

ЦОТ “БЕЛХАРД”

**Программирование на Java**

Учебно-методическое пособие для слушателей курса “Основы  
программирования на Java” ЦОТ “БЕЛХАРД”

Минск 2016

УДК 004.737  
ББК 32.979.26-017.1

Программирование на Java: учебно-методическое пособие для слушателей курса «Основы программирования на Java» ЦОТ ОДО «БелХард» / Е.В. Карсека. – Минск : ОДО «БелХард», 2015г. – 276 с.  
ISBN 987-985-6644-73-1

Пособие предназначено для слушателей курса «Основы программирования на Java» ЦОТ ОДО «БелХард» начинающих и продолжающих изучение технологий Java под руководством преподавателя. В нем рассматриваются основы языка Java и концепции объектно-ориентированного программирования, изложены важнейшие аспекты применения библиотек классов языка Java, включая коллекции, swing и jdbc.

Среди теоретического материала приводится много заданий и примеров готовых реализаций. В конце каждой главы даются выводы к главе и задания для самостоятельного выполнения.

УДК 004.737  
ББК 32.979.26-017.1

ISBN 987-985-6644-73-1

© ОДО «БелХард», 2016

## СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ.....</b>	<b>6</b>
<b>Глава 1 ВВЕДЕНИЕ В JAVA. ОСНОВЫ ЯЗЫКА. ....</b>	<b>7</b>
Тема 1.1 Язык программирования java. ....	7
Тема 1.2 Состав пакета Java2. ....	8
Тема 1.3 Настройка среды окружения. ....	9
Тема 1.4 Структура Java-программы. ....	12
Тема 1.5 Набор текста, запуск и компиляция простейшей программы. ....	13
Тема 1.6 Подробное рассмотрение кода простейшей программы. ....	14
Тема 1.7. Создание программы в разных средах разработки. ....	18
Тема 1.8 Лексические основы языка ....	23
Тема 1.9 Элементарные типы данных.....	31
Тема 1.10 Преобразование типов. ....	36
Тема 1.11 Консольный ввод с помощью класса java.util.Scanner ....	38
Тема 1.12 Операторы ....	41
1.12.1 Блок.....	42
1.12.2 Условный оператор if ....	42
1.12.3 if-else и ?.....	45
1.12.4 Оператор цикла while.....	46
1.12.5 Оператор цикла do-while ....	48
1.12.6 Оператор цикла for.....	49
1.12.7 Оператор continue и метки ....	50
1.12.8 Оператор break.....	51
1.12.9 Оператор варианта switch.....	52
Тема 1.13 Статический импорт.....	53
Тема 1.14 Класс Math.....	54
Тема 1.15 Псевдослучайные числа.....	56
Тема 1.16 Генерация случайных чисел ....	57
Тема 1.17 Массивы в Java.....	58
1.17.1 Объявление и заполнение массива.....	58
1.17.2 Сортировка массива.....	61
1.17.3 Многомерные массивы ....	65
1.17.4 Нерегулярные массивы.....	67
<b>Глава 2 КЛАССЫ ....</b>	<b>76</b>
Тема 2.1 Основы классов.....	76
Тема 2.2 Общая форма класса.....	76
Тема 2.3 Объявление объектов ....	80
Тема 2.4 Более подробное рассмотрение операции new ....	81
Тема 2.5 Присваивание переменных объектных ссылок ....	82
Тема 2.7 Возвращение значения из метода ....	85
Тема 2.8 Добавление метода, принимающего параметры ....	87

Тема 2.9 Конструкторы.....	89
2.9.1 Конструкторы без параметров .....	89
2.9.2 Конструкторы с параметрами .....	91
2.9.3 Ключевое слово <code>this</code> .....	93
2.9.4 Соккрытие переменной экземпляра .....	93
Тема 2.10 Сборка мусора.....	94
Тема 2.11 Перегрузка методов.....	95
Тема 2.12 Перегрузка конструкторов.....	99
Тема 2.13 Использование объектов в качестве параметров .....	101
Тема 2.14 Более пристальный взгляд на передачу аргументов.....	104
Тема 2.16 Рекурсия.....	106
Тема 2.18 Ключевое слово <code>static</code> .....	111
Тема 2.19 Ключевое слово <code>final</code> .....	114
Тема 2.20 Использование массива объектов .....	114
Тема 2.21 Аргументы переменной длины .....	117
Тема 2.22 Классы-оболочки .....	123
Тема 2.23 Автоупакока и автораспаковка. ....	132
Тема 2.24 Строки и числа .....	133
Тема 2.25 Нумерованные типы .....	147
Тема 2.26 Регулярные выражения .....	154
2.26.1 Метасимволы .....	154
2.26.2 Квантификаторы.....	155
2.26.3 Классы регулярных выражений в Java.....	156
2.26.4 Валидация емейла. ....	158
<b>Глава 3 НАСЛЕДОВАНИЕ И ИНТЕРФЕЙСЫ.....</b>	<b>162</b>
Тема 3.1 Основы наследования.....	162
Тема 3.2 Наследование и доступ к членам класса .....	165
Тема 3.3 Конструкторы и наследование .....	168
Тема 3.4 Использование ключевого слова <code>super</code> для вызова конструктора суперкласса .....	170
Тема 3.5 Использование ключевого слова <code>super</code> для доступа к членам суперкласса .....	174
Тема 3.6 Многоуровневая иерархия.....	175
Тема 3.7 Когда вызываются конструкторы .....	178
Тема 3.8 Объекты подклассов и ссылки на суперклассы .....	179
Тема 3.9 Переопределение методов .....	184
Тема 3.10 Переопределение методов и поддержка полиморфизма .....	187
Тема 3.11 Использование абстрактных классов .....	193
Тема 3.12 Использование ключевого слова <code>final</code> .....	197
Тема 3.13 Предотвращение переопределения методов.....	197
Тема 3.14 Предотвращение наследования.....	198
Тема 3.15 Класс <code>Object</code> .....	198
Тема 3.16 Интерфейсы.....	203
3.16.1 Объявление интерфейса. ....	204
3.16.2 Реализация интерфейсов .....	205

3.16.3 Использование ссылок на интерфейсы.....	208
3.16.4 Переменные в составе интерфейсов.....	209
3.16.5 Наследование интерфейсов.....	211
Тема 3.17 Пакеты и ограничение доступа .....	212
Тема 3.18 Внутренние классы.....	214
3.18.1 Внутренние (inner) классы .....	215
3.18.2 Вложенные (nested) классы.....	217
3.18.3 Анонимные (anonymous) классы .....	218
<b>Глава 4 ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ .....</b>	<b>224</b>
Тема 4.1 Исключения в Java.....	224
Тема 4.2 Типы исключений.....	224
Тема 4.3 Неперехваченные исключения.....	225
Тема 4.4 Ключевые слова try и catch.....	226
Тема 4.5 Вложенные операторы try.....	227
Тема 4.6 Ключевое слово throw .....	229
Тема 4.7 Ключевое слово throws.....	230
Тема 4.8 Ключевое слово finally .....	231
<b>Глава 5 УНИВЕРСАЛЬНЫЕ ТИПЫ. КОЛЛЕКЦИИ.....</b>	<b>235</b>
Тема 5.1 Общие сведения об универсальных типах.....	235
Тема 5.2 Универсальный класс с двумя параметрами типа .....	240
Тема 5.3 Ограниченные типы .....	241
Тема 5.4 Использование групповых параметров.....	243
Тема 5.5 Универсальные методы.....	245
Тема 5.6 Универсальные интерфейсы.....	248
Тема 5.7 Ошибки неоднозначности.....	251
Тема 5.8 Ограничения универсальных типов.....	252
Тема 5.9 Краткий обзор коллекций .....	254
5.9.1 Класс <i>ArrayList</i> .....	262
5.9.2 Класс <i>LinkedList</i> .....	263
5.9.3 Класс <i>HashSet</i> .....	264
5.9.4 Класс <i>TreeSet</i> .....	265
5.9.5 Доступ к коллекции через итератор.....	267
5.9.6 Алгоритмы коллекций .....	270
<b>ЗАКЛЮЧЕНИЕ.....</b>	<b>275</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....</b>	<b>276</b>

## **ВВЕДЕНИЕ**

Создание языка Java – это действительно один из самых значительных шагов вперед в области разработки сред программирования за последние 20 лет. Язык HTML (Hypertext Markup Language – язык разметки гипертекста) был необходим для статического размещения страниц во «Всемирной паутине» WWW (World Wide Web). Язык Java потребовался для качественного скачка в создании интерактивных продуктов для сети Internet. Язык Java воплощает в себе следующие качества: простоту и мощь, безопасность, объектную ориентированность, надежность, интерактивность, архитектурную независимость, возможность интерпретации, высокую производительность и легкость в изучении. После освоения основных понятий объектно-ориентированного программирования вы быстро научитесь программировать на Java.

Данное методическое пособие предназначено для слушателей курса «Основы программирования на Java» ЦОТ ЗАО «БелХард Групп», обладающих базовыми знаниями по Си или по любому другому алгоритмическому языку программирования. В нем рассмотрены только базовые вопросы JavaSE.

Изучение JavaSE – лишь первый шаг на Вашем пути в карьере программиста.

# Глава 1 ВВЕДЕНИЕ В JAVA. ОСНОВЫ ЯЗЫКА.

## Тема 1.1 Язык программирования java.

Java – объектно-ориентированный язык программирования, разрабатываемый компанией Sun Microsystems с 1991 года и официально выпущенный 23 мая 1995 года. Изначально новый язык программирования назывался Oak (James Gosling) и разрабатывался для бытовой электроники, но впоследствии был переименован в Java и стал использоваться для написания апплетов, приложений и серверного программного обеспечения.

Программы на Java могут быть транслированы в байт-код, выполняемый на виртуальной java-машине (JVM) – программе, обрабатывающей байт-код и передающей инструкции оборудованию, как интерпретатор, но с тем отличием, что байт-код, в отличие от текста, обрабатывается значительно быстрее.

Язык Java зародился как часть проекта создания передового программного обеспечения для различных бытовых приборов. Реализация проекта была начата на языке C++, но вскоре возник ряд проблем, наилучшим средством борьбы с которыми было изменение самого инструмента – языка программирования. Стало очевидным, что необходим платформу-независимый язык программирования, позволяющий создавать программы, которые не приходилось бы компилировать отдельно для каждой архитектуры и можно было бы использовать на различных процессорах под различными операционными системами.

Язык Java потребовался для создания интерактивных продуктов для сети Internet. Фактически, большинство архитектурных решений, принятых при создании Java, было продиктовано желанием предоставить синтаксис, сходный с C и C++. В Java используются практически идентичные соглашения для объявления переменных, передачи параметров, операторов и для управления потоком выполнением кода. В Java добавлены все хорошие черты C++.

Три ключевых элемента объединились в технологии языка Java

- Java предоставляет для широкого использования свои апплеты (applets) – небольшие, надежные, динамичные, не зависящие от платформы активные сетевые приложения, встраиваемые в страницы Web. Апплеты Java могут настраиваться и распространяться потребителям с такой же легкостью, как любые документы HTML

- Java высвобождает мощь объектно-ориентированной разработки приложений, сочетая простой и знакомый синтаксис с надежной и удобной в работе средой разработки. Это позволяет широкому кругу программистов быстро создавать новые программы и новые апплеты

- Java предоставляет программисту богатый набор классов объектов для ясного абстрагирования многих системных функций, используемых при работе с окнами, сетью и для ввода-вывода. Ключевая черта этих классов

заключается в том, что они обеспечивают создание независимых от используемой платформы абстракций для широкого спектра системных интерфейсов

## **Тема 1.2 Состав пакета Java2.**

На конференции разработчиков 15 июня 1999 года компания Sun объявила о разделении развития платформы Java 2 на три направления:

- J2SE (Java2 Platform, Standart Edition) – предназначен для использования на рабочих станциях и персональных компьютерах, используется для разработки настольных и сетевых приложений;
- J2EE (Java2 Platform, Enterprise Edition) – содержит все необходимое для создания сложных, высоконадежных, распределенных серверных приложений; J2EE и Java Web Services (JWS) – используются для разработки корпоративных Web и Internet приложений, а также web-служб;
- J2ME (Java2 Platform, Micro Edition) – усеченная SE, содержит все необходимое для удовлетворения жестким аппаратным условиям небольших устройств, таких как карманные компьютеры и сотовые телефоны, используется для разработки мобильных приложений для беспроводных устройств.

JDK (Java Development Kit) – программный инструментарий (набор) для полноценной работы с языком, который, наряду с компилятором, интерпретатором и отладчиком и другими инструментами включает в себя обширнейшую библиотеку классов Java. Набор программ и классов JDK в основном содержат:

- компилятор `javac` из исходного кода в байт-коды;
- интерпретатор `java`, содержащий интерпретацию JVM (java virtual machine);
- облеченный интерпретатор `jre`;
- программу просмотра апплетов `appletviewer`;
- отладчик `jdb`;
- дисассемблер `javap`;
- программу архивации и сжатия `jar`;
- программу сбора документации `javadoc`;
- программу `javah` генерации заголовочных файлов языка Си;
- программу `javakey` добавления электронной подписи;
- программу `native2ascii`, преобразующую бинарники в текстовые файлы;
- программы `rmic` и `rmiregistry` для работы с удаленными объектами;
- программу `serialver`, определяющую номер версии класса;



- библиотеки и заголовочные файлы “родных” методов;
- библиотеку классов Java API.

JRE (Java Runtime Environment) – среда исполнения java-приложений, которая должна быть установлена на компьютере для запуска Java приложений. JDK включает в себя JRE.

### Тема 1.3 Настройка среды окружения.

Для настройки переменных среды окружения в Windows XP откройте компонент Система панели управления (рис. 1.1).

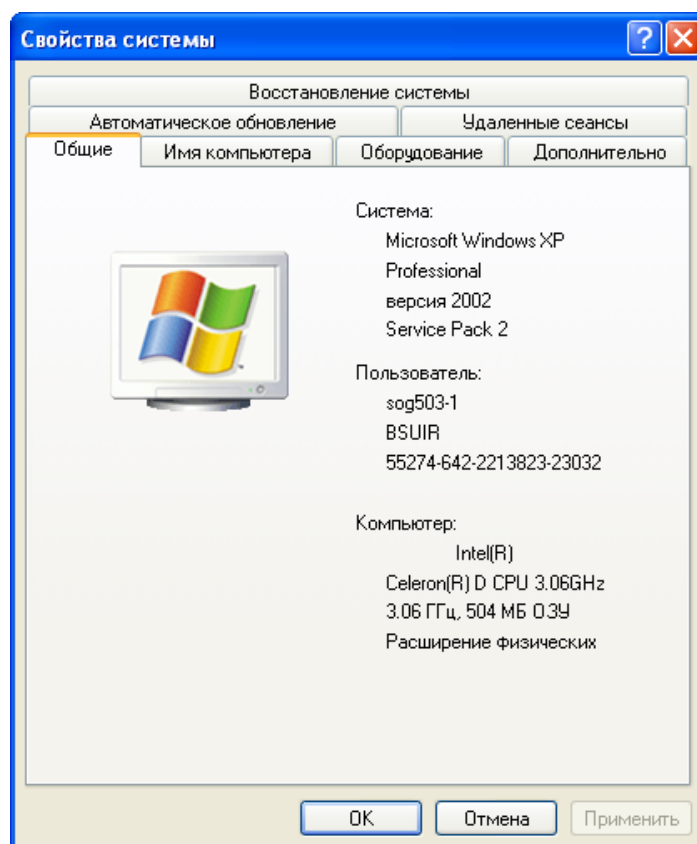


Рисунок 1.1 – Свойства системы

На вкладке Дополнительно нажмите кнопку Переменные среды (рис. 1.2).

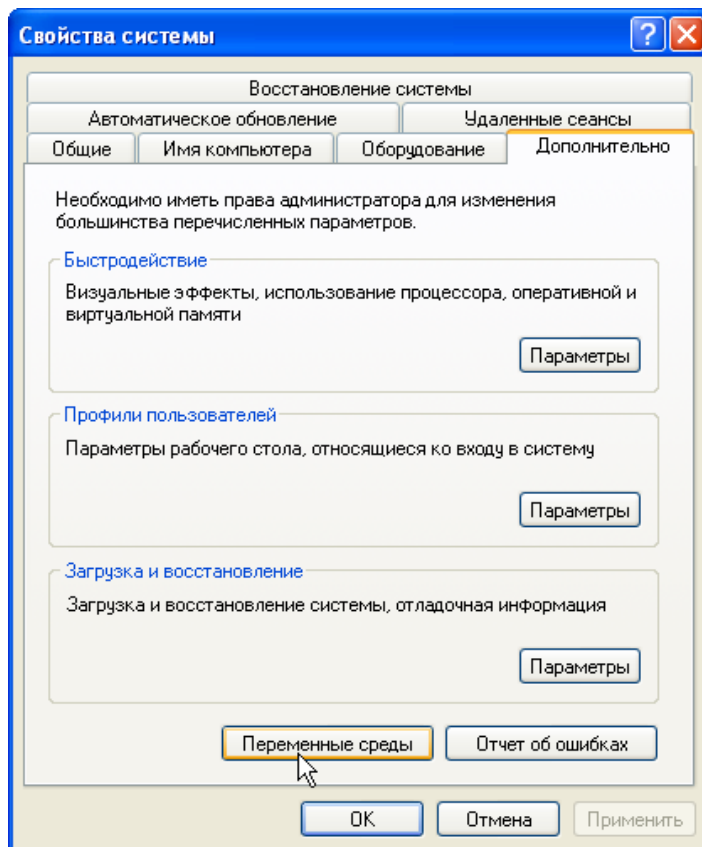


Рисунок 1.2 – Вкладка Дополнительно окна Свойства системы

В окне переменные среды в группе значений Системные переменные Выберите переменную Path и нажмите кнопку Изменить (рис. 1.3).

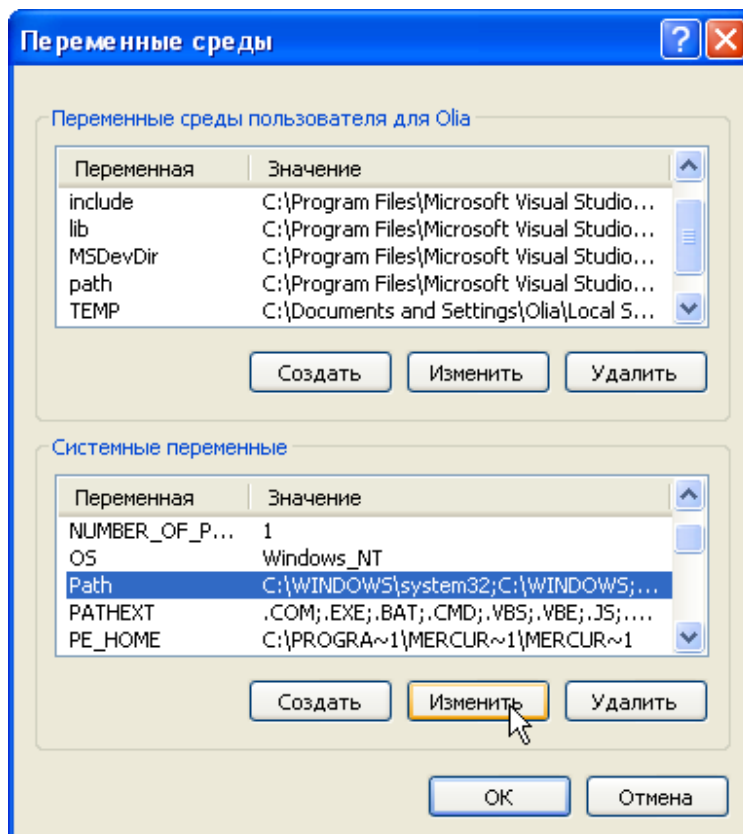


Рисунок 1.3 – Переменные среды

Измените значения этого поля, добавив путь к каталогу bin, содержащему утилиты JDK (java, javac, jar и т.п.) (обычно, это C:\Program Files\Java\jdk\_\_\_\_\bin) и путь к рабочему каталогу (рис. 1.4).

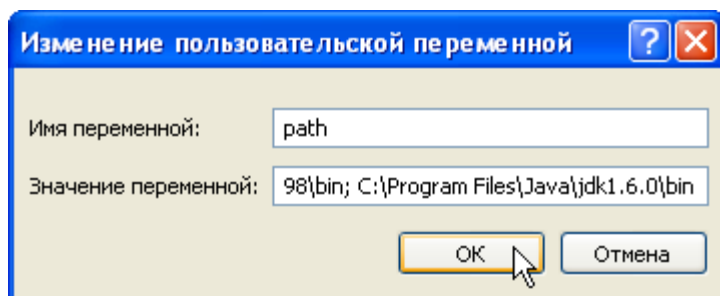


Рисунок 1.4 – Изменение пользовательской переменной path

Нажмите OK, а затем кнопку Создать. Создайте три переменные со следующими значениями.

SWING\_HOME=C:\Program Files\Java\jdk\_\_\_\_\LIB; (рис. 1.5)

CLASSPATH=.; C:\Program Files\Java\jdk\_\_\_\_; C:\Program Files\Java\jdk\_\_\_\_\LIB (рис. 1.6)

JDBCHOME=C:\WINDOWS\SYSTEM; (рис. 1.7)

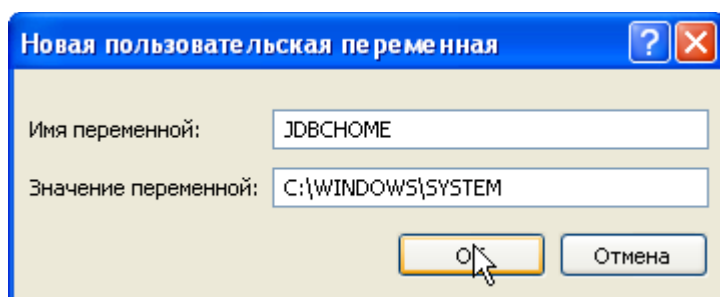


Рисунок 1.5 – Создание новой пользовательской переменной JDBCHOME

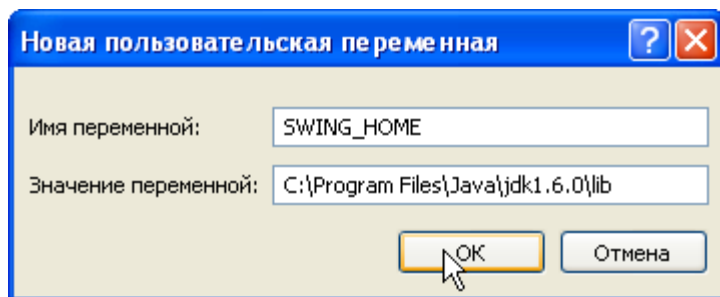


Рисунок 1.6 – Создание новой пользовательской переменной SWING\_HOME

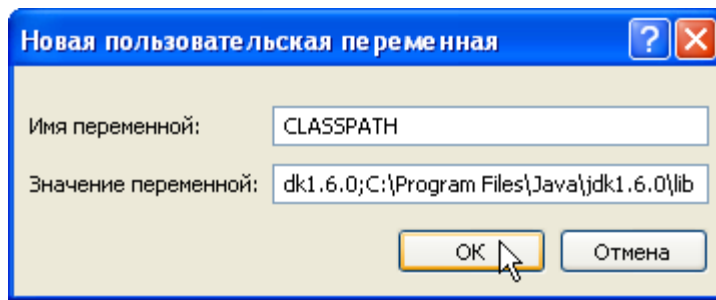


Рисунок 1.7 – Создание новой пользовательской переменной CLASSPATH

Все, переменные среды окружения установлены.

### Тема 1.4 Структура Java-программы.

Для большинства языков файл, содержащий исходный код программы, может иметь произвольное имя. Однако для Java действуют другие правила. Работая с Java, надо знать, что имя исходного файла очень важно. В данном примере файл, содержащий исходный код, имеет имя `MyFirstProgram.java`. Рассмотрим, почему следует выбрать именно такое имя.

В Java исходный файл называется модулем компиляции. Это текстовый файл, содержащий определения одного или нескольких классов. Компилятор Java требует, чтобы исходный файл имел расширение `java`. Обратив внимание на код программы, вы увидите, что класс имеет имя `MyFirstProgram`. Это не совпадение. В Java-программах весь код находится в составе классов. По соглашениям имя класса должно совпадать с именем файла, содержащего текст программы. Нетрудно убедиться, что имя файла соответствует имени класса. В языке Java имена и другие идентификаторы зависят от регистра символов. Соглашение о соответствии имен классов именам файлов может на первый взгляд показаться надуманным, однако оно упрощает организацию и сопровождение программ.

*Особенности Java-программы:*

1. Всякая программа состоит из одного или нескольких классов.
2. Начало класса отмечается служебным словом `class`, за которым следует имя класса, выбираемое произвольно. Все, что содержится в классе, записывается в фигурных скобках и составляет *тело класса* (class body).
3. Все действия производятся с помощью методов обработки информации, коротко говорят просто *метод* (method), которые являются аналогами "функций", применяемых в других языках.
4. Методы различаются по именам. Один из методов обязательно должен называться *main*, с него начинается выполнение программы.
5. Как и положено функции, метод всегда выдает в результате (чаще говорят, *возвращает* (returns)) только одно значение, тип которого обязательно указывается перед именем метода. Метод может и не возвращать никакого значения, играя роль процедуры, как в данном случае. Тогда вместо

типа возвращаемого значения записывается слово *void*, как это и сделано в примере.

6. После имени метода в скобках, через запятую, перечисляются *аргументы* (arguments), или *параметры* метода. Для каждого аргумента указываются его тип и через пробел имя.

7. Перед типом возвращаемого методом значения могут быть записаны *модификаторы* (modifiers). Модификаторы вообще необязательны, но для метода *main* () они необходимы.

## Тема 1.5 Набор текста, запуск и компиляция простейшей программы.

Текст простейшей программы на языке Java приведен в листинге 1.1

### Листинг 1.1

```
/*
```

Это пример программы на языке Java.

Поместите код в файл MyFirstProgram.Java.

```
*/
```

```
public class MyFirstProgram {
```

```
// Выполнение каждой Java-программы начинается
```

```
// с вызова метода main()
```

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

```
        System.out.println("Hello, world!");
```

```
    }
```

```
}
```

Наберите этот код в текстовом редакторе и сохраните файл как MyFirstProgram.java.

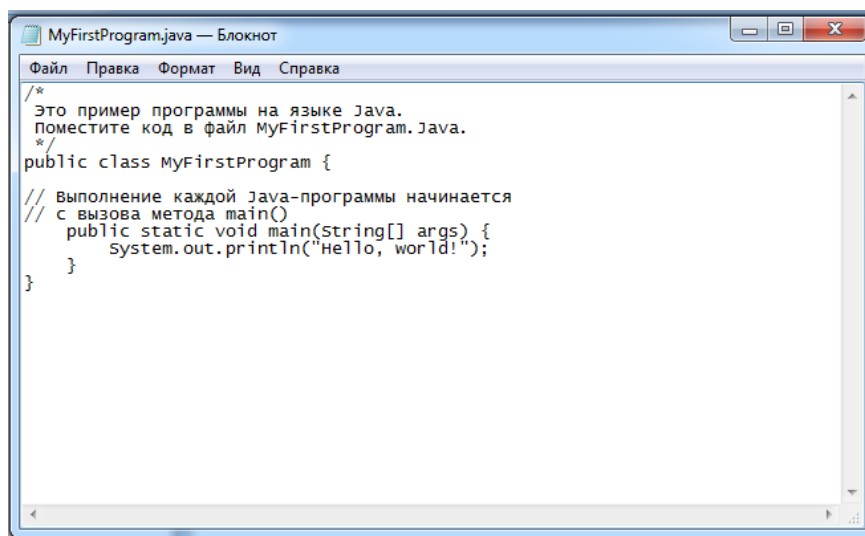


Рисунок 1.8 – Программа на Java

Для компиляции и запуска программу необходимо поместить в файл, имя которого совпадает с именем класса. Исходные тексты программ

хранятся в файлах с расширением \*.java, результат компиляции или байт-код помещается в файл с расширением \*.class. Компиляция производится не в машинный код, а в некий промежуточный код, который может понимать интерпретатор. Этим достигается переносимость – без перекомпиляции код работает на любой платформе, где есть соответствующий интерпретатор. Этот интерпретатор называют еще виртуальной машиной

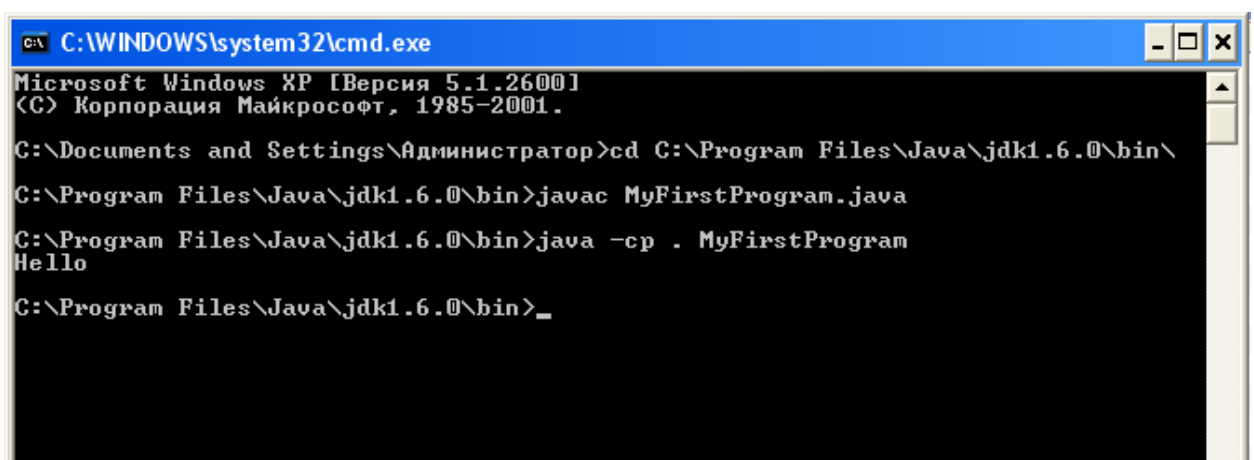
Скомпилируйте программу командой:

javac MyFirstProgram.java (рис. 1.9)

После успешной компиляции создается файл MyFirstProgram.class. Если такой файл не создался, то, значит, код содержит ошибки, которые необходимо устранить и ещё раз скомпилировать программу.

Запустите программу командой:

java MyFirstProgram



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [Версия 5.1.2600]
(C) Корпорация Майкрософт, 1985-2001.

C:\Documents and Settings\Администратор>cd C:\Program Files\Java\jdk1.6.0\bin\
C:\Program Files\Java\jdk1.6.0\bin>javac MyFirstProgram.java
C:\Program Files\Java\jdk1.6.0\bin>java -cp . MyFirstProgram
Hello
C:\Program Files\Java\jdk1.6.0\bin>_
```

Рисунок 1.9 – Компиляция программы

## Тема 1.6 Подробное рассмотрение кода простейшей программы.

Несмотря на то, что программа MyFirstProgram.java очень проста и объем кода невелик, она обладает рядом характеристик, общих для всех Java-программ. Рассмотрим подробно каждый фрагмент программы.

Исходный код начинается следующими строками:

```
/*
```

Это пример программы на языке Java.

Поместите код в файл MyFirstProgram.Java.

```
*/
```

Это комментарии. Подобно большинству других языков программирования, Java позволяет комментировать исходный текст программы. Компилятор игнорирует комментарии. Текст комментариев описывает действия, выполняемые программой, в результате исходный код становится более простым для восприятия. В данном случае комментарии сообщают, что программа проста и исходный код должен содержаться в файле MyFirstProgram.java. Конечно, в реальных приложениях комментарии

описывают, какие функции выполняют фрагменты программы или зачем использована та или иная языковая конструкция.

Java поддерживает три типа комментариев. Первый тип, приведенный в начале рассматриваемой программы, позволяет задавать комментарии, состоящие из нескольких строк. Они начинаются символами `/*` и заканчиваются символами `*/`. Любое содержимое, находящееся между этими ограничителями, компилятор игнорирует.

Следующая строка кода имеет такой вид:

```
public class MyFirstProgram {
```

Ключевое слово `public` называется модификатором доступа. Модификатор доступа определяет правила обращения к членам класса из других частей программы. Если член класса предваряется ключевым словом `public`, то к нему может производиться обращение из-за пределов класса.

Ключевое слово `class`, используемое для определения нового класса. Как было сказано ранее, класс является основной языковой конструкцией Java, поддерживающей инкапсуляцию. `MyFirstProgram` – это имя класса. Определение класса начинается открывающей фигурной скобкой `{` и заканчивается закрывающей фигурной скобкой `}`. Элементы, находящиеся между ними, являются членами класса. На данный момент не пытайтесь глубоко разобраться в особенностях классов, заметьте лишь, что любой код, какие бы действия он ни выполнял, находится в составе класса. Такая организация свидетельствует в пользу утверждения о том, что любая Java-программа является в той или иной мере объектно-ориентированной.

Следующая строка программы – комментарии. Данный тип комментариев предполагает, что они состоят лишь из одной строки.

```
// Выполнение каждой Java-программы начинается  
// с вызова метода main()
```

Это второй тип комментариев, поддерживаемых в языке Java. Комментарии, состоящие из одной строки, начинаются с символов `//` и продолжаются до конца строки. Как правило, многострочные комментарии применяются для размещения текста сравнительно большого объема, а однострочные комментарии – для коротких заметок.

Следующая строка кода выглядит так, как показано ниже.

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

Строка начинается с метода `main()`. Как было сказано ранее, в терминологии, используемой для языка Java, функции принято называть методами. Именно с данной строки начинается работа программы, и приведенные выше комментарии подтверждают это. Выполнение любого Java-приложения начинается с вызова метода `main( )`. Назначение каждого компонента данной строки пока рассматриваться не будет, для того, чтобы понять их, надо рассмотреть некоторые другие средства Java, но мы вкратце обсудим структуру соответствующей строки.

Ключевое слово `public` называется модификатором доступа. Модификатор доступа определяет правила обращения к членам класса из других частей программы. Если член класса предваряется ключевым словом

public, то к нему может производиться обращение из-за пределов класса. (Модификатор доступа private выполняет действия, противоположные public, а именно запрещает доступ к членам класса извне.) В данном случае метод main( ) должен быть объявлен как public, поскольку при выполнении программы он вызывается из-за пределов класса. Ключевое слово static допускает вызов метода main( ) до создания экземпляра класса. Указывать его необходимо, поскольку метод main( ) вызывается виртуальной машиной еще до того, как будут созданы какие-либо объекты. Ключевое слово void лишь сообщает компилятору о том, что метод main( ) не возвращает значение. Как вы увидите далее, для некоторых методов предусматривается возвращаемое значение.

Как было сказано ранее, метод main( ) вызывается в начале выполнения Java-приложения. Любая информация, которую необходимо передать данному методу, задается с помощью переменных, указанных в круглых скобках, которые следуют за именем метода. Эти переменные называются параметрами. Если для какого-либо метода параметры не предусмотрены, то после имени метода указывается пара круглых скобок. В данном случае для метода main( ) задан один параметр с именем args: String args[ ]. Это массив объектов типа String. (Массив – набор объектов одного типа.) Объекты типа String хранят последовательности символов. В данном случае с помощью args методу передаются параметры, указанные в командной строке при запуске программы. Рассматриваемая нами программа не использует полученную информацию, но в других примерах будет продемонстрирована обработка данных из командной строки.

Завершается строка символом "{". Он означает начало тела метода main(). Весь код, содержащийся в составе метода, размещается между открывающей и закрывающей фигурными скобками.

Очередная строка кода приведена ниже. Заметьте, что она содержится в составе метода main( ).

