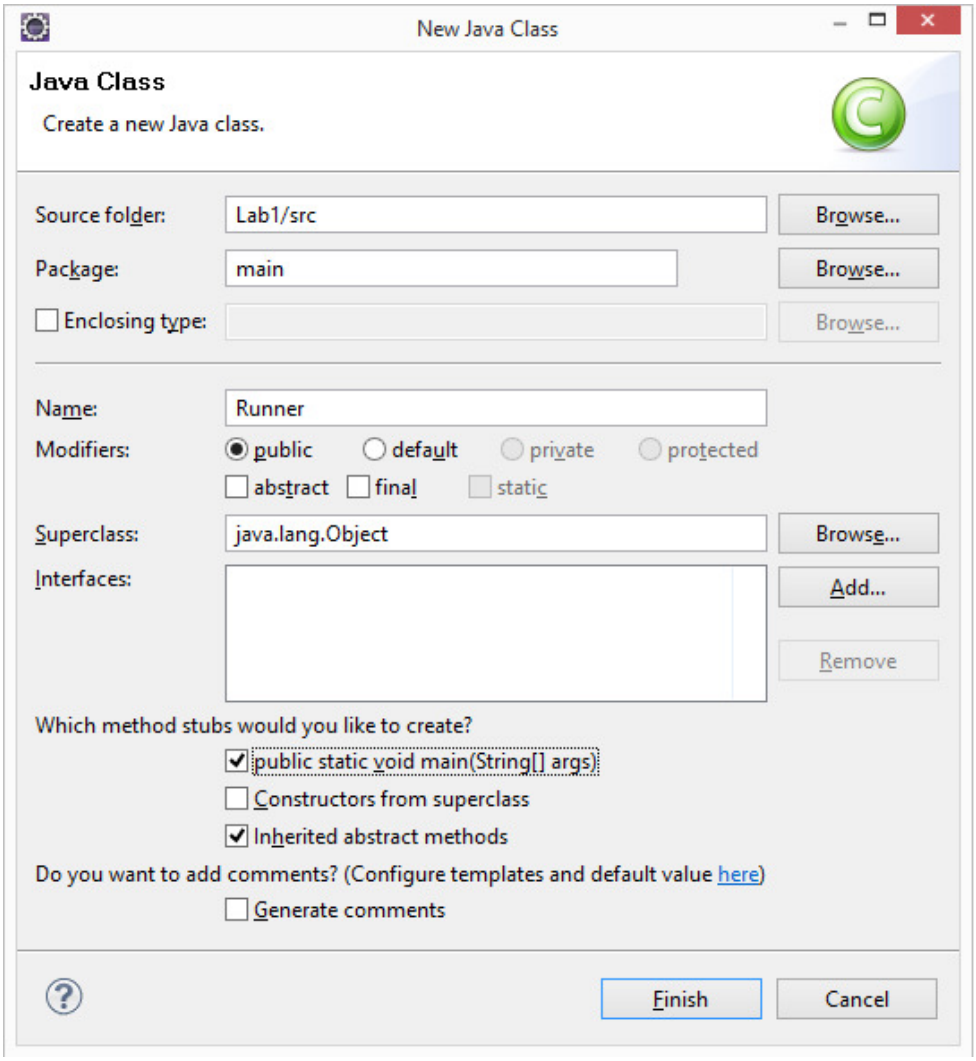

Рисунок 5 – Окно создания класса

На закладке редактирования текста класса вводится требуемый код. Запуск простого проекта осуществляется кнопкой **Run** (Выполнить). Пример текста простого приложения и результат его выполнения в Eclipse приведен на рисунке 6.

На рисунке 7 показана структура каталогов, полученная в результате создания проекта и пакета в нем. По умолчанию файлы с кодом классов (java-файлы) находятся в каталоге «src», скомпилированные class-файлы находятся в каталоге «bin». Настроить эти правила можно в свойствах проекта.

Кодировка исходных программных файлов настраивается в свойствах проекта или рабочего пространства. Наиболее распространенные кодировки – UTF-8 и cp1251.