```
System.out.println("Hello, world!");
```

Она выводит на экран строку "Hello, world!", завершая ее символом перевода строки. Вывод информации осуществляет встроенный метод println( ). В данном случае println( ) отображает переданную ему строку. Как вы вскоре увидите, с помощью метода println( ) можно выводить не только строки символов, но и данные других типов. Строка начинается с имен System.out. На данный момент объяснить их назначение достаточно трудно. System – это предопределенный класс, предоставляющий доступ к системным средствам, а out – выходной поток, связанный с консолью. Таким образом, System.out – это объект, инкапсулирующий вывод на консоль. Тот факт, что для определения консоли Java использует объект, еще раз подчеркивает объектную ориентацию данного языка.

Как вы, вероятно, догадались, в реальных Java-программах и апплетах вывод на консоль (как и ввод с консоли) используется достаточно редко. Поскольку современные вычислительные среды, как правило, реализуют оконные и графические системы, обмен данными с консолью используется



лишь в простых утилитах и в демонстрационных программах. Несколько позже вы узнаете о других способах генерации выходных данных средствами Java, но на данный момент мы ограничимся использованием методов, поддерживающих вывод на консоль.

Обратите внимание на то, что за выражением `println( )` следует точка с запятой. Этим символом заканчиваются все выражения Java. В других строках точка с запятой отсутствует лишь потому, что они не являются выражениями.

Первая закрывающая фигурная скобка завершает метод `main( )`, а вторая такая же скобка является признаком окончания определения класса `MyFirstProgram`.

В языке Java учитывается регистр символов. Если вы забудете это правило, то столкнетесь с серьезными проблемами. Например, если вместо `main` вы случайно введете слово `Main` или укажете `PrintLn` вместо `println`, код программы будет некорректным. Заметьте также, что компилятор Java компилирует классы, не содержащие метода `main( )`, но запустить их на выполнение невозможно. Таким образом, если вы допустите ошибку в имени `main`, компилятор скомпилирует программу и не сообщит об ошибке. Об ошибке вас проинформирует интерпретатор, когда не сможет найти метод `main ( )`.

#### Программа использующая аргументы командной строки

Создайте файл с java-программой, используя код из листинга 1.2. Правильно назовите файл.

##### *Листинг 1.2*

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

Запустите программу с аргументами командной строки, как показано на рис 1.10

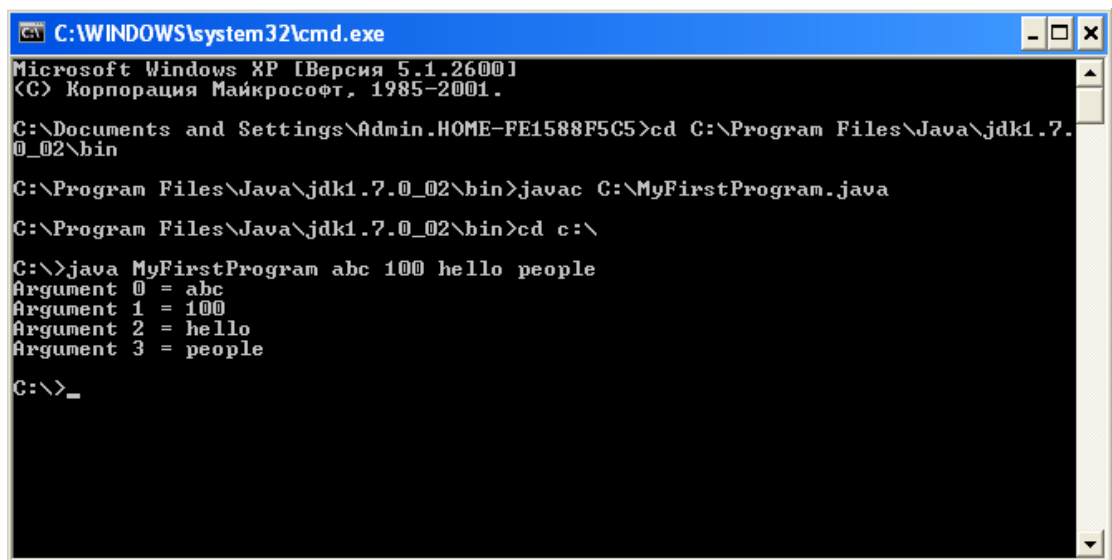


Рисунок 1.10 – Компиляция программы

## Тема 1.7. Создание программы в разных средах разработки.

### Создание простейшей программы в IDE Eclipse.

- 1) Создайте новый проект (рис. 1.11).
- 2) Выберите тип проекта – Java Project. (рис. 1.12.)
- 3) Выберите настройки Java-проекта (рис. 1.13)
- 4) Установите настройки построения проекта (рис. 1.14)
- 5) Создайте новый java-класс (рис. 1.15)
- 6) Задайте начальные характеристики класса (рис. 1.16)
- 7) Наберите текст программы (рис. 1.17)
- 8) На панели инструментов нажмите кнопку <Run>.
- 9) В открывшемся окне выберите тип проекта Java Application и нажмите кнопку <New launch configuration> (рис. 1.18)
- 10) Выберите запускаемый класс (рис. 1.19) и нажмите <OK> (рис. 1.20)
- 11) Результат работы программы будет показан в консольном окне (рис. 1.21).

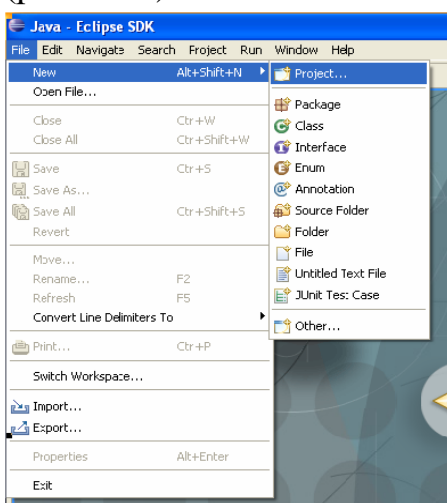
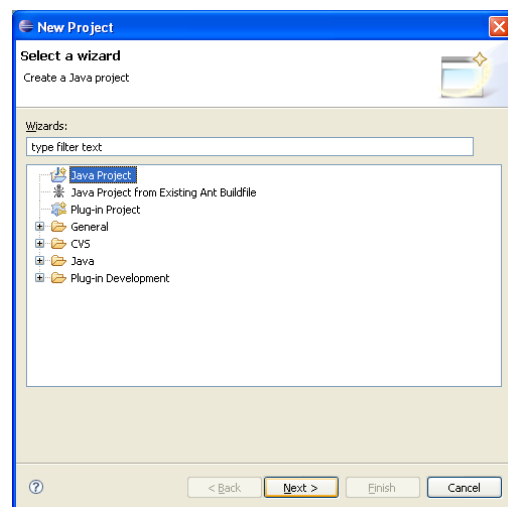


Рисунок 1.11



Рисинок 1.12

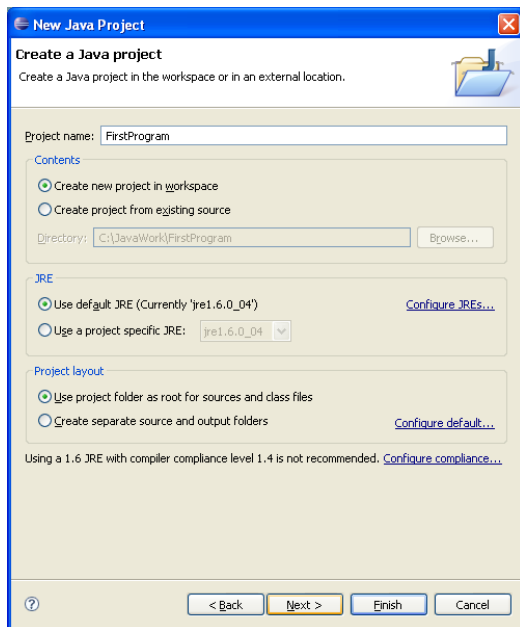


Рисунок 1.13

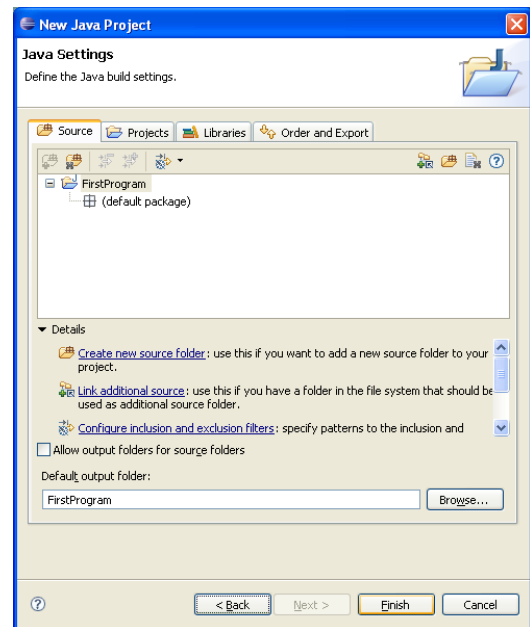


Рисунок 1.14

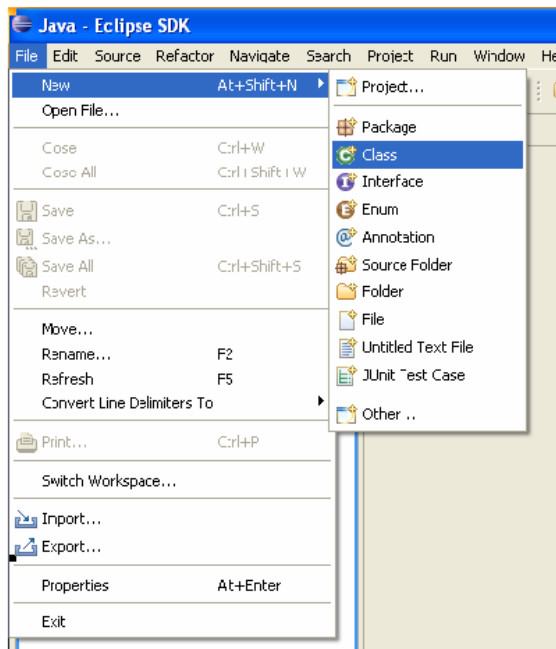


Рисунок 1.15

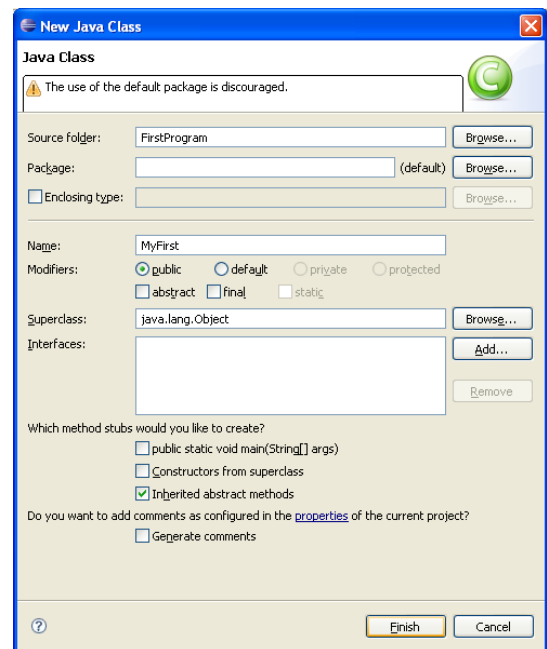


Рисунок 1.16

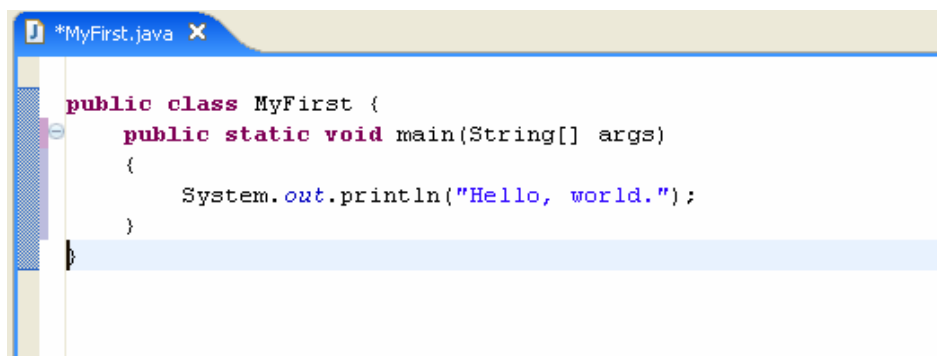


Рисунок 1.17

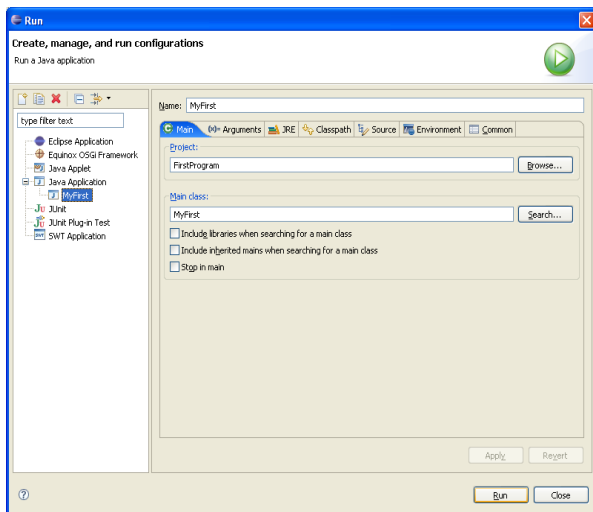


Рисунок 1.18

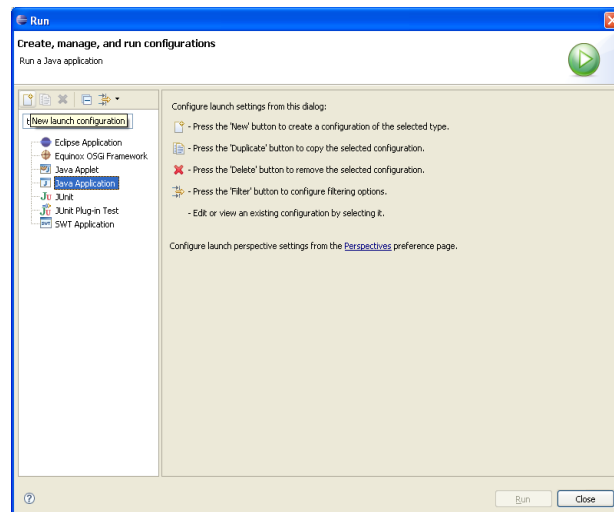


Рисунок 1.19

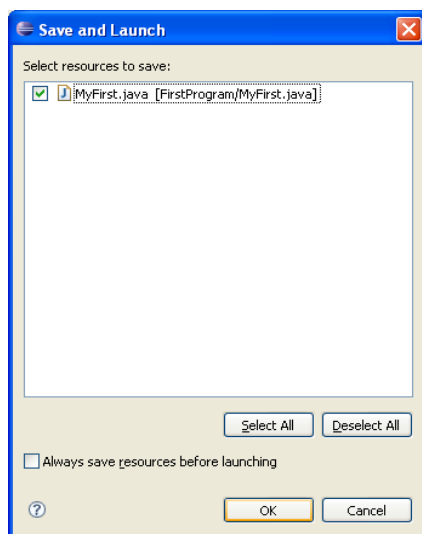


Рисунок 1.20



Рисунок 1.21

## Создание простейшего приложения в IDE NetBeans.

- 1) Создайте новый проект (рис. 1.22)
- 2) Укажите тип проекта <Java Application> (рис.1.23)
- 3) Укажите имя проекта и его расположение (рис. 1.24)
- 4) Напишите код приложения (рис. 1.25)
- 5) Постройте проект (рис. 1.26)
- 6) Запустите проект (рис. 1.27)
- 7) Результат работы программы будет выведен в консольное окно (рис. 1.28)
- 8) Ввод аргументов в IDE NetBeans (рис. 1.29 и рис. 1.30)

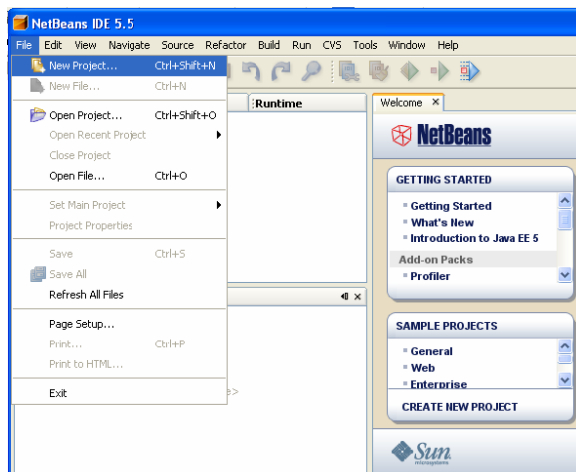


Рисунок 1.22

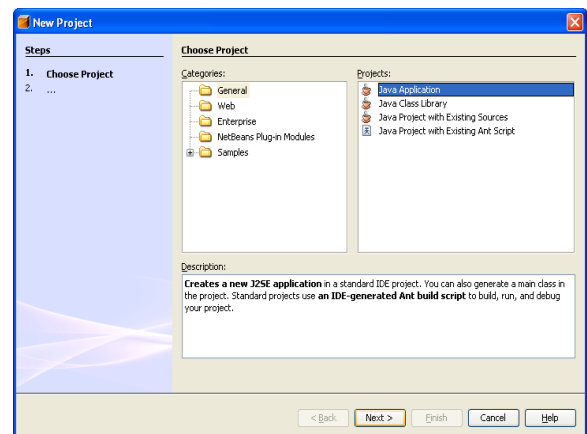


Рисунок 1.23

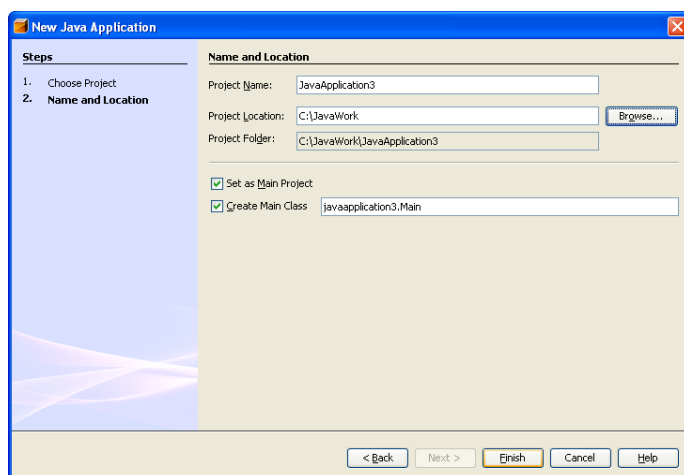


Рисунок 1.24

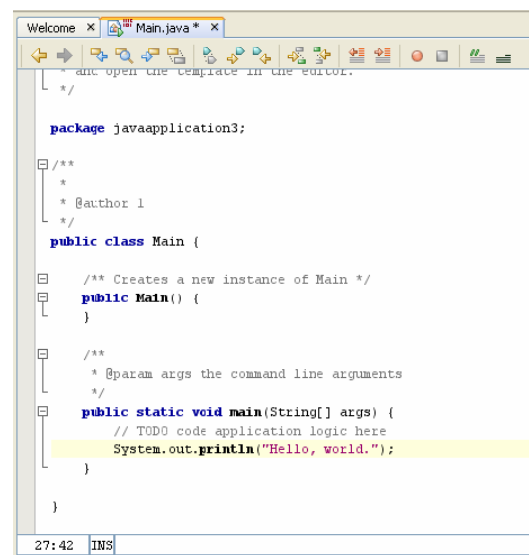


Рисунок 1.25

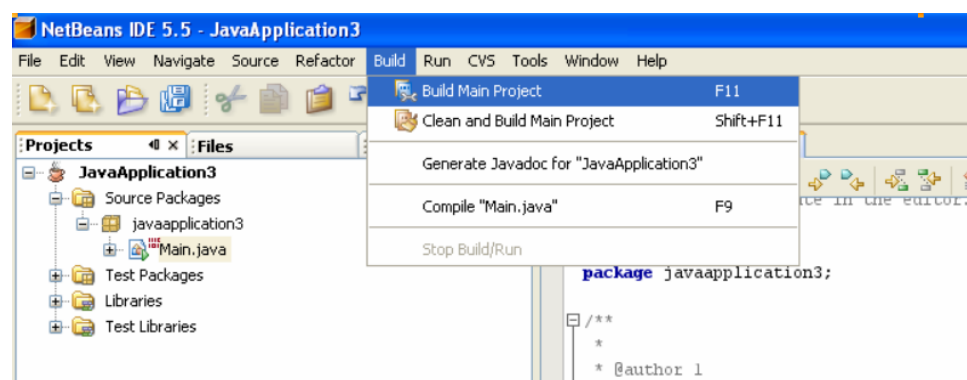


Рисунок 1.26

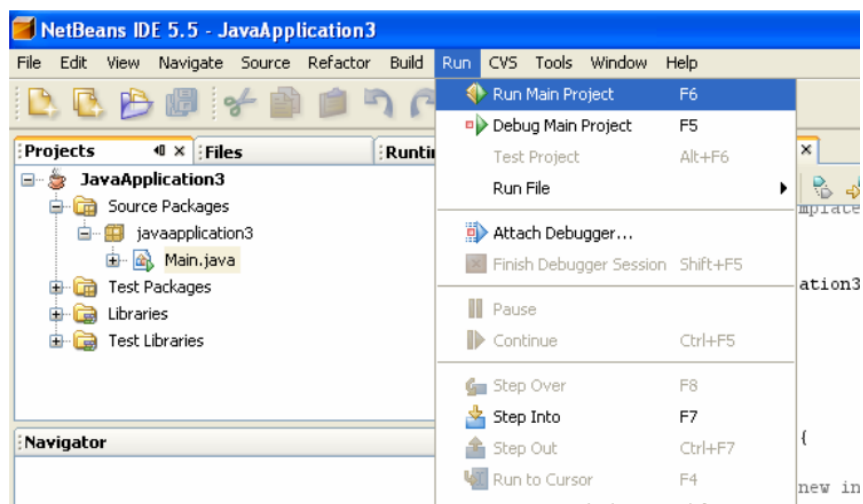


Рисунок 1.27

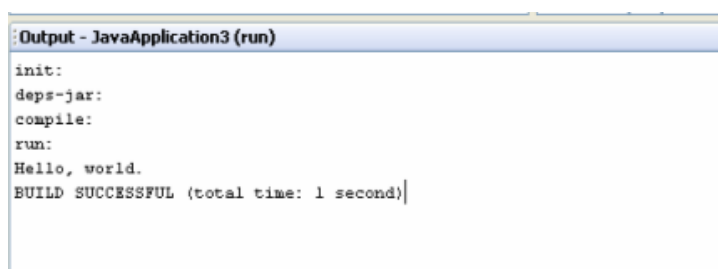


Рисунок 1.28

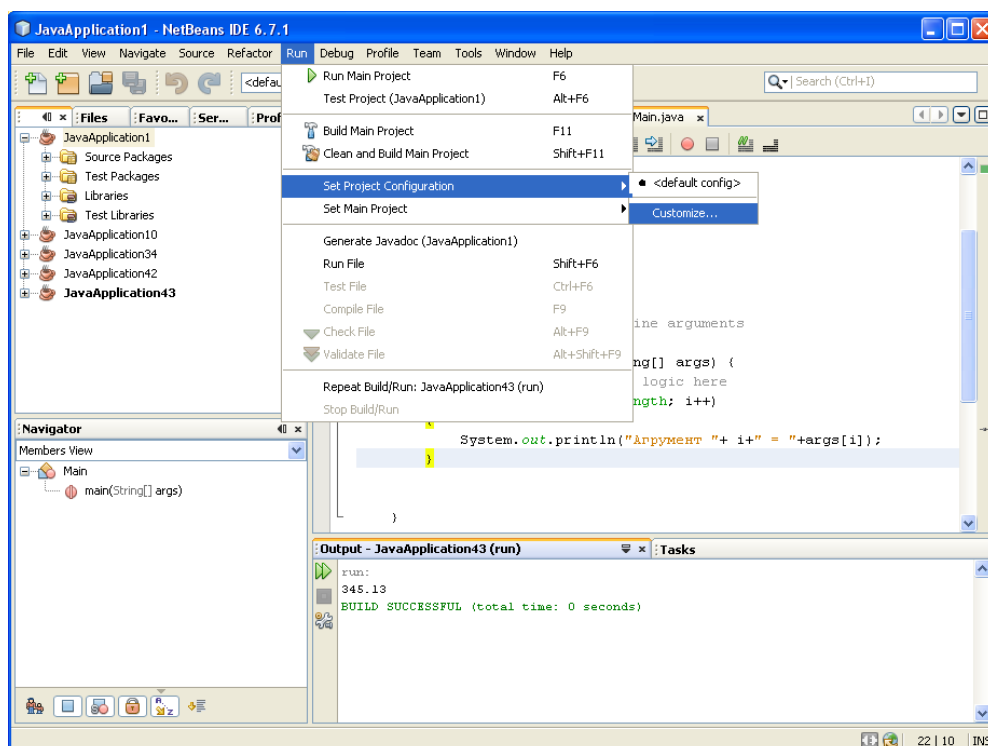


Рисунок 1.29

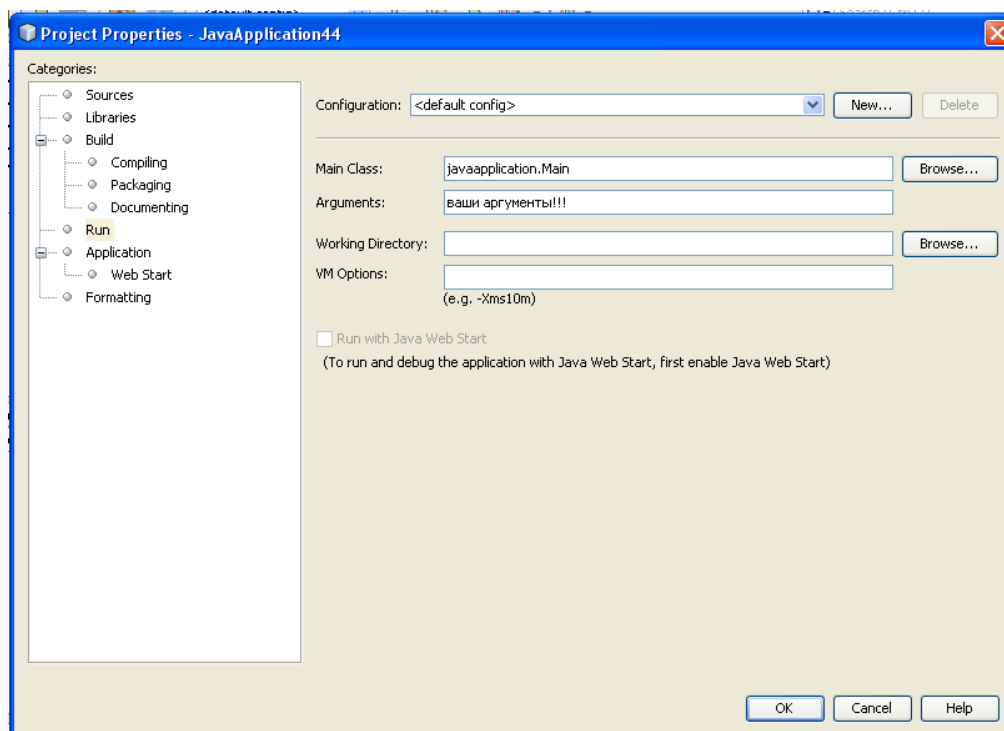


Рисунок 1.30

## Тема 1.8 Лексические основы языка

Программы на Java – это набор пробелов, комментариев, ключевых слов, идентификаторов, литеральных констант, операторов и разделителей.

### 1). Пробелы

Java – язык, который допускает произвольное форматирование текста программ. Для того, чтобы программа работала нормально, нет никакой необходимости выравнивать ее текст специальным образом. Например, класс HelloWorld можно было записать в двух строках или любым другим способом, который придется вам по душе. И он будет работать точно так же при условии, что между отдельными лексемами (между которыми нет операторов или разделителей) имеется по крайней мере по одному пробелу, символу табуляции или символу перевода строки.

### 2). Комментарии

Хотя комментарии никак не влияют на исполняемый код программы, при правильном использовании они оказываются весьма существенной частью исходного текста. Существует три разновидности комментариев: комментарии в одной строке, комментарии в нескольких строках и, наконец, комментарии для документирования. Комментарии, занимающие одну строку, начинаются с символов `//` и заканчиваются в конце строки. Такой стиль комментирования полезен для размещения кратких пояснений к отдельным строкам кода:

```
a = 42; // если 42 - ответ, то каков же был вопрос?
```

Для более подробных пояснений вы можете воспользоваться комментариями, размещенными на нескольких строках, начав текст

комментариев символами `/*` и закончив символами `*/` При этом весь текст между этими парами символов будет расценен как комментарий и транслятор его проигнорирует.

```
/*
 * Этот код несколько замысловат...
 * Попробую объяснить:
 * ....
 */
```

Третья, особая форма комментариев, предназначена для сервисной программы *javadoc*, которая использует компоненты Java-транслятора для автоматической генерации документации по интерфейсам классов. Соглашение, используемое для комментариев этого вида, таково: для того, чтобы разместить перед объявлением открытого (`public`) класса, метода или переменной документирующий комментарий, нужно начать его с символов `/**` (косая черта и две звездочки). Заканчивается такой комментарий точно так же, как и обычный комментарий – символами `*/`. Программа *javadoc* умеет различать в документирующих комментариях некоторые специальные переменные, имена которых начинаются с символа `@`. Вот пример такого комментария:

```
/**
 * Этот класс умеет делать замечательные вещи. Советуем всякому, кто
 * захочет написать еще более совершенный класс, взять его в качестве
 * базового.
 * @see Java. applet. Applet
 * ©author Patrick Naughton
 * @version 1. 2
 */
class CoolApplet extends Applet { /**
 * У этого метода два параметра:
 * @param key - это имя параметра.
 * @param value - это значение параметра с именем key.
 */ void put (String key, Object value) {
```

### 3). Зарезервированные ключевые слова

Зарезервированные ключевые слова – это специальные идентификаторы, которые в языке Java используются для того, чтобы идентифицировать встроенные типы, модификаторы и средства управления выполнением программы. На сегодняшний день в языке Java имеется 59 зарезервированных слов (см. таблицу 1). Эти ключевые слова совместно с синтаксисом операторов и разделителей входят в описание языка Java. Они могут применяться только по назначению, их нельзя использовать в качестве идентификаторов для имен переменных, классов или методов.



Таблица 1.1 – Зарезервированные слова Java

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

Отметим, что слова `byvalue`, `cast`, `const`, `future`, `generic`, `goto`, `inner`, `operator`, `outer`, `rest`, `var` зарезервированы в Java, но пока не используются. Кроме этого, в Java есть зарезервированные имена методов (эти методы наследуются каждым классом, их нельзя использовать, за исключением случаев явного переопределения методов класса `Object`).

Таблица 1.2 – Зарезервированные имена методов Java

<code>clone</code>	<code>equals</code>	<code>finalize</code>	<code>getClass</code>	<code>hashCode</code>
<code>notify</code>	<code>notifyAll</code>	<code>toString</code>	<code>wait</code>	

#### 4). Идентификаторы

Идентификаторы используются для именования классов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов `_` (подчеркивание) и `$` (доллар). Идентификаторы не должны начинаться с цифры, чтобы транслятор не перепутал их с числовыми литеральными константами, которые будут описаны ниже. Java – язык, чувствительный к регистру букв. Это означает, что, к примеру, `Value` и `VALUE` – различные идентификаторы.

#### 5). Литералы

Константы в Java задаются их литеральным представлением (рис 1.31). Целые числа, числа с плавающей точкой, логические значения, символы и строки можно располагать в любом месте исходного кода. Типы будут рассмотрены далее.

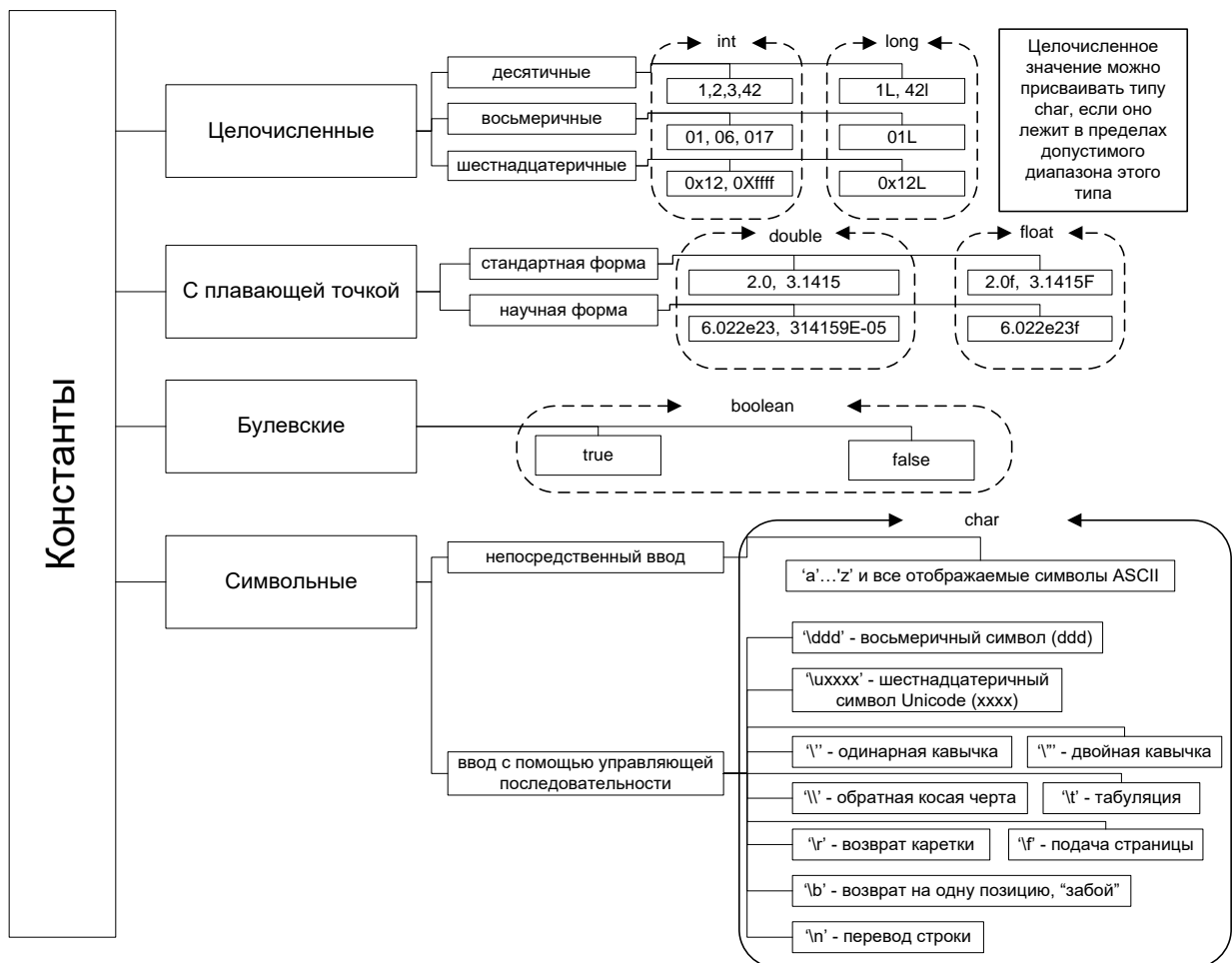


Рис 1.31 – Константы

### 5.1). Целые литералы

Целые числа – это тип, используемый в обычных программах наиболее часто. Любое целочисленное значение, например, 1, 2, 3, 42 – это целый литерал. В данном примере приведены десятичные числа, то есть числа с основанием 10 – именно те, которые мы повседневно используем вне мира компьютеров. Кроме десятичных, в качестве целых литералов могут использоваться также числа с основанием 8 и 16 – восьмеричные и шестнадцатеричные. Java распознает восьмеричные числа по стоящему впереди нулю. Нормальные десятичные числа не могут начинаться с нуля, так что использование в программе внешне допустимого числа 09 приведет к сообщению об ошибке при трансляции, поскольку 9 не входит в диапазон 0..7, допустимый для знаков восьмеричного числа. Шестнадцатеричная константа различается по стоящим впереди символам нуль-х (0x или 0X). Диапазон значений шестнадцатеричной цифры – 0..15, причем в качестве цифр для значений 10..15 используются буквы от A до F (или от a до f). С помощью шестнадцатеричных чисел вы можете в краткой и ясной форме представить значения, ориентированные на использование в компьютере, например, написав 0xffff вместо 65535.

Целые литералы являются значениями типа `int`, которое в Java хранится в 32-битовом слове. Если вам требуется значение, которое по модулю больше, чем приблизительно 2 миллиарда, необходимо воспользоваться константой типа `long`. При этом число будет храниться в 64-битовом слове. К числам с любым из названных выше оснований вы можете приписать справа строчную или прописную букву `L`, указав таким образом, что данное число относится к типу `long`. Например, `0x7fffffffffffffffL` или `9223372036854775807L` – это значение, наибольшее для числа типа `long`.

#### 5.2). Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения, у которых есть дробная часть. Их можно записывать либо в обычном, либо экспоненциальном форматах. В обычном формате число состоит из некоторого количества десятичных цифр, стоящей после них десятичной точки, и следующих за ней десятичных цифр дробной части. Например, `2.0`, `3.14159` и `.6667` – это допустимые значения чисел с плавающей точкой, записанных в стандартном формате. В экспоненциальном формате после перечисленных элементов дополнительно указывается десятичный порядок. Порядок определяется положительным или отрицательным десятичным числом, следующим за символом `E` или `e`. Примеры чисел в экспоненциальном формате: `6.022e23`, `314159E-05`, `2e+100`. В Java числа с плавающей точкой по умолчанию рассматриваются, как значения типа `double`. Если вам требуется константа типа `float`, справа к литералу надо приписать символ `F` или `f`. Если вы любитель избыточных определений – можете добавлять к литералам типа `double` символ `D` или `d`. Значения используемого по умолчанию типа `double` хранятся в 64-битовом слове, менее точные значения типа `float` – в 32-битовых.

#### 5.3). Логические литералы

У логической переменной может быть лишь два значения – `true` (истина) и `false` (ложь). Логические значения `true` и `false` не преобразуются ни в какое числовое представление. Ключевое слово `true` в Java не равно 1, а `false` не равно 0. В Java эти значения могут присваиваться только переменным типа `boolean` либо использоваться в выражениях с логическими операторами.

#### 5.4). Символьные литералы

Символы в Java – это индексы в таблице символов `UNICODE`. Они представляют собой 16-битовые значения, которые можно преобразовать в целые числа и к которым можно применять операторы целочисленной арифметики, например, операторы сложения и вычитания. Символьные литералы помещаются внутри пары апострофов (`' '`). Все видимые символы таблицы `ASCII` можно прямо вставлять внутрь пары апострофов: `'a'`, `'z'`, `'@'`. Для символов, которые невозможно ввести непосредственно, предусмотрено несколько управляющих последовательностей.

Таблица 1.3 – Управляющие последовательности символов

Управляющая последовательность	Описание
<code>\ddd</code>	Восьмеричный символ (ddd)
<code>\uxxxx</code>	Шестнадцатиричный символ UNICODE (xxxx)
<code>\'</code>	Апостроф
<code>\"</code>	Кавычка
<code>\\</code>	Обратная косая черта
<code>\r</code>	Возврат каретки (carriage return)
<code>\n</code>	Перевод строки (line feed, new line)
<code>\f</code>	Перевод страницы (form feed)
<code>\t</code>	Горизонтальная табуляция (tab)
<code>\b</code>	Возврат на шаг (backspace)

### 5.5). Строчные литералы

Строчные литералы в Java выглядят точно также, как и во многих других языках – это произвольный текст, заключенный в пару двойных кавычек ("""). Хотя строчные литералы в Java реализованы весьма своеобразно (Java создает объект для каждой строки), внешне это никак не проявляется. Примеры строчных литералов: "Hello World!"; "две\строки; \ А это в кавычках\"". Все управляющие последовательности и восьмеричные / шестнадцатиричные формы записи, которые определены для символьных литералов, работают точно так же и в строках. Строчные литералы в Java должны начинаться и заканчиваться в одной и той же строке исходного кода. В этом языке, в отличие от многих других, нет управляющей последовательности для продолжения строкового литерала на новой строке.

### 6). Операторы

Оператор – это нечто, выполняющее некоторое действие над одним или двумя аргументами и выдающее результат. Синтаксически операторы чаще всего размещаются между идентификаторами и литералами. Детально операторы будут рассмотрены далее, их перечень приведен в таблице 4.

Таблица 1.4 – Операторы языка Java

+	+=	-	-=
*	*=	/	/=
	=	^	^=
&	&=	%	%=

>	>=	<	<=
!	!=	++	--
>>	>>=	<<	<<=
>>>	>>>=	&&	
==	=	~	?:
	instanceof	[ ]	

### 7). Разделители

Лишь несколько групп символов, которые могут появляться в синтаксически правильной Java-программе, все еще остались неназванными. Это – простые разделители, которые влияют на внешний вид и функциональность программного кода.

Таблица 1.5 – Разделители языка Java

Символы	Название	Для чего применяются
( )	круглые скобки	Выделяют списки параметров в объявлении и вызове метода, также используются для задания приоритета операций в выражениях, выделения выражений в операторах управления выполнением программы, и в операторах приведения типов.
{ }	фигурные скобки	Содержат значения автоматически инициализируемых массивов, также используются для ограничения блока кода в классах, методах и локальных областях видимости.
[ ]	квадратные скобки	Используются в объявлениях массивов и при доступе к отдельным элементам массива.
;	точка с запятой	Разделяет операторы.
,	запятая	Разделяет идентификаторы в объявлениях переменных, также используется для связи операторов в заголовке цикла for.
.	точка	Отделяет имена пакетов от имен подпакетов и классов, также используется для отделения имени переменной или метода от имени переменной.

### 8). Переменные

Переменная – это основной элемент хранения информации в Java-программе. Переменная характеризуется комбинацией идентификатора, типа и области действия. В зависимости от того, где вы объявили переменную, она

может быть локальной, например, для кода внутри цикла for, либо это может быть переменная экземпляра класса, доступная всем методам данного класса. Локальные области действия объявляются с помощью фигурных скобок. Объявление переменной показано на рис 1.32.



Рисунок 1.32

Основная форма объявления переменной такова:

тип идентификатор [ = значение ] [, идентификатор [ = значение ]...];

*Тип* – это либо один из встроенных типов, то есть, byte, short, int, long, char, float, double, boolean, либо имя класса или интерфейса. Мы подробно обсудим все эти типы далее. Ниже приведено несколько примеров объявления переменных различных типов. Обратите внимание на то, что некоторые примеры включают в себя инициализацию начального значения. Переменные, для которых начальные значения не указаны, автоматически инициализируются нулем.

Таблица 1.6 – Пример объявления переменных разных типов

int a, b, c;	Объявляет три целых переменных a, b, c.
int d = 3, e, f = 5;	Объявляет еще три целых переменных, инициализирует d и f.
byte z = 22;	Инициализирует z.
double pi = 3. 14159;	Объявляет число пи (не очень точное, но все таки пи).
char x = 'x';	Переменная x получает значение 'x'.

В данном простом примере(листинг 1.3) объявлены две переменные. Первой переменной присваивается значение 2012, второй – 2012/2. Значения обоих переменных выводятся на консоль.

Листинг 1.3

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

```

int a;
int b;
a = 2012;
System.out.println("a = " + a);
b = a / 2;
System.out.print("b = a/2 = " + b);
}
}

```

## Задачи

1. Создать класс Hello, который будет приветствовать кого угодно, в зависимости от пожелания пользователя.
2. Написать приложение, которое отображает в окне консоли аргументы командной строки метода main() в обратном порядке.
3. Написать приложение, которое отображает в окне консоли только последний аргумент командной строки метода main().
4. Написать приложение, которое будет вычислять и выводить значение по формуле:  

$$a = 4 * (b + c - 1) / 2;$$
b и c задаем константами в коде самостоятельно.

## Тема 1.9 Элементарные типы данных.

Типы данных в Java платформенно-независимые. Элементарных, или простых, типов всего восемь (рис. 1.33)

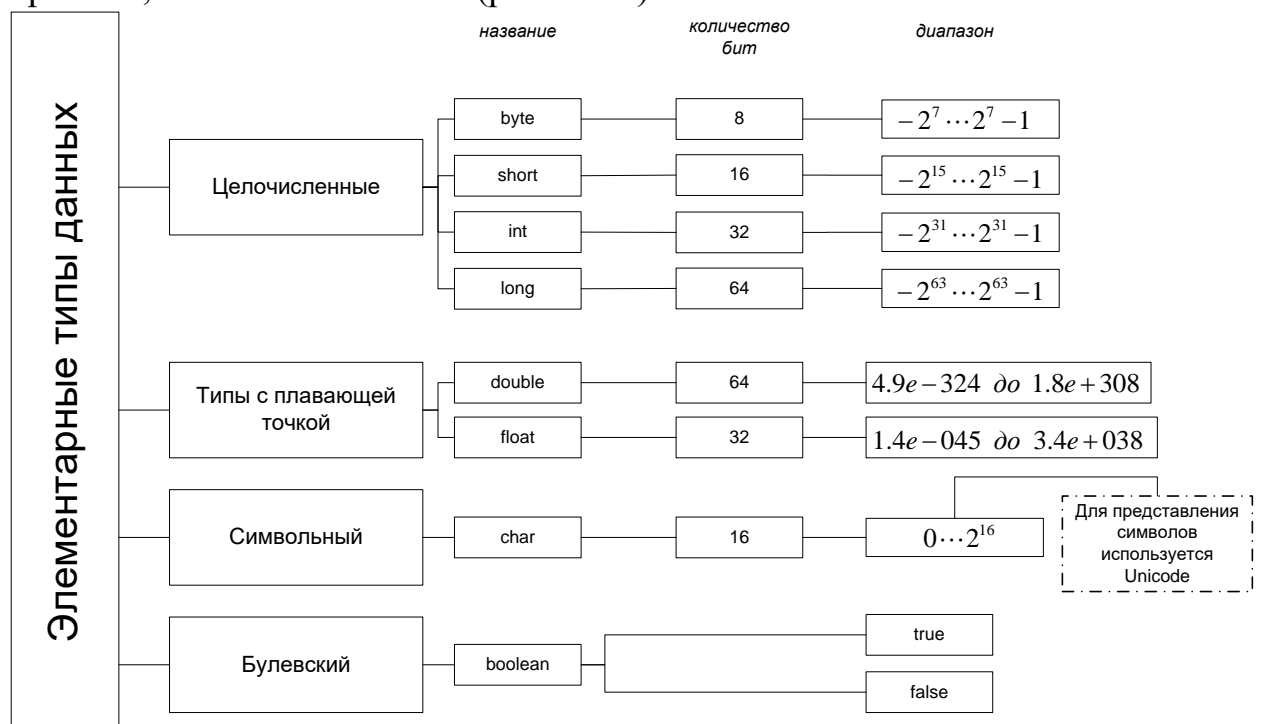


Рисунок 1.33 – Элементарные типы данных

Все типы исходных данных, встроенные в язык Java, разделяются на две группы: *элементарные типы* (primitive types) и *ссылочные типы* (reference types).

Ссылочные типы разделяют на *массивы* (arrays), *классы* (classes) и *интерфейсы* (interfaces).

В Java имеется восемь простых типов: – byte, short, int, long, char, float, double и boolean. Их можно разделить на четыре группы:

Целые. К ним относятся типы – byte, short, int и long. Эти типы предназначены для целых чисел со знаком.

Типы с плавающей точкой – float и double. Они служат для представления чисел, имеющих дробную часть.

Символьный тип char. Этот тип предназначен для представления элементов из таблицы символов, например, букв или цифр.

Логический тип boolean. Это специальный тип, используемый для представления логических величин.

В Java, в отличие от некоторых других языков, отсутствует автоматическое приведение типов. Несовпадение типов приводит не к предупреждению при трансляции, а к сообщению об ошибке. Для каждого типа строго определены наборы допустимых значений и разрешенных операций.

### **Целые числа**

В языке Java понятие беззнаковых чисел отсутствует. Все числовые типы этого языка – знаковые. Например, если значение переменной типа byte равно в шестнадцатичном виде 0x80, то это – число -1.

Единственная реальная причина использования беззнаковых чисел – это использование иных, по сравнению со знаковыми числами, правил манипуляций с битами при выполнении операций сдвига. Пусть, например, требуется сдвинуть вправо битовый массив mask, хранящийся в целой переменной и избежать при этом расширения знакового разряда, заполняющего старшие биты единицами. Стандартный способ выполнения этой задачи в C – ((unsigned) mask) >> 2. В Java для этой цели введен новый оператор беззнакового сдвига вправо. Приведенная выше операция записывается с его помощью в виде mask>>>2. Детально мы обсудим все операторы далее.

Отсутствие в Java беззнаковых чисел вдвое сокращает количество целых типов. В языке имеется 4 целых типа, занимающих 1, 2, 4 и 8 байтов в памяти. Для каждого типа – byte, short, int и long, есть свои естественные области применения.

#### **1). byte**

Тип byte – это знаковый 8-битовый тип. Его диапазон – от -128 до 127. Он лучше всего подходит для хранения произвольного потока байтов, загружаемого из сети или из файла.

*byte a;*

*byte b = 0x55;*



*byte c = 120;*

Если речь не идет о манипуляциях с битами, использования типа *byte*, как правило, следует избегать. Для нормальных целых чисел, используемых в качестве счетчиков и в арифметических выражениях, гораздо лучше подходит тип *int*.

#### 2). *short*

*Short* – это знаковый 16-битовый тип. Его диапазон – от -32768 до 32767. Это, вероятно, наиболее редко используемый в Java тип, поскольку он определен, как тип, в котором старший байт стоит первым.

*short a;*

*short b = 0x55aa;*

*short c = 32120;*

Случилось так, что на ЭВМ различных архитектур порядок байтов в слове различается, например, старший байт в двухбайтовом целом *short* может храниться первым, а может и последним. Первый случай имеет место в архитектурах SPARC и Power PC, второй – для микропроцессоров Intel x86. Переносимость программ Java требует, чтобы целые значения одинаково были представлены на ЭВМ разных архитектур.

#### 3). *int*

Тип *int* служит для представления 32-битных целых чисел со знаком. Диапазон допустимых для этого типа значений – от -2147483648 до 2147483647. Чаще всего этот тип данных используется для хранения обычных целых чисел со значениями, достигающими двух миллиардов. Этот тип прекрасно подходит для использования при обработке массивов и для счетчиков. В ближайшие годы этот тип будет прекрасно соответствовать машинным словам не только 32-битовых процессоров, но и 64-битовых с поддержкой быстрой конвейеризации для выполнения 32-битного кода в режиме совместимости. Всякий раз, когда в одном выражении фигурируют переменные типов *byte*, *short*, *int* и целые литералы, тип всего выражения перед завершением вычислений приводится к *int*.

*int a;*

*int b = 0x55aa0000;*

*int c = 2000000000;*

#### 4). *long*

Тип *long* предназначен для представления 64-битовых чисел со знаком. Его диапазон допустимых значений достаточно велик даже для таких задач, как подсчет числа атомов во вселенной.

*long a;*

*long b = 0x55aa000055aa0000;*

Не надо отождествлять *разрядность* целочисленного типа с занимаемым им количеством памяти. Исполняющий код Java может использовать для ваших переменных то количество памяти, которое сочтет нужным, лишь бы только их поведение соответствовало *поведению* типов, заданных вами. Фактически, нынешняя реализация Java из соображений эффективности хранит переменные типа *byte* и *short* в виде 32-битовых значений, поскольку

этот размер соответствует машинному слову большинства современных компьютеров (СМ – 8 бит, 8086 – 16 бит, 80386/486 – 32 бит, Pentium – 64 бит).

### Числа с плавающей точкой

Числа с плавающей точкой, часто называемые в других языках вещественными числами, используются при вычислениях, в которых требуется использование дробной части. В Java реализован стандартный (IEEE-754) набор типов для чисел с плавающей точкой – float и double и операторов для работы с ними.

#### 1). float

В переменных с обычной, или *одинарной точностью*, объявляемых с помощью ключевого слова float, для хранения вещественного значения используется 32 бита.

```
float a;
```

```
float b = 3.14F; // обратите внимание на F, т.к. по умолчанию все  
литералы double
```

#### 2). double

В случае *двойной точности*, задаваемой с помощью ключевого слова double, для хранения значений используется 64 бита. Все *трансцендентные* математические функции, такие, как sin, cos, sqrt, возвращают результат типа double.

```
double a;
```

```
double b = 3.14159265358979323846;
```

К обычным вещественным числам добавляются еще три значения»:

1. Положительная бесконечность – POSITIVE\_INFINITY , которая возникает при переполнении положительного значения, например, в результате операции умножения 3.0\*6e307.

2. Отрицательная бесконечность – NEGATIVE\_INFINITY.

3. "Не число", записываемое константой NaN (Not a Number), которое возникает при делении вещественного числа на нуль или умножении нуля на бесконечность.

Кроме того, стандарт различает положительный и отрицательный нули, возникающие при делении на бесконечность соответствующего знака, хотя сравнение 0.0 == -0.0 дает true.

Операции с бесконечностями выполняются по обычным математическим правилам. Во всем остальном вещественные типы – это обычные, вещественные значения, к которым применимы все арифметические операции и сравнения, перечисленные для целых типов.

Простая программа(листинг 1.4), помогающая понять, что такое POSITIVE\_INFINITY, NEGATIVE\_INFINITY и NaN.

*Листинг 1.4*

```
public class MyFirst {  
    public static void main(String[] args) {  
        double i = 7.0;  
        double j, z, k;
```

```

        j = i / 0;
        z = -i / 0;
        k = Math.sqrt(-i);
        if (j == Double.POSITIVE_INFINITY) {
            System.out.println("Мы получили положительную
бесконечность.");
        }
        if (z == Double.NEGATIVE_INFINITY) {
            System.out.println("Мы получили отрицательную
бесконечность.");
        }
        if (Double.isNaN(k)) {
            System.out.println("Мы получили не число.");
        }
        System.out.println("j=" + j + " z=" + z + " k=" + k);
    }
}

```

## Символы

Поскольку в Java для представления символов в строках используется кодировка Unicode, разрядность типа `char` в этом языке – 16 бит. В нем можно хранить десятки тысяч символов интернационального набора символов Unicode. Диапазон типа `char` – 0..65536. Unicode – это объединение десятков кодировок символов, он включает в себя латинский, греческий, арабский алфавиты, кириллицу и многие другие наборы символов.

```

char a;
char b = 0xf132;
char c = 'a';
char d = '\n';

```

Хотя величины типа `char` и не используются, как целые числа, вы можете оперировать с ними так, как если бы они были целыми. Это дает вам возможность сложить два символа вместе, или инкрементировать значение символьной переменной. В приведенном ниже фрагменте кода мы, располагая базовым символом, прибавляем к нему целое число, чтобы получить символьное представление нужной нам цифры.

```

int three = 3;
char one = '1';
char four = (char) (three + one);

```

В результате выполнения этого кода в переменную `four` заносится символьное представление нужной нам цифры – '4'. Обратите внимание – тип переменной `one` в приведенном выше выражении повышается до типа `int`, так что перед занесением результата в переменную `four` приходится использовать оператор явного приведения типа.

## Тип `boolean`

Значения логического типа `boolean` возникают в результате различных сравнений, вроде `2 > 3`, и используются, главным образом, в условных операторах и операторах циклов. Логических значений всего два: `true` (истина) и `false` (ложь). Это служебные слова. Описание переменных этого типа выглядит так:

```
boolean a = true;  
boolean b = false;
```

Над логическими данными можно выполнять операции присваивания, например, `bool2 = true`, в том числе и составные с логическими операциями; сравнение на равенство `b == bb` и на неравенство `b != bb`, а также логические операции.

### Тема 1.10 Преобразование типов.

Java запрещает смешивать в выражениях величины разных типов, однако при числовых операциях такое часто бывает необходимо. Различают повышающее (разрешенное, неявное) преобразование и понижающее приведение типа.

Повышающее преобразование осуществляется автоматически по следующему правилу (рис. 1.34). Серыми стрелками обозначены преобразования, при которых может произойти потеря точности.

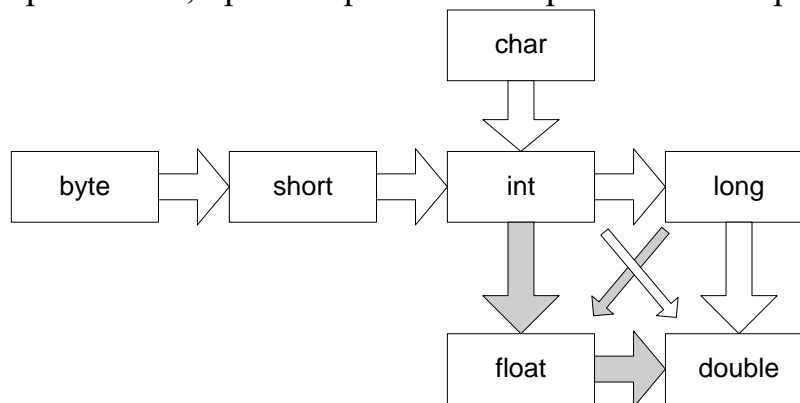


Рисунок 1.34 – Преобразование типов

Преобразования между типами, несоединенными стрелками, неявно запрещены. Программист, проводя явное преобразование, берет ответственность за корректность преобразования на себя. Явное приведение типа имеет вид:

*(целевой тип)значение*

Для примера, рассмотрим программу, приведенную в листинге 1.5.

*Листинг 1.5*

```
public class Conversion {  
    public static void main(String[] args) {  
        byte b;  
        int i=257;
```

```

double d=323.142;
b=(byte)i;
i=(int)d;
b=(byte)d;
}
}

```

### Задание:

Модифицируйте код программы так, чтобы видеть на консоли результаты приведения типов. Прокомментируйте результаты.

### Арифметические операции

К арифметическим операциям относятся:

сложение + (плюс);  
 вычитание - (дефис);  
 умножение \* (звездочка);  
 деление / (наклонная черта – слэш);  
 взятие остатка от деления (деление по модулю) % (процент);  
 инкремент (увеличение на единицу) ++;  
 декремент (уменьшение на единицу) --.

Между сдвоенными плюсами и минусами нельзя оставлять пробелы. Сложение, вычитание и умножение целых значений выполняются как обычно, а вот деление целых значений в результате даст опять целое (так называемое "*целое деление*"), например, 5/2 даст в результате 2, а не 2.5, а 5/(-3) даст -1. Дробная часть числа попросту отбрасывается, происходит усечение частного, т.к. в Java принято целочисленное деление. Это странное для математики правило естественно для программирования: если оба операнда имеют один и тот же тип, то и результат имеет тот же тип. Достаточно написать 5/2.0 или 5.0/2 или 5.0/2.0 и получим 2.5 как результат деления вещественных чисел.

Операция *деление по модулю* определяется так:

$a \% b = a - (a / b) * b$

Например, 5%2 даст в результате 1, а 5% (-3) даст, 2, т.к.

$5 = (-3)*(-1) + 2$ , но  $(-5)\%3$  даст -2, поскольку  $-5 = 3 * (-1) - 2$ .

Операции *инкремент* и *декремент* означают увеличение или уменьшение значения переменной на единицу и применяются только к переменным, но не к константам или выражениям, нельзя написать 5++ или (a + b)++.

При *постфиксной* первой форме записи в выражении участвует старое значение переменной и только потом происходит увеличение или уменьшение ее значения. При второй форме записи (*префиксной*) сначала изменится переменная, и ее новое значение будет участвовать в выражении.

Например, k=10000; (k++) + 5 даст в результате 10005, а переменная k примет значение 10001. Но в той же исходной ситуации (++k) + 5 даст 10006, а переменная k станет равной 10001.

### *Операции сравнения*

В языке Java шесть обычных операций сравнения целых чисел по величине:

больше >;  
меньше <;  
больше или равно >=;  
меньше или равно <=;  
равно ==;  
не равно !=.

Сдвоенные символы записываются без пробелов, их нельзя переставлять местами, запись => будет неверной.

Результат сравнения будет иметь логическое значение: true для выражения  $3 \neq 5$  ; или false , например, для  $3 == 5$ .

Для записи сложных сравнений следует использовать логические операции. Например, в вычислениях часто приходится делать проверки вида  $a < x < b$ . Подобная запись на языке Java приведет к сообщению об ошибке, поскольку первое сравнение  $a < x$  даст true или false, а Java не знает, больше это, чем b, или меньше. В данном случае следует написать выражение  $(a < x) \&\& (x < b)$ , причем здесь скобки можно опустить, написав  $a < x \&\& x < b$ .

### **Задачи**

1. В переменной n хранится двузначное число. Создайте программу, вычисляющую и выводящую на экран сумму цифр n.
2. В переменной n хранится трёхзначное число. Создайте программу, вычисляющую и выводящую на экран сумму цифр n.
3. В переменной n хранится вещественное число с ненулевой дробной частью. Создайте программу, округляющую число n до ближайшего целого и выводящую результат на экран.
4. В переменных q и w хранятся два натуральных числа. Создайте программу, выводящую на экран результат деления q на w с остатком. Пример вывода программы (для случая, когда в q хранится 21, а в w хранится 8):

$21 / 8 = 2$  и 5 в остатке

## **Тема 1.11 Консольный ввод с помощью класса java.util.Scanner**

Для ввода данных используется класс Scanner из библиотеки пакетов java.util.

Этот класс надо импортировать в той программе, где он будет использоваться. Это делается до начала открытого класса в коде программы.

В классе есть методы для чтения очередного символа заданного типа со стандартного потока ввода, а также для проверки существования такого символа.

Для работы с потоком ввода необходимо создать объект класса `Scanner`, при создании указав, с каким потоком ввода он будет связан. Стандартный поток ввода (клавиатура) в Java представлен объектом – `System.in`. А стандартный поток вывода (дисплей) – уже знакомым вам объектом `System.out`. Есть ещё стандартный поток для вывода ошибок – `System.err`, но он будет рассмотрен в теме обработка исключительных ситуаций.

*Листинг 1.9*

```
import java.util.Scanner; // импортируем класс
public class ScannerDemo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in); // создаём объект класса
        Scanner
        int i = 2;
        System.out.print("Введите целое число: ");
        if (sc.hasNextInt()) { // возвращает истинну если с потока ввода
            можно считать целое число
            i = sc.nextInt(); // считывает целое число с потока ввода и
            сохраняем в переменную
            System.out.println(i * 2);
        } else {
            System.out.println("Вы ввели не целое число");
        }
    }
}
```

Метод `hasNextDouble()`, применённый объекту класса `Scanner`, проверяет, можно ли считать с потока ввода вещественное число типа `double`, а метод `nextDouble()` – считывает его. Если попытаться считать значение без предварительной проверки, то во время исполнения программы можно получить ошибку (отладчик заранее такую ошибку не обнаружит). Например, попробуйте в представленной далее программе (листинг 1.10) ввести какое-то вещественное число

*Листинг 1.10*

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        double i = sc.nextDouble(); // если ввести букву s, то случится
        ошибка во время исполнения
        System.out.println(i / 3);
    }
}
```

Имеется также метод `nextLine()`, позволяющий считывать целую последовательность символов, т.е. строку, а, значит, полученное через этот метод значение нужно сохранять в объекте класса `String`. В следующем

примере создаётся два таких объекта, потом в них поочередно записывается ввод пользователя, а далее на экран выводится одна строка, полученная объединением введённых последовательностей символов.

*Листинг 1.11*

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

Существует и метод `hasNext()`, проверяющий остались ли в потоке ввода какие-то символы.



## Тема 1.12 Операторы

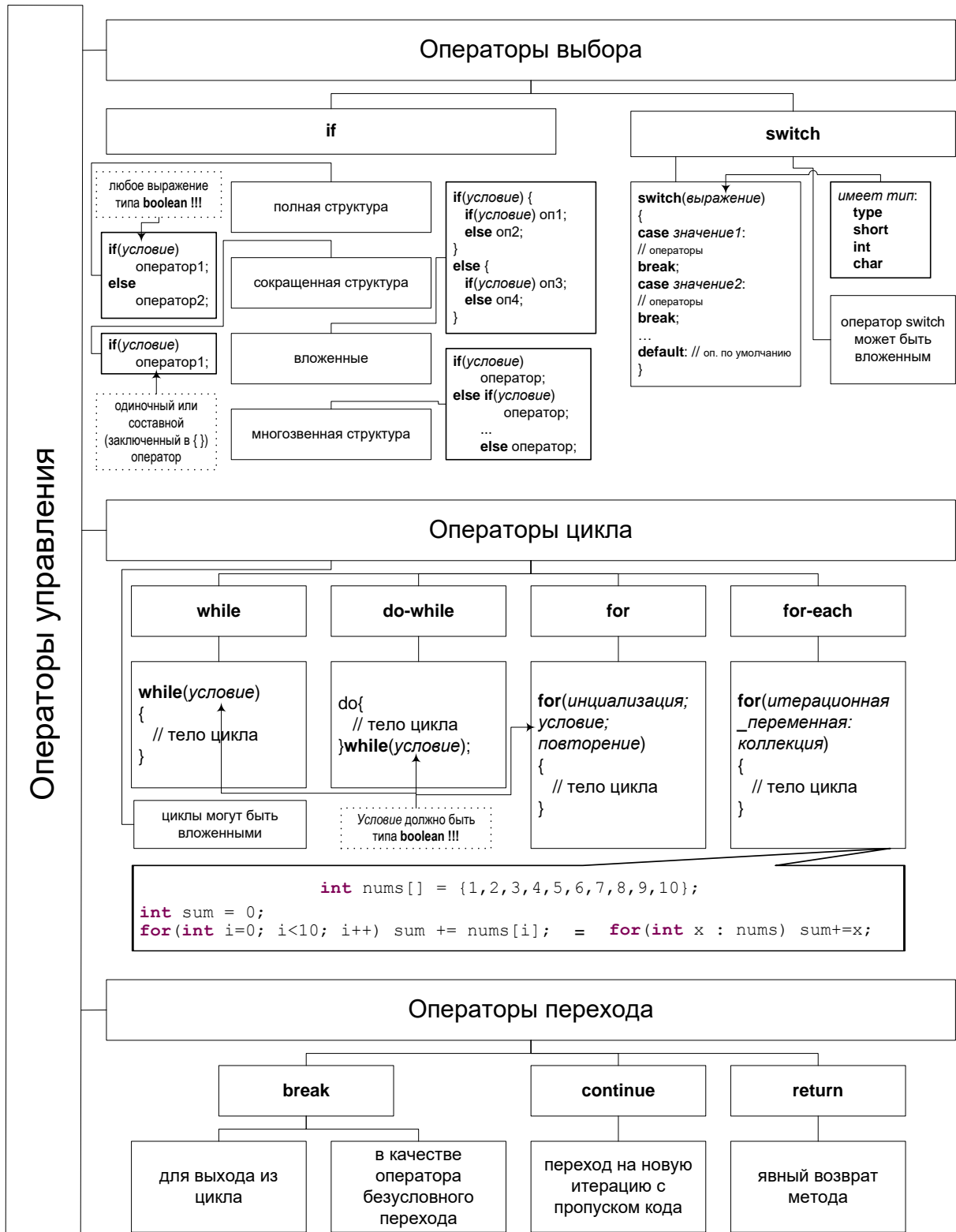


Рисунок 1.36 – Операторы управления

Как известно, любой алгоритм, можно разработать, используя линейные вычисления, разветвления и циклы. Записать его можно в разных формах: в виде блок-схемы, на псевдокоде или на обычном языке.

Всякий язык программирования должен иметь средства записи алгоритмов. Они называются *операторами* (statements) языка. Операторы управления языка Java показаны на рис 1.36. Минимальный набор

операторов должен содержать оператор для записи линейных вычислений, условный оператор для записи разветвлений и оператор цикла.

Обычно состав операторов языка программирования шире: для удобства записи алгоритмов в язык включаются несколько операторов цикла, оператор варианта, операторы перехода, операторы описания объектов. В языке Java нет оператора goto.

Все операторы языка Java можно разделить на:

- операторы описания переменных и других объектов;
- операторы-выражения;
- операторы присваивания;
- условные операторы if;
- операторы циклов while, do-while, for;
- операторы варианта switch;
- операторы перехода break, continue и return;
- блок {};
- пустые операторы – просто точка с запятой.

Всякий оператор завершается точкой с запятой. Можно поставить точку с запятой в конце любого выражения, и оно станет оператором. Но смысл это имеет только для операций присваивания, инкремента и декремента и вызовов методов. В остальных случаях это бесполезно, потому что вычисленное значение выражения потеряется. Точка с запятой в Java не разделяет операторы, а является частью оператора.

Линейное выполнение алгоритма обеспечивается последовательной записью операторов. Переход со строки на строку в исходном тексте не имеет никакого значения для компилятора, он осуществляется только для наглядности и читаемости текста.

### **1.12.1 Блок**

Блок может содержать в себе нуль или несколько операторов с целью их использования как одного оператора в тех местах, где по правилам языка можно записать только один оператор. Например, {x=5; y=6;}. Можно записать и пустой блок – {}.

Блоки операторов часто используются для ограничения области действия переменных и просто для улучшения читаемости текста программы.

### **1.12.2 Условный оператор if**

Условный оператор в Java записывается так:

```
if (условие) {  
    оператор1;  
}else {  
    оператор2;  
}
```

и действует следующим образом. Сначала проверяется условие. Если результат true, то выполняется *оператор1* и на этом выполнение условного оператора завершается, *оператор2* не выполняется, далее будет выполняться следующий за if оператор. Если результат false, то действует *оператор2*, при этом *оператор1* вообще не выполняется.

Условный оператор может быть сокращенным:

```
if (условие){  
    оператор;  
}
```

и в случае false не выполняется ничего.

Если оператор содержит только 1 строку, то фигурные скобки можно не ставить. Рекомендуется всегда использовать фигурные скобки и размещать оператор на нескольких строках с отступами, как показано в следующем примере:

```
if (a < x) {  
    x = a + b;  
} else {  
    x = a - b;  
}
```

Это облегчает добавление операторов в каждую ветвь при изменении алгоритма. Очень часто одним из операторов является снова условный оператор, например:

```
if (n == 0) {  
    sign = 0;  
} else if (n < 0) {  
    sign = -1;  
} else {  
    sign = 1;  
}
```

Вообще не стоит увлекаться сложными вложенными условными операторами, т.к. проверки условий занимают много времени. По возможности лучше использовать логические операции, например, в данном примере лучше написать следующее:

```
if (ind >= 10 && ind <= 20) {  
    x = 0;  
} else {  
    x = 1;  
}
```

**Задания:**

Определить, какое значение будет в переменной dd

1).

```
int dd, k1 = 3, k2 = 5, k3 = 7;
    if (k1 > 1) {
        if (k2 != 3) {
            dd = 4;
        } else {
            dd = 3;
        }
    } else {
        dd = 2;
    }
}
```

2).

```
int dd, k1 = 3, k2 = 5, k3 = 7;
if (k1 > 10) dd = 1;
else if (k1 % 3 == 1) dd = 2;
else if (k2 % 5 == 1) dd = 3;
else if (k3 % 3 == 2) dd = 4;
else dd = 5;
```

3).

```
int dd, k1 = 3, k2 = 5, k3 = 7;
if (k1 > 10) dd = 1;
else if (true) dd = 2;
else if (k2 % 5 == 1) dd = 3;
else if (k3 % 3 == 2) dd = 4;
else dd = 5;
```

4).

```
int dd, k1 = 6, k2 = 10, k3 = 21;
if (k1 > k2) dd = 1;
else if (k1 > k3) dd = 2;
else if (k1 + k2 > k3) dd = 3;
else dd = 4;
```

5).

```
int dd = 0;
int k = 44;
if (k % 2 == 0) dd = dd + 1;
if (k % 3 == 1) dd = dd + 2;
if (k % 5 == 4) dd = dd + 5;
if (k % 6 == 4) dd = dd + 9;
```

6).

```
int dd=0;
int k1=6;   int k2=10;   int k3=21;
if (k1>k2 || k3>k2) dd=dd+1;
if (k1 <k3&& k2!=10) dd=dd+2;
else dd=dd+3;
if (k1 % 5 == 4) dd=dd+5;
if (k2% 6 != 4) dd=dd+9;
```

### 1.12.3 if-else и ?

Оператор `?` называется тернарным оператором, поскольку он обрабатывает три операнда. Этот оператор записывается в следующем форме:

*выражение\_1 ? выражение\_2 : выражение\_3;*

где первое выражение должно быть логическим, т.е. возвращать тип `boolean` а второе и третье выражения, разделяемые двоеточием, могут быть любого за исключением `void`. Необходимо только, чтобы тип второго и третьего выражения совпадал.

Значение выражения `?` определяется следующим образом. Сначала вычисляется первое выражение. Если оно возвращает значение `true`, то вычисляется второе выражение и значение, возвращаемое им, становится значением всего выражения? Если значение первого выражения – `false`, то вычисляется третье выражение, которое становится значением всего выражения `?`. Рассмотрим пример, в котором вычисляется абсолютное значение `val` и присваивается переменной `absval`:

```
absval = val < 0 ? -val : val; // Получение абсолютного значения val.
```

где переменной `absval` присваивается значение `val`, если переменная `val` больше или равна нулю. Если значение `val` отрицательное, то переменной `absval` присваивается значение `val` со знаком "минус" (что в результате даст положительную величину). Код, решающий ту же самую задачу, но использующий структуру `if-else`, будет выглядеть следующим образом:

```
if(val < 0) absval = -val;
else absval = val;
```

Рассмотрим еще один пример использования оператора `?`. В листинге 1.18 выполняется деление двух чисел, но не допускается деление на нуль.

*Листинг 1.18*

*// Использование оператора ? для предотвращения деления на нуль*

```
public class NoZeroDiv {
    public static void main(String args[]) {
        int result;
        for (int i = -5; i < 6; i++) {
            result = i != 0 ? 100 / i : 0;
            if (i != 0) {
```

```

        System.out.println("100 / " + i + " is " + result);
    } else {
        System.out.println("i = " + result);
    }
}
}
}

```

Обратите внимание на следующую строку кода:

```
result = i != 0 ? 100 / i : 0;
```

где переменной `result` присваивается результат деления 100 на `i`. Однако деление выполняется только в том случае, если значение переменной `i` не равно нулю. В противном случае переменной `result` присваивается нулевое значение.

Выражение, возвращаемое оператором `?`, не обязательно присваивать переменной. Вы можете применить его, например, в качестве параметра при вызове метода. Если же все три выражения имеют тип `boolean`, то само выражение `?` может быть использовано в качестве условия для цикла или оператора `if`. Ниже приведена предыдущая программа, код которой несколько видоизменен (листинг 1.19). Она генерирует те же данные, которые были представлены ранее.

*Листинг 1.19*

// Использование оператора `?` для предотвращения деления на нуль

```

public class NoZeroDiv {
    public static void main(String args[]) {
        for (int i = -5; i < 6; i++) {
            if (i != 0 ? true : false) {
                System.out.println("100 / " + i + " is " + 100 / i);
            }
        }
    }
}

```

Обратите внимание на выражение `if`. Если значение переменной `i` равно нулю, то выражение `?` возвращает значение `false`, что предотвращает деление на нуль, и результат не отображается. В противном случае осуществляется обычное деление.

#### 1.12.4 Оператор цикла `while`

Основной оператор цикла – оператор `while` – выглядит так:

```

while (условие){
    оператор;
}

```

Вначале проверяется условие; если его значение `true`, то выполняется оператор, образующий цикл. Затем снова проверяется условие и действует оператор, и так до тех пор, пока не получится значение `false`. Если *логВыр*

изначально равняется false, то *оператор* не будет выполнен ни разу. Предварительная проверка обеспечивает безопасность выполнения цикла, позволяет избежать переполнения, деления на нуль и других неприятностей. Поэтому оператор while является основным, а в некоторых языках и единственным оператором цикла.

```
int i = 1, sum = 0;
while (i < 5 && sum < 4) {
    sum += i;
    i++;
}
System.out.println(sum);
```

Можно организовать бесконечный цикл:

while (true) оператор;

Конечно, из такого цикла следует предусмотреть какой-то выход, например, используя оператор break. В противном случае программа заикнется, в результате чего придется принудительно прекращать ее выполнение. В листинге 1.20 показаны примеры использования бесконечного цикла.

Листинг 1.20

```
public class Main {
    public static void main(String args[]) {
        int i = 0;
        while (true) {
            System.out.println("i=" + i);
            i++;
            if (i == 10) {
                break;
            }
        }
    }
}
```

или так:

```
public class Main {
    public static void main(String args[]) {
        int i = 0;
        boolean d = true;
        while (d) {
            System.out.println("i=" + i);
            i++;
            if (i == 10) {
                d = false;
            }
        }
    }
}
```

Если в цикл необходимо включить несколько операторов, то следует использовать блок операторов {}.

**Задания:**

1) С помощью цикла `while` и оператора `if` определяйте четность чисел и выводите их (числа от 1 до 10).

1-нечетное

2-четное

И т.д.

2) Напишите цикл `while` выводящий числа последовательности 1, 4, 7, 10... до тех пор, пока их произведение не превысит 300 или сумма 200. Выведите количество этих чисел.

3) Организуйте бесконечный цикл вычисляющий факториал числа введенного с клавиатуры. Выход из цикла с помощью `break`.

### 1.12.5 Оператор цикла `do-while`

Второй оператор цикла – оператор **`do-while`** – имеет вид

```
do{  
    оператор  
} while (условие);
```

Здесь сначала выполняется оператор, а потом происходит проверка условия. Цикл выполняется, пока *условие* остается равным `true`.

Существенное различие между этими двумя операторами цикла заключается в том, что в цикле `do-while` оператор обязательно выполнится хотя бы один раз. В листинге 1.21 показан вывод чисел кратных 3 в диапазоне от -10 до 10.

Листинг 1.21

```
public class Main {  
    public static void main(String args[]) {  
        int i = -10;  
        do {  
            if (i % 3 == 0) {  
                System.out.println("i=" + i);  
            }  
            i++;  
        } while (i <= 10);  
    }  
}
```

**Задание:**

С помощью цикла `do-while` создайте программу, выводящую на экран первые 10 элементов последовательности 2 4 8 16 32 64 128 ....



### 1.12.6 Оператор цикла for

Третий оператор цикла – оператор **for** – выглядит так:

```
for ( списокВыр ; условие; списокВыр2) {  
    оператор;  
}
```

Перед выполнением цикла вычисляется список выражений *списокВыр1*. Это нуль или несколько выражений, перечисленных через запятую. Они вычисляются слева направо, и в следующем выражении уже можно использовать результат предыдущего выражения. Как правило, здесь задаются начальные значения переменным цикла.

Затем проверяется условие. Если оно истинно, true, то выполняется оператор, потом вычисляются слева направо выражения из списка выражений *списокВыр2*. Далее снова проверяется *условие*. Если оно истинно, то выполняется оператор и *списокВыр2* и т. д. Как только *условие* станет равным false, выполнение цикла заканчивается.

Вместо *списокВыр1* может стоять одно определение переменных обязательно с начальным значением. Такие переменные известны только в пределах этого цикла.

```
for (int i = 0, j=10; i < j; i++, j--){  
    System.out.println("i="+i+", j="+j);  
}
```

Любая часть оператора for может отсутствовать: цикл может быть пустым, выражения в заголовке тоже, при этом точки с запятой сохраняются. Можно задать бесконечный цикл:

```
for (;;) оператор;
```

В этом случае в теле цикла следует предусмотреть какой-нибудь выход.

```
int i=0;  
for (;;) {  
    if (i==5) break;  
    i++;  
    System.out.println(i);  
}
```

Хотя в операторе for заложены большие возможности, используется он, главным образом, для перечислений, когда их число известно, например, фрагмент кода

```
int s=0, N=7;  
for (int k = 1; k <= N; k++)  
    s += k * k;  
вычисляет сумму квадратов первых N чисел.
```

В листинге 1.22 показан пример вывода чисел в диапазоне от -10 до 10 с шагом 2.

Листинг 1.22

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

```

        for (int i = -10; i <= 10; i += 2) {
            System.out.println("i=" + i);
        }
    }
}

```

#### **Задание:**

- 1). Вывести все четные числа от -50 до 50.
- 2). Вывести все числа кратные 5 от 0 до введенного числа и посчитать их количество.
- 3). С помощью цикла for подсчитайте сумму всех четных чисел в диапазоне от -20 до 20.
- 4). Вывести все делители введенного числа.
- 5). Выведите все простые числа до 100.

### **1.12.7 Оператор continue и метки**

Оператор continue используется только в операторах цикла. Он имеет две формы. Первая форма состоит только из слова continue и осуществляет немедленный переход к следующей итерации цикла. В листинге 1.23 оператор continue позволяет обойти деление на нуль:

Листинг 1.23

```

public class Main {
    public static void main(String args[]) {
        for (int i = 0, j = 4, s = 0; i < 5; i++, j--) {
            if (i == j) {
                System.out.println("del na 0 ");
                continue;
            }
            s = 100 / (i - j);
            System.out.println("s = " + s);
        }
    }
}

```

Листинг 1.24 содержит метку:

continue метка;

Листинг 1.24

```

public class Main {
    public static void main(String args[]) {
        label:
        for (int i = 0; i < 10; i++) {
            for (int j = 0; j < 10; j++) {
                if (i == 5) {
                    continue label;
                }
            }
        }
    }
}

```

```

    }
    System.out.println(i);
}
}
}

```

*Метка* записывается, как все идентификаторы, из букв Java, цифр и знака подчеркивания, но не требует никакого описания. Метка ставится перед оператором или открывающей фигурной скобкой и отделяется от них двоеточием. Так получается *помеченный оператор* или *помеченный блок*.

Метка не требует описания и не может начинаться с цифры. Вторая форма используется только в случае нескольких вложенных циклов для немедленного перехода к очередной итерации одного из объемлющих циклов, а именно, помеченного цикла.

### 1.12.8 Оператор break

Оператор break используется в операторах цикла и операторе варианта для немедленного выхода из этих конструкций.

Оператор *break метка* применяется внутри помеченных операторов цикла, оператора варианта или помеченного блока для немедленного выхода за эти операторы. Следующая схема поясняет эту конструкцию:

```

M1: { // Внешний блок
    M2: { // Вложенный блок – второй уровень
        M3: { // Третий уровень вложенности...
            if (что-то случилось) break M2;
            // Если true, то здесь ничего не выполняется
        }
        // Здесь тоже ничего не выполняется
    }
    // Сюда передается управление
}

```

Поначалу сбивает с толку то обстоятельство, что метка ставится перед блоком или оператором, а управление передается за этот блок или оператор. Поэтому не стоит увлекаться оператором break с меткой. В листинге 1.25 показано использование вложенных друг в друга меток.

Листинг 1.25

```

public class Main {
    public static void main(String args[]) {
        int i = 3, j = 4;
        M1:
        { // Внешний блок
            int s = i + j;
            M2:
            { // Вложенный блок – второй уровень

```

```

s = i;
M3:
{ // Третий уровень вложенности...
    if (i < j) {
        break M2;
    }
    // Если true, то здесь ничего не выполняется
    System.out.println("level 3 s = " + s);
}
// Здесь тоже ничего не выполняется
System.out.println("level 2 s = " + s);
}
// Сюда передается управление
System.out.println("level 1 s = " + s);
}
}
}

```

### 1.12.9 Оператор варианта switch

Оператор варианта switch организует разветвление по нескольким направлениям. Каждая ветвь отмечается константой или константным выражением какого-либо целого типа (кроме long) и выбирается, если значение определенного выражения совпадет с этой константой. Вся конструкция выглядит так.

```

switch (целВыр)
{
    case констВыр1: оператор1;
    case констВыр2: оператор2;
    . . . . .
    case констВырN: операторN;
    default: операторDef;
};

```

Стоящее в скобках выражение *целВыр* может быть типа byte, short, int, char, но не long. Начиная с jdk 1.7 и String! Целые числа или целочисленные выражения, составленные из констант, *констВыр* тоже не должны иметь тип long.

Оператор варианта выполняется так. Все константные выражения вычисляются заранее, на этапе компиляции, и должны иметь отличные друг от друга значения. Сначала вычисляется целочисленное выражение *целВыр*. Если оно совпадает с одной из констант, то выполняется оператор, отмеченный этой константой. Затем выполняются все следующие операторы, включая и *операторDef*, и работа оператора варианта заканчивается.

Если же ни одна константа не равна значению выражения, то выполняется *операторDef* и все следующие за ним операторы. Поэтому ветвь `default` должна записываться последней. Ветвь `default` может отсутствовать, тогда в этой ситуации оператор варианта вообще ничего не делает.

Таким образом, константы в вариантах `case` играют роль только меток, точек входа в оператор варианта, а далее выполняются все оставшиеся операторы в порядке их записи.

После выполнения одного варианта оператор `switch` продолжает выполнять все оставшиеся варианты. Чаще всего необходимо "пройти" только одну ветвь операторов. В таком случае используется оператор `break`, сразу же прекращающий выполнение оператора `switch`. Если необходимо выполнить один и тот же оператор в разных ветвях `case`, то следует поставить несколько меток `case` подряд, как показано в листинге 1.26.

Листинг 1.26

```
public class Main {
    public static void main(String args[]) {
        for (int dayOfWeek = 1; dayOfWeek <= 8; dayOfWeek++) {
            switch (dayOfWeek) {
                case 1: case 2: case 3: case 4: case 5:
                    System.out.println("Week-day");
                    break;
                case 6: case 7:
                    System.out.println("Week-end");
                    break;
                default:
                    System.out.println("Unknown day");
            }
        }
    }
}
```

Задание

Написать программу, которая по номеру месяца будет определять пору года. Номер месяца вводится с клавиатуры. Предусмотреть проверку на некорректный ввод.

### Тема 1.13 Статический импорт

Аккуратное и правильное использование `import static` улучшит читаемость вашего кода.

Для того чтобы получить доступ к статическим членам классов, требуются указать ссылку на класс. К примеру, необходимо указать имя класса `Math`:

```
double r = Math.cos(Math.PI * theta);
```

Конструкция статического импорта позволяет получить прямой доступ к статическим полям и методам класса. Можно импортировать каждый метод отдельно:

```
import static java.lang.Math.PI;
```

или все целиком:

```
import static java.lang.Math.*;
```

Однажды импортированный статический член может быть использован без указания имени класса:

```
double r = cos(PI * theta);
```

Объявление статического импорта аналогично объявлению обычного импорта. При объявлении обычного импорта классы импортируются из пакетов, что позволяет их использовать без указания имени пакета перед именем класса. При объявлении статического импорта статические члены импортируются из классов, что позволяет им быть использованными без указания имени содержащего их класса.

Так когда же следует использовать статический импорт? *Только в некоторых случаях!* Используйте его только, если иначе вы вынуждены объявлять локальные копии констант или при неправильном использовании наследования (Constant Interface Antipattern).

Другими словами, использование его оправданно, когда требуется постоянное использование статических членов одного класса из одного или двух других классов.

Чрезмерное использование статического импорта может сделать вашу программу нечитаемой и тяжелой в поддержке ввиду того, что пространство имен увеличится из-за всех статических членов из вашего импорта. Тот, кто будет читать ваш код (включая вас, спустя несколько месяцев после написания кода) не будут знать какому из статически импортированных классов принадлежит тот или иной член. Импортирование всех статических методов из класса может быть частично вредно для читабельности; если вы нуждаетесь только в одном или двух методах, импортируйте их по отдельности. Использованный умело, статический импорт может сделать вашу программу более наглядной, исключая из программного кода нужные повторения имен классов.

### Тема 1.14 Класс Math

Разработчику на Java доступно множество готовых (или библиотечных) классов и методов, полезных для использования в собственных программах. Наличие библиотечных решений позволяет изящно решать множество типовых задач.

Далее рассмотрим класс **Math**, содержащий различные математические функции. Рассмотрим некоторые из них:

**Math.abs(n)** – возвращает модуль числа *n*.

**Math.round(n)** – возвращает целое число, ближайшее к вещественному числу *n* (округляет *n*).

`Math.ceil(n)` – возвращает ближайшее к числу `n` справа число с нулевой дробной частью (например, `Math.ceil(3.4)` в результате вернёт 4.0).

`Math.cos(n)`, `Math.sin(n)`, `Math.tan(n)` – тригонометрические функции `sin`, `cos` и `tg` от аргумента `n`, указанного в радианах.

`Math.acos(n)`, `Math.asin(n)`, `Math.atan(n)` – обратные тригонометрические функции, возвращают угол в радианах.

`Math.toDegrees(n)` – возвращает градусную меру угла в `n` радианов.

`Math.toRadians(n)` – возвращает радианную меру угла в `n` градусов.

`Math.sqrt(n)` – возвращает квадратный корень из `n`.

`Math.pow(n, b)` – возвращает значение степенной функции `n` в степени `b`, основание и показатель степени могут быть вещественными.

`Math.log(n)` – возвращает значение натурального логарифма числа `n`.

`Math.log10(n)` – возвращает значение десятичного логарифма числа `n`.

Все перечисленные методы принимают вещественные аргументы, а тип возвращаемого значения зависит от типа аргумента и от самой функции.

Кроме методов в рассматриваемом классе имеются две часто используемых константы:

`Math.PI` – число «пи», с точностью в 15 десятичных знаков.

`Math.E` – число Неппера (основание экспоненциальной функции), с точностью в 15 десятичных знаков.

В листинге 1.27 приведены примеры использования некоторых методов.

*Листинг 1.27*

```
public class Main {
    public static void main(String args[]) {
        System.out.println(Math.abs(-2.33)); // выведет 2.33
        System.out.println(Math.round(Math.PI)); // выведет 3
        System.out.println(Math.round(9.5)); // выведет 10
        System.out.println(Math.round(9.5 - 0.001)); // выведет 9
        System.out.println(Math.ceil(9.4)); // выведет 10.0
        double c = Math.sqrt(3 * 3 + 4 * 4);
        System.out.println(c); // гипотенуза треугольника с катетами 3 и 4
        double s1 = Math.cos(Math.toRadians(60));
        System.out.println(s1); // выведет косинус угла в 60 градусов
    }
}
```

## Задания

1). Вычислить и вывести на экран косинусы углов в 60, 45 и 40 градусов без использования функции `Math.toDegrees(n)`.

2). В переменных `a` и `b` лежат положительные длины катетов прямоугольного треугольника. Вычислить и вывести на экран площадь треугольника и его периметр.

3). Натуральное положительное число записано в переменную *n*. Определить и вывести на экран, сколько цифр в числе *n*.

4). В переменной *n* лежит некоторое вещественное число. Вычислить и вывести на экран значение функции «сигнум» от этого числа (-1, если число отрицательное; 0, если нулевое; 1 если, положительное).

### Тема 1.15 Псевдослучайные числа

В классе *Math* есть полезная функция без аргументов, которая позволяет генерировать псевдослучайные значения, т.е. при каждом вызове этой функции она будет возвращать новое значение, предсказать которое очень сложно (не вдаваясь в подробности можно сказать, что теоретически это всё-таки возможно, именно поэтому генерируемые функцией числа называются не случайными, а псевдослучайными).

Итак, *Math.random()* возвращает псевдослучайное вещественное число из промежутка [0;1).

Если требуется получить число из другого диапазона, то значение функции можно умножать на что-то, сдвигать и, при необходимости, приводить к целым числам.

В листинге 1.28 приведены примеры создания случайных чисел в разных диапазонах.

Листинг 1.28

```
public class Main {  
    public static void main(String args[]) {  
        System.out.println(Math.random()); // вещественное из [0;1)  
        System.out.println(Math.random() + 3); // вещественное из [3;4)  
        System.out.println((int) (Math.random() * 5)); // целое из [0;4]  
        System.out.println(Math.random() * 5 + 3); //вещественное из [3;8)  
        System.out.println((int) (Math.random() * 11) - 5); // целое из [-5;5]  
    }  
}
```

Псевдослучайные числа имеют серьезнейшие практические приложения и используются, например, в криптографии.

#### Задания:

1). Создайте программу, которая будет генерировать и выводить на экран вещественное псевдослучайное число из промежутка [-3;3).

2). Натуральное положительное число записано в переменную *n*. Создайте программу, которая будет генерировать и выводить на экран целое псевдослучайное число из отрезка [-*n*; *n*].

3). В переменные *a* и *b* записаны целые числа, при этом *b* больше *a*. Создайте программу, которая будет генерировать и выводить на экран целое псевдослучайное число из отрезка [*a*; *b*].



## Тема 1.16 Генерация случайных чисел

В пакете `java.util` описан класс `Random`, являющийся генератором случайных чисел. На самом деле в силу своей природы ЭВМ не может генерировать истинно случайные числа. Числа генерируются определенным алгоритмом, причем каждое следующее число зависит от предыдущего, а самое первое – от некоторого числа, называемого инициализатором. Две последовательности «случайных» чисел, сгенерированных на основе одного инициализатора, будут одинаковы.

Класс `Random` имеет два конструктора

`Random()` – создает генератор случайных чисел, использующий в качестве инициализатора текущую дату (число миллисекунд с 1 января 1970);

`Random(long seed)` – создает генератор случайных чисел, использующий в качестве инициализатора число `seed`.

Рекомендуется использовать первый конструктор, чтобы генератор выдавал разные случайные числа при каждом новом запуске программы. От генератора можно получать случайные числа нужного типа с помощью методов `nextBoolean()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`.

Вещественные числа генерируются в диапазоне от 0 до 1 (не включая 1), а целые – из всего диапазона возможных значений. Можно сгенерировать целое число в нужном диапазоне (от 0 до `max-1`) методом `nextInt(int max)` или `nextLong(long max)`.

Наконец, можно заполнить случайными числами целый массив (предварительно созданный), воспользовавшись методом `nextBytes(byte[] arr)`. Элементы массива `arr` должны иметь тип `byte`.

В листинге 1.29 показан пример использования случайных чисел.

Листинг 1.29

```
import java.util.Random;

public class Test {
    public static void main(String[] args) {
        Random r = new Random(100);
        for (int cnt = 0; cnt < 9; cnt++) {
            System.out.print(r.nextInt() + " ");
        }
        System.out.println();
        r = new Random(100);
        for (int cnt = 0; cnt < 9; cnt++) {
            System.out.print(r.nextInt() + " ");
        }
        System.out.println();
        byte[] randArray = new byte[8];
        r.nextBytes(randArray);
    }
}
```

## Тема 1.17 Массивы в Java

### 1.17.1 Объявление и заполнение массива

Массив – это конечная последовательность упорядоченных элементов одного типа, доступ к каждому элементу в которой осуществляется по его индексу.

Размер или длина массива – это общее количество элементов в массиве. Размер массива задаётся при создании массива и не может быть изменён в дальнейшем, т. е. нельзя убрать элементы из массива или добавить их туда, но можно в существующие элементы присвоить новые значения.

Индекс начального элемента – 0, следующего за ним – 1 и т. д. Индекс последнего элемента в массиве – на единицу меньше, чем размер массива.

В Java массивы являются объектами. Это значит, что имя, которое даётся каждому массиву, лишь указывает на адрес какого-то фрагмента данных в памяти. Кроме адреса в этой переменной ничего не хранится. Индекс массива, фактически, указывает на то, насколько надо отступить от начального элемента массива в памяти, чтоб добраться до нужного элемента.

Чтобы создать массив надо объявить для него подходящее имя, а затем с этим именем связать нужный фрагмент памяти, где и будут друг за другом храниться значения элементов массива.

Возможные следующие варианты объявления массива:

*тип*[] *имя*;

*тип* *имя*[];

Где *тип* – это тип элементов массива, а *имя* – уникальный (незанятый другими переменными или объектами в этой части программы) идентификатор, начинающийся с буквы.

Примеры:

`int[] a;`

`double[] ar1;`

`double ar2[];`

В примере мы объявили имена для трёх массивов. С первым именем **a** сможет быть далее связан массив из элементов типа `int`, а с именами `ar1` и `ar2` далее смогут быть связаны массивы из вещественных чисел (типа `double`). Пока мы не создали массивы, а только подготовили имена для них.

Теперь создать (или как ещё говорят инициализировать) массивы можно следующим образом:

`a = new int[10];` // массив из 10 элементов типа `int`

`int n = 5;`

`ar1 = new double[n];` // Массив из 5 элементов `double`

`ar2 = {3.14, 2.71, 0, -2.5, 99.123};` // Массив из 6 элементов типа `double`

То есть при создании массива мы можем указать его размер, либо сразу перечислить через запятую все желаемые элементы в фигурных скобках (при этом размер будет вычислен автоматически на основе той

последовательности элементов, которая будет указана). Обратите внимание, что в данном случае после закрывающей фигурной скобки ставится точка с запятой, чего не бывает, когда это скобка закрывает какой-то блок.

Если массив был создан с помощью оператора **new**, то каждый его элемент получает значение по умолчанию. Каким оно будет определяется на основании типа данных (0 для `int`, 0.0 для `double` и т. д.).

Объявить имя для массива и создать сам массив можно было на одной строке по следующей схеме:

```
тип[] имя = new тип[размер];  
тип[] имя = {эл0, эл1, ..., элN};
```

Примеры:

```
int[] mas1 = {10,20,30};  
int[] mas2 = new int[3];
```

Чтобы обратиться к какому-то из элементов массива для того, чтобы прочитать или изменить его значение, нужно указать имя массива и за ним индекс элемента в квадратных скобках. Элемент массива с конкретным индексом ведёт себя также, как переменная. Например, чтобы вывести последний элемент массива `mas1` мы должны написать в программе:

```
System.out.println("Последний элемент массива " + mas1[2]);
```

А вот так мы можем положить в массив `mas2` тот же набор значений, что хранится в `mas1`:

```
mas2[0] = 10;  
mas2[1] = 20;  
mas2[2] = 30;
```

Уже из этого примера видно, что для того, чтоб обратиться ко всем элементам массива, нам приходится повторять однотипные действия. Как вы помните для многократного повторения операций используются циклы. Соответственно, мы могли бы заполнить массив нужными элементами с помощью цикла:

```
for(int i=0; i<=2; i++) {  
    mas2[i] = (i+1) * 10;  
}
```

Понятно, что если бы массив у нас был не из 3, а из 100 элементов, без цикла мы бы просто не справились.

Длину любого созданного массива не обязательно запоминать, потому что имеется свойство, которое его хранит. Обратиться к этому свойству можно дописав `.length` к имени массива. Например:

```
int razmer = mas1.length;
```

Это свойство нельзя изменять (т. е. ему нельзя ничего присваивать), можно только читать. Используя это свойство можно писать программный код для обработки массива даже не зная его конкретного размера.

Например, так можно вывести на экран элементы любого массива с именем `ar2`:

```
for(int i = 0; i <= ar2.length - 1; i++) {
    System.out.print(ar2[i] + " ");
}
```

Для краткости удобнее менять нестрогое неравенство на строгое, тогда не нужно будет вычитать единицу из размера массива. Давайте заполним массив целыми числами от 0 до 9 и выведем его на экран:

```
for(int i = 0; i < ar1.length; i++) {
    ar1[i] = Math.floor(Math.random() * 10);
    System.out.print(ar1[i] + " ");
}
```

Обратите внимание, на каждом шаге цикла мы сначала отправляли случайное значение в элемент массива с *i*-ым индексом, а потом этот же элемент выводили на экран. Но два процесса (наполнения и вывода) можно было проделать и в разных циклах. Например:

```
for(int i = 0; i < ar1.length; i++) {
    ar1[i] = Math.floor(Math.random() * 9);
}
for(int i = 0; i < ar1.length; i++) {
    System.out.print(ar1[i] + " ");
}
```

В данном случае более рационален первый способ (один проход по массиву вместо двух), но не всегда возможно выполнить требуемые действия в одном цикле.