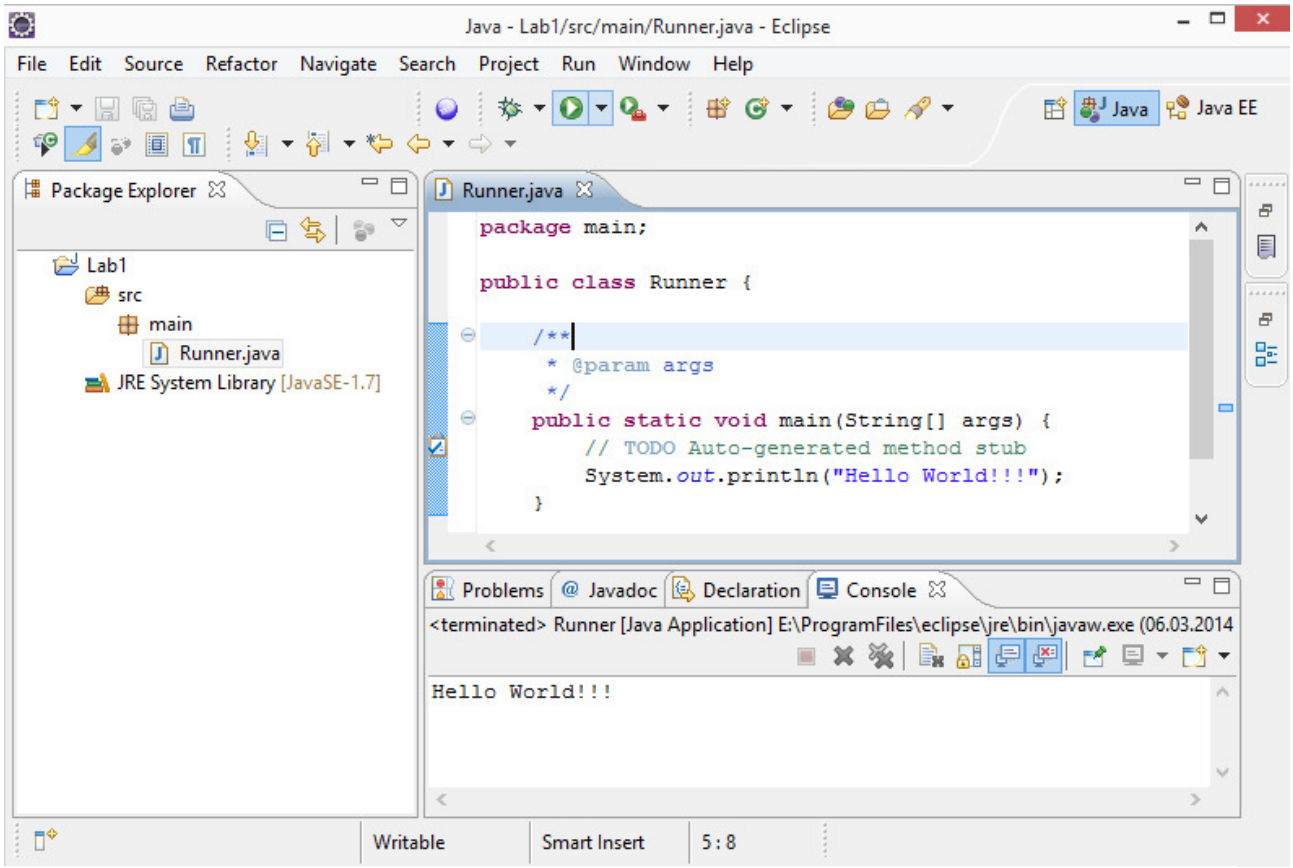

Рисунок 6 – Запуск простого приложения в Eclipse

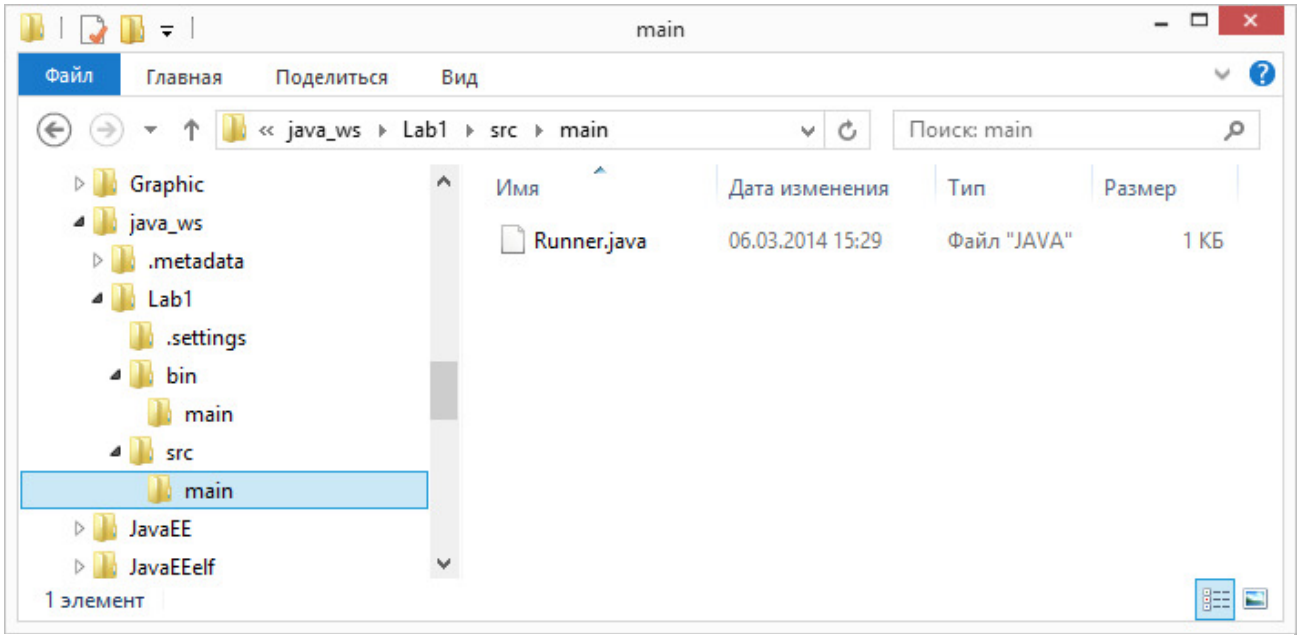

Рисунок 7 – Структура каталогов проекта

1.2 Требования к оформлению кода

В настоящее время существует много стандартов наименования переменных, но два из них являются наиболее популярными среди программистов: это «camel case» («верблюжья» нотация) и «underscore» (именование переменных с использованием символа нижнего подчеркивания в качестве разделителя).

В языке Java стандартом является «верблюжья» нотация. Согласно этому стандарту, все слова в названии начинаются с прописной буквы, кроме первого. При этом не используется никаких разделителей вроде нижнего подчеркивания, в именах классов (структур, интерфейсов) первое слово также начинается с прописной буквы.

Названия констант пишутся полностью прописными буквами, в качестве разделителя слов используется символ подчеркивания.

Отступы, открывающие/закрывающие скобки легко форматируются в Eclipse сочетанием клавиш **Ctrl-Shift-F** (Контрл-Шифт-F). Правила форматирования для проекта определяются в его свойствах **Java Code Style** (Стиль Java кода) – **Formatter** (Форматирование). Где можно выбрать встроенный стиль оформления **Java Code Conventions** (Соглашения о Java-коде) (рисунок 8).

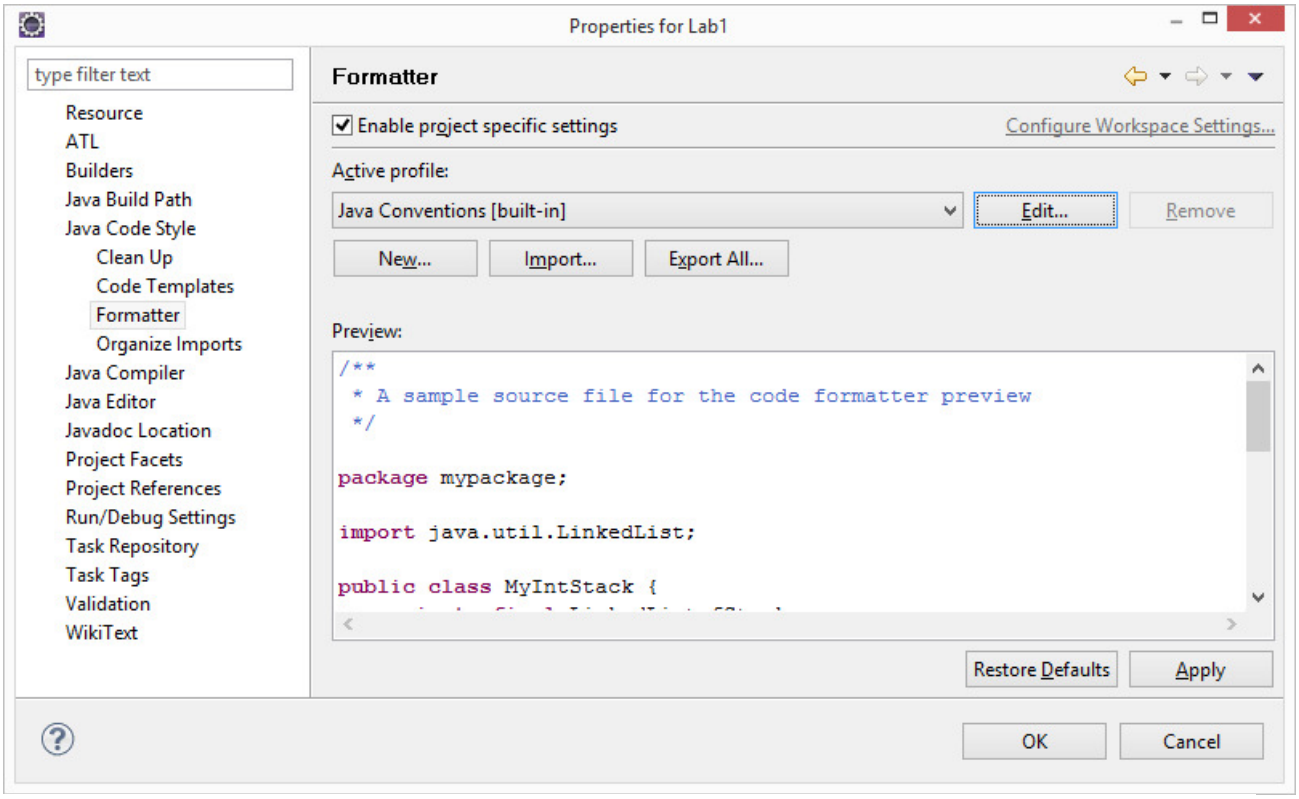

Рисунок 8 – Окно свойств проекта

2 Рекомендации по выполнению лабораторных работ

2.1 Лабораторная работа №1. Простое консольное приложение

Аргументы командной строки передаются в приложение через параметр `argv (массив строк)` метода

```
public static void main(String[] argv)
```

Настроить передаваемые параметры в Eclipse можно через свойства проекта **Run** (Выполнить) **– Debug Settings** (Параметры отладки) – Имя Класса, кнопка **Edit** (Редактировать), закладка **Arguments** (Аргументы) (рисунок 9).