Для обработки массивов всегда используются циклы типа «*n* раз» (for) потому, что нам заранее известно сколько раз должен повториться цикл (столько же раз, сколько элементов в массиве).

В листинге 1.30 приведен пример создания массива целых чисел в диапазоне от 50 до 150. Количество элементов в массиве задает пользователь.

Листинг 1.30

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        System.out.println("Введите количество элементов массива:");
        Scanner sc = new Scanner(System.in);
        int x = 0, mas[];
        if (sc.hasNextInt()) {
            x = sc.nextInt();
        }
        mas = new int[x];
        for (int i = 0; i < mas.length; i++) {
            mas[i] = (int) (Math.random() * 100 + 50);
        }
        for (int i = 0; i < mas.length; i++) {
            System.out.println("mas[" + i + "]=" + mas[i] + ";");
        }
    }
}
```

```
    }
  }
}
```

### **Задачи**

1). Создайте массив из всех чётных чисел от 2 до 20 и выведите элементы массива на экран сначала в строку, отделяя один элемент от другого пробелом, а затем в столбик (отделяя один элемент от другого началом новой строки). Перед созданием массива подумайте, какого он будет размера.

2 4 6 ... 18 20

2

4

6

...

20

2). Создайте массив из всех нечётных чисел от 1 до 99, выведите его на экран в строку, а затем этот же массив выведите на экран тоже в строку, но в обратном порядке (99 97 95 93 ... 7 5 3 1).

3). Создайте массив из 15 случайных целых чисел из отрезка [0;9]. Выведите массив на экран. Подсчитайте сколько в массиве чётных элементов и выведите это количество на экран на отдельной строке.

## **1.17.2 Сортировка массива**

Сортировкой называется такой процесс перестановки элементов массива, когда все его элементы выстраиваются по возрастанию или по убыванию.

Сортировать можно не только числовые массивы, но и, например, массивы строк (по тому же принципу, как расставляют книги на библиотечных полках). Вообще сортировать можно элементы любого множества, где задано отношение порядка.

Существуют универсальные алгоритмы, которые выполняют сортировку вне зависимости от того, каким было исходное состояние массива. Но кроме них существуют специальные алгоритмы, которые, например, очень быстро могут отсортировать почти упорядоченный массив, но плохо справляются с сильно перемешанным массивом (или вообще не справляются). Специальные алгоритмы нужны там, где важна скорость и решается конкретная задача, их подробное изучение выходит за рамки нашего курса.

### **Сортировка выбором**

Рассмотрим пример сортировки по возрастанию. То есть на начальной позиции в массиве должен стоять минимальный элемент, на следующей – больший или равный и т. д., на последнем месте должен стоять наибольший элемент.

Суть алгоритма такова. Во всём отыскиваем минимальный элемент, меняем его местами с начальным. Затем в оставшейся части массива (т. е. среди всех элементов кроме начального) снова отыскиваем минимальный элемент, меняем его местами уже со вторым элементом в массиве. И так далее (листинг 1.31).

Листинг 1.31

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        System.out.println("Введите количество элементов массива:");
        Scanner sc = new Scanner(System.in);
        int x = 0, a[];
        if (sc.hasNextInt()) {
            x = sc.nextInt();
        }
        a = new int[x];
        for (int i = 0; i < a.length; i++) {
            a[i] = (int) (Math.random() * 100 + 50);
        }
        System.out.println("Исходный массив:");
        for (int i = 0; i < a.length; i++) {
            System.out.println("mas[" + i + "]=" + a[i] + ";");
        }
        for (int i = 0; i < a.length; i++) {
            /* Предполагаем, что начальный элемент рассматриваемого
             * фрагмента и будет минимальным.
             */
            int min = a[i]; // Предполагаемый минимальный элемент
            int imin = i; // Индекс минимального элемента
            /* Просматриваем оставшийся фрагмент массива и ищем там
             * элемент, меньший предположенного минимума.
             */
            for (int j = i + 1; j < a.length; j++) {
                /* Если находим новый минимум, то запоминаем его индекс.
                 * И обновляем значение минимума.
                 */
                if (a[j] < min) {
                    min = a[j];
                    imin = j;
                }
            }
            /* Проверяем, нашёлся ли элемент меньше, чем стоит на
             * текущей позиции. Если нашёлся, то меняем элементы местами.
             */
        }
    }
}
```

```

        if (i != imin) {
            int temp = a[i];
            a[i] = a[imin];
            a[imin] = temp;
        }
    }
    System.out.println("Отсортированный массив:");
    for (int i = 0; i < a.length; i++) {
        System.out.println("mas[" + i + "]=" + a[i] + ";");
    }
}
}

```

### Сортировка методом пузырька

Суть алгоритма такова. Если пройдемся по любому массиву сравнив первый элемент с остальными, то после первого прохода на первом месте массива гарантированно будет стоять нужный элемент (самый большой для сортировки по возрастанию или самый маленький для сортировки по убыванию). Далее сравниваем второй элемент с остальными, и после этого прохода на второй позиции будет стоять правильный элемент. И так далее.

Пример:

2 9 1 4 3 5 → порядок правильный, не будет перестановки

2 9 1 4 3 5 → 1 9 2 4 3 5

1 9 2 4 3 5 → порядок правильный, не будет перестановки

1 9 2 4 3 5 → порядок правильный, не будет перестановки

1 9 2 4 3 5 → порядок правильный, не будет перестановки

После первого прохода первый элемент самый минимальный, теперь сравниваем второй с остальными

1 9 2 4 3 5 → 1 2 9 4 3 5

1 2 9 4 3 5 → порядок правильный, не будет перестановки

1 2 9 4 3 5 → порядок правильный, не будет перестановки

1 2 9 4 3 5 → порядок правильный, не будет перестановки

Теперь второй элемент на своей позиции, ищем третий и т.д.

1 2 9 4 3 5 → 1 2 4 9 3 5

1 2 4 9 3 5 → 1 2 3 9 4 5

1 2 3 9 4 5 → порядок правильный, не будет перестановки

Теперь третий элемент на своей позиции,

1 2 9 4 5 3 → 1 2 4 9 5 3

1 2 4 9 5 3 → порядок правильный, не будет перестановки

1 2 4 9 5 3 → 1 2 3 9 5 4

1 2 3 9 5 4 → 1 2 3 5 9 4

1 2 3 5 9 4 → 1 2 3 4 9 5

1 2 3 4 9 5 → 1 2 3 4 5 9

Массив будет отсортирован за количество элементов-1 раз.

Реализация алгоритма приведена в листинге 1.32

*Листинг 1.32*

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        System.out.println("Введите количество элементов массива:");
        Scanner sc = new Scanner(System.in);
        int x = 0, a[];
        if (sc.hasNextInt()) {
            x = sc.nextInt();
        }
        a = new int[x];
        for (int i = 0; i < a.length; i++) {
            a[i] = (int) (Math.random() * 100 + 50);
        }
        System.out.println("Исходный массив:");
        for (int i = 0; i < a.length; i++) {
            System.out.println("mas[" + i + "]=" + a[i] + ";");
        }
        /* Внешний цикл отсчитывает количество повторений, а их будет
        * a.length - 1
        */
        for (int i = 0; i < a.length - 1; i++) {
            /* Во внутреннем цикле мы переберем все элементы массива,
            * начиная с текущего +1
            */
            for (int j = i+1; j < a.length - 1; j++) {
                /* Если порядок соседних элементов не правильный, то их
                * надо обменять местами.
                */
                if (a[j] < a[i]) {
                    int temp = a[j];
                    a[j] = a[i];
                    a[i] = temp;
                }
            }
        }
        System.out.println("Отсортированный массив:");
        for (int i = 0; i < a.length; i++) {
            System.out.println("mas[" + i + "]=" + a[i] + ";");
        }
    }
}
```



### 1.17.3 Многомерные массивы

Массив может состоять не только из элементов какого-то встроенного типа (`int`, `double` и пр.), но и, в том числе, из объектов какого-то существующего класса и даже из других массивов.

Массив который в качестве своих элементов содержит другие массивы называется многомерным массивом.

Чаще всего используются двумерные массивы. Такие массивы можно легко представить в виде матрицы. Каждая строка которой является обычным одномерным массивом, а объединение всех строк – двумерным массивом в каждом элементе которого хранится ссылка на какую-то строку матрицы.

Трёхмерный массив можно представить себе как набор матриц, каждую из которых мы записали на библиотечной карточке. Тогда чтобы добраться до конкретного числа сначала нужно указать номер карточки (первый индекс трёхмерного массива), потому указать номер строки (второй индекс массива) и только затем номер элемент в строке (третий индекс).

Соответственно, для того, чтобы обратиться к элементу *n*-мерного массива нужно указать *n* индексов.

Объявляются массивы так:

```
int[] d1; //Обычный, одномерный
```

```
int[][] d2; //Двумерный
```

```
double[][] d3; //Трёхмерный
```

```
int[][][][] d5; //Пятимерный
```

При создании массива можно указать явно размер каждого его уровня:

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

Но можно указать только размер первого уровня:

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

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

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

В результате получим такой вот массив:

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

Мы могли создать массив явно указав его элементы. Например так:

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

При этом можно обратиться к элементу с индексом 4 во второй строке `ddd2[1][4]`, но если мы обратимся к элементу `ddd2[0][4]` или

ddd2[2][4] – произойдёт ошибка, поскольку таких элементов просто нет. При этом ошибка это будет происходить уже во время исполнения программы (т. е. компилятор её не увидит).

Обычно всё же используются двумерные массивы с равным количеством элементов в каждой строке.

Для обработки двумерных массивов используются два вложенных друг в друга цикла с разными счётчиками.

В листинге 1.33 приведен пример создания двумерного массива  $m \times n$ , заполнение случайными числами и вывод.

Листинг 1.33

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        System.out.println("Введите количество элементов массива:");
        Scanner sc = new Scanner(System.in);
        int m = 0, n = 0, a[][];
        if (sc.hasNextInt()) {
            m = sc.nextInt();
            n = sc.nextInt();
        }
        a = new int[m][n];
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a[i].length; j++) {
                a[i][j] = (int) (Math.random() * 100 + 50);
            }
        }
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a[i].length; j++) {
                System.out.print(a[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
```

В работе с массивами удобно использовать цикл другой вариант цикла for.

Для одномерного массива:

```
int ar[] = new int[5];
for ( int i = 0; i < ar.length; i++ )
{
    ar[i] = ( int ) ( Math.random () * 100 );
}
//Вывод массива
for ( int i : ar )
```

```

    {
        System.out.println ( i );
    }

```

В переменную *i* попадают копии элементов массива, поэтому его удобно использовать для вывода, но невозможно – для заполнения.

Для двумерного массива:

```

int ar[][] = {{2,6,1,3},{3,8,5,6},{1,2},{4,8,2}};
//вывод массива
for ( int i[] : ar )
{
    for ( int j : i )
    {
        System.out.println ( j );
    }
    System.out.println ( "" );
}

```

### Задания

1). Создать двумерный массив из 8 строк по 5 столбцов в каждой из случайных целых чисел из отрезка [10;99]. Вывести массив на экран.

2). Создать двумерный массив из 5 строк по 8 столбцов в каждой из случайных целых чисел из отрезка [-99;99]. Вывести массив на экран. После на отдельной строке вывести на экран значение максимального элемента этого массива (его индекс не имеет значения).

3). Создать двумерный массив из 7 строк по 4 столбца в каждой из случайных целых чисел из отрезка [-5;5]. Вывести массив на экран. Определить и вывести на экран индекс строки с наибольшим по модулю произведением элементов. Если таких строк несколько, то вывести индекс первой встретившейся из них.

## 1.17.4 Нерегулярные массивы

Выделяя память под многомерный массив, сначала необходимо указать лишь размерность по одной (а именно по самой левой) координате. Остальные размерности можно указывать по отдельности. Например, в приведенном ниже фрагменте кода задается только размерность массива *table* по первой координате. Размерность по второй координате задается вручную.

```

int table[ ] [ ] = new int[ 3] [ ];
table[0] = new int[4];
table[1] = new int[4];
table[2] = new int[4];

```

В данном случае, объявляя массив подобным образом, мы не получаем каких преимуществ, однако в некоторых ситуациях такое объявление вполне оправдано. В частности, есть возможность установить различную длину массивов, являющихся элементами первого массива. Как вы помните, многомерный массив представляет собой массив массивов, поэтому вы

можете, контролировать размер каждого из них. Предположим, например, что вам надо написать программу, в процессе работы которой будет сохраняться число пассажиров, перевезенных микроавтобусом к самолетам. Если микроавтобус делает 10 рейсов в день будние дни и по два рейса в субботу и воскресенье, то вы можете объявить мерности массивов так, как это показано в приведенном ниже фрагменте кода. Обратите внимание на то, что размерность по второй координате для первых пяти значений индекса равна 10, а для остальных элементов — 2.

*Листинг 1.34*

// Указание размерностей по второй координате вручную

```
public class Main {
    public static void main(String args[]) {
        int riders[][] = new int[7][];
        // размерность по второй координате равна 10
        riders[0] = new int[10];
        riders[1] = new int[10];
        riders[2] = new int[10];
        riders[3] = new int[10];
        riders[4] = new int[10];
        // Для остальных двух элементов
        // размерность по второй координате равна 2.
        riders[5] = new int[2];
        riders[6] = new int[2];
        int i, j;
        // Формирование произвольных данных
        for (i = 0; i < 5; i++) {
            for (j = 0; j < 10; j++) {
                riders[i][j] = i + j + 10;
            }
        }
        for (i = 5; i < 7; i++) {
            for (j = 0; j < 2; j++) {
                riders[i][j] = i + j + 10;
            }
        }
        System.out.println("Riders per trip during the week:");
        for (i = 0; i < 5; i++) {
            for (j = 0; j < 10; j++) {
                System.out.print(riders[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
        System.out.println("Riders per trip on the weekend:");
        for (i = 5; i < 7; i++) {
```

```

        for (j = 0; j < 2; j++) {
            System.out.print(riders[i][j] + " ");
        }
        System.out.println();
    }
}

```

В результате выполнения данной программы получим:

Riders per trip during the week:

```

10 11 12 13 14 15 16 17 18 19
11 12 13 14 15 16 17 18 19 20
12 13 14 15 16 17 18 19 20 21
13 14 15 16 17 18 19 20 21 22
14 15 16 17 18 19 20 21 22 23

```

Riders per trip on the weekend:

```

15 16
16 17

```

Для большинства приложений использовать нерегулярные массивы не рекомендуется, поскольку это затрудняет восприятие кода другими программистами. Однако в некоторых случаях такие массивы вполне уместны и могут существенно повысить эффективность программ. Например, если вам надо создать большой двумерный массив, в котором используются не все элементы, то нерегулярный массив позволит существенно сэкономить память.

#### **Задание:**

1. Нужно создать нерегулярный массив, подобный table:

```

int table[ ] [ ] = new int[ 4 ] [ ];
table[0] = new int[7];
table[1] = new int[3];
table[2] = new int[5];
table[3] = new int[1];

```

Количество строк вводится, а количество элементов в каждой строке задается случайным образом в диапазоне от 1 до 10.

2. Заполнить его какими-нибудь элементами.

3. Вывести массив красиво (чтоб он выглядел как массив, а не строка или столбец ).

4. Необходимо переставить строки массива, чтобы их размер возрастал (или убывал).

5. Вывести получившийся массив.

Для массива table результат будет выглядеть так:

```

table[0] = new int[1];
table[1] = new int[3];
table[2] = new int[5];
table[3] = new int[7];

```

## **Выводы к главе:**

- Любая Java-программа – это класс.
- Класс пишется в файле, по названию совпадающем с именем класса и с расширением java.
- При помощи компилятора `javac.exe` исходный код компилируется в промежуточный байт-код (получаем файл с расширением `class`).
- При помощи интерпретатора `javac.exe` происходит выполнение кода.
- Запуск кода начинается с вызова метода `main`.
- Исходными параметрами в этот метод является массив строк.
- Прежде чем использовать какую-нибудь переменную, ее следует объявить, т.е. указать ее тип и название.
- Прimitивных типов всего восемь. Все остальное – это классы.
- Для каждого примитивного типа есть класс-оболочка. При преобразовании из примитивного типа в класс-оболочку и обратно происходит автоупаковка и автораспаковка.
- В качестве названия переменной нельзя использовать зарезервированные слова и начинать название с цифры.
- Во время явного приведения типов может произойти потеря данных.
- Повторяющиеся много раз действия необходимо помещать в цикл. Любой цикл можно принудительно прервать с помощью оператора `break`.
- Массив – это набор данных одного типа. Массив необходимо объявить перед использованием и выделить под него память с помощью оператора `new`.

### Задания к главе:

- 1) Написать программу, которая выводит на экран первые четыре степени числа  $n$ .
- 2) Составить линейную программу, печатающую значение `true`, если указанное высказывание является истинным, и `false` – в противном случае.
  - 2.1) Сумма двух первых цифр заданного четырехзначного числа равна сумме двух его последних цифр.
  - 2.2) Сумма цифр данного трехзначного числа  $N$  является четным числом.
  - 2.3) Целое число  $N$  является четным двузначным числом.
  - 2.4) Данная тройка натуральных чисел  $a$ ,  $b$ ,  $c$  является тройкой Пифагора, т.е.  $c^2 = a^2 + b^2$ .
  - 2.5) Все цифры данного четырехзначного числа  $N$  различны.
  - 2.6) Данное четырехзначное число читается одинаково слева направо и справа налево.
- 3) Даны действительные числа  $x$  и  $y$ , не равные друг другу. Меньшее из этих двух чисел заменить половиной их суммы, а большее – их удвоенным произведением.
- 4) Составить программу, которая запрашивает пароль (например, четырехзначное число) до тех пор, пока он не будет правильно введен.
- 5) Найти сумму всех  $n$ -значных чисел ( $1 \leq n \leq 4$ ).
- 6) Найти сумму всех  $n$ -значных чисел, кратных  $k$  ( $1 \leq n \leq 4$ ).
- 7) Написать программу, которая загадывает случайное целое число из отрезка  $[1;10]$  и просит пользователя его угадать, вводя варианты с клавиатуры, пока пользователь не угадает число, программа будет ему подсказывать, сообщая больше или меньше число загаданное, чем то, что ввёл пользователь.
- 8). Создайте программу, выводящую на экран все четырёхзначные числа последовательности 1000 1003 1006 1009 1012 1015 ...
- 9). Создайте программу, выводящую на экран первые 55 элементов последовательности 1 3 5 7 9 11 13 15 17 ...
- 10). Создайте программу, выводящую на экран все неотрицательные элементы последовательности 90 85 80 75 70 65 60 ...
- 11). Выведите на экран все члены последовательности  $2a_{n-1}-1$ , где  $a_1=2$ , которые меньше 10000.
- 12). Выведите на экран все двузначные члены последовательности  $2a_{n-1}+200$ , где  $a_1=-166$ .
- 13). Выведите на экран все положительные делители натурального числа, введенного пользователем с клавиатуры.
- 14). Проверьте, является ли введенное пользователем с клавиатуры натуральное число – простым. Постарайтесь не выполнять лишних действий (например, после того, как вы нашли хотя бы один нетривиальный делитель уже ясно, что число составное и проверку продолжать не нужно). Также учтите, что наименьший делитель натурального числа  $n$ , если он вообще

имеется, обязательно располагается в отрезке  $[2; \sqrt{n}]$ . Если число не простое – выведите все его простые делители.

15). Выведите на экран первые 11 членов последовательности Фибоначчи. Напоминаем, что первый и второй члены последовательности равны единицам, а каждый следующий – сумме двух предыдущих.

16). Для введённого пользователем с клавиатуры натурального числа посчитайте сумму всех его цифр (заранее не известно, сколько цифр будет в числе).

17). В городе N проезд в трамвае осуществляется по бумажным отрывным билетам. Каждую неделю трамвайное депо заказывает в местной типографии рулон билетов с номерами от 000001 до 999999. «Счастливым» считается билетик, у которого сумма первых трёх цифр номера равна сумме последних трёх цифр, как, например, в билетах с номерами 003102 или 567576. Трамвайное депо решило подарить сувенир обладателю каждого счастливого билета и теперь раздумывает, как много сувениров потребуется. С помощью программы подсчитайте, сколько счастливых билетов в одном рулоне?

18). В американской армии считается несчастливым число 13, а в японской – 4. Перед международными учениями штаб российской армии решил исключить номера боевой техники, содержащие числа 4 или 13 (например, 40123, 13313, 12345 или 13040), чтобы не смущать иностранных коллег. Если в распоряжении армии имеется 100 тыс. единиц боевой техники и каждая боевая машина имеет номер от 00001 до 99999, то сколько всего номеров придётся исключить?

19). Создать программу, которая будет проверять попало ли случайно выбранное из отрезка  $[5; 155]$  целое число в интервал  $(25; 100)$  и сообщать результат на экран.

Примеры работы программы:

Число 113 не содержится в интервале  $(25, 100)$

Число 72 содержится в интервале  $(25, 100)$

20). Создать программу, выводящую на экран случайно сгенерированное трёхзначное натуральное число и его наибольшую цифру.

Примеры работы программы:

В числе 208 наибольшая цифра 8

В числе 774 наибольшая цифра 7

21). На некотором предприятии инженер Петров создал устройство, на табло которого показывается количество секунд, оставшихся до конца рабочего дня. Когда рабочий день начинается ровно в 9 часов утра – табло отображает «28800» (т.е. остаётся 8 часов), когда времени 14:30 – на табло «9000» (т.е. остаётся два с половиной часа), а когда наступает 17 часов – на табло отображается «0» (т.е. рабочий день закончился).

Программист Иванов заметил, как страдают офисные сотрудницы – им неудобно оценивать остаток рабочего дня в секундах. Иванов вызвался помочь сотрудницам и написать программу, которая вместо секунд будет выводить на табло понятные фразы с информацией о том, сколько полных



часов осталось до конца рабочего дня. Например: «осталось 7 часов», «осталось 4 часа», «остался 1 час», «осталось менее часа».

Итак, в переменную *n* должно записываться случайное (на время тестирования программы) целое число из  $[0;28800]$ , далее оно должно выводиться на экран (для Петрова) и на следующей строке (для сотрудниц) должна выводиться фраза о количестве полных часов, содержащихся в *n* секундах.

Примеры работы программы:

23466

Осталось 6 часов

10644

Осталось 2 часа

1249

Осталось менее часа

22). Создайте массив из 8 случайных целых чисел из отрезка  $[1;10]$ . Выведите массив на экран в строку. Замените каждый элемент с нечётным индексом на ноль. Снова выведете массив на экран на отдельной строке.

23). Создайте 2 массива из 5 случайных целых чисел из отрезка  $[0;5]$  каждый, выведите массивы на экран в двух отдельных строках. Посчитайте среднее арифметическое элементов каждого массива и сообщите, для какого из массивов это значение оказалось больше (либо сообщите, что их средние арифметические равны).

24). Создайте массив из 4 случайных целых чисел из отрезка  $[10;99]$ , выведите его на экран в строку. Определите и вывести на экран сообщение о том, является ли массив строго возрастающей последовательностью.

25). Создайте массив из 20-ти первых чисел Фибоначчи и выведите его на экран. Напоминаем, что первый и второй члены последовательности равны единицам, а каждый следующий – сумме двух предыдущих.

26). Создайте массив из 12 случайных целых чисел из отрезка  $[-15;15]$ . Определите какой элемент является в этом массиве максимальным и сообщите индекс его последнего вхождения в массив.

27). Создайте два массива из 10 целых случайных чисел из отрезка  $[1;9]$  и третий массив из 10 действительных чисел. Каждый элемент с *i*-ым индексом третьего массива должен равняться отношению элемента из первого массива с *i*-ым индексом к элементу из второго массива с *i*-ым индексом. Вывести все три массива на экран (каждый на отдельной строке), затем вывести количество целых элементов в третьем массиве.

28). Создайте массив из 11 случайных целых чисел из отрезка  $[-1;1]$ , выведите массив на экран в строку. Определите какой элемент встречается в массиве чаще всего и выведите об этом сообщение на экран. Если два каких-то элемента встречаются одинаковое количество раз, то не выводите ничего.

29). Пользователь должен указать с клавиатуры чётное положительное число, а программа должна создать массив указанного размера из случайных целых чисел из  $[-5;5]$  и вывести его на экран в строку. После этого программа должна определить и сообщить пользователю о том, сумма модулей какой

половины массива больше: левой или правой, либо сообщить, что эти суммы модулей равны. Если пользователь введёт неподходящее число, то программа должна требовать повторного ввода до тех пор, пока не будет указано корректное значение.

30). Программа должна создать массив из 12 случайных целых чисел из отрезка  $[-10;10]$  таким образом, чтобы отрицательных и положительных элементов там было поровну и не было нулей. При этом порядок следования элементов должен быть случаен (т. е. не подходит вариант, когда в массиве постоянно выпадает сначала 6 положительных, а потом 6 отрицательных чисел или же когда элементы постоянно чередуются через один и пр.). Вывести полученный массив на экран.

31). Пользователь вводит с клавиатуры натуральное число большее 3, которое сохраняется в переменную  $n$ . Если пользователь ввёл не подходящее число, то программа должна просить пользователя повторить ввод. Создать массив из  $n$  случайных целых чисел из отрезка  $[0;n]$  и вывести его на экран. Создать второй массив только из чётных элементов первого массива, если они там есть, и вывести его на экран.

32). Создать двумерный массив из 6 строк по 7 столбцов в каждой из случайных целых чисел из отрезка  $[0;9]$ . Вывести массив на экран. Преобразовать массив таким образом, чтобы на первом месте в каждой строке стоял её наибольший элемент. При этом изменять состав массива нельзя, а можно только переставлять элементы в рамках одной строки. Порядок остальных элементов строки не важен (т.е. можно совершить только одну перестановку, а можно отсортировать по убыванию каждую строку). Вывести преобразованный массив на экран.

33). Для проверки остаточных знаний учеников после летних каникул, учитель младших классов решил начинать каждый урок с того, чтобы задавать каждому ученику пример из таблицы умножения, но в классе 15 человек, а примеры среди них не должны повторяться. В помощь учителю напишите программу, которая будет выводить на экран 15 случайных примеров из таблицы умножения (от  $2*2$  до  $9*9$ , потому что задания по умножению на 1 и на 10 – слишком просты). При этом среди 15 примеров не должно быть повторяющихся (примеры  $2*3$  и  $3*2$  и им подобные пары считать повторяющимися).

34). Вычислить сумму и число положительных элементов матрицы  $A[N, N]$ , находящихся над главной диагональю.

35). Дана матрица  $A$  размером  $n \times m$ . Определить  $k$  – количество особых элементов массива  $A$ , считая его элемент особым, если он больше суммы остальных элементов его столбца.

36). Задана квадратная матрица. Поменять местами строку с максимальным элементом на главной диагонали со строкой с заданным номером.

37). Дана матрица  $B[N, M]$ . Найти в каждой строке матрицы максимальный и минимальный элементы и поменять их местами с первым и последним элементом строки соответственно.

38). Дана целая квадратная матрица  $n$ -го порядка. Определить, является ли она магическим квадратом, т.е. такой, в которой суммы элементов во всех строках и столбцах одинаковы.

39). Задана матрица размером  $n \times t$ . Найти максимальный по модулю элемент матрицы. Переставить строки и столбцы матрицы таким образом, чтобы максимальный по модулю элемент был расположен на пересечении  $k$ -й строки и  $k$ -го столбца.

40). Дана квадратная матрица  $A[N, N]$ . Записать на место отрицательных элементов матрицы нули, а на место положительных – единицы.

## Глава 2 КЛАССЫ

### Тема 2.1 Основы классов

Мы пользовались классами с самого начала этого курса. Однако до сих пор применялась только наиболее примитивная форма класса. Классы, созданные в предшествующих главах, служили только в качестве контейнеров метода `main ()`, который мы использовали для ознакомления с основами синтаксиса Java. Как вы вскоре убедитесь, классы предоставляют значительно больше возможностей, чем те, которые были представлены до сих пор.

Вероятно, наиболее важное свойство класса то, что он определяет новый тип данных. После того как он определен, этот новый тип можно применять для создания объектов данного типа. Таким образом, класс – это шаблон объекта, а объект – это экземпляр класса. Поскольку объект является экземпляром класса, термины объект и экземпляр часто используются попеременно.

### Тема 2.2 Общая форма класса

При определении класса объявляют его конкретную форму и сущность. Это выполняется путем указания данных, которые он содержит, и кода, воздействующего на эти данные. Хотя очень простые классы могут содержать только код или только данные, большинство классов, применяемых в реальных программах, содержит оба эти компонента. Как будет показано в дальнейшем, код класса определяет интерфейс к его данным. Для объявления класса служит ключевое слово `class`. Упрощенная общая форма определения класса имеет следующий вид:

```
class имя_класса {  
    тип поле1;  
    тип поле2;  
    // ...  
    тип полеN;  
    тип имя_метода 1 (список_параметров) {  
        // тело метода  
    }  
    тип имя_метода2 (список_параметров) {  
        // тело метода  
    }  
    // ...  
    тип имя_методаN (список_параметров) {  
        // тело метода  
    }  
}
```

Данные, или переменные, определенные внутри класса, называются полями или переменными экземпляра. Код содержится внутри методов. Определенные внутри класса методы и переменные вместе называют членами класса. В большинстве классов действия с переменными экземпляров и доступ к ним выполняются через методы, определенные в этом классе. Таким образом, в общем случае именно методы определяют способ использования данных класса.

Определенные внутри класса переменные называют переменными экземпляра, поскольку каждый экземпляр класса (т.е. каждый объект класса) содержит собственные копии этих переменных. Таким образом, данные одного объекта отделены и отличаются от данных другого объекта. Вскоре мы вернемся к рассмотрению этой концепции, но она слишком важна, чтобы можно было обойтись без хотя бы предварительного ознакомления с ней.

Все методы имеют ту же общую форму, что и метод `main ()`, который мы использовали до сих пор. Однако большинство методов не будут указаны как `static` или `public`. Обратите внимание, что общая форма класса не содержит определения метода `main ()`. Классы Java могут и не содержать этот метод. Его обязательно указывать только в тех случаях, когда данный класс служит начальной точкой программы. Более того, апплеты вообще не требуют использования метода `main ()`.

Изучение классов начнем с простого примера. Ниже приведен код класса `Car` (Автомобиль), который определяет две переменные экземпляра: `speed` (скорость) и `color` (цвет). Для более полного описания объекта полей должно быть больше, но для простоты примера, мы ограничимся двумя. В настоящий момент `Car` не содержит никаких методов (но вскоре мы добавим в него метод).

Листинг 2.1

```
public class Car{  
    public int speed;  
    public String color;  
}
```

Как уже было сказано, класс определяет новый тип данных. В данном случае новый тип данных назван `Car`. Это имя будет использоваться для объявления объектов типа `Car`. Важно помнить, что объявление `class` создает только шаблон, но не действительный объект. Таким образом, приведенный код не приводит к появлению никаких объектов типа `Car`.

Чтобы действительно создать объект `Car`, нужно использовать оператор, подобный следующему:

```
//создание объекта bmw класса Car  
Car bmw = new Car ();
```

После выполнения этого оператора `bmw` станет экземпляром класса `Car`. То есть он обретет "физическое" существование. Но пока можете не задумываться о нюансах этого оператора.

Повторим еще раз: при каждом создании экземпляра класса мы создаем объект, который содержит собственную копию каждого поля, определенного

классом. Таким образом, каждый объект Car будет содержать собственные копии полей speed и color. Для получения доступа к этим полям применяется операция точки (.). Эта операция связывает имя объекта с именем поля. Например, чтобы занести значения в поля объекта bmw нужно использовать следующие операторы:

```
bmw.speed=100;  
bmw.color="RED";
```

Этот оператор указывает компилятору, что копии переменной speed, хранящейся внутри объекта bmw, нужно присвоить значение, равное 100, копии переменной color, нужно присвоить значение, равное "RED". В общем случае операцию точки используют для доступа как к переменным экземпляра, так и к методам внутри объекта.

В листинге 2.2 приведена полная программа, в которой используется класс Car.

Листинг 2.2

```
public class Car{  
    public int speed;  
    public String color;  
}  
public class Main{  
    public static void main ( String[] args ){  
        //создание объекта bmw класса Car  
        Car bmw = new Car ();  
        bmw.speed=100;  
        bmw.color="RED";  
        System.out.println ( "speed =" +bmw.speed+", color="+bmw.color );  
    }  
}
```

Каждый файл должен быть помещен в свой файл. Выполнив компиляцию этой программы, вы убедитесь в создании двух файлов .class: одного для Car и одного для Main. Компилятор Java автоматически помещает каждый класс в отдельный файл с расширением .class.

В результате будет получен следующий вывод:

```
speed =100, color=RED
```

### **Задание:**

Создайте класс Student, в котором будут объявлены поля(переменные экземпляра) класса – numberCourse, nameStudent. Создайте второй класс Main, в котором будет создан объект ivanov. Занесите данные в поля и выведите их.

Как было сказано ранее, каждый объект содержит собственные копии переменных экземпляра. Это означает, что при наличии двух объектов Car каждый из них будет содержать собственные копии переменных speed и color. Важно понимать, что изменения переменных экземпляра одного

объекта не влияют на переменные экземпляра другого. Например, в листинге 2.3 объявлены два объекта Car.

Листинг 2.3

```
public class Car{
    public int speed;
    public String color;
}
public class Main {
    public static void main(String[] args) {
        //создание объекта bmw класса Car
        Car bmw = new Car();
        bmw.speed = 100;
        bmw.color = "RED";
        Car lada = new Car();
        lada.speed = 75;
        lada.color = "GREEN";
        // вывод всех полей объектов
        System.out.println("bmw: speed =" + bmw.speed + ", color=" +
bmw.color);
        System.out.println("lada: speed =" + lada.speed + ", color=" +
lada.color);
        //изменение полей объекта bmw
        bmw.speed = 90;
        bmw.color = "BLACK";
        // вывод всех полей объектов после изменений
        System.out.println("bmw: speed =" + bmw.speed + ", color=" +
bmw.color);
        System.out.println("lada: speed =" + lada.speed + ", color=" +
lada.color);
    }
}
```

В результате выполнения программы, получим:

```
bmw: speed =100, color=RED
lada: speed =75, color=GREEN
bmw: speed =90, color=BLACK
lada: speed =75, color=GREEN
```

Как видите, изменение полей в объекте bmw никак не влияет на поля объекта lada.

### **Задание:**

Создайте в нашем классе Student второй объект – petrov. Занесите для него значение в поля. Выведите данные для обоих объектов.

## Тема 2.3 Объявление объектов

Как мы уже отмечали, при создании класса вы создаете новый тип данных. Этот тип можно использовать для объявления объектов данного типа. Однако создание объектов класса – двухступенчатый процесс. Вначале необходимо объявить переменную типа класса. Эта переменная не определяет объект. Она представляет собой всего лишь переменную, которая может ссылаться на объект. Затем потребуется получить действительную, физическую копию объекта и присвоить ее этой переменной. Это можно выполнить с помощью операции `new`. Эта операция динамически (т.е. во время выполнения) распределяет память под объект и возвращает ссылку на него. В общих чертах эта ссылка представляет собой адрес объекта в памяти, распределенной операцией `new`. Затем эта ссылка сохраняется в переменной. Таким образом, в Java все объекты классов должны распределяться динамически. Рассмотрим эту процедуру более подробно.

В приведенном ранее примере программы строка, подобная следующей, используется для объявления объекта типа `Car`:

```
Car bmw = new Car();
```

Этот оператор объединяет только что описанные шаги. Чтобы каждый из шагов был более очевидным, его можно было переписать следующим образом:

```
Car    bmw;    // объявление ссылки на объект
bmw = new Car(); // распределение памяти для объекта Car
```

Первая строка объявляет `bmw` в качестве ссылки на объект типа `Car`. После выполнения этой строки переменная `bmw` содержит значение `null`, свидетельствующее о том, что она еще не указывает на реальный объект. Любая попытка использования `bmw` на этом этапе приведет к возникновению ошибки времени компиляции. Следующая строка распределяет память под реальный объект (создает объект в памяти) и присваивает переменной `bmw` ссылку на этот объект. После выполнения второй строки переменную `bmw` можно использовать, как если бы она была объектом `Car`. Но в действительности переменная `bmw` просто содержит адрес памяти реального объекта `Car`. Эффект выполнения этих двух строк кода показан на рис. 2.1

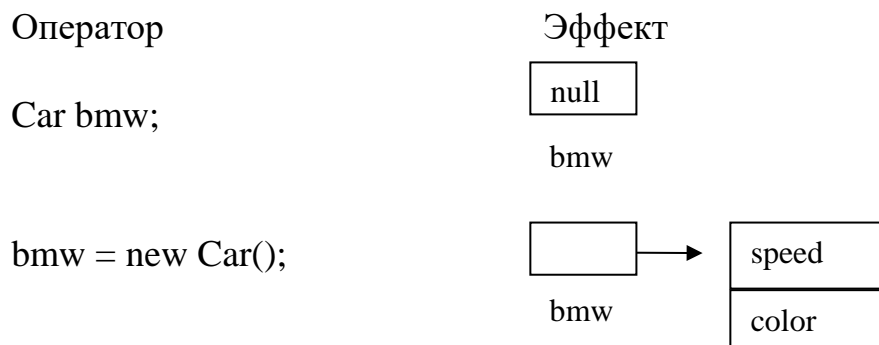


Рисунок 2.1 – Объявление объекта типа `Car`



## Тема 2.4 Более подробное рассмотрение операции new

Как было сказано, операция new динамически распределяет память для объекта. Общая форма этой операции имеет следующий вид:

```
переменная_класса = new имя_класса();
```

Здесь переменная\_класса – переменная создаваемого типа класса. Имя\_класса – это имя класса, конкретизация которого выполняется. Имя класса, за которым следуют круглые скобки, указывает конструктор данного класса. Конструктор определяет действия, выполняемые при создании объекта класса. Конструкторы – важная часть всех классов, и они обладают множеством важных атрибутов. Большинство классов, используемых в реальных программах, явно определяют свои конструкторы внутри своего определения класса. Однако если никакой явный конструктор не указан, Java автоматически предоставит конструктор, используемый по умолчанию. Это же происходит в случае объекта `String`. Пока мы будем пользоваться конструктором, заданным по умолчанию. Вскоре научимся определять собственные конструкторы.

Может возникнуть вопрос, почему не требуется использовать операцию new для таких элементов, как целые числа или символы. Это обусловлено тем, что элементарные типы Java реализованы не в виде объектов, а в виде "обычных" переменных. Это сделано для повышения эффективности. Как вы убедитесь, объекты обладают множеством свойств и атрибутов, которые требуют, чтобы Java-программа обрабатывала их иначе, чем элементарные типы. Отсутствие накладных расходов, связанных с обработкой объектов, при обработке элементарных типов позволяет эффективнее реализовать элементарные типы. Несколько позже мы приведем объектные версии элементарных типов, которые могут пригодиться в ситуациях, когда требуются полноценные объекты этих типов.

Важно понимать, что операция new распределяет память для объекта во время выполнения. Преимущество этого подхода состоит в том, что программа может создавать ровно столько объектов, сколько требуется во время ее выполнения. Однако поскольку объем памяти ограничен, возможна ситуация, когда операция new не сможет выделить память для объекта из-за ее нехватки. В этом случае возникнет исключение времени выполнения. В примерах программ, приведенных в этой книге, можно не беспокоиться по поводу недостатка объема памяти, но в реальных программах эту возможность придется учитывать.

Еще раз рассмотрим различие между классом и объектом. Класс создает новый тип данных, который можно использовать для создания объектов. То есть класс создает логический каркас, определяющий взаимосвязь между его членами. При объявлении объекта класса мы создаем экземпляр этого класса. Таким образом, класс – это логическая конструкция. А объект обладает физической сущностью. (То есть объект занимает область в памяти.) Важно помнить об этом различии.

## Тема 2.5 Присваивание переменных объектных ссылок

При выполнении присваивания переменные объектных ссылок действуют иначе, чем можно было бы представить.

```
Car bmw= new Car();
```

```
Car lada=bmw;
```

Можно подумать, что переменной lada присваивается ссылка на копию объекта, на которую ссылается переменная bmw. То есть может показаться, что bmw и lada ссылаются на отдельные и различные объекты. Однако это не так. После выполнения этого фрагмента кода обе переменные будут ссылаться на один и тот же объект. Эта операция присваивания приводит лишь к тому, что переменная lada ссылается на тот же объект, что и переменная bmw. Таким образом, любые изменения, выполненные в объекте через переменную lada, окажут влияние на объект, на который ссылается переменная bmw, поскольку это – один и тот же объект. Эта ситуация отражена на рис. 2.2.

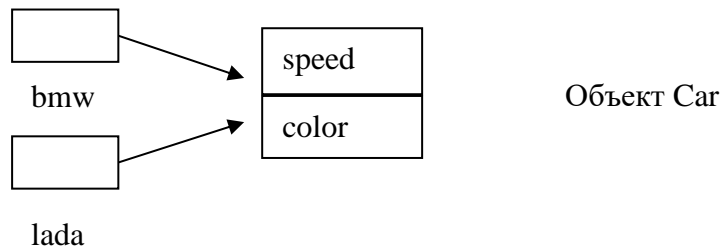


Рисунок 2.2 – Использование переменных объектных ссылок

Хотя и bmw и lada ссылаются на один и тот же объект, эти переменные не связаны между собой никаким другим образом. Например, следующая операция присваивания значения переменной bmw просто разорвет связь переменной bmw с исходным объектом, не оказывая влияния на сам объект или на переменную lada:

```
Car bmw= new Car();
```

```
Car lada=bmw;
```

```
bmw=null;
```

В этом примере значение bmw установлено равным null, но переменная lada по-прежнему указывает на исходный объект.

Присваивание ссылочной переменной одного объекта ссылочной переменной другого объекта не ведет к созданию копии объекта, а лишь создает копию ссылки.

### Задание:

Создайте в нашем классе Student объект – petrov просто присвоив ему ivanov. Выведите данные про оба объекта. Затем измените данные через ссылку ivanov и снова выведите данные. Проанализируйте полученные результаты.

## Тема 2.6 Знакомство с методами

Как было сказано в начале этой главы, обычно классы состоят из двух элементов: переменных экземпляра (полей) и методов. Поскольку язык Java предоставляет им столь большие возможности и гибкость, тема методов очень обширна. Однако чтобы можно было приступить к добавлению методов к своим классам, необходимо ознакомиться с рядом их основных характеристик.

Общая форма объявления метода выглядит следующим образом:

```
спецификатор_доступа тип имя (список_параметров) {  
    // тело метода  
}
```

Здесь *тип* указывает тип данных, возвращаемых методом. Он может быть любым допустимым типом, в том числе типом класса, созданным программистом. Если метод не возвращает значение, типом его возвращаемого значения должен быть `void`. *Имя* служит для указания имени метода. Оно может быть любым допустимым идентификатором, кроме тех, которые уже используются другими элементами в текущей области определения. *Список\_параметров* – *последовательность пар "тип-идентификатор", разделенных запятыми*. По сути, параметры – это переменные, которые принимают значения аргументов, переданных методу во время его вызова. Если метод не имеет параметров, список параметров будет пустым. Методы, тип возвращаемого значения которых отличается от `void`, возвращают значение вызывающей процедуре с помощью следующей формы оператора `return`: *return значение*; Здесь значение – это возвращаемое значение. В нескольких последующих разделах мы рассмотрим создание различных типов методов, в том числе как принимающие параметры, так и возвращающие значения.

Хотя было бы весьма удобно создать класс, который содержит только данные, в реальных программах подобное встречается редко. В большинстве случаев для осуществления доступа к переменным экземпляра, определенным классом, придется использовать методы. Фактически методы определяют интерфейсы большинства классов. Это позволяет программисту, который реализует класс, скрывать конкретную схему внутренних структур данных за более понятными абстракциями метода. Кроме определения методов, которые обеспечивают доступ к данным, можно определять также методы, используемые внутренне самим классом.

Теперь приступим к добавлению метода в класс `Car`. Просматривая предшествующие программы, легко прийти к выводу, что класс `Car` мог бы лучше справиться с выводом данных об объекте, чем класс `Main`. Для этого в класс `Car` нужно добавить метод, как показано в листинге 2.4.

Листинг 2.4

```
// Эта программа содержит метод внутри класса Car.  
public class Car {  
    public int speed;
```

```

    public String color;
    public void show() {
        System.out.println("speed =" + speed + ", color=" + color);
    }
}

```

```

public class Main {
    public static void main(String[] args) {
        //создание объекта bmw класса Car
        Car bmw = new Car();
        bmw.speed = 100;
        bmw.color = "RED";
        Car lada = new Car();
        lada.speed = 75;
        lada.color = "GREEN";
        // вывод всех полей объектов
        bmw.show();
        lada.show();
    }
}

```

Эта программа генерирует следующий вывод:

```

speed =100, color=RED
speed =75, color=GREEN

```

В первой строке `bmw.show()`; присутствует обращение к методу `show()`, по отношению к объекту `bmw`, для чего было использовано имя объекта, за которым следует символ операции точки. Таким образом, обращение к `bmw.show()` отображает параметры автомобиля, определенного объектом `bmw`, а обращение к `lada.show()` – параметры автомобиля, определенного объектом `lada`. При каждом вызове метода `show ()` он отображает параметры указанного автомобиля.

В методе `show()` следует обратить внимание на один очень важный нюанс: ссылка на переменные экземпляра `speed` и `color` выполняется непосредственно, без указания перед ними имени объекта или операции точки. Когда метод использует переменную экземпляра, которая определена его классом, он выполняет это непосредственно, без указания явной ссылки на объект и без применения операции точки. Это становится понятным, если немного подумать. Метод всегда вызывается по отношению к какому-то объекту его класса. Как только этот вызов выполнен, объект известен. Таким образом, внутри метода вторичное указание объекта совершенно излишне. Это означает, что `speed` и `color` неявно ссылаются на копии этих переменных, хранящиеся в объекте, который вызывает метод `show ()`.

Подведем краткие итоги. Когда обращение к переменной экземпляра выполняется кодом, не являющимся частью класса, в котором определена переменная экземпляра, оно должно выполняться посредством объекта с

применением операции точки. Однако когда это обращение осуществляется кодом, который является частью того же класса, где определена переменная экземпляра, ссылка на переменную может выполняться непосредственно. Эти же правила применимы и к методам.

### **Задание:**

Создайте в нашем классе Student метод выводящий параметры студента на консоль(тип возвращаемого значения у метода void).

## **Тема 2.7 Возвращение значения из метода**

Добавим функциональности в наш класс Car. Напишем метод, который будет вычислять за какое время автомобиль проедет необходимое расстояние. Метод будет возвращать полученный результат. Программа из листинга 2.5 – усовершенствованная версия предыдущей программы – выполняет именно эту задачу.

### **Листинг 2.5**

```
public class Car {
    public int speed;
    public String color;

    public void show() {
        System.out.println("speed =" + speed + ", color=" + color);
    }
    //метод вычисляет за какое время автомобиль
    // проедет путь равный length
    public double time() {
        double length=0;
        System.out.println("введите расстояние");
        Scanner sc =new Scanner(System.in);
        if(sc.hasNextDouble()){
            length=sc.nextDouble();
        }
        return length / speed;
    }
}

public class Main {
    public static void main(String args[]) {
        //создание объекта bmw класса Car
        Car bmw = new Car();
        bmw.speed = 100;
        bmw.color = "RED";
        //создание объекта lada класса Car
```

```

Car lada = new Car();
lada.speed = 75;
lada.color = "GREEN";
// вывод всех полей объектов
bmw.show();
lada.show();
//определение времени за которое проедет расстояние
// каждый автомобиль
double time1=bmw.time();
double time2=lada.time();
System.out.println("bmw проедет расстояние за "+time1);
System.out.println("lada проедет расстояние за "+time2);

    }
}

```

В результате работы этой программы получим:

```

speed =100, color=RED
speed =75, color=GREEN
введите расстояние
200
введите расстояние
200
bmw проедет расстояние за 2.0
lada проедет расстояние за 2.6666666666666665

```

Как видите, вызов метода `time ()` выполняется в правой части оператора присваивания.левой частью этого оператора является переменная, в данном случае `time1`, которая будет принимать значение, возвращенное методом `time ()`. Таким образом, после выполнения такого оператора:

```
double time1=bmw.time();
```

значение `bmw.time()` равно 2.0, и это время сохраняется в переменной `time1`.

При работе с возвращаемыми значениями следует учитывать два важных обстоятельства.

1). Тип данных, возвращаемых методом, должен быть совместим с возвращаемым типом, указанным методом. Например, если возвращаемым типом какого-либо метода является `boolean`, нельзя возвращать целочисленное значение.

2). Переменная, принимающая возвращенное методом значение (такая как `time1` в данном случае), также должна быть совместима с возвращаемым типом, указанным для метода.

Предыдущую программу можно было бы записать в несколько более эффективной форме, поскольку в действительности переменная `time1` совершенно не нужна. Обращение к методу `time ()` можно было бы

использовать в операторе `println ()` непосредственно, как в следующей строке кода:

```
System.out.println("bmw проедет расстояние за "+ bmw.time());
```

В этом случае при выполнении оператора `println()` метод `bmw.time()` будет вызываться автоматически, а возвращаемое им значение будет передаваться методу `println ()`.

### **Задание:**

Добавьте в нашем классе `Student` метод вычисляющий средний бал студента, по введенному массиву оценок.

## **Тема 2.8 Добавление метода, принимающего параметры**

Хотя некоторые методы не нуждаются в параметрах, большинство требует их передачи. Параметры позволяют обобщать метод. То есть метод с параметрами может работать с различными данными и/или применяться в ряде несколько различных ситуаций. В качестве иллюстрации рассмотрим очень простой пример. Ниже показан метод, который возвращает квадрат числа 10.

```
public int square () {  
    return 10 * 10;  
}
```

Хотя этот метод действительно возвращает 102, его применение очень ограничено. Однако если его изменить так, чтобы он принимал параметр, как показано в следующем примере, метод `square ()` может стать значительно более полезным.

```
public int square(int i) {  
    return i * i;  
}
```

Теперь метод `square ()` будет возвращать квадрат любого значения, с которым он вызван. То есть теперь метод `square ()` является методом общего назначения, который может вычислять квадрат любого целочисленного значения, а не только числа 10.

Приведем примеры:

```
int x, y;  
x = square(5);  
// x равно 25  
x = square(9);  
// x равно 81  
y = 2;  
x = square(y);  
// x равно 4
```

В первом обращении к методу `square ()` значение 5 будет передано параметру `i`. Во втором обращении параметр `i` примет значение, равное 9. Третий вызов метода передает значение переменной `y`, которое в этом

примере составляет 2. Как видно из этих примеров, метод square () способен возвращать квадрат любых переданных ему данных.

Важно различать два термина: параметр и аргумент. Параметр – это переменная, определенная методом, которая принимает значение при вызове метода. Например, в методе square () параметром является i. Аргумент – это значение, передаваемое методу при его вызове. Например, square (100) передает 100 в качестве аргумента. Внутри метода square () параметр i получает это значение.

Методом с параметрами можно воспользоваться для усовершенствования класса Car. В предыдущей версии программы мы вводили значение length прямо внутри метода ,а это не всегда удобно, т.к. значение может быть известно заранее, либо определяться каким-либо другим методом. Изменим метод time(), чтобы значение length передавалось в метод.

#### Листинг 2.6.

// Эта программа использует метод с параметрами.

```
public class Car {
    public int speed;
    public String color;

    public void show() {
        System.out.println("speed =" + speed + ", color=" + color);
    }
    //вычисление времени по переданному параметру
    public double time(double length) {
        return length / speed;
    }
}

public class Main {
    public static void main(String args[]) {
        //создание объекта bmw класса Car
        Car bmw = new Car();
        bmw.speed = 100;
        bmw.color = "RED";
        //создание объекта lada класса Car
        Car lada = new Car();
        lada.speed = 75;
        lada.color = "GREEN";
        // вывод всех полей объектов
        bmw.show();
        lada.show();
        //определение времени за которое проедет 200км
        // каждый автомобиль
```



```

        double time1=bmw.time(200);
        double time2=lada.time(200);
        System.out.println("bmw проедет 200км за "+time1);
        System.out.println("lada проедет 200км за "+time2);
    }
}

```

В результате работы этой программы получим:

```

speed =100, color=RED
speed =75, color=GREEN
bmw проедет 200км за 2.0
lada проедет 200км за 2.6666666666666665

```

Как видите, метод `time(double length)` более универсален и прост в использовании.

### **Задание:**

Создайте в нашем классе `Student` метод вычисляющий средний бал по переданному в метод массиву оценок.

## **Тема 2.9 Конструкторы**

### **2.9.1 Конструкторы без параметров**

Инициализация всех переменных класса при каждом создании его экземпляра может оказаться утомительным процессом. Поскольку необходимость инициализации возникает столь часто, Java позволяет объектам выполнять собственную инициализацию при их создании. Эта автоматическая инициализация осуществляется с помощью конструктора.

Конструктор инициализирует объект непосредственно во время создания. Его имя совпадает с именем класса, в котором он находится, а синтаксис аналогичен синтаксису метода. Как только он определен, конструктор автоматически вызывается непосредственно после создания объекта, перед завершением выполнения операции `new`. Конструкторы выглядят несколько непривычно, поскольку не имеют ни возвращаемого типа, ни даже типа `void`. Это обусловлено тем, что неявно заданный возвращаемый тип конструктора класса – тип самого класса. Именно конструктор инициализирует внутреннее состояние объекта так, чтобы код, создающий экземпляр, с самого начала содержал полностью инициализированный, пригодный к использованию объект.

Пример класса `Car` можно изменить, чтобы значения параметров автомобиля присваивались при конструировании объекта. Для этого напишем конструктор. Вначале определим простой конструктор, который

просто устанавливает одинаковые параметры для всех автомобилей. Эта версия программы приведена в листинге 2.7.

#### Листинг 2.7

// Эта программа содержит конструктор

```
public class Car {
    public int speed;
    public String color;

    //конструктор без параметров
    public Car() {
        speed=100;
        color="RED";
    }

    public void show() {
        System.out.println("speed =" + speed + ", color=" + color);
    }

    public double time(double length) {
        return length / speed;
    }
}

public class Main {
    public static void main(String args[]) {
        //создание объекта bmw класса Car
        Car bmw = new Car();
        //создание объекта lada класса Car
        Car lada = new Car();
        // вывод всех полей объектов
        bmw.show();
        lada.show();
        //определение времени за которое проедет 200км
        // каждый автомобиль
        double time1=bmw.time(200);
        double time2=lada.time(200);
        System.out.println("bmw проедет 200км за "+time1);
        System.out.println("lada проедет 200км за "+time2);

    }
}
```

В результате работы этой программы получим:

speed =100, color=RED

speed =100, color=RED

bmw проедет 200км за 2.0

lada проедет 200км за 2.6666666666666665

Как видите, и bmw, и lada были инициализированы конструктором Car () при их создании. Поскольку конструктор присваивает всем автомобилям одинаковые значения, то мы и получили два объекта с одинаковыми параметрами, что и продемонстрировал метод show().

Прежде чем продолжить, еще раз рассмотрим операцию new. Как вы уже знаете, при распределении памяти для объекта используют следующую общую форму:

*переменная\_класса = new имя\_класса() ;*

Теперь вам должно быть ясно, почему после имени класса требуются круглые скобки. В действительности этот оператор вызывает конструктор класса. Таким образом, в строке:

Car bmw = new Car();

операция new Car () вызывает конструктор Car (). Если конструктор класса не определен явно, Java создает для класса конструктор, который будет использоваться по умолчанию. Именно поэтому приведенная строка кода работала в предшествующих версиях класса Car, в которых конструктор не был определен. Конструктор, используемый по умолчанию, инициализирует все переменные экземпляра нулевыми значениями. Зачастую конструктора, используемого по умолчанию, вполне достаточно для простых классов, чего обычно нельзя сказать о более сложных. Как только конструктор определен, конструктор, заданный по умолчанию, больше не используется.

### **Задание:**

Создайте в нашем классе Student конструктор без параметров, задающий начальные значения для студента.

## **2.9.2 Конструкторы с параметрами**

Хотя в предыдущем примере конструктор Car () инициализирует объект Car, он не особенно полезен – все автомобили получают одинаковые размеры. Следовательно, необходим способ конструирования объектов Car с различными размерами. Простейшее решение этой задачи – добавление к конструктору параметров. Как легко догадаться, это делает конструктор значительно более полезным. Например, следующая версия класса Car (листинг 2.8) определяет конструктор с параметрами, который устанавливает параметры автомобиля. Обратите особое внимание на способ создания объектов Car.

Листинг 2.8

//В этой программе класс Car использует конструктор с параметрами

```
public class Car {  
    public int speed;  
    public String color;
```

```

//конструктор с параметрами
public Car(int s, String c) {
    speed = s;
    color = c;
}

public void show() {
    System.out.println("speed =" + speed + ", color=" + color);
}

public double time(double length) {
    return length / speed;
}
}

```

```

public class Main {
    public static void main(String[] args) {
        //создание объекта bmw класса Car
        Car bmw = new Car(100,"RED");
        //создание объекта lada класса Car
        Car lada = new Car(75, "GREEN");
        // вывод всех полей объектов
        bmw.show();
        lada.show();
        //определение времени за которое проедет 200км
        //каждый автомобиль
        double time1=bmw.time(200);
        double time2=lada.time(200);
        System.out.println("bmw проедет 200км за "+time1);
        System.out.println("lada проедет 200км за "+time2);

    }
}

```

Как видите, инициализация каждого объекта выполняется в соответствии со значениями, указанными в параметрах его конструктора. Например, в следующей строке:

```
Car bmw = new Car(100,"RED");
```

значения 100 и "RED" передаются конструктору Car () при создании объекта с помощью операции new. Таким образом, копии переменных speed и color будут содержать соответственно значения 100 и "RED".

### **Задание:**

Создайте в нашем классе Student конструктор с параметрами.

### 2.9.3 Ключевое слово **this**

Иногда будет требоваться, чтобы метод ссылался на вызвавший его объект. Чтобы это было возможно, в Java определено ключевое слово **this**. Оно может использоваться внутри любого метода для ссылки на текущий объект. То есть **this** всегда служит ссылкой на объект, для которого был вызван метод. Ключевое слово **this** можно использовать везде, где допускается ссылка на объект типа текущего класса.

Для пояснения рассмотрим следующую версию конструктора `Car ()`:

```
// Избыточное применение ключевого слова this.  
public Car(int s, String c) {  
    this.speed = s;  
    this.color = c;  
}
```

Эта версия конструктора `Car ()` действует точно так же, как предыдущая. Применение ключевого слова **this** избыточно, но совершенно правильно. Внутри метода `Car ()` ключевое слово **this** всегда будет ссылаться на вызывающий объект. Хотя в данном случае это и излишне, в других случаях, один из которых рассмотрен в следующем разделе, ключевое слово **this** весьма полезно.

### 2.9.4 Соккрытие переменной экземпляра

Как вы знаете, в Java не допускается объявление двух локальных переменных с одним и тем же именем в одной и той же или во включающих одна другую областях определения. Интересно отметить, что могут существовать локальные переменные, в том числе формальные параметры методов, которые перекрываются с именами переменных экземпляра класса. Однако когда имя локальной переменной совпадает с именем переменной экземпляра, локальная переменная скрывает переменную экземпляра. Именно поэтому внутри класса `Car` переменные `speed` и `color` не были использованы в качестве имен параметров конструктора `Car ()`. В противном случае переменная `speed` ссылалась бы на формальный параметр, скрывая переменную экземпляра `speed`. Хотя обычно проще использовать различные имена, существует и другой способ выхода из подобной ситуации. Поскольку ключевое слово **this** позволяет ссылаться непосредственно на объект, его можно применять для разрешения любых конфликтов пространства имен, которые могут возникать между переменными экземпляра и локальными переменными. Например, ниже показана еще одна версия метода `Car ()`, в которой имена `speed` и `color` использованы в качестве имен параметров, а ключевое слово **this** служит для обращения к переменным экземпляра по этим же именам.

```
// Этот код служит для разрешения конфликтов пространства имен.  
public Car(int speed, String color) {
```

```
    this.speed = speed;  
    this.color = color;  
}
```

Небольшое предостережение: иногда подобное применение ключевого слова `this` может приводить к недоразумениям, и некоторые программисты стараются не применять имена локальных переменных и параметров, скрывающие переменные экземпляров. Конечно, множество программистов придерживаются противоположного мнения и считают целесообразным в целях облегчения понимания программ использовать одни и те же имена, а для предотвращения скрытия переменных экземпляров применяют ключевое слово `this`.

### **Задание:**

Измените в нашем классе `Student` конструктор с параметрами (имена параметров и полей совпадают) используя ключевое слово `this`.

## **Тема 2.10 Сборка мусора**

Поскольку распределение памяти для объектов осуществляется динамически посредством операции `new`, у читателей может возникнуть вопрос, как уничтожаются такие объекты и как их память освобождается для последующего распределения. В некоторых языках, подобных C++, динамически распределенные объекты нужно освобождать вручную с помощью операции `delete`. В Java применяется другой подход. Освобождение памяти выполняется автоматически. Используемая для выполнения этой задачи технология называется сборкой мусора. Процесс проходит следующим образом: при отсутствии каких-либо ссылок на объект программа заключает, что этот объект больше не нужен, и занимаемую объектом память можно освободить. В Java не нужно явно уничтожать объекты, как это делается в C++. Во время выполнения программы сборка мусора выполняется только изредка (если вообще выполняется). Она не будет выполняться лишь потому, что один или несколько объектов существуют и больше не используются. Более того, в различных реализациях системы времени выполнения Java могут применяться различные подходы к сборке мусора, но в большинстве случаев при написании программ об этом можно не беспокоиться.

Иногда при уничтожении объект должен будет выполнять какое-либо действие. Например, если объект содержит какой-то ресурс, отличный от ресурса Java (вроде файлового дескриптора или шрифта), может требоваться гарантия освобождения этих ресурсов перед уничтожением объекта. Для подобных ситуаций Java предоставляет механизм, называемый финализацией. Используя финализацию, можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком мусора.

Чтобы добавить в класс средство выполнения финализации, достаточно определить метод `finalize ()`. Среда времени выполнения Java вызывает этот метод непосредственно перед удалением объекта данного класса. Внутри метода `finalize ()` нужно указать те действия, которые должны быть выполнены перед уничтожением объекта. Сборщик мусора запускается периодически, проверяя наличие объектов, на которые отсутствуют ссылки как со стороны какого-либо текущего состояния, так и косвенные ссылки через другие ссылочные объекты. Непосредственно перед освобождением ресурсов среда времени выполнения Java вызывает метод `finalize ()` по отношению к объекту.

Общая форма метода `finalize ()` имеет следующий вид:

```
protected void finalize () {  
    // здесь должен находиться код финализации  
}
```

В этой синтаксической конструкции ключевое слово `protected` – спецификатор, который предотвращает доступ к методу `finalize ()` со стороны кода, определенного вне его класса.

Важно понимать, что метод `finalize ()` вызывается только непосредственно перед сборкой мусора. Например, он не вызывается при выходе объекта за рамки области определения. Это означает, что неизвестно, когда будет – и, даже будет ли вообще – выполняться метод `finalize ()`. Поэтому программа должна предоставлять другие средства освобождения используемых объектом системных ресурсов и тому подобного. Нормальная работа программы не должна зависеть от метода `finalize ()`.

## Тема 2.11 Перегрузка методов

Java разрешает определение внутри одного класса двух или более методов с одним именем, если только объявления их параметров различны. В этом случае методы называют перегруженными, а процесс – перегрузкой методов. Перегрузка методов – один из способов поддержки полиморфизма в Java. Тем читателям, которые никогда не использовали язык, допускающий перегрузку методов, эта концепция вначале может показаться странной. Но, как вы вскоре убедитесь, перегрузка методов – одна из наиболее впечатляющих и полезных функциональных возможностей Java.

При вызове перегруженного метода для определения нужной версии Java использует тип и/или количество аргументов метода. Следовательно, перегруженные методы должны различаться по типу и/или количеству их параметров. Хотя возвращаемые типы перегруженных методов могут быть различны, самого возвращаемого типа недостаточно для различения двух версий метода. Когда Java встречает вызов перегруженного метода, она просто выполняет ту его версию, параметры которой соответствуют аргументам, использованным в вызове.

Листинг 2.9 иллюстрирует перегрузку метода.

Листинг 2.9

```
// Демонстрация перегрузки методов.
public class OverloadDemo {
    public void test() {
        System.out.println("Параметры отсутствуют");
    }
    // Проверка перегрузки на наличие одного целочисленного параметра.
    public void test(int a) {
        System.out.println("a: " + a);
    }
    // Проверка перегрузки на наличие двух целочисленных параметров.
    public void test(int a, int b) {
        System.out.println("a и b: " + a + " " + b);
    }
    // Проверка перегрузки на наличие параметра типа double
    public double test(double a) {
        System.out.println("double a: " + a);
        return a * a;
    }
}

public class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;
        // вызов всех версий метода test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Результат ob.test(123.25) : " + result);
    }
}
```

Эта программа генерирует следующий вывод:

Параметры отсутствуют

a: 10

a и b: 10 20

double a: 123.25

Результат ob.test(123.25): 15190.5625

Как видите, метод test () перегружается четыре раза. Первая версия не принимает никаких параметров, вторая принимает один целочисленный параметр, третья – два целочисленных параметра, а четвертая – один параметр типа double. То, что четвертая версия метода test () возвращает также значение, не имеет никакого значения для перегрузки, поскольку возвращаемый тип никак не влияет на разрешение перегрузки.



При вызове перегруженного метода Java ищет соответствие между аргументами, которые были использованы для вызова метода, и параметрами метода. Однако это соответствие не обязательно должно быть полным. В некоторых случаях к разрешению перегрузки может применяться автоматическое преобразование типов Java. Применение автоматического преобразования типов показано в листинге 2.10 :

Листинг 2.10

// Применение автоматического преобразования типов к перегрузке.

```
public class OverloadDemo {
    public void test() {
        System.out.println("Параметры отсутствуют");
    }
    // Проверка перегрузки на наличие двух целочисленных параметров.
    public void test(int a, int b) {
        System.out.println("a и b: " + a + " " + b);
    }
    // Проверка перегрузки на наличие параметра типа double

    public double test(double a) {
        System.out.println("Внутреннее преобразование test(double) a:" + a);
        return a * a;
    }
}
```

```
public class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;
        ob.test();
        ob.test(10, 20);
        ob.test(i);
        // этот оператор вызовет test(double)
        ob.test(123.2);
        // этот оператор вызовет test(double)
    }
}
```

Программа генерирует следующий вывод:

Параметры отсутствуют

a и b: 10 20

Внутреннее преобразование test(double) a: 88.0

Внутреннее преобразование test(double) a: 123.2

Как видите, эта версия класса OverloadDemo не определяет перегрузку test (int). Поэтому при вызове метода test () с целочисленным аргументом внутри класса Overload какой-то соответствующий метод отсутствует. Однако Java может автоматически преобразовывать тип integer в тип double,

и это преобразование может использоваться для разрешения вызова. Поэтому после того, как версия `test (int)` не обнаружена, Java повышает тип `i` до `double`, а затем вызывает метод `test (double)`. Конечно, если бы метод `test (int)` был определен, вызвался бы он. Java будет использовать автоматическое преобразование типов только при отсутствии полного соответствия.

Перегрузка методов поддерживает полиморфизм, поскольку это один из способов реализации в Java концепции "один интерфейс, несколько методов". Для пояснения приведем следующие рассуждения. В тех языках, которые не поддерживают перегрузку методов, каждому методу должно быть присвоено уникальное имя. Однако часто желательно реализовать, по сути, один и тот же метод для различных типов данных. Например, рассмотрим функцию вычисления абсолютного значения. Обычно в языках, которые не поддерживают перегрузку, существует три или более версии этой функции со слегка различающимися именами. Например, в C функция `abs ()` возвращает абсолютное значение значения типа `integer`, `labs ()` – значения типа `long integer`, а `fabs ()` – значения с плавающей точкой. Поскольку язык C не поддерживает перегрузку, каждая функция должна обладать собственным именем, несмотря на то, что все три функции выполняют по существу одно и то же действие. В результате в концептуальном смысле ситуация становится более сложной, чем она есть на самом деле. Хотя каждая из функций построена на основе одной и той же концепции, программист вынужден помнить три имени. В Java подобная ситуация не возникает, поскольку все методы вычисления абсолютного значения могут использовать одно и то же имя. И действительно, стандартная библиотека классов Java содержит метод вычисления абсолютного значения, названный `abs ()`. Перегрузки этого метода для обработки всех численных типов определены в Java-классе `Math`. Java выбирает для вызова нужную версию метода `abs ()` в зависимости от типа аргумента.

Перегрузка ценна тем, что она позволяет обращаться к схожим методам по общему имени. Таким образом, имя `abs` представляет общее действие, которое должно выполняться. Выбор нужной конкретной версии для данной ситуации – задача компилятора. Программисту нужно помнить только об общем выполняемом действии. Полиморфизм позволяет свести несколько имен к одному. Хотя приведенный пример весьма прост, если эту концепцию расширить, легко убедиться в том, что перегрузка может облегчить выполнение более сложных задач.

При перегрузке метода каждая версия этого метода может выполнять любые необходимые действия. Не существует никакого правила, в соответствии с которым перегруженные методы должны быть связаны между собой. Однако со стилистической точки зрения перегрузка методов предполагает определенную связь. Таким образом, хотя одно и то же имя можно использовать для перегрузки несвязанных методов, поступать так не следует. Например, имя `sqrt` можно было бы использовать для создания методов, которые возвращают квадрат целочисленного значения и квадратный корень значения с плавающей точкой. Но эти две операции

принципиально различны. Такое применение перегрузки методов противоречит ее исходному назначению. В частности, следует перегружать только тесно связанные операции.

### **Задание:**

Напишите статический перегруженный метод для расчета площади фигуры. Расчет площади квадрата – передаем 1 параметр double; круга – 1 параметр int; прямоугольника – 2 параметра double

## **Тема 2.12 Перегрузка конструкторов**

Наряду с перегрузкой обычных методов можно также выполнять перегрузку методов конструкторов. Фактически перегруженные конструкторы станут нормой, а не исключением, для большинства классов, которые вам придется создавать для реальных программ. Чтобы это утверждение было понятным, вернемся к классу Car, разработанному ранее (листинг 2.11).

### **Листинг 2.11**

```
public class Car {
    public int speed;
    public String color;

    //конструктор с параметрами
    public Car(int speed, String color) {
        this.speed = speed;
        this.color = color;
    }

    public void show() {
        System.out.println("speed =" + speed + ", color=" + color);
    }

    public double time(double length) {
        return length / speed;
    }
}

public class Main {
    public static void main(String[] args) {
        //создание объекта bmw класса Car
        Car bmw = new Car(100,"RED");
        //создание объекта lada класса Car
        Car lada = new Car(75, "GREEN");
        // вывод всех полей объектов
    }
}
```

```

    bmw.show();
    lada.show();
    //определение времени за которое проедет 200км
    //каждый автомобиль
    double time1=bmw.time(200);
    double time2=lada.time(200);
    System.out.println("bmw проедет 200км за "+time1);
    System.out.println("lada проедет 200км за "+time2);

}
}

```

Как видите, конструктор Car () требует передачи двух параметров. Это означает, что все объявления объектов Vox должны передавать конструктору Car () два аргумента. Например, следующий оператор недопустим:

```
Car ob = new Car () ;
```

Поскольку конструктор Car () требует передачи двух аргументов, его вызов без аргументов – ошибка.

К счастью, решение подобных проблем достаточно просто: достаточно перегрузить конструктор Car, чтобы он учитывал только что описанные ситуации. Ниже приведена программа, которая содержит усовершенствованную версию класса Car (листинг 2.12).

Листинг 2.12

```

public class Car {
    public int speed;
    public String color;

    //конструктор с параметрами
    public Car(int speed, String color) {
        this.speed = speed;
        this.color = color;
    }
    //конструктор без параметров
    public Car() {

    }

    public void show() {
        System.out.println("speed =" + speed + ", color=" + color);
    }

    public double time(double length) {
        return length / speed;
    }
}

```

```

public class Main {
    public static void main(String[] args) {
        //создание объекта bmw класса Car
        Car bmw = new Car(100,"RED");
        //создание объекта lada класса Car
        Car lada = new Car();
        lada.speed=75;
        lada.color="GREEN";
        // вывод всех полей объектов
        bmw.show();
        lada.show();
        //определение времени за которое проедет 200км
        //каждый автомобиль
        double time1=bmw.time(200);
        double time2=lada.time(200);
        System.out.println("bmw проедет 200км за "+time1);
        System.out.println("lada проедет 200км за "+time2);

    }
}

```

Как видите, соответствующий перегруженный конструктор вызывается в зависимости от параметров, указанных при выполнении операции new.

### **Задание:**

Напишите перегруженные конструкторы в классе Student

## **Тема 2.13 Использование объектов в качестве параметров**

До сих пор в качестве параметров методов мы использовали только простые типы. Однако передача методам объектов – и вполне допустима, и достаточно распространена. Рассмотрим программу листинга 2.13:

### **Листинг 2.13**

// Методам можно передавать объекты.

```

public class Test {
    public int a, b;
    public Test(int i, int j) {
        a = i;
        b = j;
    }
    // возврат значения true, если параметр o равен вызывающему объекту
    public boolean equals(Test o) {
        if (o.a == a && o.b == b) {
            return true;
        }
    }
}

```

```

        } else {
            return false;
        }
    }
}

```

```

public class PassOb {
    public static void main(String args[]) {
        Test obi = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
        System.out.println("obi == ob2: " + obi.equals(ob2));
        System.out.println("obi == ob3 : " + obi.equals(ob3));
    }
}

```

Эта программа создает следующий вывод:

```

obi == ob2: true
obi == ob3: false

```

Как видите, метод `equals ()` внутри метода `Test` проверяет равенство двух объектов и возвращает результат этой проверки. То есть он сравнивает вызывающий объект с тем, который был ему передан. Если они содержат одинаковые значения, метод возвращает значение `true`. В противном случае он возвращает значение `false`. Обратите внимание, что параметр `o` в методе `equals ()` указывает `Test` в качестве типа. Хотя `Test` – тип класса, созданный программой, он используется совершенно так же, как встроенные типы `Java`.

Одно из наиболее часто встречающихся применений объектов-параметров – в конструкторах. Часто приходится создавать новый объект так, чтобы вначале он не отличался от какого-то существующего объекта. Для этого потребуется определить конструктор, который в качестве параметра принимает объект его класса. В листинге 2.14 показана инициализация одного объекта другим:

Листинг 2.14

```

public class Car {
    public int speed;
    public String color;
    // Обратите внимание, этот конструктор использует объект типа Car:
    public Car (Car ob) { // передача объекта конструктору
        this.speed = ob.speed;
        this.color = ob.color;
    }
    //конструктор с параметрами
    public Car(int speed, String color) {
        this.speed = speed;
        this.color = color;
    }
}

```

```

    //конструктор без параметров
public Car() {
    }

    public void show() {
        System.out.println("speed =" + speed + ", color=" + color);
    }

    public double time(double length) {
        return length / speed;
    }
}

public class Main {
    public static void main(String[] args) {
        //создание объекта bmw класса Car
        Car bmw = new Car(100,"RED");
        //создание объекта lada класса Car
        Car lada = new Car();
        lada.speed=75;
        lada.color="GREEN";
        Car lada2 = new Car(lada);
        // вывод всех полей объектов
        bmw.show();
        lada.show();
        lada2.show();
        //определение времени за которое проедет 200км
        //каждый автомобиль
        double time1=bmw.time(200);
        double time2=lada.time(200);
        System.out.println("bmw проедет 200км за "+time1);
        System.out.println("lada проедет 200км за "+time2);

    }
}

```

Как вы убедитесь, приступив к созданию собственных классов, чтобы объекты можно было конструировать удобным и эффективным образом, нужно располагать множеством форм конструкторов.

### **Задание:**

Добавьте в класс Student конструктор создающий копию объекта.

## Тема 2.14 Более пристальный взгляд на передачу аргументов

В общем случае существует два способа, которыми компьютерный язык может передавать аргументы подпрограмме. Первый способ – вызов по значению. При использовании этого подхода значение аргумента копируется в формальный параметр подпрограммы. Следовательно, изменения, выполненные в параметре подпрограммы, не влияют на аргумент. Второй способ передачи аргумента – вызов по ссылке. При использовании этого подхода параметру передается ссылка на аргумент (а не его значение). Внутри подпрограммы эта ссылка используется для обращения к реальному аргументу, указанному в вызове. Это означает, что изменения, выполненные в параметре, будут влиять на аргумент, использованный в вызове подпрограммы. Как вы убедитесь, в Java применяются оба подхода, в зависимости от передаваемых данных.

В Java элементарный тип передается методу по значению. Таким образом, все происходящее с параметром, который принимает аргумент, не оказывает влияния вне метода(листинг 2.15).

Листинг 2.15

// Элементарные типы передаются по значению.

```
public class Test {
    public void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}

class CallByValue {
    public static void main(String args[]) {
        Test ob = new Test();
        int a = 15, b = 20;
        System.out.println("a и b перед вызовом: " + a + " " + b);
        ob.meth(a, b);
        System.out.println("a и b после вызова: " + a + " " + b);
    }
}
```

Вывод этой программы имеет следующий вид:

a и b перед вызовом: 15 20

a и b после вызова: 15 20

Как видите, выполняемые внутри метода meth () операции не влияют на значения a и b, использованные в вызове. Их значения не изменились на 30 и 10.

При передаче объекта методу ситуация изменяется коренным образом, поскольку по существу объекты передаются посредством вызова по ссылке. Следует помнить, что при создании переменной типа класса создается лишь ссылка на объект. Таким образом, при передаче этой ссылки методу,



принимающий ее параметр будет ссылаться на тот же объект, на который ссылается аргумент. По сути это означает, что объекты передаются методам посредством вызова по ссылке. Изменения объекта внутри метода влияют на объект, использованный в качестве аргумента(листинг 2.16).

Листинг 2.16

// Объекты передаются по ссылке.

```
public class Test {  
    public int a, b;  
    public Test(int i, int j) {  
        a = i;  
        b = j;  
    }  
    // передача объекта  
    public void meth(Test o) {  
        o.a *= 2;  
        o.b /= 2;  
    }  
}
```

```
public class CallByRef {  
    public static void main(String args[]) {  
        Test ob = new Test(15, 20);  
        System.out.println("ob.a и ob.b перед вызовом: " + ob.a + " " + ob.b);  
        ob.meth(ob);  
        System.out.println("ob.a и ob.b после вызова: " + ob.a + " " + ob.b);  
    }  
}
```

Эта программа генерирует следующий вывод:

ob.a и ob.b перед вызовом: 15 20

ob.a и ob.b после вызова: 30 10

Как видите, в данном случае действия внутри метода meth () влияют на объект, использованный в качестве аргумента.

Интересно отметить, что когда ссылка на объект передается методу, сама ссылка передается посредством вызова по значению. Однако поскольку передаваемое значение ссылается на объект, копия этого значения все равно будет ссылаться на тот же объект, что и соответствующий аргумент.

Когда элементарный тип передается методу, это выполняется посредством вызова по значению. Объекты передаются неявно с помощью вызова по ссылке.

## Тема 2.15 Возврат объектов

Метод может возвращать любой тип данных, в том числе созданные типы классов. Например, в следующей программе(листинг 2.17) метод

incrByTen () возвращает объект, в котором значение переменной a на 10 больше значения этой переменной в вызывающем объекте.

Листинг 2.17

// Возвращение объекта.

```
public class Test {
    public int a;
    public Test(int i) {
        a = i;
    }
    public Test incrByTen() {
        Test temp = new Test(a + 10);
        return temp;
    }
}

public class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a после второго увеличения: " + ob2.a);
    }
}
```

Эта программа генерирует следующий вывод:

ob1.a: 2

ob2.a: 12

ob2.a после второго увеличения: 22

Как видите, при каждом вызове метода incrByTen () программа создает новый объект и возвращает ссылку на него вызывающей процедуре.

Приведенная программа иллюстрирует еще один важный момент: поскольку все объекты распределяются динамически с помощью операции new, программисту не нужно беспокоиться о том, чтобы объект не вышел за пределы области определения, т.к. выполнение метода, в котором он был создан, прекращается. Объект будет существовать до тех пор, пока где-либо в программе будет существовать ссылка на него. При отсутствии какой-либо ссылки на него объект будет уничтожен во время следующей уборки мусора.

## Тема 2.16 Рекурсия

В Java поддерживается рекурсия. Рекурсия – это процесс определения чего-либо в терминах самого себя. Применительно к программированию на

Java рекурсия – это атрибут, который позволяет методу вызывать самого себя. Такой метод называют рекурсивным.

Классический пример рекурсии – вычисление факториала числа. Факториал числа  $N$  – это произведение всех целых чисел от 1 до  $N$ . Например, факториал 3 равен  $1 \times 2 \times 3$ , или 6. В листинге 2.18 показано как можно вычислить факториал, используя рекурсивный метод.

Листинг 2.18

```
public class Factorial {
    // это рекурсивный метод
    public int fact(int n) {
        int result;
        if (n == 1) {
            return 1;
        }
        result = fact(n - 1) * n;
        return result;
    }
}

public class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();
        System.out.println("Факториал 3 равен " + f.fact(3));
        System.out.println("Факториал 4 равен " + f.fact(4));
        System.out.println("Факториал 5 равен " + f.fact(5));
    }
}
```

Вывод этой программы имеет вид:

```
Факториал 3 равен 6
Факториал 4 равен 24
Факториал 5 равен 120
```

Для тех, кто не знаком с рекурсивными методами, работа метода `fact ()` может быть не совсем понятна. Вот как работает этот метод. При вызове метода `fact ()` с аргументом, равным 1, функция возвращает 1. В противном случае она возвращает произведение `fact (n-1) *n`. Для вычисления этого выражения программа вызывает метод `fact ()` с аргументом 2. Это приведет к третьему вызову метода с аргументом, равным 1. Затем этот вызов возвратит значение 1, которое будет умножено на 2 (значение  $n$  во втором вызове метода). Этот результат (равный 2) возвращается исходному вызову метода `fact ()` и умножается на 3 (исходное значение  $n$ ). В результате мы получаем ответ, равный 6. В метод `fact ()` можно было бы вставить операторы `println ()`, которые будут отображать уровень каждого вызова и промежуточные результаты.

Когда метод вызывает самого себя, новым локальным переменным и параметрам выделяется место в стеке и код метода выполняется с этими

новыми начальными значениями. При каждом возврате из рекурсивного вызова старые локальные переменные и параметры удаляются из стека, и выполнение продолжается с момента вызова внутри метода. Рекурсивные методы выполняют действия, подобные выдвиганию и складыванию телескопа.

Из-за дополнительной перегрузки ресурсов, связанной с дополнительными вызовами функций, рекурсивные версии многих подпрограмм могут выполняться несколько медленнее их итерационных аналогов. Большое количество обращений к методу могут вызвать переполнение стека. Поскольку параметры и локальные переменные сохраняются в стеке, а каждый новый вызов создает новые копии этих значений, это может привести к переполнению стека. В этом случае система времени выполнения Java будет генерировать исключение. Однако, вероятно, об этом можно не беспокоиться, если только рекурсивная подпрограмма не начинает себя вести странным образом.

Основное преимущество применения рекурсивных методов состоит в том, что их можно использовать для создания более понятных и простых версий некоторых алгоритмов, чем при использовании итерационных аналогов. Например, алгоритм быстрой сортировки достаточно трудно реализовать итерационным методом. А некоторые типы алгоритмов, связанных с искусственным интеллектом, легче всего реализовать именно с помощью рекурсивных решений.

При использовании рекурсивных методов нужно позаботиться о том, чтобы где-либо в программе присутствовал оператор `if`, осуществляющий возврат из рекурсивного метода без выполнения рекурсивного вызова. В противном случае, будучи вызванным, метод никогда не выполнит возврат. Эта ошибка очень часто встречается при работе с рекурсией. Поэтому во время разработки советуем как можно чаще использовать операторы `println()`, чтобы можно было следить за происходящим и прервать выполнение в случае ошибки.

Рассмотрим еще один пример рекурсии(листинг 2.19). Рекурсивный метод `printArray()` выводит первые `i` элементов массива `values`.

*Листинг 2.19*

```
public class RecTest {
    public int values[];
    public RecTest(int i) {
        values = new int[i];
    }
    // рекурсивное отображение элементов массива
    public void printArray(int i) {
        if (i == 0) {
            return;
        } else {
            printArray(i - 1);
        }
    }
}
```

```

        System.out.println(" [" + (i - 1) + "] " + values[i - 1]);
    }
}

public class Recursion2 {
    public static void main(String args[]) {
        RecTest ob = new RecTest(10);
        int i;
        for (i = 0; i < 10; i++) {
            ob.values[i] = i;
        }
        ob.printArray(10);
    }
}

```

Эта программа генерирует следующий вывод:

```

[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9

```

## Тема 2.17 Введение в управление доступом

Как вы уже знаете, инкапсуляция связывает данные с манипулирующим ими кодом. Однако инкапсуляция предоставляет еще один важный атрибут: управление доступом. Посредством инкапсуляции можно управлять тем, какие части программы могут получать доступ к членам класса. Управление доступом позволяет предотвращать злоупотребления. Например, предоставляя доступ к данным только посредством четко определенного набора методов, можно предотвратить злоупотребление этими данными. Таким образом, если класс реализован правильно, он создает "черный ящик", который можно использовать, но внутренний механизм которого защищен от повреждения. Однако представленные ранее классы не полностью соответствуют этой цели.

Способ доступа к члену класса определяется спецификатором доступа, который изменяет его объявление. В Java определен обширный набор спецификаторов доступа. Некоторые аспекты управления доступом связаны главным образом с наследованием и пакетами. (Пакет — это, по сути, группирование классов.) Эти составляющие механизма управления доступом

Java будут рассмотрены в последующих разделах. А пока начнем с рассмотрения применения управления доступа к отдельному классу. Когда основы управления доступом станут понятными, освоение других аспектов не представит особой сложности.

Спецификаторами доступа Java являются `public` (общедоступный), `private` (приватный) и `protected` (защищенный). Java определяет также уровень доступа, предоставляемый по умолчанию. Спецификатор `protected` применяется только при использовании наследования. Остальные спецификаторы доступа описаны далее в этой главе.

Начнем с определения спецификаторов `public` и `private`. Когда член класса изменяется спецификатором доступа `public`, он становится доступным для любого другого кода. Когда член класса указан как `private`, он доступен только другим членам этого же класса. Теперь вам должно быть понятно, почему методу `main ()` всегда предшествует спецификатор `public`. Этот метод вызывается кодом, расположенным вне данной программы – т.е. системой времени выполнения Java. При отсутствии спецификатора доступа по умолчанию член класса считается общедоступным внутри своего класса.

В уже разработанных нами классах все члены класса использовали режим доступа, определенный по умолчанию, который, по сути, является общедоступным. Однако, как правило, это не будет соответствовать реальным требованиям. Обычно будет требоваться ограничить доступ к членам данных класса – предлагая доступ только через методы. Кроме того, в ряде случаев придется определять приватные методы класса.

Спецификатор доступа предшествует остальной спецификации типа члена. То есть оператор объявления члена должен начинаться со спецификатора доступа. Например:

```
public int i;
private double j ;
private int myMethod(int a, char, b) {
//...
}
```

Чтобы влияние использования общедоступного и приватного доступа было понятно, рассмотрим следующую программу(листинг 2.20):

Листинг 2.20

//Эта программа демонстрирует различие между спецификаторами `public` и `private`.

```
public class Test {
    int a; // доступ, определенный по умолчанию
    public int b; // общедоступный доступ
    private int c; // приватный доступ
    // методы доступа к c
    public void setC(int i) {
    // установка значения переменной c
        c = i;
    }
}
```

```

    public int getC() {
// получение значения переменной c
        return c;
    }
}

class AccessTest {
    public static void main(String args[]) {
        Test ob = new Test();
// Эти операторы правильны, а и b доступны непосредственно
        ob.a = 10;
        ob.b = 20;
// Этот оператор неверен и может вызвать ошибку
// ob.c = 100;
// Ошибка!
// Доступ к объекту c должен осуществляться посредством методов его
класса
        ob.setC(100);
        System.out.println("a, b, и c: " + ob.a + " " + ob.b + " " + ob.getC());
    }
}

```

Как видите, внутри класса Test использован метод доступа, заданный по умолчанию, что в данном примере равносильно указанию доступа public. Объект b явно указан как public. Объект c указан как приватный. Это означает, что он недоступен для кода, переделенного вне его класса. Поэтому внутри класса AccessTest объект c не может применяться непосредственно. Доступ к нему должен осуществляться посредством его общедоступных методов setC () и getC (). Удаление символа комментария из начала строки:

```
ob.c = 100; // Ошибка!
```

сделало бы компиляцию этой программы невозможной из-за нарушений правил доступа.

## Тема 2.18 Ключевое слово static

В некоторых случаях желательно определить член класса, который будет использоваться независимо от любого объекта этого класса. Обычно обращение к члену класса должно выполняться только в сочетании с объектом его класса. Однако можно создать член класса, который может использоваться самостоятельно, без ссылки на конкретный экземпляр. Чтобы создать такой член, в начало его объявления нужно поместить ключевое слово static. Когда член класса объявлен как static (статический), он доступен до создания каких-либо объектов его класса и без ссылки на какой-либо объект. Статическими могут быть объявлены как методы, так и переменные. Наиболее распространенный пример статического члена – метод main ().

Этот метод объявляют как `static`, поскольку он должен быть объявлен до создания любых объектов.

Переменные экземпляров, объявленные как `static`, по существу являются глобальными переменными. При объявлении объектов их класса программа не создает никаких копий переменной `static`. Вместо этого все экземпляры класса совместно используют одну и ту же статическую переменную.

На методы, объявленные как `static`, накладывается ряд ограничений.

- Они могут вызывать только другие статические методы.
- Они должны осуществлять доступ только к статическим переменным.
- Они никоим образом не могут ссылаться на члены типа `this` или `super`.

(Ключевое слово `super` связано с наследованием и описывается далее.)

Если для инициализации переменных типа `static` нужно выполнить вычисления, можно объявить статический блок, который будет выполняться только один раз при первой загрузке класса. В следующем примере показан класс, который содержит статический метод, несколько статических переменных и статический блок инициализации (листинг 2.21).