В Java используются объектные и примитивные типы данных.

Примитивные типы данных Java приведены в таблице 1.

Таблица 1 – примитивные типы данных Java

Тип	Размер (бит)	По умолчанию	Мин	Макс	Тип упаковки
boolean	-	false	-	-	Boolean
char	16	'\u0000'	Unicode 0	Unicode $2^{16}-1$	Character
byte	8	0	-128	127	Byte
short	16	0	-2^{15}	$2^{15}-1$	Short
int	32	0	-2^{31}	$2^{31}-1$	Integer
long	64	0	-2^{63}	$2^{63}-1$	Long
float	32	0.0	IEEE754	IEEE754	Float
double	64	0.0	IEEE754	IEEE754	Double
void	-		-	-	Void

Все примитивные типы можно «упаковывать» в специальные классы-оболочки, тем самым приводя к объектному типу.

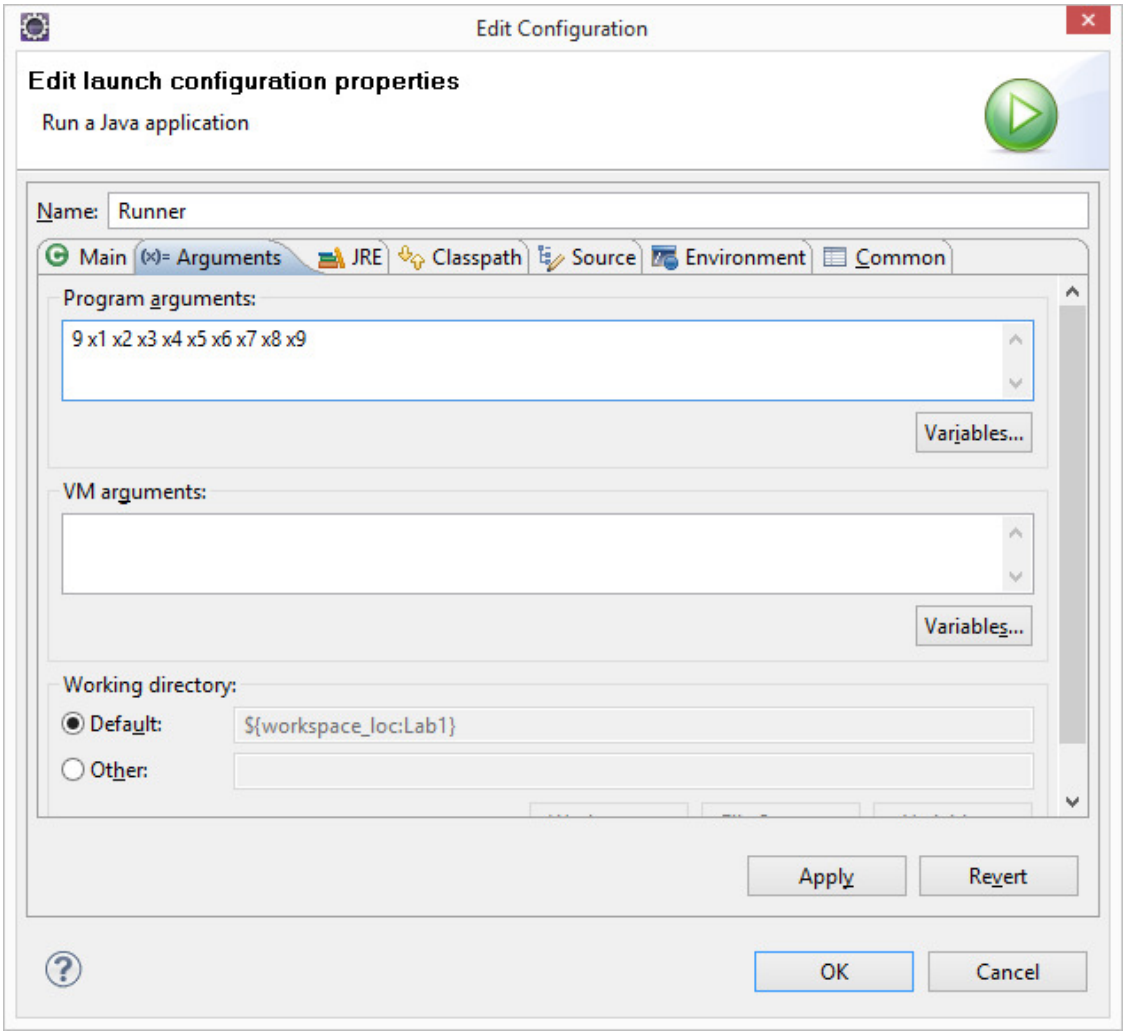

Рисунок 9 – Окно свойств проекта

Преобразование строки в число возможно функцией:

```
Byte.valueOf(str);
Integer.valueOf(str);
Float.valueOf(str);
Double.valueOf(str);
```

или при помощи конструктора:

```
new Byte(str);
new Integer(str);
new Float(str);
new Double(str);
```

Логические операторы и операторы сравнения Java приведены в таблице 2.

Таблица 2 – Логические операторы и операторы сравнения Java

Оператор	Действие	Оператор	Действие
<	Меньше	>	Больше
<=	Меньше или равно	>=	Больше или равно
==	Равно	!=	Не равно
||	Или	&&	И
!	Унарное отрицание		

Арифметические операторы Java приведены в таблице 3.

Таблица 3 – Арифметические операторы Java

Оператор	Действие	Оператор	Действие
+	Сложение	/	Деление
+=	Сложение (с присваиванием)	/=	Деление (с присваиванием)
-	Вычитание	%	Остаток от деления
-=	Вычитание (с присваиванием)	%=	Остаток от деления (с присваиванием)
*	Умножение	++	Инкремент
*=	Умножение (с присваиванием)	--	Декремент

Операторы управления в Java включают в себя операторы ветвления, выбора и перебора.

Полный оператор ветвления:

```
if (boolexp) { /*операторы*/}
else { /*операторы*/ }
```

Полный оператор выбора:

```
switch(exp) {
    case exp1:{/*операторы*/}
    break;
    case expN:{/*операторы*/}
    break;
    default: {/*операторы*/}
}
```

Операторы перебора:

1) цикл с предусловием

```
while (boolexp) { /*операторы*/ }
```

2) цикл с постусловием

```
do { /*операторы*/ } while (boolexp);
```

3) инкрементный перебор элементов с условием

```
for(exp1; boolexp; exp3){ /*операторы*/ }
```

4) перебор элементов коллекции (массива)

```
for(ТипДанных имя : имяОбъекта){ /*операторы*/ }
```

Класс содержит описание данных и операций над ними. В классе дается обобщенное описание некоторого набора родственных, реально существующих объектов.

Объект – конкретный экземпляр класса, т.е. обычно обладающий именем набор данных (полей объекта), физически находящихся в памяти компьютера, и методов, имеющих доступ к ним. Имя объекта (переменной) используется для доступа к полям и методам, составляющим объект

Пример 1. В качестве аргументов командной строки получить N – количество строк, затем получить эти строки и вывести каждую вторую в обратном порядке.

Решение.

Ввод аргуметров командной строки отображен на рисунке 2.1.1.

```
package main;

public class Task1 {

    // точка входа в программу
    public static void main(String[] args) {

        // проверяем наличие аргументов командной строки
        if (args.length > 0) {

            // первый аргумент должен содержать количество строк
            // преобразуем содержимое первого аргумента в число
            int n = new Integer(args[0]);

            // оставшиеся аргументы в обратном порядке через один
            for (int i = n; i > 0; i -= 2) {

                // выводим в консоль
                System.out.print(args[i] + " ");
            }
        }
    }
}
```

Результат работы приложения – строка в консоли:

```
x9 x7 x5 x3 x1
```

Пример 2. Создать класс *песня* с полями *название* и *продолжительность*. Создать главный класс с методом *main*, в котором получить в качестве аргументов командной строки название и продолжительность, создать объект *песня*, заполнить соответствующие поля, вывести значение объекта *песня* в консоль.