Листинг 2.21

// Демонстрация статических переменных, методов и блоков инициализации.

```
public class UseStatic {
    public static int a = 3;
    public static int b;
    public int c, d;
    public static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {
        System.out.println("Статический блок инициализирован.");
        b = a * 4;
    }
    //блок инициализации
    {
        System.out.println("блок инициализирован.");
        c = 10;
        d = c + 5;
    }
    public static void main(String args[]) {
        meth(42);
        UseStatic ss = new UseStatic();
        System.out.println("c = " + ss.c);
        System.out.println("d = " + ss.d);
    }
}
```



Сразу после загрузки класса UseStatic программа выполняет все операторы static. Вначале значение a устанавливается равным 3, затем программа выполняет блок static, который выводит сообщение, а затем инициализирует переменную b значением a\*4, или 12. Затем программа вызывает метод main (), который обращается к методу meth (), передавая параметру x значение 42. Три оператора println () ссылаются на две статических переменные a и b на локальную переменную x.

Вывод этой программы имеет такой вид:

Статический блок инициализирован.

x = 42

a = 3

b = 12

блок инициализирован.

c = 10

d = 15

За пределами класса, в котором они определены, статические методы и переменные могут использоваться независимо от какого-либо объекта. Для этого достаточно указать имя их класса, за которым должна следовать операция точки. Например, если метод типа static нужно вызвать извне его класса, это можно выполнить, используя следующую общую форму:

*имя\_класса.метод()*

Здесь имя\_класса – имя класса, в котором объявлен метод тип static. Как видите, этот формат аналогичен применяемому для вызова нестатических методов через переменные объектных ссылок. Статическая переменная доступна аналогичным образом – посредством операции точки, следующей за именем класса. Так в Java реализованы управляемые версии глобальных методов и переменных.

В листинге 2.22 показано обращение к статическому методу callme () и статической переменной b осуществляется посредством имени их класса StaticDemo.

Листинг 2.22

```
public class StaticDemo {
    public static int a = 42;
    public static int b = 99;
    public static void callme() {
        System.out.println("a = " + a);
    }
}

public class StaticByName {
    public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}
```

Вывод этой программы выглядит следующим образом:

a = 42

b = 99

## Тема 2.19 Ключевое слово `final`

Переменная может быть объявлена как `final` (окончательная). Это позволяет предотвратить изменение содержимого переменной. Это означает, что переменная типа `final` должна быть инициализирована во время ее объявления. Например:

```
final int FILE_NEW = 1;
```

```
final int FILE_OPEN = 2;
```

```
final int FILE_SAVE = 3;
```

```
final int FILE_SAVEAS = 4;
```

```
final int FILE_QUIT = 5;
```

Теперь все последующие части программы могут пользоваться переменной `FILE_OPEN` и прочими так, как если бы они были константами, без риска изменения их значений.

В практике программирования на Java принято идентификаторы всех переменных типа `final` записывать прописными буквами. Переменные, объявленные как `final`, не занимают отдельную область памяти для каждого экземпляра – т.е., по сути, они являются константами.

Ключевое слово `final` можно применять также к методам, но в этом случае его значение существенно отличается от применяемого к переменным. Это второе применение ключевого слова `final` описано в главе, посвященной наследованию.

## Тема 2.20 Использование массива объектов

Методы работающие с массивом, не работают на прямую с полями объекта(т.е. объект, который необходим для вызова метода никакой роли не играет), то эти методы сделаем статическими и будем обращаться к ним через имя класса.

### Листинг 2.23

```
import java.util.Scanner;
```

```
public class Car {
```

```
    public int speed;
```

```
    public String color;
```

```
    // Обратите внимание, этот конструктор использует объект типа Car:
```

```
    public Car(Car ob) { // передача объекта конструктору
```

```

        this.speed = ob.speed;
        this.color = ob.color;
    }

    //конструктор с параметрами
    public Car(int speed, String color) {
        this.speed = speed;
        this.color = color;
    }

    //конструктор без параметров
    public Car() {
    }

    //метод для ввода всех параметров
    public void create() {
        System.out.println("введите скорость");
        Scanner sc = new Scanner(System.in);
        if (sc.hasNextInt()) {
            speed = sc.nextInt();
        }
        System.out.println("введите цвет");
        sc = new Scanner(System.in);
        if (sc.hasNextLine()) {
            color = sc.nextLine();
        }
    }

    //метод для вывода всех параметров
    public void show() {
        System.out.println("speed =" + speed + ", color=" + color);
    }

    //метод для вычисления времени в пути
    public double time(double length) {
        return length / speed;
    }

    //создание массива объектов
    public static Car[] createMas(int n) {
        Car mas[] = new Car[n];
        for (int i = 0; i < mas.length; i++) {
            mas[i] = new Car();
            mas[i].create();
        }
    }

```

```

        return mas;
    }

    //ВЫВОД МАССИВА ОБЪЕКТОВ
    public static void showMas(Car[] mas) {
        for (int i = 0; i < mas.length; i++) {
            mas[i].show();
        }
    }

    //ВЫВОД ПОИСК САМОГО БЫСТРОГО АВТОМОБИЛЯ
    public static Car bestCar(Car[] mas) {
        Car best = mas[0];
        for (int i = 1; i < mas.length; i++) {
            if (mas[i].speed > best.speed) {
                best = mas[i];
            }
        }
        return best;
    }
}

public class Main {
    public static void main(String[] args) {
        Car mas[]=Car.createMas(3);
        Car.showMas(mas);
        Car best=Car.bestCar(mas);
        System.out.println("параметры самого быстрого автомобиля");
        best.show();
    }
}

```

В результате работы данной программы получим

```

введите скорость
100
введите цвет
RED
введите скорость
150
введите цвет
WHITE
введите скорость
120
введите цвет

```

GREEN

speed =100, color=RED

speed =150, color=WHITE

speed =120, color=GREEN

параметры самого быстрого автомобиля

speed =150, color=WHITE

### **Задания:**

1) Напишите класс Животные, содержащий поля – Имя, Возраст, Вес, Рост. Объявите три константы Возраст, Вес, Рост в которые запишите пороговые значения. Создайте несколько объектов класса. Напишите конструктор без параметров и конструктор с параметрами. Напишите метод, выводящий все данные о животном на консоль. Метод, определяющий одинаковые ли имена у двух животных. Метод, сравнивающий поля объекта с пороговыми значениями Возраст, Вес, Рост (пример вывода : старше 5 лет; легче 2 кг; выше 20 см ).

2) Определите класс для комплексных чисел. Напишите методы для выведения числа в комплексном виде, сложения, вычитания и умножения комплексных чисел. Во все методы передается объекты.

Для тех, кто забыл комплексные числа

$$(x, y) + (x', y') = (x + x', y + y');$$

$$(x, y) \cdot (x', y') = (xx' - yy', xy' + yx').$$

3) Определите класс для матриц. Напишите перегруженные конструктор для создания одномерной и двумерной матриц. В конструкторы передаются размерности матриц. (В конструктор для одномерной матрицы передается один параметр, для двумерной два). Напишите методы для выведения, сложения, вычитания и умножения матриц.

## **Тема 2.21 Аргументы переменной длины**

В JDK 5 была добавлена новая функциональная возможность, которая упрощает создание методов, принимающих переменное количество аргументов. Эта функциональная возможность получила название `varargs` (сокращение термина `variable-length arguments` – аргументы переменной длины). Метод, который принимает переменное число аргументов, называют методом переменной аргументности, или просто методом `varargs`.

Ситуации, в которых методу нужно передавать переменное количество аргументов, встречаются не так уж редко. Например, метод, который открывает подключение к Internet, может принимать имя пользователя, пароль, имя файла, протокол и тому подобное, но применять значения, заданные по умолчанию, если какие-либо из этих сведений опущены. В этой ситуации было бы удобно передавать только те аргументы, для которых заданные по умолчанию значения не применимы. Еще один пример – метод `printf()`, входящий в состав библиотеки ввода-вывода Java. До появления

версии J2SE 5 обработка аргументов переменной длины могла выполняться двумя способами, ни один из которых не был особенно удобным. Во-первых, если максимальное количество аргументов было небольшим и известным, можно было создавать перегруженные версии метода – по одной для каждого возможного способа вызова метода. Хотя этот способ подходит для ряда случаев, он применим только к узкому классу ситуаций.

В тех случаях, когда максимальное число возможных аргументов было большим или неизвестным, применялся второй подход, при котором аргументы помещались в массив, а затем массив передавался методу. Листинг 2.23 иллюстрирует этот подход.

Листинг 2.23

```
// Использование массива для передачи методу переменного
// количества аргументов. Это старый стиль подхода
// к обработке аргументов переменной длины.
public class PassArray {
    public static void vaTest(int v[]) {
        System.out.print("Количество аргументов: " + v.length + "
Содержимое: ");
        for (int x : v) {
            System.out.print(x + " ");
        }
        System.out.println();
    }

    public static void main(String args[]) {
        // Обратите внимание на способ создания массива
        // для хранения аргументов.
        int n1[] = { 10 };
        int n2[] = { 1, 2, 3 };
        int n3[] = { };
        vaTest(n1); // 1 аргумент
        vaTest(n2); // 3 аргумента
        vaTest(n3); // без аргументов
    }
}
```

Эта программа создает следующий вывод:

Количество аргументов: 1

Содержимое: 10

Количество аргументов: 3

Содержимое: 1 2 3

Количество аргументов: 0

Содержимое:

В программе методу `vaTest ()` аргументы передаются через массив `v`. Этот старый подход к обработке аргументов переменной длины позволяет методу `vaTest ()` принимать любое число аргументов. Однако он требует,

чтобы эти аргументы были вручную помещены в массив до вызова метода `vaTest ()`. Создание массива при каждом вызове метода `vaTest ()` не только трудоемкая, но и чреватая ошибками задача. Функциональная возможность использования методов `varargs` обеспечивает более простой и эффективный подход.

Для указания аргумента переменной длины используют три точки (...). Например, вот как метод `vaTest ()` можно записать с использованием аргумента Переменной длины:

```
public static void vaTest(int ... v) {
```

Эта синтаксическая конструкция указывает компилятору, что метод `vaTest ()` может вызываться с нулем или более аргументов. В результате `v` неявно объявляется как массив типа `int []`. Таким образом, внутри метода `vaTest ()` доступ к `v` осуществляется с использованием синтаксиса обычного массива. Предыдущая программа с применением метода `vararg` приобретает следующий вид(листинг 2.24):

Листинг 2.24

```
public class VarArgs {
    // теперь vaTest () использует аргументы переменной длины
    public static void vaTest(int... v) {
        System.out.print("Количество аргументов: " + v.length + "
Содержимое: ");
        for (int x : v) {
            System.out.print(x + " ");
        }
        System.out.println();
    }
    public static void main(String args[]) {
        // Обратите внимание на возможные способы вызова
        // vaTest () с переменным числом аргументов.
        vaTest(10); //1 аргумент
        vaTest(1, 2, 3); //3 аргумента
        vaTest(); // без аргументов
    }
}
```

Вывод этой программы совпадает с выводом исходной версии.

Отметим две важные особенности этой программы. Во-первых, как уже было сказано, внутри метода `vaTest ()` переменная `v` действует как массив. Это обусловлено тем, что `v` является массивом. Синтаксическая конструкция `...` просто указывает компилятору, что метод будет использовать переменное количество аргументов, и что эти аргументы будут храниться в массиве, на который ссылается переменная `v`. Во-вторых, в методе `main ()` метод `vaTest ()` вызывается с различным количеством аргументов, в том числе, и вовсе без аргументов. Аргументы автоматически помещаются в массив и передаются переменной `v`. В случае отсутствия аргументов длина массива равна нулю.

Наряду с параметром переменной длины массив может содержать "нормальные" параметры. Однако параметр переменной длины должен быть последним параметром, объявленным методом. Например, следующее объявление метода вполне допустимо:

```
int dolt(int a, int b, double c, int ... vals) {
```

В данном случае первые три аргумента, указанные в обращении к методу `dolt ()`, соответствуют первым трем параметрам. Все остальные аргументы считаются принадлежащими параметру `vals`.

Помните, что параметр `vararg` должен быть последним. Например, следующее объявление записано неправильно:

```
int dolt (int a, int b, double c, int ... vals, boolean stopFlag) {
```

*// Ошибка!*

В этом примере предпринимается попытка объявления обычного параметра после параметра типа `vararg`, что недопустимо.

Существует еще одно ограничение, о котором следует знать: метод должен содержать только один параметр типа `varargs`. Например, следующее объявление также неверно:

```
int dolt (int a, int b, double c, int ... vals, double ... morevals) {
```

*// Ошибка!*

Попытка объявления второго параметра типа `vararg` недопустима. Рассмотрим измененную версию метода `vaTest ()`, которая принимает обычный аргумент и аргумент переменной длины в листинге 2.25.

Листинг 2.25

```
public class VarArgs {  
    // Использование- аргумента переменной длины совместно со  
    стандартными аргументами.
```

```
    //В этом примере msg – обычный параметр, а v – параметр vararg.
```

```
    public static void vaTest(String msg, int... v) {  
        System.out.print(msg + v.length + " Содержимое: ");  
        for (int x : v) {  
            System.out.print(x + " ");  
        }  
        System.out.println();  
    }  
    public static void main(String args[]) {  
        vaTest("Один параметр vararg: ", 10);  
        vaTest("Три параметра vararg: ", 1, 2, 3);  
        vaTest("Всё параметров vararg: ");  
    }  
}
```

Вывод этой программы имеет вид:

Один параметр vararg: 1

Содержимое: 10

Три параметра vararg: 3

Содержимое: 12 3



Вез параметров vararg: 0

Метод, который принимает аргумент переменной длины, можно перегружать.

Типы его параметра vararg могут быть различными. Именно это имеет место в вариантах vaRest (int ...) и vaTest (boolean ...). Помните, что конструкция ... вынуждает компилятор обрабатывать параметр как массив указанного типа. Поэтому, подобно тому, как можно выполнять перегрузку методов, используя различные типы параметров массива, можно выполнять перегрузку методов vararg, используя различные типы аргументов переменной длины. В этом случае система Java использует различие в типах для определения нужного варианта перегруженного метода.

Второй способ перегрузки метода vararg – добавление обычного параметра. Именно это было сделано для vaTest (String, int ...). В данном случае для определения нужного метода система Java использует и количество аргументов, и их тип.

Метод, поддерживающий varargs, может быть перегружен также методом, который не поддерживает эту функциональную возможность. Например, в приведенной ранее программе метод vaTest () может быть перегружен методом vaTest (int x). Эта специализированная версия вызывается только при Наличии аргумента int. В случае передаче методу двух и более аргументов int программа будет использовать varargs-версию метода vaTest (int. . .v).

При перегрузке метода, принимающего аргумент переменной длины, могут случаться непредвиденные ошибки. Они связаны с неопределенностью, которая может возникать при вызове перегруженного метода с аргументом переменной длины. Например, рассмотрим следующую программу(листинг 2.26):

Листинг 2.26

```
// Аргументы переменной длины, перегрузка и неопределенность.  
//Эта программа содержит ошибку, и ее компиляция  
// будет невозможна!  
public class VarArgs {  
    public static void vaTest(int... v) {  
        System.out.print("vaTest (int ...): " + "Количество аргументов: " +  
v.length + " Содержимое: ");  
        for (int x : v) {  
            System.out.print(x + " ");  
        }  
        System.out.println();  
    }  
    public static void vaTest(boolean... v) {  
        System.out.print("vaTest(boolean ...) " + "Количество аргументов: " +  
v.length + " Содержимое: ");  
        for (boolean x : v) {
```

```

        System.out.print(x + " ");
    }
    System.out.println();
}

public static void main(String args[]) {
    vaTest(1, 2, 3); //OK
    vaTest(true, false, false); // OK
    vaTest(); // Ошибка: неопределенность!
}
}

```

В этой программе перегрузка метода `vaTest ()` выполняется вполне корректно. Однако ее компиляция будет невозможна из-за следующего вызова:

```
vaTest (); // Ошибка: неопределенность!
```

Поскольку параметр типа `vararg` может быть пустым, этот вызов может быть преобразован в обращение к `vaTest (int . . .)` или к `vaTest (boolean . . .)`. Оба варианта допустимы. Поэтому вызов принципиально неоднозначен.

Рассмотрим еще один пример неопределенности. Следующие перегруженные версии метода `vaTest ()` изначально неоднозначны, несмотря на то, что одна из них принимает обычный параметр:

```

static void vaTest(int ... v) {
    // ...
}

static void vaTest(int n, int ... v) {
    //...
}

```

Хотя списки параметров метода `vaTest ()` различны, компилятор не имеет возможности разрешения следующего вызова:

```
vaTest(1)
```

Должен ли он быть преобразован в обращение к `vaTest (int . . .)` с одним аргументом переменной длины или в обращение к `vaTest (int, int . . .)` без аргументов переменной длины? Компилятор не имеет возможности ответить на этот вопрос. Таким образом ситуация неоднозначна.

Из-за ошибок неопределенности, подобных описанным, в некоторых случаях придется пренебрегать перегрузкой и просто использовать два различных имени метода. Кроме того, в некоторых случаях ошибки неопределенности служат признаком концептуальных изъянов программы, которые можно устранить путем более тщательного построения решения задачи.

### **Задание:**

Напишите методы с переменным числом параметров:

1). выводящий все параметры на консоль,

- 2). вычисляющий сумму всех параметров,
- 3). произведение всех параметров,
- 4). сортирующий параметры по возрастанию.

## Тема 2.22 Классы-оболочки

Java – полностью объектно-ориентированный язык. Это означает, что все, что только можно, в Java представлено объектами.

Восемь примитивных типов нарушают это правило. Они оставлены в Java из-за многолетней привычки к числам и символам. Да и арифметические действия удобнее и быстрее производить с обычными числами, а не с объектами классов.

Но и для этих типов в языке Java есть соответствующие классы – *классы-оболочки* примитивных типов. Конечно, они предназначены не для вычислений, а для действий, типичных при работе с классами – создания объектов, преобразования объектов, получения численных значений объектов в разных формах и передачи объектов в методы по ссылке.

На рис. 2.3 показана одна из ветвей иерархии классов Java. Для каждого примитивного типа есть соответствующий класс. Числовые классы имеют общего предка – абстрактный класс `Number`, в котором описаны шесть методов, возвращающих числовое значение, содержащееся в классе, приведенное к соответствующему примитивному типу: `byteValue()`, `doublievalue()`, `floatValue()`, `intValue()`, `longValue()`, `shortValue()`. Эти методы переопределены в каждом из шести числовых классов-оболочек.

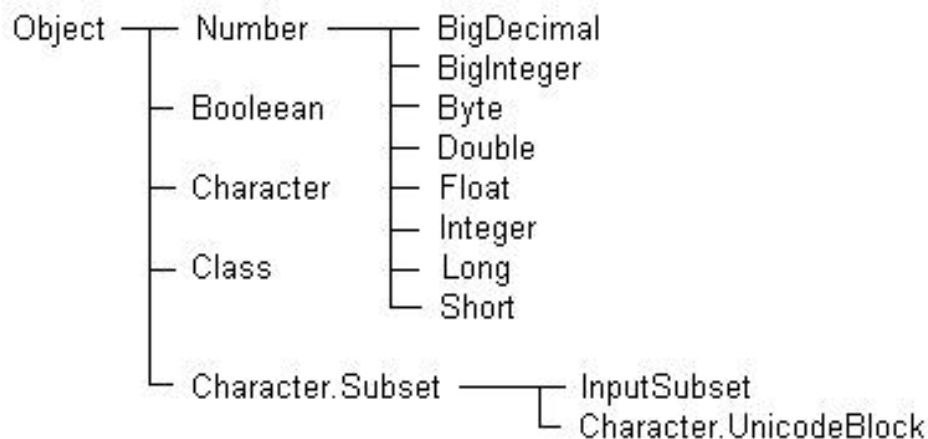


Рисунок 2.3 – Классы примитивных типов

Помимо метода сравнения объектов `equals()`, переопределенного из класса `Object`, все описанные в этой главе классы, кроме `Boolean` и `Class`, имеют метод `compareTo()`, сравнивающий числовое значение, содержащееся в данном объекте, с числовым значением объекта – аргумента метода `compareTo()`. В результате работы метода получается целое значение:

- 0, если значения равны;

- отрицательное число (−1), если числовое значение в данном объекте меньше, чем в объекте-аргументе;
- положительное число (+1), если числовое значение в данном объекте больше числового значения, содержащегося в аргументе.

В каждом из шести числовых классов-оболочек есть статические методы преобразования строки символов типа String представляющей число, в соответствующий примитивный тип: Byte.parseByte(), Double.parseDouble(), Float.parseFloat(), Integer.parseInt(), Long.parseLong(), Short.parseShort(). Исходная строка типа String, как всегда в статических методах, задается как аргумент метода. Эти методы полезны при вводе данных в поля ввода, обработке параметров командной строки, т. е. всюду, где числа представляются строками цифр со знаками плюс или минус и десятичной точкой.

В каждом из этих классов есть статические константы MAX\_VALUE и MIN\_VALUE, показывающие диапазон числовых значений соответствующих примитивных типов. В классах Double и Float есть еще константы POSITIVE\_INFINITY, NEGATIVE\_INFINITY, NaN и логические методы проверки isNaN(), isInfinite().

Если вы хорошо знаете двоичное представление вещественных чисел, то можете воспользоваться статическими методами floatToIntBits() и doubleToLongBits(), преобразующими вещественное значение в целое. Вещественное число задается как аргумент метода. Затем вы можете изменить отдельные биты побитными операциями и преобразовать измененное целое число обратно в вещественное значение методами intBitsToFloat() и longBitsToDouble().

Статическими методами toBinaryString(), toHexString() и toOctalString() классов Integer и Long можно преобразовать целые значения типов int и long, заданные как аргумент метода, в строку символов, показывающую двоичное, шестнадцатеричное или восьмеричное представление числа.

В листинге 2.27 показано применение этих методов

Листинг 2.27

```
class NumberTest{
public static void main(String[] args) {
    int i = 0;
    short sh = 0;
    double d = 0;
    Integer k1 = new Integer(55);
    Integer k2 = new Integer(100);
    Double dl = new Double(3.14);
    i = Integer.parseInt(args[0]);
    sh = Short.parseShort(args[0]);
    d = Double.parseDouble(args[1]);
    dl = new Double(args[1]);
    k1 = new Integer(args[0]);
    double x = 1.0 / 0.0;
```

```

        System.out.println("i = " + i);
        System.out.println("sh - " + sh);
        System.out.println("d. = " + d);
        System.out.println("kl.intValue() = " + kl.intValue());
        System.out.println("dl.intValue() = " + dl.intValue());
        System.out.println("kl > k2? " + kl.compareTo(k2));
        System.out.println("x = " + x);
        System.out.println("x isNaN? " + Double.isNaN(x));
        System.out.println("x isInfinite? " + Double.isInfinite(x));
        System.out.println("x      ==      Infinity?      "+      (x      ==
Double.POSITIVE_INFINITY));
        System.out.println("d = " + Double.doubleToLongBits(d));
        System.out.println("i = " + Integer.toBinaryString(i));
        System.out.println("i = " + Integer.toHexString(i));
        System.out.println("i = " + Integer.toOctalString(i));
    }
}

```

Класс Boolean очень небольшой класс, предназначенный главным образом для того, чтобы передавать логические значения в методы по ссылке.

Конструктор Boolean (String s) создает объект, содержащий значение true, если строка s равна "true" в любом сочетании регистров букв, и значение false – для любой другой строки.

Логический метод booleanValue() возвращает логическое значение, хранящееся в объекте.

В классе Character собраны статические константы и методы для работы с отдельными символами.

Статический метод digit(char ch, int radix) переводит цифру ch системы счисления с основанием radix в ее числовое значение типа int.

Статический метод forDigit(int digit, int radix) производит обратное преобразование целого числа digit в соответствующую цифру (тип char) в системе счисления с основанием radix.

Основание системы счисления должно находиться в диапазоне от Character.MIN\_RADIX до Character.MAX\_RADIX.

Метод toString() переводит символ, содержащийся в классе, в строку с тем же символом.

Статические методы toLowerCase(), toUpperCase(), toTitleCase() возвращают символ, содержащийся в классе, в указанном регистре. Последний из этих методов предназначен для правильного перевода в верхний регистр четырех кодов Unicode, не выражающихся одним символом.

Множество статических логических методов проверяют различные характеристики символа, переданного в качестве аргумента метода:

- isDefined() – выясняет, определен ли символ в кодировке Unicode;
- isDigit() – проверяет, является ли символ цифрой Unicode;

- `isIdentifierIgnorable()` – выясняет, нельзя ли использовать символ в идентификаторах;
- `isIsoControl()` – определяет, является ли символ управляющим;
- `isJavaIdentifierPart()` – выясняет, можно ли использовать символ в идентификаторах;
- `isJavaIdentifierStart()` – определяет, может ли символ начинать идентификатор;
- `isLetter()` – проверяет, является ли символ буквой Java;
- `isLetterOrDigit()` – Проверяет, является ли символ буквой или цифрой Unicode;
- `isLowerCase()` – определяет, записан ли символ в нижнем регистре;
- `isSpaceChar()` – выясняет, является ли символ пробелом в смысле Unicode;
- `isTitieCase()` – проверяет, является ли символ титульным;
- `isUnicodeIdentifierPart()` – выясняет, можно ли использовать символ в именах Unicode;
- `isUnicodeIdentifierStart()` – проверяет, является ли символ буквой Unicode;
- `isUpperCase()` – проверяет, записан ли символ в верхнем регистре;
- `isWhiteSpace()` – выясняет, является ли символ пробельным.

Точные диапазоны управляющих символов, понятия верхнего и нижнего регистра, титульного символа, пробельных символов, лучше всего посмотреть по документации Java API.

Листинг 2.28 демонстрирует использование этих методов

Листинг 2.28

```
class CharacterTest{
    public static void main(String[] args){
        char ch = '9';
        Character c = new Character(ch);
        System.out.println("ch = " + ch);
        System.out.println("cl.charValue() = "+c.charValue());
        System.out.println("number of 'A' = "+ Character.digit('A', 16));
        System.out.println("digit for 12 = "+ Character.forDigit(12, 16));
        System.out.println("cl = " + c.toString());
        System.out.println("ch isDefined? "+ Character.isDefined(ch));
        System.out.println("ch isDigit? "+ Character.isDigit(ch));
        System.out.println("ch isIdentifierIgnorable? "
            + Character.isIdentifierIgnorable(ch));
        System.out.println("ch isISOControl? "+ Character.isISOControl(ch));
        System.out.println("ch isJavaIdentifierPart? "
            + Character.isJavaIdentifierPart(ch));
        System.out.println("ch isJavaIdentifierStart? "
            + Character.isJavaIdentifierStart(ch));
        System.out.println("ch isLetter? " + Character.isLetter(ch));
        System.out.println("ch isLetterOrDigit? "
```

```

        + Character.isLetterOrDigit(ch));
    System.out.println("ch isLowerCase? " + Character.isLowerCase(ch));
    System.out.println("ch isSpaceChar? " + Character.isSpaceChar(ch));
    System.out.println("ch isTitleCase? " + Character.isTitleCase(ch));
    System.out.println("ch isUnicodeIdentifierPart? "
        + Character.isUnicodeIdentifierPart(ch));
    System.out.println("ch isUnicodeIdentifierStart? "
        + Character.isUnicodeIdentifierStart(ch));
    System.out.println("ch isUpperCase? " + Character.isUpperCase(ch));
    System.out.println("ch isWhitespace? " + Character.isWhitespace(ch));
    }
}

```

В класс `Character` вложены классы `Subset` и `UnicodeBlock`, причем класс `Unicode` и еще один класс, `InputSubset`, являются расширениями класса `Subset`, как это видно на рис. 1.35. Объекты этого класса содержат подмножества `Unicode`.

Вместе с классами-оболочками удобно рассмотреть два класса для работы со сколь угодно большими числами.

**Класс `BigInteger`.** Все примитивные целые типы имеют ограниченный диапазон значений. В целочисленной арифметике Java нет переполнения, целые числа приводятся по модулю, равному диапазону значений.

Для того чтобы было можно производить целочисленные вычисления с любой разрядностью, в состав Java API введен класс `BigInteger`, хранящийся в пакете `java.math`. Этот класс расширяет класс `Number`, следовательно, в нем переопределены методы `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`. Методы `byteValue()` и `shortValue()` не переопределены, а прямо наследуются от класса `Number`.

Действия с объектами класса `BigInteger` не приводят ни к переполнению, ни к приведению по модулю. Если результат операции велик, то число разрядов просто увеличивается. Числа хранятся в двоичной форме с дополнительным кодом.

Перед выполнением операции числа выравниваются по длине распространением знакового разряда.

Шесть конструкторов класса создают объект класса `BigDecimal` из строки символов (знака числа и цифр) или из массива байтов.

Две константы – `ZERO` и `ONE` – моделируют нуль и единицу в операциях с объектами класса `BigInteger`.

Метод `toArray()` преобразует объект в массив байтов.

Большинство методов класса `BigInteger` моделируют целочисленные операции и функции, возвращая объект класса `BigInteger`:

- `abs()` – возвращает объект, содержащий абсолютное значение числа, хранящегося в данном объекте `this`;
- `add(x)` – операция `this + x`;
- `and(x)` – операция `this & x`;

- `andNot(x)` – операция `this & (~x)` ;
- `divide (x)` – операция `this / x`;
- `divideAndRemainder(x)` – возвращает массив из двух объектов класса `BigInteger`, содержащих частное и остаток от деления `this` на `x`;
- `gcd(x)` – наибольший общий делитель, абсолютных, значений объекта `this` и аргумента `x`;
- `max(x)` – наибольшее из значений объектов `this` и аргумента `x`;
- `min(x)` – наименьшее из значений объектов `this` и аргумента `x`;
- `mod(x)` – остаток от деления объекта `this` на аргумент метода `x`;
- `modInverse(x)` – остаток от деления числа, обратного объекту `this`, на аргумент `x`;
- `modPow(n, m)` – остаток от деления объекта `this`, возведенного в степень `n`, на `m`;
- `multiply (x)` – операция `this * x`;
- `negate()` – перемена знака числа, хранящегося в объекте;
- `not()` – операция `~this`;
- `or(x)` – операция `this | x`;
- `pow(n)` – операция возведения числа, хранящегося в объекте, в степень `n`;
- `remainder(x)` – операция `this % x`;
- `shiftLeft (n)` – операция `this « n` ;
- `shiftRight (n)` – операция `this » n`;
- `signum()` – функция `sign (x)`;
- `subtract (x)` – операция `this - x`;
- `xor(x)` – операция `this ^ x`.

В листинге 2.29 приведены примеры использования данных методов

#### Листинг 2.29

```
import Java.math.BigInteger;
class BigIntegerTest{
    public static void main(String[] args){
        BigInteger a = new BigInteger("999999999999999999");
        BigInteger b = new BigInteger("8888888888888888888888");
        System.out.println("bits in a = " + a.bitLength());
        System.out.println("bits in b = " + b.bitLength());
        System.out.println("a + b = " + a.add(b));
        System.out.println("a & b = " + a.and(b));
        System.out.println("a & ~b = " + a.andNot(b));
        System.out.println("a / b = " + a.divide(b));
        BigInteger[] r = a.divideAndRemainder(b);
        System.out.println("a / b: q = " + r[0] + ", r = " + r[1]);
        System.out.println("gcd(a, b) = " + a.gcd(b));
        System.out.println("max(a, b) = " + a.max(b));
        System.out.println("min(a, b) = " + a.min(b));
        System.out.println("a mod b = " + a.mod(b));
        System.out.println("I/a mod b = " + a.modInverse(b));
    }
}
```



```

        System.out.println("a^ a mod b = " + a.modPow(a, b));
        System.out.println("a * b = " + a.multiply(b));
        System.out.println("-a = " + a.negate());
        System.out.println("~a = " + a.not());
        System.out.println("a | b = " + a.or(b));
        System.out.println("a ^ 3 = " + a.pow(3));
        System.out.println("a % b = " + a.remainder(b));
        System.out.println("a « 3 = " + a.shiftLeft(3));
        System.out.println("a » 3 = " + a.shiftRight(3));
        System.out.println("sign(a) = " + a.signum());
        System.out.println("a - b = " + a.subtract(b));
        System.out.println("a ^ b = " + a.xor(b));
    }
}

```

Обратите внимание на то, что в программу листинга 1.14 надо импортировать пакет `java.math`.

Класс `BigDecimal` расположен в пакете `java.math`.

Каждый объект этого класса хранит два целочисленных значения: мантиссу вещественного числа в виде объекта класса `BigInteger`, и неотрицательный десятичный порядок числа типа `int`.

Например, для числа 76.34862 будет храниться мантисса 7 634 862 в объекте класса `BigInteger`, и порядок 5 как целое число типа `int`. Таким образом, мантисса может содержать любое количество цифр, а порядок ограничен значением константы `Integer.MAX_VALUE`. Результат операции над объектами класса `BigDecimal` округляется по одному из восьми правил, определяемых следующими статическими целыми константами:

- `ROUND_CEILING` – округление в сторону большего целого;
- `ROUND_DOWN` – округление к нулю, к меньшему по модулю целому значению;
- `ROUND_FLOOR` – округление к меньшему целому;
- `ROUND_HALF_DOWN` – округление к ближайшему целому, среднее значение округляется к меньшему целому;
- `ROUND_HALF_EVEN` – округление к ближайшему целому, среднее значение округляется к четному числу;
- `ROUND_HALF_UP` – округление к ближайшему целому, среднее значение округляется к большему целому;
- `ROUND_UNNECESSARY` – предполагается, что результат будет целым, и округление не понадобится;
- `ROUND_UP` – округление от нуля, к большему по модулю целому значению.

В классе `BigDecimal` четыре конструктора:

- `BigDecimal (BigInteger bi)` – объект будет хранить большое целое `bi`, порядок равен нулю;

- `BigDecimal (BigInteger mantissa, int scale)` – задается мантисса `mantissa` и неотрицательный порядок `scale` объекта; если порядок `scale` отрицателен, возникает исключительная ситуация;

- `BigDecimal (double d)` – объект будет содержать вещественное число удвоенной точности `d`; если значение `d` бесконечно или NaN, то возникает исключительная ситуация;

- `BigDecimal (String val)` – число задается строкой символов `val`, которая должна содержать запись числа по правилам языка Java.

При использовании третьего из перечисленных конструкторов возникает неприятная особенность, отмеченная в документации. Поскольку вещественное число при переводе в двоичную форму представляется, как правило, бесконечной двоичной дробью, то при создании объекта, например, `BigDecimal(0.1)`, мантисса, хранящаяся в объекте, окажется очень большой. Но при создании такого же объекта четвертым конструктором, `BigDecimal ("0.1")`, мантисса будет равна просто 1.

В Классе переопределены методы `doubleValue()`, `floatValue()`, `intValue()`, `longValue()`.

Большинство методов этого класса моделируют операции с вещественными числами. Они возвращают объект класса `BigDecimal`. Здесь буква `x` обозначает объект класса `BigDecimal`, буква `n` – целое значение типа `int`, буква `r` – способ округления, одну из восьми перечисленных выше констант:

- `abs()` – абсолютное значение объекта `this`;

- `add(x)` – операция `this + x`;

- `divide(x, r)` – операция `this / x` с округлением по способу `r`;

- `divide(x, n, r)` – операция `this / x` с изменением порядка и округлением по способу `r`;

- `max(x)` – наибольшее из `this` и `x`;

- `min(x)` – наименьшее из `this` и `x`;

- `movePointLeft(n)` – сдвиг влево на `n` разрядов;

- `movePointRight(n)` – сдвиг вправо на `n` разрядов;

- `multiply(x)` – операция `this * x`;

- `negate()` – возвращает объект с обратным знаком;

- `scale()` – возвращает порядок числз;

- `setScale(n)` – устанавливает новый порядок `n` ;

- `setScale(n, r)` – устанавливает новый порядок `n` и округляет число при необходимости по способу `r`;

- `signum` – знак числа, хранящегося в объекте;

- `subtract(x)` – операция `this - x`;

- `toBigInteger()` – округление числа, хранящегося в объекте;

- `unscaledValue()` – возвращает мантиссу числа.

Листинг 2.30 показывает примеры использования этих методов

Листинг 2.30

```
import java.math.*;
class BigDecimalTest{
```

```

public static void main(String[] args) {
    BigDecimal x = new BigDecimal("-12345.67890123456789");
    BigDecimal y = new BigDecimal("345.7896e-4");
    BigDecimal z = new BigDecimal(new BigInteger("123456789"), 8);
    System.out.println("|x| = " + x.abs());
    System.out.println("x + y = " + x.add(y));
    System.out.println("x      /      y      =      "      +      x.divide(y,
BigDecimal.ROUND_DOWN));
    System.out.println("x / y = "
        + x.divide(y, 6, BigDecimal.ROUND_HALF_EVEN));
    System.out.println("max(x, y) = " + x.max(y));
    System.out.println("min(x, y) = " + x.min(y));
    System.out.println("x « 3 = " + x.movePointLeft(3));
    System.out.println("x » 3 = " + x.movePointRight(3));
    System.out.println("x * y = " + x.multiply(y));
    System.out.println("-x = " + x.negate());
    System.out.println("scale of x = " + x.scale());
    System.out.println("increase scale of x to 20 = " + x.setScale(20));
    System.out.println("decrease scale of x to 10 = "
        + x.setScale(10, BigDecimal.ROUND_HALF_UP));
    System.out.println("sign(x) = " + x.signum());
    System.out.println("x - y = " + x.subtract(y));
    System.out.println("round x = " + x.toBigInteger());
    System.out.println("mantissa of x = " + x.unscaledValue());
    System.out.println("mantissa of 0.1 =\n= "
        + new BigDecimal(0.1).unscaledValue());
    }
}

```

Приведем еще один пример. Напишем простенький калькулятор, выполняющий четыре арифметических действия с числами любой величины. Он работает из командной строки. Программа представлена в листинге 2.31.

Листинг 2.31

```

import Java.math.*;
class Calc{
    public static void main(String[] args) {
        if (args.length < 3) {
            System.err.println("Usage: Java Calc operand operator operand");
            return;
        }
        BigDecimal a = new BigDecimal(args[0]);
        BigDecimal b = new BigDecimal(args[2]);
        switch (args[1].charAt(0)) {
            case '+':
                System.out.println(a.add(b));

```

```

        break;
    case '-':
        System.out.println(a.subtract(b));
        break;
    case '*':
        System.out.println(a.multiply(b));
        break;
    case '/':
        System.out.println(a.divide(b,
            BigDecimal.ROUND_HALF_EVEN));
        break;
    default:
        System.out.println("Invalid operator");
    }
}
}

```

## Тема 2.23 Автоупаковка и автораспаковка.

Начиная с JDK 5 в Java существуют два новых средства: автоупаковка (autoboxing) и автораспаковка (autounboxing). Данная возможность позволяет не использовать методы типа `типValue()` для преобразования примитивного типа в объект. Так, автоупаковка происходит каждый раз, когда элементарный тип должен быть преобразован в примитивный тип, автораспаковка – наоборот. Автораспаковка/автораспаковка может быть осуществлена, когда аргумент передается в метод или значение из метода возвращается, или в выражениях.

*Листинг 2.32*

```

public class Main {
    public static void main(String[] args) {
        Integer objI;
        int i = 100;
        // пример ручной упаковки и распаковки
        objI = Integer.valueOf(i);
        System.out.println("1: " + objI.intValue());
        // пример автоматической упаковки/распаковки
        objI = 2000;
        i = objI;
        System.out.println("2: objI=" + objI + " i=" + i);
        // автоматическая упаковка/распаковка в методы
        double d;
        d = getDouble(2.345);
        System.out.println("3: d=" + d);
        // автоматическая упаковка/распаковка в выражении
    }
}

```

```

objI++;
i = objI / 2 + 10;
System.out.println("4: objI=" + objI + " i=" + i);
// упаковка/распаковка булевского типа
Boolean bool;
bool = true;
if (bool) {
    System.out.println("true");
} else {
    System.out.println("false");
}
// упаковка/ распаковка символьного типа
Character ch;
ch = 'd';
char c = ch;
System.out.println(c);
}

public static double getDouble(Double d) {
    return d;
}
}

```

## Тема 2.24 Строки и числа

В повседневной работе каждый программист обязательно встречается с объектами String. Объект String определяет строку символов и поддерживает операции над ней. Во многих языках программирования строка – это лишь массив символов, однако в языке Java это не так. В нем строка – это объект.

Возможно, вы не догадываетесь об этом, но реально класс String уже использовался, с самого начала. Создавая строковой литерал, вы на самом деле создавали объект String. Рассмотрим приведенное ниже выражение.

```
System.out.println(" In Java, strings are objects.");
```

Наличие в нем строки "In Java, strings are objects." автоматически приводит к созданию объекта String. Таким образом, класс String присутствовал в предыдущих программах "за сценой". В последующих разделах мы рассмотрим, как можно использовать этот класс явным образом. Помните, что в классе String предусмотрен огромный набор методов. Здесь мы вкратце обсудим лишь некоторые из них. Большую часть возможностей класса String вам предстоит изучить самостоятельно.

Объекты String создаются так же, как и объекты других типов, т.е. для этой цели используется конструктор. Например:

```
String str = new String("Hello");
```

В данном примере создается объект String с именем str, содержащий строку символов "Hello". Также есть возможность создать объект String на основе другого объекта такого же типа. Например:

```
String str = new String("Hello");  
String str2 = new String(str);
```

После выполнения этих выражений объект str2 будет также содержать строку символов "Hello".

Еще один способ создания объекта String показан ниже.

```
String str = "Java strings are powerful.";
```

В данном случае объект str инициализируется последовательностью символов "Java strings are powerful."

Создав объект String, вы можете использовать его везде, где допустим строковой литерал (последовательность символов, помещенная в кавычки). Например, объект String можно передать методу println() при его вызове так, как показано в следующем листинге 2.27.

Листинг 2.27

```
public class Main {  
    public static void main(String args[]) {  
        // Определение строк различными способами  
        String str1 = new String("Java strings are objects.");  
        String str2 = "They are constructed various ways.";  
        String str3 = new String(str2);  
        System.out.println(str1);  
        System.out.println(str2);  
        System.out.println(str3);  
    }  
}
```

Класс String содержит ряд методов, предназначенных для выполнения действий над строками. Ниже описаны некоторые из них.

*String.valueOf(параметр)* – возвращает строку типа String, являющуюся результатом преобразования параметра в строку. Параметр может быть любого примитивного или объектного типа.

*String.valueOf(charArray, index1, count)* – функция, аналогичная предыдущей для массива символов, но преобразует count символов начиная с символа, имеющего индекс index1.

У объектов типа String также имеется ряд методов. Перечислим важнейшие из них.

*s1.charAt(i)* – символ в строке s1, имеющий индекс i (индексация начинается с нуля).

*s1.endsWith(subS)* – возвращает true в случае, когда строка s1 заканчивается последовательностью символов, содержащихся в строке subS.

*s1.equals(subS)* – возвращает true в случае, когда последовательностью символов, содержащихся в строке s1, совпадает с последовательностью символов, содержащихся в строке subS.

*s1.equalsIgnoreCase(subS)* – то же, но при сравнении строк игнорируются различия в регистре символов (строчные и заглавные буквы не различаются).

*s1.indexOf(subS)* – индекс позиции, где в строке *s1* первый раз встретилась последовательность символов *subS*.

*s1.indexOf(subS,i)* – индекс позиции, начиная с *i*, где в строке *s1* первый раз встретилась последовательность символов *subS*.

*s1.lastIndexOf(subS)* – индекс позиции, где в строке *s1* последний раз встретилась последовательность символов *subS*.

*s1.lastIndexOf(subS,i)* – индекс позиции, начиная с *i*, где в строке *s1* последний раз встретилась последовательность символов *subS*.

*s1.length()* – длина строки (число 16-битных символов UNICODE, содержащихся в строке). Длина пустой строки равна нулю.

*s1.replaceFirst(oldSubS,newSubS)* – возвращает строку на основе строки *s1*, в которой произведена замена первого вхождения символов строки *oldSubS* на символы строки *newSubS*.

*s1.replaceAll(oldSubS,newSubS)* – возвращает строку на основе строки *s1*, в которой произведена замена всех вхождений символов строки *oldSubS* на символы строки *newSubS*.

*s1.split(separator)* – возвращает массив строк *String[]*, полученный разделением строки *s1* на независимые строки по местам вхождения сепаратора, задаваемого строкой *separator*. При этом символы, содержащиеся в строке *separator*, в получившиеся строки не входят. Пустые строки из конца получившегося массива удаляются.

*s1.split(separator, i)* – то же, но положительное *i* задаёт максимальное допустимое число элементов массива. В этом случае последним элементом массива становится окончание строки *s1*, которое не было расщеплено на строки, вместе с входящими в это окончание символами сепараторов. При *i* равном 0 ограничений нет, но пустые строки из конца получившегося массива удаляются. При *i* < 0 ограничений нет, а пустые строки из конца получившегося массива не удаляются.

*s1.startsWith(subS)* – возвращает *true* в случае, когда строка *s1* начинается с символов строки *subs*.

*s1.startsWith(subs, index1)* – возвращает *true* в случае, когда символы строки *s1* с позиции *index1* начинаются с символов строки *subs*.

*s1.substring(index1)* – возвращает строку с символами, скопированными из строки *s1* начиная с позиции *index1*.