Решение.

Параметры командной строки устанавливаются, как показано на рисунке 2.1.1. Например: `Hero 4`

Оба класса можно описать в одном java-файле.

```
package main;

/** класс Песня */
class Song {

    private String title; // название
    private int min; // продолжительность

    // получить значение названия
    public void setTitle(String title) {
        this.title = title;
    }

    // получить значение продолжительности
    public void setMin(int min) {
        this.min = min;
    }

    // метод наследуется от класса Object и должен быть
    // переопределен для представления информации об объекте
    @Override
    public String toString() {
        return "Song: название " + title + ", продолжительность "
                + min;
    }
}

/** главный класс */
public class Task2 {

    // точка входа приложения
    public static void main(String[] args){

        // проверка количества аргументов командной строки
        if (args.length == 2) {

            // создание объекта класса Песня
            Song song = new Song();

            // установка значения поля Название
            song.setTitle(args[0]);

            // установка значения поля Продолжительность
            // значение из командной строки преобразуется к
            // числовому типу
            song.setMin(new Integer(args[1]));

            // вывод в консоль описания объекта
            System.out.print(song.toString());
        }
    }
}
```

Результат работы приложения – строка в консоли:

```
Song: название Hero, продолжительность 4
```

2.2 Лабораторная работа №2. Простые классы

Классы в языке Java содержат конструкторы, поля, методы, логические блоки, внутренние классы. Все функции определяются внутри классов и называются методами. Спецификаторы доступа (`public, private, protected`) воздействуют только на те объявления полей, методов и классов, перед которыми они стоят, а не на участок от одного до другого спецификатора. Элементы по умолчанию не устанавливаются в `private`, а доступны для классов из данного пакета.

Определение простейшего **класса** выглядит следующим образом:

```
class ИмяКласса {
    { } // логические блоки
        // дружественные данные и методы
    private   // закрытые данные и методы
    protected // защищенные данные и методы
    public    // открытые данные и методы
}
```

Общая спецификация класса выглядит как:

```
[спецификаторы] class ИмяКласса [extends СуперКласс]
[implements список_интерфейсов]
```

Спецификаторами к классу могут быть:

– `public` – класс доступен в данном пакете и вне пакета;

– `final` – класс не может иметь подклассов;

– `abstract` – класс может содержать абстрактные методы, объект такого класса создать нельзя.

По умолчанию спецификатор устанавливается как дружественный (friendly). Такой класс доступен только в текущем пакете.

Конструктор – это метод, который автоматически вызывается при создании объекта класса и выполняет действия по инициализации объекта. Конструктор имеет то же имя, что и класс, вызывается не по имени, а только вместе с ключевым словом **new** при создании экземпляра класса. Конструктор не возвращает значение, но может иметь параметры и быть перегружаемым.

Оператор **new** вызывает конструктор, поэтому в круглых скобках могут стоять аргументы, передаваемые конструктору.

Если конструктор в классе не определен, Java предоставляет конструктор по умолчанию без параметров, который инициализирует поля класса значениями по умолчанию, например.

Если же конструктор с параметрами определен, то *конструктор по умолчанию становится недоступным* и для его вызова необходимо явное объявление такого конструктора.

Конструктор подкласса всегда вызывает конструктор суперкласса.

Этот вызов может быть явным или неявным и всегда располагается в первой строке кода конструктора.

Если конструктору суперкласса нужно передать параметры, то необходим явный вызов:

```
super(параметры);
```

Объявление **поля класса** выглядит следующим образом:

```
[спецификаторы] тип имя;
```

Одновременно можно применить **спецификаторы поля**:

– `static` – статическая переменная класса, такие поля общие для всех объектов класса и называются переменными класса;

– `final` – константа;

– спецификатор доступа (`public, private, protected`).

Поля, объявленные как **final**, не имеют значения по умолчанию и должны быть проинициализированы только один раз в одном из следующих мест:

– при объявлении;

– в логическом блоке;

– в конструкторе.

Модификатор **final** также может быть использован при объявлении локальных переменных внутри метода и при описании параметров методов. Значения констант нельзя изменять при помощи оператора присваивания.

Простейшее **определение метода** выглядит следующим образом:

```
ТипВозвращаемогоЗначения имяМетода(список_параметров) {
    // тело метода
    return value; // если нужен возврат значения
                  //т.е. ТипВозвращаемогоЗначения не void
}
```

Вызов методов осуществляется из объекта или класса:

```
имяОбъекта.имяМетода(); ИмяКласса.имяМетода();
```

Общее определение метода

```
[спецификаторы] ТипВозвращаемогоЗначения имяМетода
(список_параметров) [throws список_исключений]{
    // тело метода
    return value;
}
```

Переменная базового типа всегда передается в метод по значению, а переменная класса-оболочки – по ссылке.

Одновременно можно применить с**пецификаторы к методу**:

– спецификатор доступа (`public, private, protected`).

– `static` – статический метод, такие методы называются методами класса, не привязаны ни к какому объекту и не содержат указателя **this** на конкретный объект, вызвавший метод;

– `abstract` – абстрактный метод, не имеет реализации (тела), такие методы должны быть реализованы в подклассах, класс с абстрактным методом должен быть тоже абстрактным;

– `final` – метод нельзя замещать (переопределять) в подклассах;

– `native` – метод реализован в другом месте на языке C++;

– `synchronized` – метод синхронизирован для работы в разных потоках.

Обращаться к статическим членам класса следует через имя класса.

Логические блоки могут быть статическими и не статическими:

```
static { /* код */ }
{ /* код */ }
```

Они используются в качестве инициализаторов полей, могут содержать вызовы методов и обращения к полям текущего класса.

Блоки **static** вызываются только один раз в жизненном цикле приложения при загрузке класса (при создании первого объекта или при обращении к статическому члену класса).

Код логических блоков вызывается на выполнение последовательно, в порядке размещения блоков, сначала все статические, потом остальные. После выполнения последнего блока вызывается конструктор класса.

Операции с полями класса внутри логического блока до явного объявления этого поля возможны только при использовании ссылки **this**.

Переопределение и перегрузка методов. Метод называется *перегруженным*, если существует несколько его версий с одним и тем же именем, но с различным списком параметров. *Переопределение* – объявление методов с одинаковыми именами и совпадающими списками параметров в разных классах одной цепочки наследования.

Статические методы могут перегружаться нестатическими, и наоборот без ограничений. Перегрузка реализует *раннее связывание*.

Массив в Java представляет собой объект, где имя массива является объектной ссылкой. Элементами массива могут быть значения базового типа или объекты. Индексирование элементов начинается с нуля.

Все массивы в языке Java динамические, поэтому для создания массива требуется выделение памяти с помощью оператора **new** или прямой инициализации.

Значения элементов неинициализированного массива, для которого выделена память, устанавливаются в значения по умолчанию для массива базового типа или **null** для массива объектных ссылок.

Размерность массива хранится в его свойстве **length**.

У класса **System** есть статический метод, ***arraycopy*** который копирует данные из одного массива в другой:

public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

Object src - массив откуда копировать

Object dest - массив куда копировать

int srcPos - исходная позиция в исходном массиве

int destPos - исходная позиция в массиве адресата

int length - число элементов для копирования

Пример. Создать класс **Book**, содержащий информацию: Название, Автор(ы), Издательство, Год издания, Количество страниц, Цена. В этом классе объявить несколько конструкторов и методы **set***Поле*(), **get***Поле*(), **toString**(). Создать класс, содержащий массив книг, и имеющий методы для выбора книг в соответствии с критерием выбора: по автору, по издательству, позже заданного года.

Задать критерии выбора данных и вывести эти данные на консоль.

Решение. Описание класса книги:

```
package task02;

/** класс Книга */
public class Book {
    private String title;       // название
    private String[] authors;   // список авторов
    private String publisher;   // издательство
    private int year;           // год издания
    private int pageCount;      // количество страниц
    private double cost;        // стоимость

    /** конструктор по умолчанию без параметров,
        его надо описать явно, поскольку
        у класса есть конструкторы с параметрами */
```

```
    public Book() {
        title = "";
        authors = new String[0];
        publisher = "";
    }
    /** конструктор с инициализацией всех полей */
    public Book(String title, String[] authors, String publisher,
            int year, int pageCount, double cost) {
        super();
        this.title = title;
        this.authors = authors;
        this.publisher = publisher;
        this.year = year;
        this.pageCount = pageCount;
        this.cost = cost;
    }
    /** получить значение title*/
    public String getTitle() {
        return title;
    }
    /** установить значение title*/
    public void setTitle(String title) {
        this.title = title;
    }
    // ...
    // аналогичные методы доступа к полям класса
    /* функция для определения
       есть ли фамилия в списке авторов книги */
    public boolean hasAuthor(String author) {
        boolean res = false;
        int i = 0;
        // пока не найден автор и не вышли за пределы массива
        while ((!res) && (i < authors.length)) {
            // res будет true в случае,
            // если фамилия текущего автора
            // совпадает с искомой фамилией
            res = author.equals(authors[i]);
            i++;
        }
        return res;
    }
    /** метод для получения списка авторов в виде строки */
    private String getListAuthors() {
        String list = "";
        for (int i = 0; i < authors.length; i++) {
            list += authors[i] + ", ";
        }
        return list;
    }
    @Override
    public String toString() {
        return "Книга: название=" + title + ", авторы=["
                + getListAuthors() + "], издатель="
                + publisher + ", год издания=" + year
                + ", кол-во страниц=" + pageCount
                + ", стоимость=" + cost + "]";
    }
}
```

```
package task02;
public class BookArray {
    private Book books[];
    public BookArray(Book[] books) {
        this.books = books;
    }
    public Book[] getBooks() { return books; }
    public void setBooks(Book[] books) { this.books = books; }

    /** поиск книг определенного автора */
    public BookArray getByAuthor(String author) {
        Book[] list = new Book[books.length];
        int count = 0;
        for (Book b : books) {
            if (b.hasAuthor(author)) {
                list[count] = b;
                count++;
            }
        }
        Book[] res = new Book[count];
        System.arraycopy(list, 0, res, 0, count);
        return new BookArray(res);
    }
    //  . . .
    // аналогично выглядят методы поиска по другим условиям
    @Override
    public String toString() {
        String res = "";
        for (int i = 0; i < books.length; i++) {
            res += books[i].toString() + '\n';
            // '\n' добавляется для перехода на новую строку
        }
        return res;
    }
}

package task02;
public class Task02 {
    public static void main(String[] args) {
      Book books[] = new Book[4];
      String[] author1 = { "Гради Буч" };
      books[0] = new Book("Объектно-ориентированный анализ...",
      author1, "Вильямс", 2010, 720, 534.6);
      String[] author2 = { "Гради Буч", "Джеймс Рамбо", "Ивар Якобсон"};
      books[1] = new Book("Язык UML. Руководство пользователя", author2,
                 "ДМК Пресс", 2007, 496, 425.9);
      books[2] = new Book("UML. Классика CS", author2, "Питер", 2006,
                 736, 648.5);
      books[3] = new Book("Введение в UML от создателей языка", author2,
                 "ДМК Пресс", 2011, 496, 328.6);
      BookArray bookArray = new BookArray(books);
      System.out.println("\n-- весь список --");
      System.out.println(bookArray.toString());
      System.out.println("\n-- по автору --");
      System.out.println(bookArray.getByAuthor("Джеймс Рамбо"));
      System.out.println("\n-- по издательству --");
      System.out.println(bookArray.getByPublisher("ДМК Пресс"));
    }
}
```

2.3 Лабораторная работа №3. Наследование и полиморфизм

Подкласс **наследует** переменные и методы суперкласса, используя ключевое слово **extends**. Класс может также реализовывать любое число интерфейсов, используя ключевое слово – **implements**.

Полиморфизм – возможность работать с несколькими типами так, как будто это один и тот же тип и в то же время поведение каждого типа будет уникальным в зависимости от его реализации.

Ключевое слово **super** используется для вызова конструктора суперкласса и для доступа к члену суперкласса.

Каждый объект (экземпляр класса) имеет неявную ссылку **this** на себя, которая передается также и методам. После этого метод «знает», какой объект его вызвал.

Способность Java делать выбор метода, исходя из типа объекта во время выполнения, называется *поздним связыванием*. Выбор версии переопределенного метода производится на этапе выполнения кода.

Все методы Java являются виртуальными (ключевое слово virtual, как в C++, не используется).

Статические методы могут быть переопределены в подклассе, но не могут быть полиморфными, так как их вызов не затрагивает объекты. Их следует вызывать только с использованием имени класса.

При использовании ссылки для доступа к статическому члену компилятор при выборе метода или поля учитывает тип ссылки, а не тип объекта, ей присвоенного.

Абстрактные классы объявляются с ключевым словом **abstract** и могут содержать объявления абстрактных методов, которые не реализованы в этих классах, а будут реализованы в подклассах. Абстрактные классы могут содержать и полностью реализованные методы, а также конструкторы и поля данных.

Объекты абстрактных классов создать нельзя, но можно создать объекты подклассов, которые реализуют эти методы.

При этом допустимо объявлять ссылку на абстрактный класс, но инициализировать ее можно только объектом производного от него класса.

Все классы Java наследуют класс Object и, соответственно, все его методы. Из этих методов следует выделить методы **equals**(Object ob) и **hashCode**().

Переопределение этих методов необходимо, если при создании класса предполагается проверка логической эквивалентности объектов, которая

не выполнена в суперклассе, или логика приложения предусматривает использование элементов в коллекциях

Метод **equals** при сравнении двух объектов должен возвращать истину, если содержимое объектов эквивалентно, и ложь – в противном случае.

При переопределении метода **equals** должны выполняться правила:

– *рефлексивность* – объект равен самому себе;

– *симметричность* – если **x.equals(y)** возвращает значение **true**, то и **y.equals(x)** всегда возвращает значение **true**;

– *транзитивность* – если метод **equals()** возвращает значение **true** при сравнении объектов **x** и **y**, а также **y** и **z**, то и при сравнении **x** и **z** будет возвращено значение **true**;

– *непротиворечивость* – при многократном вызове метода для двух неподвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;

– ненулевая ссылка при сравнении с литералом **null** всегда возвращает значение **false**.

Метод **hashCode** переопределен, как правило, в каждом классе и возвращает число, являющееся уникальным идентификатором объекта, зависящим в большинстве случаев только от значения объекта. Его следует переопределять всегда, когда переопределен метод **equals**.

Метод **hashCode** возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

– во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен;

– все одинаковые по содержанию объекты одного типа **должны** иметь одинаковые хэш-коды;

– различные по содержанию объекты одного типа **могут** иметь различные хэш-коды.

Метод **toString()** класса **Object** возвращает строку с описанием объекта в виде:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Метод **toString** вызывается автоматически, когда объект выводится в поток вывода методами **println()**, **print()** и некоторыми другими.