*s1.substring(index1,index2)* – возвращает строку с символами, скопированными из строки *s1* начиная с позиции *index1* и кончая позицией *index2*.

*s1.toCharArray()* – возвращает массив символов, скопированных из строки *s1*.

*s1.toLowerCase()* – возвращает строку с символами, скопированными из строки *s1*, и преобразованными к нижнему регистру (строчным буквам).

*s1.toUpperCase()* – возвращает строку с символами, скопированными из строки *s1*, и преобразованными к верхнему регистру (заглавным буквам).

*s1.trim()* – возвращает копию строки *s1*, из которой убраны ведущие и завершающие пробелы.

*Листинг 2.28*

```
public class Main {
    public static void main(String args[]) {
        String str1, str2;
        // преобразование символа в строку
        str1 = String.valueOf('f');
        char s[] = {'a', 'D', 'S', 'a', 'q', 'w'};
        System.out.println(str1);
        // преобразование массива символов в строку
        str2 = String.valueOf(s, 0, s.length);
        System.out.println(str2);
        // вывод 5-го элемента строки
        System.out.println(str2.charAt(4));
        // проверяет чем заканчивается строка
        System.out.println(str2.endsWith("aqw"));
        System.out.println(str2.toLowerCase());
        // разделение строки на массив строк
        String a[] = str2.split("a");
        for (String q : a) {
            System.out.println(q);
        }
    }
}
```

Кроме указанных выше имеется ряд строковых операторов, заданных в оболочечных числовых классах. Например, методы преобразования строковых представлений чисел в числовые значения

*Byte.parseByte(строка)*

*Short.parseShort(строка)*

*Integer.parseInt(строка)*

*Long.parseLong(строка)*

*Float.parseFloat(строка)*

*Double.parseDouble(строка)*

и метод *valueOf(строка)*, преобразующий строковые представления чисел в числовые объекты – экземпляры оболочечных классов *Byte*, *Short*, *Character*, *Integer*, *Long*, *Float*, *Double*. Например,

*Byte.valueOf(строка)* , и т.п.

Кроме того, имеются методы классов *Integer* и *Long* для преобразования чисел в двоичное и шестнадцатеричное строковое представление:

*Integer.toBinaryString(число)*

*Integer.toHexString(число)*

*Long.toBinaryString(число)*

*Long.toHexString(число)*



Имеется возможность обратного преобразования – из строки в объект соответствующего класса (Byte, Short, Integer, Long) с помощью метода decode:

*Byte.decode(строка)* , и т.п.

Также полезны методы для анализа отдельных символов:

*Character.isDigit(символ)* – булевская функция, проверяющая, является ли символ цифрой.

*Character.isLetter(символ)* – булевская функция, проверяющая, является ли символ буквой.

*Character.isLetterOrDigit(символ)* – булевская функция, проверяющая, является ли символ буквой или цифрой.

*Character.isLowerCase(символ)* – булевская функция, проверяющая, является ли символ символом в нижнем регистре.

*Character.isUpperCase(символ)* – булевская функция, проверяющая, является ли символ символом в верхнем регистре.

*Character.isWhitespace(символ)* – булевская функция, проверяющая, является ли символ “пробелом в широком смысле” – пробелом, символом табуляции, перехода на новую строку и т.д.

Использование некоторых методов демонстрирует листинг 2.29.

*Листинг 2.29*

```
public class Main {
    public static void main(String args[]) {
        String str1 = "When it comes to Web programming, Java is #1.";
        String str2 = new String(str1);
        String str3 = "Java strings are powerful.";
        int result, idx;
        char ch;
        System.out.println("Length of str1: " + str1.length());
        // Отображение str1 по одному символу
        for (int i = 0; i < str1.length(); i++) {
            System.out.print(str1.charAt(i));
        }
        System.out.println();
        if (str1.equals(str2)) {
            System.out.println("str1 equals str2");
        } else {
            System.out.println("str1 does not equal str2");
        }
        if (str1.equals(str3)) {
            System.out.println("str1 equals str3");
        } else {
            System.out.println("str1 does not equal str3");
        }
        result = str1.compareTo(str3);
        if (result == 0) {
```

```

        System.out.println("str1 and str3 are equal");
    } else if (result < 0) {
        System.out.println("str1 is less than str3");
    } else {
        System.out.println("str1 is greater than str3");
    }
}
// Присвоение переменной str2 ссылки на новую строку
str2 = "One Two Three One";
idx = str2.indexOf("One");
System.out.println("Index of first occurrence of One: " + idx);
idx = str2.lastIndexOf("One");
System.out.println("Index of last occurrence of One: " + idx);
}
}

```

Конкатенацию (объединение) двух строк обеспечивает оператор +. Например приведенная ниже последовательность выражений инициализирует переменную str4 строкой "OneTwoThree".

```

String str1 = "One";
String str2 = "Two";
String str3 = "Three";
String str4 = str1 + str2 + str3;

```

Для сравнения строк определен метод equals(). Метод equals() сравнивает последовательности символов, содержащиеся в двух объектах String, и проверяет, совпадают ли они. Так как оператор == лишь определит, ссылаются ли две переменные на один и тот же объект.

Подобно другим типам данных, строки можно объединять в массивы. В листинге 2.30 приведен пример использования массивов.

*Листинг 2.30*

```

public class Main {
    public static void main(String args[]) {
        String strs[] = {"This", "is", "a", "test."};
        System.out.println("Original array: ");
        for (String s : strs) {
            System.out.print(s + " ");
        }
        System.out.println("\n");
        // Внесение изменений
        strs[1] = "was";
        strs[3] = "test, too!";
        System.out.println("Modified array: ");
        for (String s : strs) {
            System.out.print(s + " ");
        }
    }
}

```

Ниже приведен код программы, демонстрирующей использование метода `substring()` и принцип неизменности строк.

Листинг 2.31

```
public class Main {
    public static void main(String args[]) {
        String orgstr = "Java makes the Web move.";
        // Формирование подстроки
        String substr = orgstr.substring(5, 18);
        System.out.println("orgstr: " + orgstr);
        System.out.println("substr: " + substr);
    }
}
```

Как видите, исходная строка, `orgstr`, остается в прежнем виде, а объект `substr` содержит подстроку.

Перевести строковое значение в величину типа `int` или `double` можно с помощью методов `parseInt()` и `parseDouble()` классов `Integer` и `Double`. Обратное преобразование возможно при использовании метода `valueOf()` класса `String`. Кроме того, любое значение можно преобразовать в строку путем конкатенации его (+) с пустой строкой ("").

Листинг 2.32

```
public class Main {
    public static void main(String args[]) {
        String strInt = "123";
        String strDouble = "123.456";
        int x;
        double y;
        x = Integer.parseInt(strInt);
        y = Double.parseDouble(strDouble);
        System.out.println("x=" + x);
        System.out.println("y=" + y);
        strInt = String.valueOf(x + 1);
        strDouble = String.valueOf(y + 1);
        System.out.println("strInt=" + strInt);
        System.out.println("strDouble=" + strDouble);

        String str;
        str = "num=" + 345;
        System.out.println(str);
    }
}
```

Для преобразования целого числа в десятичную, двоичную, шестнадцатеричную и восьмеричную строки используются методы `toString()`, `toBinaryString()`, `toHexString()` и `toOctalString()`.

Листинг 2.33

```
public class Main {
```

```

public static void main(String[] args) {
    System.out.println(Integer.toString(262));
    System.out.println(Integer.toBinaryString(262));
    System.out.println(Integer.toHexString(267));
    System.out.println(Integer.toOctalString(267));
}
}

```

В листинге 2.34 приведен пример использования строк и switch. Программа определяет, образуют ли цифры данного четырехзначного числа N строго возрастающую последовательность.

*Листинг 2.34*

```

public class Main {
    public static void main(String[] args) {
        String str = "";
        Scanner sc = new Scanner(System.in);
        System.out.print("Введите строку из 4-х символов: ");
        if (sc.hasNextLine()) {
            str = sc.nextLine();
        }
        if (str.length() != 4) {
            System.err.println("Строка не содержит четырех символов.");
            //обработчик системной ошибки
            return;
        }
        for (int i = 0; i < str.length() - 1; i++) {
            if (str.charAt(i) > str.charAt(i + 1))//взять символ
            {
                System.out.println("false");
                return;
            }
        }
        System.out.println("true");
    }
}

```

В листинге 2.35 приведена программа, которая читает натуральное число в десятичном представлении, а на выходе выдает это же число в десятичном представлении и на естественном языке.

Например:

7 семь

204 двести четыре

52 пятьдесят два

*Листинг 2.35*

```

public class Main {
    public static void main(String[] args) {

```

```

String num = "";
char[] achNum;
char curSymbol;
Scanner sc = new Scanner(System.in);
System.out.print("Введите число: ");
if (sc.hasNextLine()) {
    num = sc.nextLine();
}
StringBuffer strBuf = new StringBuffer(num);
strBuf.reverse();
num = null;
for (int i = strBuf.length() - 1; i >= 0; i--) {
    curSymbol = strBuf.charAt(i);
    switch (i) {
        case 2: // сотни
            printSotni(curSymbol);
            break;
        case 1: // десятки
            if (curSymbol != '1') {
                printDeciatk(curSymbol);
            } else {
                i--;
                curSymbol = strBuf.charAt(i);
                printDeciatkWithEdinic(curSymbol);
            }
            break;
        case 0: // единицы
            printEdinic(curSymbol);
            break;
        default:
            System.out.println("Такие числа программа пока не
ВЫВОДИТ.");
            return;
    }
}
}
// печатает названия единиц
public static void printEdinic(char i) {
    switch (i) {
        case '1':
            System.out.print("один ");
            break;
        case '2':
            System.out.print("два ");
            break;
    }
}

```

```

        case '3':
            System.out.print("три ");
            break;
        case '4':
            System.out.print("четыре ");
            break;
        case '5':
            System.out.print("пять ");
            break;
        case '6':
            System.out.print("шесть ");
            break;
        case '7':
            System.out.print("семь ");
            break;
        case '8':
            System.out.print("восемь ");
            break;
        case '9':
            System.out.print("девять ");
            break;
    }
}
// печатает названия десятков
public static void printDeciatk(char i) {
    switch (i) {
        case '1':
            System.out.print("десять ");
            break;
        case '2':
            System.out.print("двадцать ");
            break;
        case '3':
            System.out.print("тридцать ");
            break;
        case '4':
            System.out.print("сорок ");
            break;
        case '5':
            System.out.print("пятьдесят ");
            break;
        case '6':
            System.out.print("шестьдесят ");
            break;
        case '7':

```

```

        System.out.print("семьдесят ");
        break;
    case '8':
        System.out.print("восемьдесят ");
        break;
    case '9':
        System.out.print("девяносто ");
        break;
    }
}

public static void printDeciatkWithEdinic(char i) {
    switch (i) {
        case '0':
            System.out.print("десять ");
            break;
        case '1':
            System.out.print("одиннадцать ");
            break;
        case '2':
            System.out.print("двенадцать ");
            break;
        case '3':
            System.out.print("тринадцать ");
            break;
        case '4':
            System.out.print("четырнадцать ");
            break;
        case '5':
            System.out.print("пятнадцать ");
            break;
        case '6':
            System.out.print("шестнадцать ");
            break;
        case '7':
            System.out.print("семнадцать ");
            break;
        case '8':
            System.out.print("восемнадцать ");
            break;
        case '9':
            System.out.print("девятнадцать ");
            break;
    }
}
// печатает сотни

```

```

public static void printSotni(char i) {
    switch (i) {
        case '1':
            System.out.print("сто ");
            break;
        case '2':
            System.out.print("двести ");
            break;
        case '3':
            System.out.print("триста ");
            break;
        case '4':
            System.out.print("четыреста ");
            break;
        case '5':
            System.out.print("пятьсот ");
            break;
        case '6':
            System.out.print("шестьсот ");
            break;
        case '7':
            System.out.print("семьсот ");
            break;
        case '8':
            System.out.print("восемьсот ");
            break;
        case '9':
            System.out.print("девятьсот ");
            break;
    }
}

```

Классы `StringBuilder` и `StringBuffer` являются “близнецами” и по своему предназначению близки к классу `String`, но, в отличие от последнего, содержимое и размеры объектов классов `StringBuilder` и `StringBuffer` можно изменять.

Основным и единственным отличием `StringBuilder` от `StringBuffer` является потокобезопасность последнего. В версии 1.5.0 был добавлен непотокобезопасный (следовательно, более быстрый в обработке) класс `StringBuilder`, который следует применять, если не существует вероятности использования объекта в конкурирующих потоках.

С помощью соответствующих методов и конструкторов объекты классов `StringBuffer`, `StringBuilder` и `String` можно преобразовывать друг в друга. Конструктор класса `StringBuffer` (также как и `StringBuilder`) может принимать в качестве параметра объект `String` или неотрицательный размер буфера.



Объекты этого класса можно преобразовать в объект класса `String` методом `toString()` или с помощью конструктора класса `String`.

Некоторые полезные методы:

`void setLength(int n)` – установка размера буфера;

`void ensureCapacity(int minimum)` – установка гарантированного минимального размера буфера;

`int capacity()` – возвращение текущего размера буфера;

`StringBuffer append(параметры)` – добавление к содержимому объекта строкового представления аргумента, который может быть символом, значением базового типа, массивом и строкой;

`StringBuffer insert(параметры)` – вставка символа, объекта или строки в указанную позицию;

`StringBuffer deleteCharAt(int index)` – удаление символа;

`StringBuffer delete(int start, int end)` – удаление подстроки;

`StringBuffer reverse()` – обращение содержимого объекта.

В классе присутствуют также методы, аналогичные методам класса `String`, такие как `replace()`, `substring()`, `charAt()`, `length()`, `getChars()` и др.

В листинге 2.36 показано использование некоторых методов.

Листинг 2.36

```
public class DemoStringBuffer {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer();
        System.out.println("длина ->" + sb.length());
        System.out.println("размер ->" + sb.capacity());
        // sb = "Java"; // ошибка, только для класса String
        sb.append("Java");
        System.out.println("строка ->" + sb);
        System.out.println("длина ->" + sb.length());
        System.out.println("размер ->" + sb.capacity());
        System.out.println("реверс ->" + sb.reverse());
    }
}
```

Результатом выполнения данного кода будет:

длина ->0

размер ->16

строка ->Java

длина ->4

размер ->16

реверс ->avaJ

При создании объекта `StringBuffer` конструктор по умолчанию автоматически резервирует некоторый объем памяти (16 символов), что в дальнейшем позволяет быстро менять содержимое объекта, оставаясь в границах участка памяти, выделенного под объект. Размер резервируемой памяти при необходимости можно указывать в конструкторе. Если длина строки `StringBuffer` после изменения превышает его размер, то ёмкость

объекта автоматически увеличивается, оставляя при этом резерв для дальнейших изменений. С помощью метода `reverse()` можно быстро изменить порядок символов в объекте.

Если метод, вызываемый объектом `StringBuffer`, производит изменения в его содержимом, то это не приводит к созданию нового объекта, как в случае объекта `String`, а изменяет текущий объект `StringBuffer`.

Листинг 2.37

```
public class RefStringBuffer {
    public static void changeStr(StringBuffer s) {
        s.append(" Microsystems");
    }
    public static void main(String[] args) {
        StringBuffer str = new StringBuffer("Sun");
        changeStr(str);
        System.out.println(str);
    }
}
```

В результате выполнения этого кода будет выведена строка:

Sun Microsystems

Объект `StringBuffer` передан в метод `changeStr()` по ссылке, поэтому все изменения объекта сохраняются и для вызывающего метода.

Для класса `StringBuffer` не переопределены методы `equals()` и `hashCode()`, т.е. сравнить содержимое двух объектов невозможно, к тому же хэш-коды всех объектов этого типа вычисляются так же, как и для класса `Object`.

Листинг 2.38

```
public class EqualsStringBuffer {
    public static void main(String[] args) {
        StringBuffer sb1 = new StringBuffer("Sun");
        StringBuffer sb2 = new StringBuffer("Sun");
        System.out.print(sb1.equals(sb2));
        System.out.print(sb1.hashCode() == sb2.hashCode());
    }
}
```

Результатом выполнения данной программы будет дважды выведенное значение `false`.

### Задания

1). Написать класс, в котором как поле объявлен массив строк. Вводим количество строк с консоли. Организовать ввод строк с консоли до тех пор, пока в какой-то строке не встретится слово `end`, остальные строки заполняются цифрами = номер строки. Отсортируйте строки по длине. Определите, есть ли среди строк одинаковые. Выведите 3 последних элемента самой длинной строки. Преобразуйте 2 строку к верхнему регистру.

Разделите самую длинную строку на слова. Определить является ли второй символ самой короткой строки цифрой.

2). Дан массив из N строк. Строки имеют произвольную длину. Строки содержат слова, состоящие из произвольных символов, разделенных символами ' ' ', ' ' ' N≤10. Необходимо написать методы:

- 1) сортировка строк массива по количеству слов в строке.
- 2) выводящий значения длин всех строк массива.
- 3) выводящий строки с i по j из массива.
- 4) выводящий номер строки с максимальной цифрой.
- 5) удаляющий из i-ой строки все заглавные буквы.
- 6) удаляющий из i-ой строки все символы не буквы и не цифры.
- 7) выводящий из i-ой строки все слова, содержащие только цифры.
- 8) вычисляющий сумму всех цифр i-ой строки.
- 9) выводящий из массива все слова, содержащие только прописные буквы.
- 10) выводящий все числа из строк.
- 11) удаляющий из строки часть, заключенную между двумя символами, которые вводятся (например, между скобками '(' и ') или между звездочками '\*' и т.п.).
- 12) определяющий сколько в массиве одинаковых строк.
- 13) определяющий сколько в массиве одинаковых слов (выводить слово и количество повторений).
- 14) метод, объединяющий в одну строку строки с i по j.
- 15) метод, преобразовывающий i-ую строку так, чтобы все слова начинались с заглавной буквы.
- 16) метод вносящий изменение в i-ую строку (передается номер строки и новое содержание)

3). Написать класс для хранения информации о людях. В конструктор передается один параметр String, в котором через ; перечисляется имя, возраст, вес, рост – пример: ("Alex; 45; 90; 185"). В конструкторе по этому параметру заполняются соответственно 4 поля (имя, возраст, вес, рост). Метод, в который передается аналогичная строка ("Serg; 25; 80; 180") и делается сравнение всех полей для данного объекта с данными из строки. Метод, выводящий всю информацию об объекте. Методы для изменения каждого из полей.

## Тема 2.25 Нумерованные типы

В простейшем виде механизм нумерованных типов сводится к поддержке списка именованных констант, определяющих новый тип данных. Объект нумерованного типа может принимать лишь значения, содержащиеся в списке. Таким образом, нумерованные типы предоставляют возможность создавать новый тип данных, содержащий лишь фиксированный набор допустимых значений.

В повседневной жизни нумерованные типы встречается довольно часто. Например, к ним можно отнести набор денежных единиц, используемых в стране. Месяцы в году идентифицируются именами. Так же обстоит дело с днями недели.

С точки зрения программирования нумерация полезна в любом случае, когда надо определить фиксированный набор значений. Например, нумерацию можно использовать для представления набора кодов состояния (успешное завершение, ошибка, необходимость повторной попытки). Конечно, такие значения можно определить с помощью констант, но использование нумерации – более структурированный подход.

Для создания нумерованного типа используется ключевое слово `enum`. Ниже приведен пример простого нумерованного типа, в котором перечисляются различные транспортные средства.

*Листинг 2.39*

```
public enum Transport {  
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT  
}
```

Идентификаторы `CAR`, `TRUCK` и т.д. называются нумерованными константами. Каждый из них автоматически объявляется как член `Transport`, причем при объявлении предполагаются модификаторы `public` и `static`. Тип этих констант соответствует типу выражения `enum`; в данном случае это `Transport`. В терминах Java подобные константы называются самотипизированными (приставка "само" означает, что в качестве типа константы принимается тип нумерованного выражения).

Определив нумерацию, можно создать переменную данного типа. Однако, несмотря на то, что для поддержки нумерации используется класс, сформировать экземпляр `enum` с помощью оператора `new` невозможно. Переменная нумерованного типа создается подобно переменной одного из простых типов. Например, чтобы объявить переменную `tp` рассмотренного выше типа `Transport`, надо использовать выражение

```
Transport tp;
```

Поскольку переменная `tp` принадлежит типу `Transport`, ей можно присваивать только те значения, которые определены в составе нумерации. Например, в следующей строке кода данной переменной: присваивается значение `AIRPLANE`.

```
tp = Transport.AIRPLANE;
```

Обратите внимание на то, что имя `AIRPLANE` соответствует типу `Transport`.

Для проверки равенства нумерованных констант служит оператор сравнения `==`. Например, приведенное ниже выражение сравнивает содержимое переменной `tp` с константой `TRAIN`.

```
if(tp == Transport.TRAIN) // ...
```

Нумерованное значение также можно использовать в выражении `switch`. Очевидно, что при этом в выражениях `case` могут присутствовать только константы из того же выражения `enum`, которому принадлежит константа,

указанная в выражении `switch`. Например, следующий фрагмент кода составлен корректно:

```
// Использование нумерованного типа в выражении
switch(tp) {
    case CAR:
        System.out.println("A car carries people.");
        break;
    case TRUCK:
        System.out.println("A truck carries freight.");
        break;
    case AIRPLANE:
        System.out.println("An airplane flies.");
        break;
    case TRAIN:
        System.out.println("A train runs on rails.");
        break;
    case BOAT:
        System.out.println("A boat sails on water.");
        break;
}
```

Заметьте, что в выражениях `case` используются константы без указания имени типа. Например, вместо `Transport.TRUCK` указано просто `TRUCK`. Это допустимо потому, что нумерованный тип в выражении `switch` неявно определяет тип констант. Более того, если вы попытаетесь явно указать тип константы, возникнет ошибка компиляции.

При отображении нумерованной константы с помощью, например, метода `println()` выводится ее имя. Например, в результате выполнения следующего выражения отобразится имя `BOAT`:

```
System.out.println(Transport.BOAT);
```

Ниже приведен пример программы, которая демонстрирует все особенности использования нумерованного типа `Transport`.

*Листинг 2.40*

```
enum Transport {
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
public class DemoEnum {
    public static void main(String args[]) {
        // Объявление ссылки на переменную Transport
        Transport tp;
        // Присвоение переменной tp константы AIRPLANE
        tp = Transport.AIRPLANE;
        // Вывод нумерованного значения
        System.out.println("Value of tp: " + tp);
        System.out.println();
        tp = Transport.TRAIN;
```

```
// Проверка равенства двух объектов Transport
    if (tp == Transport.TRAIN) {
        System.out.println("tp contains TRAIN.\n");
    }
// Использование нумерованного типа в выражении switch
    switch (tp) {
        case CAR:
            System.out.println("A car carries people.");
            break;
        case TRUCK:
            System.out.println("A truck carries freight.");
            break;
        case AIRPLANE:
            System.out.println("An airplane flies.");
            break;
        case TRAIN:
            System.out.println("A train runs on rails.");
            break;
        case BOAT:
            System.out.println("A boat sails on water.");
            break;
    }
}
```

Перед тем как переходить к изучению дальнейшего материала, необходимо сделать одно замечание. Имена констант, принадлежащих типу `Transport`, составлены из прописных букв (например, одна из них называется `CAR`, а не `car`).

Однако это не является обязательным требованием. Не существует правила, ограничивающего регистр символов в именах нумерованных констант. Тем не менее, поскольку нумерованные константы, как правило, предназначены для замены переменных, объявленных как `final`, имена которых по традиции записываются символами верхнего регистра, в именах нумерованных констант также в большинстве случаев используются прописные буквы. Но некоторые специалисты не разделяют это мнение.

Предыдущий пример демонстрирует создание и использование нумерованных типов, однако он не дает представления обо всех их возможностях. В отличие от других языков в Java нумерованные типы реализованы в виде классов. Несмотря на то что создать экземпляр `enum` с помощью оператора `new` невозможно, во всем остальном нумерованные типы неотличимы от классов. Такой подход предоставляет нумерованным типам богатые возможности, принципиально недостижимые в других языках. Так, например, вы можете определять конструкторы нумерованных типов,

добавлять к ним переменные экземпляра и методы и даже создавать нумерованные типы, реализующие интерфейсы.

всех нумерованных типах доступны два предопределенных метода `values()` и `valueOf()`. Их заголовки имеют следующий вид:

```
public static нумерованный_тип[] values( )
public static нумерованный_тип valueOf(String str)
```

Метод `values()` возвращает массив, содержащий нумерованные константы. Метод `valueOf()` возвращает константу, значение которой соответствует строке, переданной в качестве параметра. Тип обоих методов соответствует нумерованному типу. Например, если мы используем рассмотренный выше тип `Transport`, метод `Transport.valueOf("TRAIN")` вернет значение `TRAIN` типа `Transport`.

#### *Листинг 2.41*

```
enum Transport2 {
    CAR, TRUCK, AIRPLANE, TRAIN, BOAT
}
public class EnumDemo2 {
    public static void main(String args[]) {
        Transport2 tp;
        System.out.println("Here are all Transport constants");
        // Использование метода values()
        // Получение массива констант типа Transport
        Transport2 allTransports[] = Transport2.values();
        for (Transport2 t : allTransports) {
            System.out.println(t);
        }
        System.out.println();
        // Использование метода valueOf()
        // Получение константы AIRPLANE
        tp = Transport2.valueOf("AIRPLANE");
        System.out.println("tp contains " + tp);
    }
}
```

Обратите внимание на то, что в данной программе для перебора массива констант, полученного с помощью метода `values()`, используется вариант `foreach` цикла `for`. Для того чтобы код примера был понятнее, в нем создается переменная `allTransports`, которой присваивается ссылка на массив нумерованных констант. На самом деле это не обязательно; цикл `for` можно записать так, как показано ниже. При этом необходимость в дополнительной переменной отпадает.

```
for(Transport t : Transport.values())
    System.out.println(t);
```

Обратите также внимание на то, что значение, соответствующее имени AIRPLANE, было получено путем вызова метода `valueOf( )`. `tp = Transport.valueOf("AIRPLANE");`

Как было сказано ранее, метод `valueOf( )` возвращает нумерованное значение, соответствующее имени константы, которое было передано в виде строки при вызове метода.

Важно четко представлять себе, что каждая нумерованная константа – это объект соответствующего типа, который может содержать конструкторы, методы и переменные экземпляра. Если вы определите конструктор для `enum`, то он будет вызываться при создании каждой нумерованной константы. Каждая нумерованная константа может быть использована для вызова любого метода, определенного в нумерованном типе. С ее же помощью можно обращаться к переменным экземпляра. Ниже приведен вариант нумерованного типа `Transport`, иллюстрирующий использование конструктора, переменной экземпляра и метода. Благодаря ним появляется возможность определить приблизительную скорость перемещения.

#### *Листинг 2.42*

```
enum Transport {  
    // Использование инициализационных значений  
    CAR(65), TRUCK(55), AIRPLANE(600), TRAIN(70), BOAT(22);  
    private int speed;  
    // Приблизительная скорость каждого  
    // транспортного средства  
    // Конструктор  
    private Transport(int s) {  
        speed = s;  
    }  
    // определение метода  
    public int getSpeed() {  
        return speed;  
    }  
}  
  
public class EnumDemo {  
    public static void main(String args[]) {  
        Transport tp;  
        // Отображение скорости самолета  
        // Скорость определяется посредством метода getSpeed()  
        System.out.println("Typical speed for an airplane is "  
            + Transport.AIRPLANE.getSpeed()  
            + " miles per hour.\n");  
  
        // Отображение констант типа Transport  
        // и скорости каждого транспортного средства
```



```

        System.out.println("All Transport speeds: ");
        for (Transport t : Transport.values()) {
            System.out.println(t + " typical speed is "
                               + t.getSpeed()
                               + " miles per hour.");
        }
    }
}

```

Данной версии Transport сделаны некоторые дополнения. Во-первых, переменная экземпляра speed, используемая для хранения скорости типа транспортного средства. Во-вторых, к нумерованному типу Transport добавлен конструктор, которому передается значение скорости. И в-третьих, к данному типу добавлен метод getSpeed( ), возвращающий значение переменной speed.

Когда переменная tp объявляется в методе main(), конструктор Transport() вызывается для каждой константы. Параметр, который должен быть передан конструктору, указывается в скобках после имени константы.

CAR(100), TRUCK(80), AIRPLANE(1200), TRAIN(140), BOAT(40);

Переданное число становится значением параметра конструктора Transport() и при выполнении конструктора присваивается переменной speed. Заметьте также, что список нумерованных констант завершается точкой с запятой. Последней в списке указана константа BOAT. Точка с запятой нужна в том случае, если помимо констант в нумерованном типе присутствуют и другие элементы.

Поскольку каждая нумерованная константа содержит собственную копию переменной speed, вы можете выяснить скорость соответствующего транспортного средства, вызвав метод getSpeed(). Например, в методе main() скорость самолета определяется с помощью следующего выражения:

Transport.AIRPLANE.getSpeed()

Скорость каждого транспортного средства определяется путем перебора нумерованных констант в цикле for. Поскольку каждая константа содержит свою копию speed, значения, связанные с разными константами, различаются между собой. Подобный подход позволяет достаточно просто обеспечить выполнение сложных функций, однако он возможен только в том случае, когда нумерованные типы реализованы посредством классов, как это имеет место в языке Java.

В предыдущем примере использовался только один конструктор, но нумерованные типы, как и обычные классы, допускают любое число конструкторов.

### **Задание:**

Создайте список цветов. В конструктор передаются их номера. Метод выводющий все цвета. Метод, который по введенному с консоли номеру будет печатать какой цвет выбран.

## Тема 2.26 Регулярные выражения

**Регулярные выражения** (англ. regular expressions) — формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (символов-джокеров, англ. wildcard characters). По сути это строка-образец (англ. pattern, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска.

Регулярные выражения используются в большом количестве языков программирования.

В **Java** тоже есть пакет, который позволяет работать с ними - **java.util.regex**.

*"Регулярные выражения откроют перед вами возможности, о которых вы, возможно, даже не подозревали. Ежедневно я неоднократно использую их для решения всевозможных задач – и простых, и сложных(и если бы не регулярные выражения, многие простые задачи оказались бы довольно сложными).*

*Конечно, эффектные примеры, открывающие путь к решению серьезных проблем, наглядно демонстрируют достоинства регулярных выражений. Менее очевиден тот факт, что регулярные выражения используются в повседневной работе для решения «неинтересных» задач – «неинтересных» в том смысле, что программисты вряд ли станут обсуждать их с коллегами в курилке, но без решения этих задач вы не сможете нормально работать.*

*Приведу простой пример. Однажды мне потребовалось проверить множество файлов (точнее 70 с лишним файлов с текстом этой книги) и убедиться в том, что в каждом файле строка 'SetSize' встречается ровно столько же раз, как и строка 'ResetSize'. Задача усложнялась тем, что регистр символов при подсчете не учитывался (т. е. строки 'setSIZE' и 'SetSize' считаются эквивалентными). Конечно, просматривать 32 000 строк текста вручную было бы, по меньшей мере, неразумно. Даже использование стандартных команд поиска в редакторе потребует воистину титанических усилий, учитывая количество файлов и возможные различия в регистре символов.*

*На помощь приходят регулярные выражения! Я ввожу всего одну короткую команду, которая проверяет все файлы и выдает всю необходимую информацию. Общие затраты времени – секунд 15 на ввод команды и еще 2 секунды на проверку данных. Потрясающе!" Из книги "Регулярные выражения. Дж. Фридл."*

### 2.26.1 Метасимволы

Основой синтаксиса регулярных выражений является тот факт, что некоторые символы встречающиеся в строке рассматриваются не как

обычные символы, а как имеющие специальное значение (т.н. метасимволы). Именно это решение позволяет работать всему механизму регулярных выражений. Каждый метасимвол имеет свою собственную роль.

*Вот примеры основных метасимволов:*

- ^** - (крышка, циркумфлекс) начало проверяемой строки
- \$** - (доллар) конец проверяемой строки
- .** - (точка) представляет собой сокращенную форму записи для символьного класса, совпадающего с любым символом
- |** - означает «или». Подвыражения, объединенные этим способом, называются альтернативами (alternatives)
- ?** - (знак вопроса) означает, что предшествующий ему символ является необязательным
- +** - обозначает «один или несколько экземпляров непосредственно предшествующего элемента
- \*** - любое количество экземпляров элемента (в том числе и нулевое)
- \\d** - цифровой символ
- \\D** - не цифровой символ
- \\s** - пробельный символ
- \\S** - не пробельный символ
- \\w** - буквенный или цифровой символ или знак подчёркивания
- \\W** - любой символ, кроме буквенного или цифрового символа или знака подчёркивания

## 2.26.2 Квантификаторы

Регулярные выражения предоставляют инструменты позволяющие указать сколько раз может повторяться один или несколько символов. С некоторыми мы уже встречались:

- +** - Одно или более
- \*** - Ноль или более
- ?** - Ноль или одно
- {n}** - Ровно n раз
- {m,n}** - От m до n включительно
- {m,}** - Не менее m
- {,n}** - Не более n

Теперь мы можем полностью понять регулярное выражение :

"^[a-z0-9\_-]{3,15}\$" .

Разберем её по кусочкам:

**^** - начало строки

**[a-z0-9\_-]** - символ который может быть маленькой латинской буквой или цифрой или символом подчёркивания.

{3,15} - предыдущий объект(смотри выше) может повторяться от 3х до 15раз.

### 2.26.3 Классы регулярных выражений в Java

Пакет, который позволяет работать с регулярными выражениями - **java.util.regex**.

В библиотеке регулярных выражений имеется три основных класса: Pattern, Matcher и PatternSyntaxException. (еще есть классы ASCII, MatchResult, UnicodeProp)

#### 1. Class Pattern

Регулярное выражение, которое Вы записываете в строке, должно сначала быть скомпилированным в объект данного класса. После компиляции объект этого класса может быть использован для создания объекта Matcher.

В классе Pattern объявлены следующие методы:

*Pattern compile*(String regex) – возвращает Pattern, который соответствует regex.

*Matcher matcher*(CharSequence input) – возвращает Matcher, с помощью которого можно находить соответствия в строке input.

#### 2. Class Matcher

Объект Matcher анализирует строку, начиная с 0, и ищет соответствие шаблону.

После завершения этого процесса Matcher содержит много информации о найденных (или не найденных) соответствиях в нашей входной строке. Мы можем получить эту информацию, вызывая различные методы нашего объекта Matcher:

`boolean matches()` просто указывает, соответствует ли вся входная последовательность шаблону.

`int start()` указывает значение индекса в строке, где начинается соответствующая шаблону строка.

`int end()` указывает значение индекса в строке, где заканчивается соответствующая шаблону строка плюс единица.

`String group()` - возвращает найденную строку

`String group(int group)` - если у Вас в регулярном выражении были группы, то можно вывести только кусочек строки соответствующей определенной группе.

Итак давайте попробуем разбить строку на слова с помощью регулярных выражений.

*Листинг 2.43*

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```

public class Regex {
    public static final String strRegex = "This is my small example string
which I'm going to use for pattern matching.";
    public static void main(String[] args) {

        Pattern pattern = Pattern.compile("\\w+");
        Matcher matcher = pattern.matcher(strRegex);

        while (matcher.find()) {
            System.out.print("Start index: " + matcher.start());
            System.out.print(" End index: " + matcher.end() + " ");
            System.out.println(matcher.group());
        }
        Pattern replace = Pattern.compile("\\s+");
        Matcher matcher2 = replace.matcher(strRegex);
        System.out.println(matcher2.replaceAll("\t"));
    }
}

```

Создадим строку, которую не сможем изменять в целях безопасности. Создадим паттерн нашей строки с помощью класса `Pattern` и его статического метода `compile`, который в параметр принимает строку с записанным регулярным выражением. В нашем случае оно будет «ловить» строки с буквенными последовательностями верхнего и нижнего регистра и знака нижнего подчеркивания. Матчер же с помощью одноимённого метода найдёт все совпадения шаблона(паттерна). Метод матчера `find` находит ближайшее совпадение и как видно чуть ниже записывает начальный и конечный индекс найденного шаблона в данной строке. Метод `group` формирует слово со строки по заданному шаблону. Последние 3 строки показывают как мы можем заменить все пробелы в строке на знаки табуляции.

Но тоже самое мы могли бы сделать намного проще с помощью метода `split` для строк. Но с индексами начала и конца было разобраться немного сложнее.

Типичные примеры регулярных выражений нам необходимы для валидации email и ip адресов. Давайте рассмотрим эти примеры.

#### *Листинг 2.44*

```

public class Regex {
    public static final String STR_REGEX = "001.100.05.001 192.168.1.1";
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("(25[0-5]\\.|2[0-4]\\d\\.|1\\d\\d\\.|[1-9]\\d\\.|\\d\\.){3}(25[0-5]|2[0-4]\\d|1\\d\\d|[1-9]\\d\\d)");
        Matcher matcher = pattern.matcher(STR_REGEX);
        while(matcher.find())
            System.out.println(matcher.group());
    }
}

```

```

    }
}

```

Давайте рассмотрим что у нас записано. Во первых представим задачу у себя в мозгу: айпишник состоит из четырёх групп 0-255 разделенных точками. Тогда определим для себя какой вид группы является валидным. Если число однозначное то это 1-9(а не 001-009, или 01-09) и точка – “\d\.”. Но если мы поставим это выражение первым, то будем принимать как влидную группу и 001. Если двузначное то это 10-99(а не 010-099) – “[1-9]\d\.”. Теперь можно составить общее регулярное выражение которое будет считать валидной группу 10-99. или 1-9. – “([1-9]\d\.)|(\d\.)”. Символ «|» означает или слева или справа. А это означает что нам необходимо идти от большего к меньшему. Поэтому и в дальнейшем новые комбинации мы будем добавлять в начало, а не в конец. Продолжим рассматривать случаи: это 100-199, 200-249, 250-255. Итого получим: “((25[0-5]\.)(2[0-4]\d\.)|(1\d\d\.)|([1-9]\d\.)|(\d\.))” . Таких групп должно быть 3 – {3}. Последнюю группу повторяем из первых трёх, только без знаков точки на конце.

#### 2.26.4 Валидация емейла.

```

Pattern pattern =Pattern.compile("[a-zA-Z]{1}[a-zA-Z\d\u002E\u005F]
+@([a-zA-Z]+\u002E){1,2}((net)|(com)|(org))");

```

Последовательность вида [a-zA-Z] указывает на множество. {n} говорит о том, что некоторый символ должен встретиться n раз, а {n,m} - от n до m раз. Символ \d указывает на множество цифр. “\u002E” и “\u005F” - это символы точки и подчеркивания соответственно. Знак плюс после некоторой последовательности говорит о том, что она должна встретиться один или более раз. “|” - представление логического “или”. Полное описание всех конструкций можно найти в Java API. В нашем примере под Pattern будут подходить те e-mail адреса, которые начинаются с буквы, содержат буквы, цифры, точку и подчеркивание до символа “@” и находятся в доменах com, net, org (не более третьего уровня).

## **Выводы к главе:**

- Любая Java программа это один класс или несколько классов.
- Класс является шаблоном для создания объектов класса.
- Класс – это ваш собственный тип данных.
- В классе содержатся поля и методы.
- Поля – это переменные, которые хранят какие-то характеристики объектов.
- Методы – это функции, которые выполняет объект.
- При создании объекта класса вызывается конструктор.
- Конструктор – это специальный метод класса.
- Имя конструктора совпадает с именем класса.
- У конструктора нет возвращаемого типа.
- В конструктор можно передавать параметры.
- В классе может быть несколько конструкторов (перегрузка конструктора), они должны отличаться количеством или типами параметров.
- Если ни один конструктор не описан, то используется конструктор по умолчанию.
- Если хоть один конструктор описан, то конструктор по умолчанию нет.
- Перегружать можно не только конструктор, но и другие методы. Они должны отличаться типом или количеством параметров.
- Обращение к полям осуществляется через имя объекта.
- Вызов метода осуществляется через имя объекта.
- Если поле или метод объявить как *static*, то его можно вызвать без создания объекта(через имя класса).
- Если поле объявить как *final* , то значение этого поля изменить нельзя.

### Задания к главе:

1). Создать классы, спецификации которых приведены ниже. Определить конструкторы и методы `setТип()`, `getТип()`, `toString()`. Определить дополнительно методы в классе, создающем массив объектов. Задать критерий выбора данных и вывести эти данные на консоль.

1.1). Student: id, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон, Факультет, Курс, Группа.

Создать массив объектов. Вывести:

- a) список студентов заданного факультета;
- b) списки студентов для каждого факультета и курса;
- c) список студентов, родившихся после заданного года;
- d) список учебной группы.

1.2). Patient: id, Фамилия, Имя, Отчество, Адрес, Телефон, Номер медицинской карты, Диагноз.

Создать массив объектов. Вывести:

- a) список пациентов, имеющих данный диагноз;
- b) список пациентов, номер медицинской карты у которых находится в заданном интервале.

1.3). Abiturient: id, Фамилия, Имя, Отчество, Адрес, Телефон, Оценки.

Создать массив объектов. Вывести:

- a) список абитуриентов, имеющих неудовлетворительные оценки;
- b) список абитуриентов, средний балл у которых выше заданного;
- c) выбрать заданное число  $n$  абитуриентов, имеющих самый высокий средний балл (вывести также полный список абитуриентов, имеющих полупроходной балл).

1.4). House: id, Номер квартиры, Площадь, Этаж, Количество комнат, Улица, Тип здания, Срок эксплуатации.

Создать массив объектов. Вывести:

- a) список квартир, имеющих заданное число комнат;
- b) список квартир, имеющих заданное число комнат и расположенных на этаже, который находится в заданном промежутке;
- c) список квартир, имеющих площадь, превосходящую заданную.

1.5). Car: id, Марка, Модель, Год выпуска, Цвет, Цена, Регистрационный номер.

Создать массив объектов. Вывести:

- a) список автомобилей заданной марки;
- b) список автомобилей заданной модели, которые эксплуатируются больше  $n$  лет;
- c) список автомобилей заданного года выпуска, цена которых больше указанной.

2). Разработать класс Круг, имеющий три поля. Одно поле будет хранить значение радиуса. Два других координаты центра. Конструктор без параметров, конструктор с 1 параметром – радиус, конструктор с двумя



параметрами – координаты центра, конструктор с 3 параметрами – все три поля. Написать метод выводющий все характеристики круга. Написать метод изменяющий координаты центра(передаются параметры указывающие на сколько нужно изменить координаты центра). Написать метод для изменения радиуса круга. Написать метод для расчета площади круга и метод для расчета длины окружности.

3). Разработать класс Склад. Два поля: количество единиц товара и стоимость 1 единицы. Конструктор пустой и конструктор с двумя параметрами. Написать метод позволяющий изменять количество товара. Написать метод позволяющий изменять стоимость товара. Написать метод позволяющий рассчитывать стоимость товара. Написать метод для сравнения стоимости товаров. Написать метод с переменным числом параметров определяющий общее количество товаров.

4). Разработать класс Книга. Поля – автор, название, год выпуска, количество страниц. Конструктор пустой и конструктор с 4 параметрами. Написать методы, позволяющие менять каждое из полей. Метод, который по названию книги, будет выводить всю информацию о книге. Перегрузить методы по изменению полей, так чтобы новое значение поля можно было вводить с клавиатуры.

5). Определить класс Дробь в виде пары  $(m,n)$ . Класс должен содержать несколько конструкторов. Реализовать методы для сложения, вычитания, умножения, деления и сокращения дробей. Методы сумма и произведение сделать с переменным числом параметров. Объявить массив из  $k$  дробей, ввести/вывести значения для массива дробей. Создать массив объектов и передать его в метод, который изменяет каждый элемент массива с четным индексом путем добавления следующего за ним элемента массива.

6). Определить класс Вектор размерности  $n$ . Реализовать методы сложения, вычитания, умножения, инкремента, декремента, индексирования. Определить массив из  $m$  объектов. Каждую из пар векторов передать в методы, возвращающие их скалярное произведение и длины. Вычислить и вывести углы между векторами.

7). Определить класс Множество символов мощности  $n$ . Написать несколько конструкторов. Реализовать методы для определения принадлежности заданного элемента множеству; пересечения, объединения, разности двух множеств. Создать методы сложения, вычитания, умножения (пересечения), индексирования, присваивания. Создать массив объектов и передавать пары объектов в метод другого класса, который строит множество, состоящее из элементов, входящих только в одно из заданных множеств.

8). Определить класс Квадратное уравнение. Класс должен содержать несколько конструкторов. Реализовать методы для поиска корней, экстремумов, а также интервалов убывания/возрастания. Создать массив объектов и определить наибольшие и наименьшие по значению корни.

## Глава 3 НАСЛЕДОВАНИЕ И ИНТЕРФЕЙСЫ

Наследование — один из трех базовых принципов объектно-ориентированного программирования. Благодаря ему появляется возможность создавать иерархические классы объектов. С помощью наследования можно сформировать общий класс, определяющий характерные особенности некоторого понятия. Данный класс наследуется остальными, более конкретными классами, каждый из которых уточняет это понятие и дополняет его уникальными характеристиками.

В языке Java наследуемый класс принято называть суперклассом. Его дочерние классы называются подклассами. Таким образом, подкласс — это специализированная версия суперкласса. Он наследует все переменные и методы, определенные в суперклассе, и дополняет их своими элементами.

### Тема 3.1 Основы наследования

Тот факт, что один класс наследует другой, в языке Java отражается в объявлении класса. Для этой цели служит ключевое слово `extends`. Подкласс дополняет характеристики суперкласса (расширяет его).

Рассмотрим простой пример, иллюстрирующий некоторые свойства наследования. Ниже приведен код программы, в которой определен суперкласс `TwoDShape`, хранящий сведения о ширине и высоте двумерного объекта. Там же определен и его подкласс `Triangle`. Обратите внимание на то, что в определении подкласса присутствует ключевое слово `extends`. В листинге 3.1 показана простая иерархия классов.

#### *Листинг 3.1*