Метод **toString** следует переопределять таким образом, чтобы он возвращал:

– стандартную информацию о пакете, в котором находится класс;

– само имя класса;

– значения полей объекта (то есть всю полезную информацию объекта) вместо хэш-кода, как это делается в классе **Object**.

Пример. Определить иерархию холодного оружия. Для каждого класса иерархии переопределить методы **toString**(), **equals**(), **hashCode**(). Создать класс Арсенал для работы с массивом оружия с методами:

– добавить новое оружие;
– получить список всего оружия;
– получить все оружие, сделанное только из заданного материала;
– получить оружие определенного типа;
– найти оружие, идентичное заданному;
– найти суммарный вес всего оружия.

Для класса Арсенал переопределить метод **toString**(). Заполнить арсенал, обеспечить поиск оружия по параметрам. Найти общий вес оружия.

Решение.

```
package task03.coldarm;

/** холодное оружие */
public abstract class ColdArm {
    private String name; // название
    private String material; // материал
    private int weigth; // вес

    /** конструктор по умолчанию */
    public ColdArm() {
        name = "noname " + this.getClass().getSimpleName();
    }

    /** конструктор с параметорм */
    public ColdArm(String name) {
        setName(name);
    }

    /** получить название */
    public String getName() {
        return name;
    }

    /** установить название */
    public void setName(String name) {
        this.name = name;
    }

    public int getWeigth() {
        return weigth;
    }

    public String getMaterial() {
        return material;
    }
```

```
    public void setMaterial(String material) {
        this.material = material;
    }

    public void setWeigth(int weigth) {
        this.weigth = weigth;
    }

    @Override
    public String toString() {
        return this.getClass().getName() + "\t name = " +
getName()
                + "; weigth = " + getWeigth() + "; material = " +
getMaterial();
    }

    @Override
    public int hashCode() {
        return weigth + 2 * name.hashCode() + 3 *
material.hashCode();
    }

    @Override
    public boolean equals(Object o) {
        if (o == null) {
            return false;
        } else if (this == o) {
            return true;
        } else if (o.getClass() == this.getClass()) {
            ColdArm ca = (ColdArm) o;
            // строки надо сравнивать через equals()
            if (ca.getName().equals(this.getName())
                    && ca.getMaterial().equals(this.getMaterial())
                    && ca.getWeigth() == this.getWeigth()) {
                return true;
            } else
                return false;
        } else
            return false;
    }
}

package task03.coldarm;

/** Меч */
public class Sword extends ColdArm {
    private int length; // длина меча

    public Sword() {
    }

    public Sword(String name) {
        // вызов конструктора суперкласса
        super(name);
    }

    public int getLength() {
        return length;
    }
```

```
    /** изменить значение длины меча */
    public void setLength(int length) {
        // существует логическое ограниччение на длину
        if (length > 10 && length < 500) {
            this.length = length;
        } else
            this.length = 20;
    }

    @Override
    public String toString() {
        // сначала вызывается метод суперкласса
        return super.toString() + "; length = "
                + this.getLength();
    }

    @Override
    public int hashCode() {
        // сначала вызывается метод суперкласса
        return super.hashCode() + 4 * this.getLength();
    }

    @Override
    public boolean equals(Object o) {
        // сначала вызывается метод суперкласса
        if (super.equals(o)) {
            Sword s = (Sword) o;
            if (s.getLength() == this.getLength()) {
                return true;
            } else
                return false;
        } else
            return false;
    }
}

package task03.coldarm;
/** Арбалет */
public class Arbalest extends ColdArm {
    private int arrowCount = 0; // количество стрел
    // материал стрел, по умолчанию стрелы деревянные
    private String arrowMaterial = "wood";

    public Arbalest() {

    }

    public Arbalest(String name) {
        super(name);
    }

    public int getArrowCount() {
        return arrowCount;
    }

    public void setArrowCount(int arrowCount) {
        this.arrowCount = arrowCount;
    }
```

```
    public String getArrowMaterial() {
        return arrowMaterial;
    }

    public void setArrowMaterial(String arrowMaterial) {
        this.arrowMaterial = arrowMaterial;
    }

    public String toString() {
        return super.toString() + "; arrowMaterial = "
                + this.getArrowMaterial() + "; arrowCount = " +
getArrowCount();
    }

    public int hashCode() {
        return super.hashCode() + 4 * arrowCount + 5
                * getArrowMaterial().hashCode();
    }

    public boolean equals(Object o) {
        if (super.equals(o)) {
            Arbalest a = (Arbalest) o;
            if (a.getArrowCount() == this.getArrowCount()
                    &&
a.getArrowMaterial().equals(this.getArrowMaterial())) {
                return true;
            } else
                return false;
        } else
            return false;
    }
}

package task03.arsenal;

import task03.coldarm.*; // импорт всех классов из пакета
/** Арсенал */
public class Arsenal {
    private ColdArm[] arms = new ColdArm[0];

    public void clear() {
        arms = new ColdArm[0];
    }

    /** возвращает список оружия */
    public ColdArm[] getArms() {
        return arms;
    }

    /** добавляет новое оружие */
    public void addArm(ColdArm ca) {
        ColdArm[] temp = new ColdArm[arms.length + 1];
        System.arraycopy(arms, 0, temp, 0, arms.length);
        temp[arms.length] = ca;
        arms = temp;
    }
```

```
    /** добавляет список нового оружия */
    public void addArm(ColdArm[] ca) {
        ColdArm[] temp = new ColdArm[arms.length + ca.length];
        System.arraycopy(arms, 0, temp, 0, arms.length);
        System.arraycopy(ca, 0, temp, arms.length, ca.length);
        arms = temp;
    }

    /** возвращает суммарный вес оружия */
    public int getAllWieght() {
        int res = 0;
        for (ColdArm ca : arms) {
            res += ca.getWeigth();
        }
        return res;
    }

    /** возвращает только арбалеты */
    public ColdArm[] getArbalests() {
        Arsenal res = new Arsenal();
        for (ColdArm ca : arms) {
            // используется оператор
            // принадлежности объектному типу
            if (ca instanceof Arbalest) {
                res.addArm(ca);
            }
        }
        return res.getArms();
    }

    /** возвращает только оружие из заданного материала */
    public ColdArm[] searchMaterial(String material) {
        Arsenal res = new Arsenal();
        for (ColdArm ca : arms) {
            if (material.equals(ca.getMaterial())) {
                res.addArm(ca);
            }
        }
        return res.getArms();

    }

    /** находит все оружие идентичное заданному */
    public ColdArm[] searchEqual(ColdArm cl) {
        Arsenal res = new Arsenal();
        for (ColdArm ca : arms) {
            if (ca.equals(cl)) { res.addArm(ca); }
        }
        return res.getArms();
    }

    @Override
    public String toString() {
        String str = new String();
        for (ColdArm a : arms) { str += a + "\n"; }
        return str;
    }
}
```

```
package task03.run;
import task03.arsenal.Arsenal;
import task03.coldarm.*;
/** точка входа */
public class Runner {

    public static void main(String[] args) {
        Arsenal ars = new Arsenal();
        Sword ca1 = new Sword("Arthur");
        ca1.setMaterial("wood");
        ca1.setLength(50);
        ca1.setWeigth(100);
        ars.addArm(ca1);

        Arbalest ar1 = new Arbalest("Robin");
        ar1.setMaterial("steel");
        ar1.setWeigth(40);
        ar1.setArrowCount(100);
        ar1.setArrowMaterial("steel");
        ars.addArm(ar1);

        Sword ca2 = new Sword("Jaina");
        ca3.setMaterial("silver");
        ca3.setLength(40);
        ca3.setWeigth(75);
        ars.addArm(ca3);

        Arbalest ar2 = new Arbalest("VanHelsing");
        ar2.setMaterial("wood");
        ar2.setWeigth(20);
        ar2.setArrowCount(200);
        ar2.setArrowMaterial("silver");
        ars.addArm(ar2);

        ColdArm[] arr = { ar1, ar2, ca1 };
        ars.addArm(arr);

        System.out.println("all list:");
        System.out.println(ars);

        Arsenal temp = new Arsenal();
        System.out.println("only Arbalest:");
        temp.clear();
        temp.addArm(ars.getArbalests());
        System.out.println(temp);

        System.out.println("search by material 'wood' :");
        temp.clear();
        temp.addArm(ars.searchMaterial("wood"));
        System.out.println(temp);

        System.out.println("summary weight = "
                + ars.getAllWieght());

    }

}
```

2.4 Лабораторная работа №4. Применение внутренних классов

В Java можно определить (вложить) один класс внутри определения другого класса, что позволяет группировать классы, логически связанные друг с другом, и динамично управлять доступом к ним.

Внутренние классы бывают нестатические – внутренние (**inner**) и статические – вложенные (**nested**).

Объект внутреннего (**inner**) класса всегда ассоциируется (скрыто хранит ссылку) с создавшим его объектом внешнего класса – внешним (enclosing) объектом. Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса. Доступ к элементам внутреннего класса возможен из внешнего класса только через объект внутреннего класса. Внутренние классы **не могут** содержать статические атрибуты и методы, кроме констант (`final static`).

После компиляции объектный модуль, соответствующий внутреннему классу, получит имя `Владелец$Внутренний.class`.

К **inner** классу можно применить с**пецификаторы**:

– спецификатор доступа (`public, private, protected`).

– `abstract` – абстрактный класс;

– `final` – класс нельзя наследовать.

Внутренний класс может быть объявлен также внутри метода или логического блока внешнего класса. Видимость такого класса регулируется областью видимости блока, в котором он объявлен.

Внутренний класс внутри метода или блока сохраняет доступ ко всем полям и методам внешнего класса и *константам*, объявленным в текущем блоке кода. Класс, объявленный внутри метода, **не может** быть объявлен статическим, содержать статические поля и методы.

При объявлении **вложенного (nested) класса** присутствует служебное слово **static**.

Статический вложенный класс напрямую имеет доступ только к статическим полям и методам внешнего класса, для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса

Внутренние статические и нестатические (nested и inner) классы способны наследовать другие классы, реализовывать интерфейсы, являться объектом наследования для любого класса, обладающего необходимыми правами доступа.

Подкласс внутреннего класса не способен унаследовать возможность доступа к членам внешнего класса, которыми наделен его суперкласс.

Существуют также **анонимные (anonymous) классы**. Объявление анонимного класса выполняется одновременно с созданием его объекта посредством оператора **new**.

```
package inner;
public class Runner {
    public static void main(String[] args) {
        Object o = new Object() {// анонимный класс
                                 //наследник Object
            // переопределение существующего метода
            public String toString() {
                System.out.println("Анонимный класс: Привет!");
                return "";
            }
        };
        // вызов переопределенного метода
        o.toString(); // тут нельзя вызвать метод,
                      // не объявленный в супер-классе

        final int MAX = 100;
        new Object() {// анонимный класс
                      // наследник Object
            // определение нового поля
            int newField = MAX / 2; // внутренний класс имеет
                                    // доступ к константам
                                    // текущего блока кода
            // определение нового метода
            void printNewField() {
                System.out.println("newField = " + newField);
            }
        }.printNewField(); // тут можно вызвать новый метод
    }
}
```

Анонимные классы эффективно используются, как правило, для реализации (переопределения) нескольких методов и создания собственных методов объекта. Конструкторы анонимных классов нельзя определять и переопределять. Анонимные классы допускают вложенность друг в друга, что может сильно запутать код и сделать эти конструкции непонятными.

Пример. Создать класс Студент с закрытым (private) внутренним (inner) классом, объекты которого содержат данные о сданных экзаменах. Просмотреть результаты сессии одного студента.

Решение.

```
package task04.run;

public class Student {
    private final int PASSED_MARK = 4; // мин. зачетный балл
    private int id; // ID студента
    private ExamResult[] exams; // список сданных экзаменов

    public Student(int id) {
        this.id = id;
    }

    private class ExamResult {// внутренний класс
        private String name;
        private int mark;
        private boolean passed;

        // конструктор внутреннего класса
        ExamResult(String name) {
            this.name = name;
            passed = false;
        }

        void setMark(int mark) {
            this.mark = mark;
            // из внутреннего класса есть доступ
            // ко всем полям и методам внешнего класса
            passed = (mark >= PASSED_MARK);
        }

        int getMark() {
            return mark;
        }

        String getName() {
            return name;
        }

        boolean isPassed() {
            return passed;
        }
    } // окончание внутреннего класса

    /** возвращает оценку за экзамен по названию предмета */
    public String getExamMark(int index) {
        return "Exam " + exams[index].getName() + " - "
                + exams[index].getMark();
    }
```

```
    /** заполняет список экзаменов */
    public void setExams(String[] name, int[] marks) {
        // инициализируем массив экзаменов
        exams = new ExamResult[name.length];
        // заполняем каждый экзамен названием и оценкой
        for (int i = 0; i < name.length; i++) {
            exams[i] = new ExamResult(name[i]);
            // из внешнего класса метод внутреннего класса
            // может быть вызван только у объекта
            exams[i].setMark(marks[i]);
        }
    }

    @Override
    public String toString() {
        String res = "Студент: " + id + "\n";
        for (int i = 0; i < exams.length; i++)
            if (exams[i].isPassed())
                res += exams[i].getName() + " сдал \n";
            else
                res += exams[i].getName() + " не сдал \n";
        return res;
    }
}
```

```
package task04.run;

/* точка входа */
public class Runner {
    public static void main(String[] args) {
        Student stud = new Student(822201);
        String ex[] = {"Компьютерные сети", "Методы оптимизации"};
        int marks[] = { 8, 2 };
        stud.setExams(ex, marks);
        System.out.println(stud);
        System.out.println(stud.getExamMark(0));
        System.out.println(stud.getExamMark(1));
    }

}
```

2.5 Лабораторная работа №5. Реализация интерфейсов

Интерфейсы не являются классами. Для каждого интерфейса при компиляции создается class файл. Ни один из объявленных методов не может быть реализован внутри интерфейса (до 8 версии Java).

В Java два вида интерфейсов:

– интерфейсы, определяющие контракт для классов посредством методов;
– интерфейсы, реализация которых автоматически (без реализации методов) придает классу определенные свойства.

При именовании интерфейсов рекомендуется в начало имени помещать I, например, `IPaper`.

В Java реализовано множественное наследование интерфейсов:

```
[доступ] interface ИмяИнтерфейса extends Имя1, Имя2,…, ИмяN {
     /*код интерфейса*/
}
```

Все объявленные в интерфейсе поля автоматически трактуются как `public`, `static` и `final`

Все объявленные в интерфейсе методы автоматически трактуются как `public` и `abstract`.

Класс может реализовывать любое количество интерфейсов.

Класс обязан реализовать все методы, полученные от интерфейсов, или объявить себя абстрактным классом.

```
[доступ] class ИмяКласса implements Имя1, Имя2,…, ИмяN {
    /*код класса*/
}
```

Класс можно объявить **внутри интерфейса**.

```
interface ИмяИнтерфейса [extends Интерфейс1, Интерфейс2, ...]{
    тип Поле = ЗНАЧЕНИЕ;
    тип метод();
    [спецификаторы]class ИмяКласса [extends …][implements …]{
        /* реализация класса */
    }
}
```

Все объявленные в интерфейсе классы автоматически трактуются как `public` и `static`.

Интерфейсы тоже могут быть внутренними.

```
Map.Entry<K, V>
```

Этот интерфейс описывает пару ключ-значение ассоциативного словаря.

Пример. Разработать иерархию транспортных средств (транспортное средство, машина, лодка) и интерфейс (возможность перевозить пассажиров, с методами посадить/высадить пассажира, получить количество пассажиров). Дополнить иерархию транспортных средств пассажирскими машиной и лодкой.

Решение.

```
package task05.transport;
/** базовый класс Транспорт*/
public abstract class Transport {
    private String name; // название

    public Transport(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName() + " name=" + name;
    }

}
package task05.transport;
/** лодка */
// этот класс наследует транспорт
public class Boat extends Transport{
    private double capacity; // водоизмещение
    public Boat(String name, double capacity) {
        // вызов конструктора с параметрами супер-класса
        super(name);
        this.capacity = capacity;
    }
    public double getCapacity() {
        return capacity;
    }
```

```
    @Override
    public String toString() {
        return super.toString() + " capacity=" + capacity;
    }
}

package task05.transport;
/** автомобиль */
// класс наследует транспорт
public class Avto extends Transport{
    private int axisCount; // количество осей

    public Avto(String name, int axisCount) {
        // вызов конструктора супер-класса
        super(name);
        this.axisCount = axisCount;
    }

    public int getAxisCount() {
        return axisCount;
    }

    @Override
    public String toString() {
        return super.toString() +  " axisCount="
                + axisCount + "]";
    }
}

package task05.passange;
/** возможность перевозить пассажиров */
public interface IPassange {
    // вложенный nested класс
    /** пассажир */
    class Passange {
        // тут может быть некая реализация
    }

    // еще один вложенный nested класс
    /** список пассажиров */
    class ListOfPassanges {
        private Passange[] pass;
        private int countPass = 0;

        // в конструктор необходимо передать
        // максимальное количество пассажиров
        public ListOfPassanges(int maxPassCount) {
            pass = new Passange[maxPassCount];
        }

        /** возвращает максимальное количество пассажиров */
        public int getMaxPassCount() {
            return pass.length;
        }
```

```
        /** возвращает количество пассажиров */
        public int getPassCount() {
            return countPass;
        }

        /** возвращает пассажира по номеру
         (удаляя при этом из списка) */
        public Passange getPass(int index) {
            Passange p = pass[index];
            if (p != null) {
                pass[index] = null;
                countPass--;
            }
            return p;
        }

        /** помещает пассажира в список по номеру,
         если там есть пассажир, то заменяет его */
        public void setPass(Passange p, int index) {
            Passange op = pass[index];
            pass[index] = p;
            if (op == null && p != null) {
                pass[index] = null;
                countPass++;
            }
        }
    }

    // методы интерфейса
    /** получить максимальное количество пассажиров */
    int getMaxPassCount();

    /** получить количество пассажиров */
    int getPassCount();

    /** посадить пассажира */
    void setPass(Passange p, int index);

    /** высадить пассажира */
    Passange getPass(int index);
}
```

```
package task05.passange;

// импорт класса из другого пакета
import task05.transport.Boat;

/** пассажирская лодка */
public class PassangeBoat extends Boat implements IPassange {
    // переменная - список пассажиров
    // экземпляр вложенного класса реализуемого интерфейса
    ListOfPassanges list;

    // конструктор с параметрами
    public PassangeBoat(String name, double capacity,
              int maxCountPass) {
        super(name, capacity);
        list = new ListOfPassanges(maxCountPass);
    }
    // метод должен быть реализован в классе
    @Override
    public int getMaxPassCount() {
        // выполнение метода делегируется объекту-полю класса,
        // в котором этот метод действительно реализован
        return list.getMaxPassCount();
    }

    // метод должен быть реализован в классе
    @Override
    public int getPassCount() {
        // выполнение аналогично делегируется внутреннему объекту
        return list.getPassCount();
    }

    // метод должен быть реализован в классе
    @Override
    public void setPass(Passange p, int index) {
        // выполнение аналогично делегируется внутреннему объекту
        list.setPass(p, index);
    }

    // метод должен быть реализован в классе
    @Override
    public Passange getPass(int index) {
        // выполнение аналогично делегируется внутреннему объекту
        return list.getPass(index);
    }

}
```

класс `PassangeAvto` реализован аналогично классу `PassangeBoat`.

```
package task05.run;

import java.util.Random;
import task05.passange.*;
import task05.transport.*;

// точка входа
public class Runner {
    static Random rand = new Random(100);
    // метод работает только с классами,
    // реализующими интерфейс IPassange
    public static void passangeHandle(IPassange passTr) {
        // случайное количество пассажиров,
        // не больше максимально возможного
        int count = rand.nextInt(passTr.getMaxPassCount());
        // При объявлении переменной "пассажир"
        // используется имя интерфейса,
        // поскольку класс описан внутри интерфейса
        IPassange.Passange p;
        for (int i = 0; i < count; i++) {
            // создаем нового пассажира
            p = new IPassange.Passange();
            // помещаем пассажира в транспорт
            passTr.setPass(p, i);
        }
        // вывод в консоль действительного количества пассажиров
        System.out.println("пассажиров = " +
                passTr.getPassCount());
    }

    public static void main(String[] args) {
        Transport[] list = new Transport[5];
        PassangeAvto avto1 = new PassangeAvto("Гусеничка", 4, 12);
        PassangeAvto avto2 = new PassangeAvto("Пат-пат", 2, 2);
        PassangeBoat boat1 = new PassangeBoat("Варяг", 10, 45);
        list[0] = new Boat("Аврора", 10);
        list[1] = new Avto("Антилопа Гну", 2);
        list[2] = avto1; list[3] = boat1; list[4] = avto2;

        for(Transport tr: list){
            System.out.println(tr);
        }
        // переменная avto1 объявлена как PassangeAvto
        // поэтому может быть передана в метод, т.к.
        // PassangeAvto реализует IPassange
        passangeHandle(avto1);
        passangeHandle(avto2);
        // переменную list[3] несмотря на то,
        // что она указывает на объект типа PassangeBoat,
        // необходимо явно привести к типу IPassange
        // поскольку она объявлена как Transport
        passangeHandle((IPassange) list[3]);
    }
}
```

Список литературы

1. Блинов, И.Н. Java. Промышленное программирование / И.Н. Блинов, В.С. Романчик. – Мн.: Универсал Пресс, 2007. – 124 с

2. Эккель, Б. Философия Java. 4-е издание / Б. Эккель. – СПб: Питер, 2009. – 638с.

3. Шилдт, Г. Java. Полное руководство, 8-е издание / Г. Шилдт. – СПб.: Вильямс, 2012. – 1104 с.

4. Тейт, Б. Горький вкус Java / Б. Тейт – СПб.: Питер, 2003. – 334 с.

5. Макконнел, С. Совершенный код. Мастер-класс/ С.Макконнел – М.: «Русская редакция»; СПб.: Питер, 2007. – 896 с.

6. Блох, Д. Java. Эффективное программирование / Д. Блох – М.: Лори, 2002. – 224 с.

7. Burnette, E. Eclipse IDE Pocket Guide / E. Burnette – O'Relly, 2005. – 127 p.

Исследование операций. Лабораторный практикум.
Составители:
Ерофеева Елена Анатольевна

Подписано в печать 06.01.98. Формат 60 х 80 1/16. Печать офсетная. Усл.п.л. 1,6. Уч.-изд.л. 1,3. Тираж 100 экз. Заказ 114 .

Отпечатано на ротапринте Гомельского государственного университета им. Ф. Скорины.
246699, Гомель, ул. Советская, 104