```
// Простая иерархия классов
// Класс, описывающий двумерные объекты
public class TwoDShape {
    public double width;
    public double height;
    public void showDim() {
        System.out.println("Width and height are "
            + width + " and " + height);
    }
}

// Подкласс класса TwoDShape для представления треугольников.
// Класс Triangle наследует TwoDShape
public class Triangle extends TwoDShape {
    String style;
    public double area() {
// Из класса Triangle можно обращаться к полям и методам
// класса TwoDShape так же, как и к собственным элементам
    }
}
```

```

        return width * height / 2;
    }
    public void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

public class Shapes {
    public static void main(String args[]) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();
        // Все члены Triangle, даже унаследованные от TwoDShape,
        // доступны посредством объекта Triangle
        t1.width = 4.0;
        t1.height = 4.0;
        t1.style = "isosceles";
        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = "right";
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());
        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}

```

Здесь в классе TwoDShape определены атрибуты "универсальной" двумерной фигуры, конкретным представителем которой может быть квадрат, треугольник прямоугольник и т.д. Класс Triangle представляет конкретный тип объект TwoDShape, в данном случае треугольник. Класс Triangle включает все элементы класса TwoDObject и в дополнение к ним – поле style и методы area() и showStyle(). Описание треугольника хранится в переменной style, метод area() вычисляет и возвращает площадь треугольника, а метод showStyle() отображает описание.

Поскольку класс Triangle содержит все члены суперкласса TwoDShape() в теле метода area() доступны переменные width и height. Кроме того в теле метода main () можно с помощью объектов t1 и t2 непосредственно обращаться к переменным width и height так, как будто они принадлежат классу Triangle. На рисунке 3.1 условно показано, каким образом суперкласс TwoDShape включается в состав класса Triangle.

TwoDShape	widht	Triangle
	height	
	showDim()	
	style	
	area()	
	showStyle()	

Рисунок 3.1 – Условное представление класса Triangle

Даже несмотря на то, что TwoDShape является суперклассом класса Triangle, он все-таки остается независимым классом. Тот факт, что класс является суперклассом другого класса, не означает, что он не может быть использован сам по себе. Например, следующий фрагмент кода составлен корректно:

```
TwoDShape shape = new TwoDShape();
shape.width = 10;
shape.height = 20;
shape.showDim();
```

Класс TwoDShape не имеет сведений о своих подклассах и не может обратиться к ним.

Общий формат определения класса, наследующего суперкласс, приведен ниже:

```
class имя_класса extends имя_суперкласса {
    // тело класса
}
```

Для каждого создаваемого класса можно указать только один суперкласс. Множественное наследование в Java не поддерживается, т.е. подкласс не может иметь несколько суперклассов. (язык Java отличается от языка C++, где разработчик может создать класс, дочерний по отношению сразу к нескольким классам. Это надо помнить, преобразуя код C++ в Java.) С другой стороны, многоуровневая иерархия, По которой подкласс выступает в то же время как суперкласс другого класса, вполне возможна. И конечно же, класс не может быть суперклассом для самого себя.

Главное преимущество наследования состоит в том, что суперкласс, содержащий определения атрибутов, общих для набора объектов, может быть использован для создания любого количества более конкретных подклассов. Каждый подкласс уточняет понятие, представляемое своим суперклассом. В качестве примера рассмотрим еще один подкласс класса TwoDShape, который инкапсулирует прямоугольники(листинг 3.2):

*Листинг 3.2*

```
// Подкласс класса TwoDShape, представляющий прямоугольник
public class Rectangle extends TwoDShape {
    public boolean isSquare() {
        if (width == height) {
```

```

        return true;
    }
    return false;
}
public double area() {
    return width * height;
}
}

```

Класс `Rectangle` включает элементы класса `TwoDShape` и, кроме того, содержит метод `isSquare()`, определяющий, является ли прямоугольник квадратом, и метод `area()`, вычисляющий площадь прямоугольника.

### **Задание:**

Разработать класс Животные. Поля—вес, возраст, имя. Методы — для изменения полей, для вывода всех полей.

Класс Кот наследуется от Животных. Поле — количество пойманных мышей. Методы — для изменения поля, для вывода всех полей.

## **Тема 3.2 Наследование и доступ к членам класса**

Как вы знаете, часто переменные экземпляра объявляют как `private`, чтобы предотвратить их некорректное использование. Наследование класса никак не влияет на ограничения, накладываемые модификатором доступа `private`. Несмотря на то, что подкласс содержит все члены суперкласса, он не может обращаться к закрытым классам и методам. Рассмотрим в качестве примера следующий фрагмент кода. Если в классе `TwoDShape` переменные `width`, `right` объявлены как `private`, то класс `Triangle` не имеет к ним доступа(листинг 3.3).

### *Листинг 3.3*

```

// Закрытые члены не наследуются
// Данная программа не будет компилироваться
// Класс, описывающий двумерные объекты
public class TwoDShape {
    private double width; // Теперь эти переменные
    private double height; // объявлены как private
    public void showDim() {
        System.out.println("Width and height are " +
            width + " and " + height);
    }
}
// Подкласс TwoDShape, представляющий треугольники
public class Triangle extends TwoDShape {
    public String style;
    public double area() {
        return width * height / 2; // Ошибка! Доступ запрещен
    }
}

```

```

    }
    public void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

```

Класс Triangle не будет скомпилирован, поскольку ссылки на width и height в теле класса area () нарушают правила доступа. Поскольку эти переменные объявлены как private, они доступны только членам собственного класса. Подклассам обращаться к ним запрещено. Как вы помните, член класса, объявленный как private, недоступен из-за пределов класса. Это ограничение распространяется и на подклассы.

Поначалу может показаться, что отсутствие доступа к закрытым членам суперкласса — серьезный недостаток, затрудняющий программирование. Однако это не так. Как было сказано в модуле 6, в программах на Java для обеспечения доступа к закрытым членам класса обычно используются специальные методы. Ниже показаны модифицированные классы TwoDShape и Triangle, в которых обращения к переменным width и height производится посредством специальных методов(листинг 3.4)

#### *Листинг 3.4*

```

// Использование методов доступа для установки и
// получения значений закрытых переменных
// Класс, описывающий двумерные объекты
public class TwoDShape {
    private double width; // Теперь эти переменные
    private double height; // объявлены как private
    // Методы доступа для переменных width и height
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
    public void setWidth(double w) {
        width = w;
    }
    public void setHeight(double h) {
        height = h;
    }
    public void showDim() {
        System.out.println("Width and height are "
            + width + " and " + height);
    }
}

```

```
// Подкласс класса TwoDShape, представляющий треугольники
public class Triangle extends TwoDShape {
    String style;
    public double area() {
        return getWidth() * getHeight() / 2;
    }
    public void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

public class Shapes2 {
    public static void main(String args[]) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();
        t1.setWidth(4.0);
        t1.setHeight(4.0);
        t1.style = "isosceles";
        t2.setWidth(8.0);
        t2.setHeight(12.0);
        t2.style = "right";
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());
        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}
```

Не существует четких правил, позволяющих принять безошибочное решение по данному вопросу. Следует лишь придерживаться двух общих принципов. Если переменная экземпляра используется только методами определенными в классе, то она должна быть закрытой. Если значение переменной экземпляра не должно выходить за определенные границы, то следует объявить эту переменную как `private`, а доступ к ней организовать с помощью специальных методов. Таким образом, вы предотвратите присвоение переменной недопустимых значений.

### **Задание:**

Измените в классе Животные все поля на `private`, внесите изменения в методы.

### Тема 3.3 Конструкторы и наследование

В иерархии наследования и суперклассы, и подклассы могут иметь конструкторы. В связи с этим возникает важный вопрос: какой из конструкторов отвечать за формирование экземпляра класса: конструктор этого класса, конструктор суперкласса или они оба? Ответ звучит так: конструктор суперкласса формирует часть объекта, соответствующую суперклассу, а конструктор подкласса – остальные части. Данное решение вполне оправдано, поскольку суперкласс не имеет сведений об элементах подкласса и не может обращаться к ним. Таким образом конструкторы должны работать независимо друг от друга. В примерах, рассмотренных выше, использовались конструкторы по умолчанию, автоматически создаваемые исполняющей системой Java, и их действия были скрыты от пользователя. Однако на практике для большинства классов конструкторы определяются явно. Сейчас мы рассмотрим, как следует поступать в подобных ситуациях.

Когда конструктор определен только в подклассе, его использование не вызывает вопросов: надо лишь обычным образом сформировать объект. При этом часть объекта, соответствующая суперклассу, будет создана автоматически с использованием конструктора по умолчанию. В качестве примера рассмотрим модифицированную версию класса `Triangle`, в котором определен конструктор, поскольку переменная `style` устанавливается конструктором, она объявлена как `private` (листинг 3.5).

#### *Листинг 3.5*

```
// Добавление конструктора к классу Triangle
// Класс, описывающий двумерные объекты
public class TwoDShape {
    private double width; // Теперь эти переменные
    private double height; // объявлены как private
    // Методы доступа к переменным width и height
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
    public void setWidth(double w) {
        width = w;
    }
    public void setHeight(double h) {
        height = h;
    }
    public void showDim() {
        System.out.println("Width and height are "
            + width + " and " + height);
    }
}
```



```

    }
}

// Подкласс класса TwoDShape, представляющий треугольники
public class Triangle extends TwoDShape {
    private String style;
    // Конструктор
    public Triangle(String s, double w, double h) {
        // Инициализация части объекта,
        // соответствующей классу TwoDShape
        setWidth(w);
        setHeight(h);
        style = s;
    }
    public double area() {
        return getWidth() * getHeight() / 2;
    }
    public void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

public class Shapes3 {
    public static void main(String args[]) {
        Triangle t1 = new Triangle("isosceles", 4.0, 4.0);
        Triangle t2 = new Triangle("right", 8.0, 12.0);
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());
        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}

```

Здесь конструктор класса Triangle, помимо поля style, инициализирует также унаследованные члены класса TwoDClass.

Если конструктор объявлен и в подклассе, и в суперклассе, то процесс их использования несколько сложнее. Это связано с тем, что оба конструктора должны получить управление. В таком случае приходит на помощь ключевое слово `super`, которое используется в двух формах. С помощью первой формы вызывается конструктор суперкласса. Вторая форма

используется для доступа к членам суперкласса, маскируемым членами подкласса. Рассмотрим первое применение ключевого слова `super`.

**Задание:**

Добавьте конструктор в класс `Кот`

### **Тема 3.4 Использование ключевого слова `super` для вызова конструктора суперкласса**

Для вызова конструктора суперкласса применяется выражение следующего типа:

*`super (список_параметров);`*

где список параметров содержит параметры, необходимые для работы конструктора суперкласса. Вызов `super ()` должен быть первым выражением в теле конструктора подкласса. Для того чтобы лучше понять особенности вызова `super`.

Рассмотрим вариант класса `TwoDShape` из следующей программы, в котором определен конструктор, инициализирующий переменные `width` и `height` (листинг 3.6).

*Листинг 3.6*

*// Добавление конструкторов к классу `TwoDShape`*

```
public class TwoDShape {
    private double width;
    private double height;
    // Конструктор с параметрами
    public TwoDShape(double w, double h) {
        width = w;
        height = h;
    }
    // методы доступа к переменным width и height
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
    public void setWidth(double w) {
        width = w;
    }
    public void setHeight(double h) {
        height = h;
    }
    public void showDim() {
        System.out.println("Width and height are ")
    }
}
```

```

        + width + " and " + height);
    }
}

// Подкласс класса TwoDShape, представляющий треугольники
public class Triangle extends TwoDShape {
    private String style;
    public Triangle(String s, double w, double h) {
// Использование выражения super() для вызова
// конструктора TwoDShape
        super(w, h); // Вызов конструктора суперкласса
        style = s;
    }
    public double area() {
        return getWidth() * getHeight() / 2;
    }
    public void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

public class Shapes4 {
    public static void main(String args[]) {
        Triangle t1 = new Triangle("isosceles", 4.0, 4.0);
        Triangle t2 = new Triangle("right", 8.0, 12.0);
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());
        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}

```

В конструкторе Triangle присутствует вызов super () с параметрами w и h. В результате управление получает конструктор TwoDShape(), инициализирующий переменные width и height значениями, переданными в качестве параметров. Теперь класс Triangle уже не занимается инициализацией элементов суперкласса. Он должен инициализировать только собственную переменную style. Конструктору TwoDShape() предоставляется возможность сформировать соответствующий подобъект так, как предусмотрено в данном классе. Более того, разработчик может

реализовать в TwoDShape функциональность, о которой не будут знать существующие подклассы. Это делает код более устойчивым к ошибкам.

Любой конструктор суперкласса вызывается посредством ключевого слова `super`. При этом конкретный вариант конструктора выбирается по соответствию параметров. Например, ниже приведена расширенная версия классов TwoDShape и Triangle, содержащих конструктор по умолчанию и конструктор, получающий одно значение (листинг 3.7).

*Листинг 3.7*

/ Дальнейшее добавление конструкторов к классу TwoDShape

```
public class TwoDShape {
    private double width;
    private double height;
    // Конструктор по умолчанию
    public TwoDShape() {
        width = height = 0.0;
    }
    // Конструктор с параметрами
    public TwoDShape(double w, double h) {
        width = w;
        height = h;
    }
    // Конструирование объекта с одинаковыми
    // значениями width и height
    public TwoDShape(double x) {
        width = height = x;
    }
    // Методы доступа к переменным width и height
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
    public void setWidth(double w) {
        width = w;
    }
    public void setHeight(double h) {
        height = h;
    }
    public void showDim() {
        System.out.println("Width and height are "
            + width + " and " + height);
    }
}
// Подкласс класса TwoDShape, представляющий треугольники
```

```

public class Triangle extends TwoDShape {
    private String style;
    // Конструктор по умолчанию
    public Triangle() {
        // Использование выражения super() для обращения
        // к различным вариантам конструктора TwoDShape()
        super();
        style = "null";
    }
    // Конструктор
    public Triangle(String s, double w, double h) {
        // Использование выражения super() для обращения
        // к различным вариантам конструктора TwoDShape()
        super(w, h); // Вызов конструктора суперкласса
        style = s;
    }
    // Конструктор, формирующий равнобедренный треугольник
    public Triangle(double x) {
        // Использование выражения super() для обращения
        // к различным вариантам конструктора TwoDShape()
        super(x); // Вызов конструктора суперкласса
        style = "isosceles";
    }
    public double area() {
        return getWidth() * getHeight() / 2;
    }
    public void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

public class Shapes5 {
    public static void main(String args[]) {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle("right", 8.0, 12.0);
        Triangle t3 = new Triangle(4.0);
        t1 = t2;
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());
        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
    }
}

```

```

        System.out.println("Area is " + t2.area());
        System.out.println();
        System.out.println("Info for t3: ");
        t3.showStyle();
        t3.showDim();
        System.out.println("Area is " + t3.area());
        System.out.println();
    }
}

```

Давайте еще раз вспомним основные свойства вызова `super()`. Когда данный вызов присутствует в конструкторе подкласса, управление получает конструктор его непосредственного суперкласса. Таким образом, вызывается конструктор того класса, который непосредственно породил вызывающий класс. Это справедливо и при многоуровневой иерархии. Кроме того, вызов `super()` должен быть первым выражением в теле конструктора подкласса.

#### **Задание:**

Добавьте конструктор в класс Животные и измените в классе Кот

### **Тема 3.5 Использование ключевого слова `super` для доступа к членам суперкласса**

Существует еще одна форма ключевого слова `super`, которая применяется подобно `this`, но ссылается на суперкласс класса. Формат обращения к члену суперкласса имеет следующий вид:

*super.член\_класса*

где в качестве члена класса может выступать либо метод, либо переменная экземпляра.

Данная форма `super` применяется тогда, когда член подкласса маскирует член суперкласса. Рассмотрим следующую несложную иерархию классов(листинг 3.8):

*Листинг 3.8*

```

// Использование ключевого слова super
// для предотвращения маскировки имен
public class A {
    public int i;
}
// Создание подкласса, расширяющего класс A
public class B extends A {
    public int i; // Данная переменная i маскирует
// переменную i в составе класса A
    public B(int a, int b) {
// Выражение super.i ссылается на переменную i в классе A

```

```

        super.i = a; // i в классе A
        i = b; // i в классе B
    }
    public void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}

class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);
        subOb.show();
    }
}

```

Несмотря на то что переменная экземпляра *i* в классе *B* маскирует одноименную переменную в классе *A*, ключевое слово *super* позволяет обращаться к переменной в составе суперкласса. Точно так же данное ключевое слово можно использовать для вызова методов, маскируемых методами подкласса.

### Тема 3.6 Многоуровневая иерархия

До сих пор мы имели дело с простыми иерархиями классов, состоящими только из суперкласса и подкласса. Однако можно создать иерархическую структуру, насчитывающую как угодно много уровней наследования. Как было сказано ранее, подкласс может выступать в роли суперкласса для другого класса. Так, например, среди классов *A*, *B* и *C* класс *C* может быть подклассом класса *B*, а он, в свою очередь, – подклассом класса *A*. В подобных ситуациях каждый подкласс наследует характеристики всех суперклассов. В частности, класс *C* наследует элементы *B* и *A*.

Для того чтобы лучше понять использование многоуровневой иерархии, рассмотрим следующую программу. В ней подкласс *Triangle* выступает в роли суперкласса для класса *ColorTriangle*. Класс *ColorTriangle* наследует все свойства *Triangle* и *TwoDShape*, и, кроме того, в нем имеется поле *color*, определяющее цвет треугольника (листинг 3.9).

#### *Листинг 3.9*

```

// Многоуровневая иерархия
public class TwoDShape {
    private double width;
    private double height;
    // Конструктор по умолчанию
    public TwoDShape() {
        width = height = 0.0;
    }
}

```

```

    }
    // Конструктор с параметрами
    public TwoDShape(double w, double h) {
        width = w;
        height = h;
    }
    // Формирование объекта с одинаковыми
    // значениями width и height
    public TwoDShape(double x) {
        width = height = x;
    }
    // Методы доступа к переменным width и height
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
    public void setWidth(double w) {
        width = w;
    }
    public void setHeight(double h) {
        height = h;
    }
    public void showDim() {
        System.out.println("Width and height are "
            + width + " and " + height);
    }
}

// Подкласс класса TwoDShape
public class Triangle extends TwoDShape {
    private String style;
    // Конструктор по умолчанию
    public Triangle() {
        super();
        style = "null";
    }
    public Triangle(String s, double w, double h) {
        super(w, h); // Вызов конструктора суперкласса
        style = s;
    }
    // Формирование равнобедренного треугольника
    public Triangle(double x) {
        super(x); // Вызов конструктора суперкласса
    }
}

```



```

        style = "isosceles";
    }
    public double area() {
        return getWidth() * getHeight() / 2;
    }
    public void showStyle() {
        System.out.println("Triangle is " + style);
    }
}

```

// Подкласс класса Triangle.  
 // Класс ColorTriangle является подклассом Triangle  
 // который, в свою очередь, расширяет класс TwoDShape.  
 // Следовательно, ColorTriangle содержит переменные  
 // и методы как класса Triangle, так и класса TwoDShape.

```

public class ColorTriangle extends Triangle {
    private String color;
    public ColorTriangle(String c, String s,
        double w, double h) {
        super(s, w, h);
        color = c;
    }
    public String getColor() {
        return color;
    }
    public void showColor() {
        System.out.println("Color is " + color);
    }
}

```

```

public class Shapes6 {
    public static void main(String args[]) {
        ColorTriangle t1 = new ColorTriangle("Blue", "right", 8.0, 12.0);
        ColorTriangle t2 = new ColorTriangle("Red", "isosceles", 2.0, 2.0);
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        t1.showColor();
        System.out.println("Area is " + t1.area());
        System.out.println();
        System.out.println("Info for t2: ");
        // Из объекта ColorTriangle можно вызывать методы,
        // принадлежащие как ему самому, так и его суперклассам
        t2.showStyle();
        t2.showDim();
    }
}

```

```

        t2.showColor();
        System.out.println("Area is " + t2.area());
    }
}

```

Благодаря наследованию `ColorTriangle` может использовать ранее определенные классы `Triangle` и `TwoDShape`, добавляя лишь информацию, специфическую для той задачи, для которой создавался `ColorTriangle`. Таким образом, наследование способствует повторному использованию кода.

Данный пример также иллюстрирует еще одну важную деталь: выражение `super()` всегда ссылается на конструктор ближайшего суперкласса. Другими словами, выражение `super()` в `ColorTriangle` означает вызов конструктора `Triangle`. В классе же `Triangle` вызов `super()` относится к конструктору `TwoDShape`. Если в иерархии классов для конструктора суперкласса предусмотрены параметры, то все суперклассы должны передавать их вверх по иерархической структуре. Это правило не зависит от того, нужны ли параметры самому подклассу.

#### **Задание:**

Добавьте класс `Котенок` (потомок класса `Кот`) с полем `время кормления`. Напишите конструктор.

### **Тема 3.7 Когда вызываются конструкторы**

В разговоре о наследовании и иерархии классов может возникнуть очень важный вопрос, какой конструктор выполняется первым при создании экземпляра подкласса: тот, который принадлежит подклассу, или тот, который определен в суперклассе? Пусть, например, у нас есть подкласс `В` и суперкласс `А`. Будет ли конструктор `А` вызван перед конструктором `В` или наоборот? Ответ звучит так: в иерархии классов конструкторы вызываются вниз по структуре наследования – от суперклассов к подклассам. Более того, поскольку выражение `super()` должно быть расположено первым в конструкторе подкласса, порядок, в котором конструкторы получают управление, остается неизменным, независимо от того, используется ли вызов `super()`. Если выражение `super()` отсутствует, то выполняется конструктор каждого суперкласса по умолчанию (т.е. конструктор, не имеющий параметров). Листинг 3.10 демонстрирует вызов конструкторов.

#### *Листинг 3.10*

```

// Демонстрация порядка вызова конструкторов
// Создание суперкласса
public class A {
    public A() {
        System.out.println("Constructing A.");
    }
}

```

```
// Создание подкласса, расширяющего класс A
public class B extends A {
    public B() {
        System.out.println("Constructing B.");
    }
}
// Создание подкласса, расширяющего класс B
public class C extends B {
    public C() {
        System.out.println("Constructing C.");
    }
}
public class OrderOfConstruction {
    public static void main(String args[]) {
        C c = new C();
    }
}
```

Как видите, конструкторы вызываются в том же порядке, в котором происходит наследование.

Если подумать, то станет ясно, почему выбран именно такой порядок вызова конструкторов. Поскольку суперкласс не имеет сведений ни об одном подклассе, то его инициализация должна выполняться перед инициализацией подклассов. Следовательно, конструктор суперкласса должен вызываться первым.

### **Тема 3.8 Объекты подклассов и ссылки на суперклассы**

Как вы уже знаете, Java – строго типизированный язык. Соответствие типов соблюдается всегда, за исключением автоматического преобразования и тех случаев, когда используется явное приведение типов. Таким образом, переменная, в качестве типа которой указано имя класса, не может ссылаться на экземпляра другого класса. Рассмотрим в качестве примера листинг 3.11.

*Листинг 3.11*

```
// Этот код не будет скомпилирован
public class X {
    private int a;
    public X(int i) {
        a = i;
    }
}
public class Y {
    private int a;
    public Y(int i) {
        a = i;
    }
}
```

```

    }
}
public class IncompatibleRef {
    public static void main(String args[]) {
        X x = new X(10);
        X x2;
        Y y = new Y(5);
        x2 = x; // Допустимо; обе переменные имеют
// один и тот же тип
        x2 = y; // ошибка; переменные разных типов
    }
}

```

Несмотря на то что классы X и Y имеют одинаковые элементы, невозможно присвоить переменной типа X ссылку на объект Y, так как типы объектов различаются. Объектная ссылка может указывать только на один тип объектов.

Существует, однако, важное исключение из этого правила: *переменная, типом которой является суперкласс, может получать ссылку на переменную, типом которой указан любой подкласс этого класса.* Это иллюстрирует листинг 3.12:

*Листинг 3.12*

*// Ссылка на суперкласс может указывать*

*// на экземпляр подкласса*

```

public class X {
    private int a;
    public X(int i) {
        a = i;
    }
    public int getA() {
        return a;
    }
    public void setA(int a) {
        this.a = a;
    }
}
public class Y extends X {
    private int b;
    public Y(int i, int j) {
        super(j);
        b = i;
    }
    public int getB() {
        return b;
    }
    public void setb(int b) {

```

```

        this.b = b;
    }
}
public class SupSubRef {
    public static void main(String args[]) {
        X x = new X(10);
        X x2;
        Y y = new Y(5, 6);
        x2 = x; // Допустимо; переменные одного типа OK, both of same type
        System.out.println("x2.a: " + x2.getA());
        // Поскольку Y является подклассом X, то переменные
        // x2 и y могут ссылаться на один и тот же объект
        x2 = y; // Допустимо
        System.out.println("x2.a: " + x2.getA());
        // Ссылка на X имеет сведения только о членах класса X
        x2.setA(19); // Допустимо
        // x2.b = 27; // Ошибка; b не является членом класса X
    }
}

```

Здесь Y является подклассом X, следовательно, переменной x2 можно присвоить ссылку на объект Y. Однако важно понимать, что в случае, когда переменная суперкласса ссылается на объект подкласса, с ее помощью доступны не переменные и методы подкласса, а лишь члены суперкласса. Таким образом, присваивая ссылку на экземпляр подкласса переменной суперкласса, вы сужаете доступ к элементам объекта, ограничивая его лишь элементами, унаследованными от суперкласса. Именно поэтому, несмотря на то, что переменная x2 ссылается на объект Y, она не позволяет обращаться к переменной b. Это вполне оправдано; ведь суперкласс не имеет сведений о том, какие подклассы порождены из него. Именно поэтому последняя строка кода в программе закомментирована.

Приведенные выше рассуждения могут показаться отвлеченными, однако они имеют важные практические применения. Одно из них мы рассмотрим сейчас.

Процедура присвоения ссылок на подкласс переменным суперкласса важна при вызове конструкторов в иерархической структуре классов. Как вы знаете, в составе классов часто присутствуют конструкторы, которым в качестве параметра передается экземпляр класса. Это позволяет формировать копии объектов. Такая возможность часто используется и в подклассах. Рассмотрим, например приведенные ниже варианты классов TwoDShape и Triangle. В обоих есть конструкторы, параметрами которых являются объекты.

### *Листинг 3.13*

```

public class TwoDShape {
    private double width;
    private double height;
}

```

```

// Конструктор по умолчанию
public TwoDShape() {
    width = height = 0.0;
}
// Конструктор с параметрами
public TwoDShape(double w, double h) {
    width = w;
    height = h;
}
// Формирование объекта с одинаковыми значениями width и height
public TwoDShape(double x) {
    width = height = x;
}
// Формирование объекта на основе другого объекта
public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
}
// Методы для доступа к переменным width и height
public double getWidth() {
    return width;
}
public double getHeight() {
    return height;
}
public void setWidth(double w) {
    width = w;
}
public void setHeight(double h) {
    height = h;
}
public void showDim() {
    System.out.println("Width and height are "
        + width + " and " + height);
}
}
// Подкласс класса TwoDShape, представляющий треугольники
public class Triangle extends TwoDShape {
    private String style;
// Конструктор по умолчанию

    public Triangle() {
        super();
        style = "null";
    }
}

```

```

// Конструктор класса Triangle
public Triangle(String s, double w, double h) {
    super(w, h); // Вызов конструктора суперкласса
    style = s;
}
// Формирование равнобедренного треугольника
public Triangle(double x) {
    super(x); // Вызов конструктора суперкласса
    style = "isosceles";
}
// Формирование объекта на основе другого объекта
public Triangle(Triangle ob) {
    super(ob); // Передача ссылки на объект Triangle
// конструктору TwoDShape
    style = ob.style;
}
public double area() {
    return getWidth() * getHeight() / 2;
}
public void showStyle() {
    System.out.println("Triangle is " + style);
}
}

public class Shapes7 {
    public static void main(String args[]) {
        Triangle t1 = new Triangle("right", 8.0, 12.0);
        // Создание копии объекта t1
        Triangle t2 = new Triangle(t1);
        System.out.println("Info for t1: ");
        t1.showStyle();
        t1.showDim();
        System.out.println("Area is " + t1.area());
        System.out.println();
        System.out.println("Info for t2: ");
        t2.showStyle();
        t2.showDim();
        System.out.println("Area is " + t2.area());
    }
}
// Формирование объекта на основе другого объекта
public Triangle(Triangle ob) {
    super(ob); // Передача ссылки на объект Triangle
// конструктору TwoDShape
    style = ob.style;
}

```

```
}
```

В качестве параметра данному конструктору передается объект Triangle, который затем с помощью вызова `super()` передается конструктору `TwoDShape`.

```
// Формирование объекта на основе другого объекта
```

```
public TwoDShape(TwoDShape ob) {
```

```
    width = ob.width;
```

```
    height = ob.height;
```

```
}
```

Заметьте, что конструктор `TwoDShape()` должен получить объект `TwoDShape`, однако `Triangle()` передает ему объект `Triangle`. Тем не менее проблемы не возникают, поскольку переменная суперкласса может быть ссылкой на объект подкласса. Вследствие этого конструктору `TwoDShape()` можно передать ссылку на экземпляр класса, являющегося подклассом `TwoDShape`. Поскольку конструктор `TwoDShape()` инициализирует лишь часть объекта подкласса, а именно члены, унаследованные от класса `TwoDShape`, не важно, имеет ли объект, переданный в качестве параметра, дополнительные переменные и методы.

### **Задание:**

Добавьте во все классы конструкторы, параметрами которых является объект.

## **Тема 3.9 Переопределение методов**

В иерархии классов часто бывает так, что и в суперклассе, и в подклассе метод с одинаковой сигнатурой и одинаковым возвращаемым значением. В этом случае говорят, что метод суперкласса переопределяется в подклассе. Если переопределенный метод вызывается из подкласса, то получает управление тот вариант метода, который был определен в подклассе. Вариант метода, определенный в суперклассе, оказывается замаскированным. Рассмотрим листинг 3.14.

*Листинг 3.14*

```
// Переопределение метода
```

```
public class A {
```

```
    private int i, j;
```

```
    public A(int a, int b) {
```

```
        i = a;
```

```
        j = b;
```

```
    }
```

```
// Отображение i и j
```

```
    public void show() {
```

```
        System.out.println("i and j: " + i + " " + j);
```

```
    }
```



```

}
public class B extends A {
    private int k;
    public B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // Отображение k; данный метод переопределяет
    // метод show() из класса A
    public void show() {
        System.out.println("k: " + k);
    }
}
public class Override {

    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); // Вызов метода show() класса B
    }
}

```

Когда метод `show ()` вызывается посредством ссылки на объект `B`, управление получает вариант этого метода, определенный в классе `B`. Таким образом, метод `show()` в составе класса `B` переопределяет одноименный метод, определенный в классе `A`. Если вы хотите обратиться к исходному варианту переопределенного метода, тому, который определен в суперклассе, вам надо использовать ключевое слово `super`. Например, в приведенном ниже варианте класса `B` из метода `show ()` вызывается вариант того же метода, определенный в суперклассе. При этом отображаются все переменные экземпляра.

### *Листинг 3.15*

```

public class B extends A {
    private int k;
    public B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    public void show() {
        // Использование ключевого слова super для
        // вызова метода show(), определенного в классе A
        super.show();
        System.out.println("k: " + k);
    }
}

```

Если вы замените новой модификацией метода `show()` метод с тем же именем из предыдущей программы, то выходные данные изменятся и будут иметь следующий вид:

i and j: 1 2

k: 3

где `super.show ()` – это вызов метода `show()`, определенного в суперклассе.

Переопределение метода происходит только в том случае, если сигнатуры переопределяемого и переопределяющего метода совпадают. В противном случае происходит обычная перегрузка методов. Рассмотрим приведенную ниже модификацию предыдущего примера (листинг 3.16).

*Листинг 3.16*

// Методы с разными сигнатурами не переопределяются, а перегружаются

```
public class A {
    private int i, j;
    public A(int a, int b) {
        i = a;
        j = b;
    }
    // Отображение i и j
    public void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}
// Создание подкласса, расширяющего класс A
public class B extends A {
    private int k;
    public B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // Поскольку сигнатуры данного метода
    // и метода show() из класса A различаются,
    // вместо переопределения происходит перегрузка
    public void show(String msg) {
        System.out.println(msg + k);
    }
}
public class Overload {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show("This is k: "); // Вызов метода show() из B
        subOb.show(); // Вызов метода show() из A
    }
}
```

}

На этот раз в варианте метода `show()`, принадлежащем классу `B`, предусмотрен строковый параметр. Сигнатура метода отличается от сигнатуры метода `show()` из класса `A`, для которого параметры не предусмотрены. В результате переопределение не происходит.

#### **Задание:**

Переопределите методы для вывода всех полей.

### **Тема 3.10 Переопределение методов и поддержка полиморфизма**

Пример, приведенный в предыдущем разделе, демонстрирует переопределение методов, но из него трудно понять, насколько богатые возможности представляет данный механизм. Действительно, если переопределение методов используется только для соблюдения соглашений об именовании, то его можно рассматривать как интересную, но почти бесполезную деталь. Однако это не так. Переопределение методов лежит в основе одного из наиболее мощных средств динамического доступа к методам. Динамический доступ к методам – это механизм вызова переопределенных методов, причем выбор конкретного метода может осуществляется не на этапе компиляции, а в процессе выполнения программы. Динамический доступ к методам очень важен, поскольку именно посредством его в языке `Java` реализован полиморфизм.

Давайте начнем с того, что вспомним один очень важный принцип: *переменная суперкласса может ссылаться на экземпляр подкласса*. В языке `Java` этот факт используется для вызова переопределенных методов на этапе выполнения. Если переопределенный метод вызывается посредством ссылки на суперкласс, то исполняющая система `Java` по типу объекта определяет, какой вариант метода следует вызвать, причем выясняет это в процессе работы программы. Если ссылки указывают на различные типы объектов, то вызваны будут разные версии переопределенных методов. Другими словами, вариант переопределенного метода для вызова определяет не тип переменной, а тип объекта, на который она ссылается. Таким образом, если суперкласс содержит метод, переопределенный в подклассе, то вызывается метод, соответствующий тому типу объекта, на который указывает переменная суперкласса.

Ниже приведен простой пример, демонстрирующий динамический доступ к методам.

#### *Листинг 3.17*

// Демонстрация динамического доступа к методам

```
public class Sup {  
    public void who() {  
        System.out.println("who() in Sup");  
    }  
}
```

```

    }
    public class Sub1 extends Sup {
        public void who() {
            System.out.println("who() in Sub1");
        }
    }
    public class Sub2 extends Sup {
        public void who() {
            System.out.println("who() in Sub2");
        }
    }
    public class DynDispDemo {
        public static void main(String args[]) {
            Sup superOb = new Sup();
            Sub1 subOb1 = new Sub1();
            Sub2 subOb2 = new Sub2();
            Sup supRef;
            // В каждом из приведенных ниже вызовов конкретный
            // вариант метода who() определяется на этапе
            // выполнения программы
            supRef = superOb;
            supRef.who();
            supRef = subOb1;
            supRef.who();
            supRef = subOb2;
            supRef.who();
        }
    }
}

```

В данной программе определен суперкласс Sup и два подкласса, Sub1 и Sub2. В классе Sup определен метод who(), который переопределяется в подклассах. В теле метода main() созданы объекты типа Sup, Sub1 и Sub2. Там же присутствует ссылка на тип Sup с именем supRef. Затем в методе main () переменной supRef последовательно присваиваются ссылки на объекты различного типа и эти ссылки используются для вызова метода who(). Как видно из выходных данных, вариант метода who(), который должен быть вызван, определяется типом объекта, на который ссылается переменная supRef в момент вызова, а не типом этой переменной.

Переопределяемые методы обеспечивают поддержку полиморфизма в языке Java. Полиморфизм в объектно-ориентированных программах имеет большое значение потому, что благодаря ему становится возможным объявить в классе методы, общие для всех подклассов, но позволить подклассам определять специфические реализации всех этих методов или некоторых из них. Переопределение методов – один из способов, которыми в языке Java реализуется принцип полиморфизма "один интерфейс – много методов".

Одним из залогов успешного использования полиморфизма является понимание того, что суперклассы и подклассы формируют иерархию в направлении от меньшей специализации к большей. Будучи сформированным корректно, суперкласс предоставляет подклассу все элементы, которые последний может непосредственно использовать. В нем также определяются те методы, которые должны быть по-другому реализованы в дочерних классах. Таким образом, подклассы получают достаточную гибкость, выражающуюся в возможности определять собственные методы, и в то же время оказываются вынуждены реализовывать требуемый интерфейс. Объединяя наследование с переопределением методов, суперкласс определяет общий формат методов, который должен быть использован во всех его подклассах.

Для того чтобы лучше понять, насколько мощным является механизм переопределения методов, применим его к классу `TwoDShape`. В приведенных ранее примерах в каждом классе, дочернем по отношению к классу `TwoDShape`, определялся метод `area()`. Теперь мы знаем, что в этом случае имеет смысл включить метод `area()` в состав класса `TwoDShape` и позволить каждому подклассу переопределить этот метод, в частности реализовать вычисление площади в зависимости от конкретного типа геометрической фигуры. Такой подход реализован в приведенной ниже программе. Для удобства к классу `TwoDShape` добавлено поле `name`, которое упрощает написание демонстрационного кода.

*Листинг 3.18*

// Использование динамического доступа к методу

```
public class TwoDShape {  
    private double width;  
    private double height;  
    private String name;
```

// Конструктор по умолчанию

```
    public TwoDShape() {  
        width = height = 0.0;  
        name = "null";  
    }
```

// Конструктор с параметрами

```
    public TwoDShape(double w, double h, String n) {  
        width = w;  
        height = h;  
        name = n;  
    }
```

// Формирование объекта с равными

// значениями `width` и `height`

```
    public TwoDShape(double x, String n) {  
        width = height = x;  
        name = n;  
    }
```

```

// Формирование объекта на основе другого объекта
public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
}
// Методы для доступа к переменным width и height
public double getWidth() {
    return width;
}
public double getHeight() {
    return height;
}
public void setWidth(double w) {
    width = w;
}
public void setHeight(double h) {
    height = h;
}
public String getName() {
    return name;
}
public void showDim() {
    System.out.println("Width and height are "
        + width + " and " + height);
}
// Метод area() определен в классе TwoDShape
public double area() {
    System.out.println("area() must be overridden");
    return 0.0;
}
}
// Подкласс класса TwoDShape для представления треугольников
public class Triangle extends TwoDShape {
    private String style;
// Конструктор по умолчанию
    public Triangle() {
        super();
        style = "null";
    }
// Конструктор класса Triangle
    public Triangle(String s, double w, double h) {
        super(w, h, "triangle");
        style = s;
    }
}

```

```

// формирование равнобедренных треугольников
public Triangle(double x) {
    super(x, "triangle"); // Вызов конструктора суперкласса
    style = "isosceles";
}
// формирование объекта на основе другого объекта
public Triangle(Triangle ob) {
    super(ob); // Передача объекта конструктору TwoDShape
    style = ob.style;
}
// Переопределение метода area() для класса Triangle
public double area() {
    return getWidth() * getHeight() / 2;
}
void showStyle() {
    System.out.println("Triangle is " + style);
}
}
// Подкласс класса TwoDShape для представления прямоугольников
public class Rectangle extends TwoDShape {
    // Конструктор по умолчанию
    public Rectangle() {
        super();
    }
    // Конструктор класса Rectangle
    public Rectangle(double w, double h) {
        super(w, h, "rectangle"); // Вызов конструктора суперкласса
    }
    // Формирование квадрата
    public Rectangle(double x) {
        super(x, "rectangle"); // Вызов конструктора суперкласса
    }
    // Формирование объекта на основе другого объекта
    public Rectangle(Rectangle ob) {
        super(ob); // Передача объекта конструктору TwoDShape
    }
    public boolean isSquare() {
        if (getWidth() == getHeight()) {
            return true;
        }
        return false;
    }
}
// Переопределение метода area() для класса Rectangle
public double area() {
    return getWidth() * getHeight();
}

```

```

    }
}

public class DynShapes {
    public static void main(String args[]) {
        TwoDShape shapes[] = new TwoDShape[5];
        shapes[0] = new Triangle("right", 8.0, 12.0);
        shapes[1] = new Rectangle(10);
        shapes[2] = new Rectangle(10, 4);
        shapes[3] = new Triangle(7.0);
        shapes[4] = new TwoDShape(10, 20, "generic");
        for (int i = 0; i < shapes.length; i++) {
            System.out.println("object is " + shapes[i].getName());
            // Требуемый вариант area() определяется
            // для каждой геометрической фигуры
            System.out.println("Area is " + shapes[i].area());
            System.out.println();
        }
    }
}

```

Рассмотрим код программы более подробно. Теперь, как и предполагалось написании программы, метод `area()` входит в состав класса `TwoDShape` переопределяется в классах `Triangle` и `Rectangle`. В классе `TwoDShape` метод `area()` играет роль заглушки и лишь информирует пользователя о том, что метод должен быть переопределен в подклассе. При каждом переопределении метода `area()` в нем реализуются средства, необходимые для того типа объекта, который представляется подклассом. Таким образом, если вам надо будет реализовать класс, представляющий эллипс, вам придется переопределить метод `area()` так, чтобы он вычислял площадь данной фигуры.

В рассматриваемой программе есть еще одна важная особенность. Обратите внимание на то, что в методе `main()` фигуры объявлены как массив объектов `TwoDShape`. Однако на самом деле элементами массива являются ссылки на объекты `Triangle`, `Rectangle` и `TwoDShape`. Это допустимо, поскольку переменная суперкласса может быть ссылкой на объект подкласса. В программе организован перебор элементов массива в цикле и вывод информации о каждом объекте. Несмотря на то что пример достаточно прост, он иллюстрирует как возможности наследования, так и переопределения методов. Тип объекта, на который указывает переменная суперкласса, определяется на этапе выполнения программы и обрабатывается соответствующим образом. Если объект является дочерним по отношению к `TwoDShape`, то его площадь вычисляется путем вызова метода `area()`. Интерфейс для данной операции общий и не зависит от того, с какой геометрической фигурой мы имеем дело в каждый конкретный момент.



### Тема 3.11 Использование абстрактных классов

Когда вы начнете создавать свои библиотеки классов, то увидите, что ситуация, при которой в контексте суперкласса невозможно реализовать тот или иной метод, встречается очень часто. Справиться с такой проблемой можно двумя способами. Один из них был рассмотрен в предыдущем примере. Он состоит в том что метод лишь отображает предупреждающее сообщение. Такой подход может быть полезным в ряде случаев, например при отладке, но в большинстве ситуаций он совершенно неприменим. Поэтому чаще всего применяется второй способ, позволяющий создать метод, который должен быть переопределен в подклассе там, где он приобретает конкретный смысл. Рассмотрим класс `Triangle`. Он будет неполон, если не определить метод `area()`. Нужен механизм, позволяющий убедиться в том, что в подклассе переопределены требуемые методы. В языке Java этот механизм носит название абстрактный метод.

При создании абстрактного метода используется модификатор `abstract`. Тело абстрактного метода отсутствует, поэтому он не реализуется в суперклассе. В подклассе он должен быть переопределен. Использовать вариант метода, объявленный в суперклассе, невозможно. Абстрактный метод объявляется в следующем формате:

```
abstract тип имя (список_параметров);
```

Как видите, тело метода отсутствует. Модификатор `abstract` применим только с обычными методами. Он не может быть использован со статическими методами или с конструкторами.

Класс, содержащий хотя бы один абстрактный метод, также должен быть объявлен абстрактным, т.е. в определении класса должен присутствовать спецификатор `abstract`. Поскольку в абстрактном классе отсутствует определение хотя бы одного метода, экземпляр такого класса создать невозможно. Попытка сформировать соответствующий объект посредством оператора `new` приведет к возникновению ошибки на этапе компиляции.

Подкласс, наследующий абстрактный класс, должен реализовать все абстрактные методы. Если это не будет сделано, то подкласс также должен быть абстрактным. Таким образом, подклассы останутся абстрактными до тех пор, пока все абстрактные методы не будут реализованы. Теперь мы можем модифицировать класс `TwoDShape`, используя модификатор `abstract`. Поскольку говорить о площади фигуры невозможно до тех пор, пока тип фигуры не будет конкретизирован, в классе `TwoDShape` есть смысл добавить абстрактный метод `area()`, а в определении самого класса также использовать модификатор `abstract`. Это будет означать, что во всех подклассах класса `TwoDShape` должен быть переопределен метод `area()`.

*Листинг 3.19*

```
// Создание абстрактного класса
public abstract class TwoDShape {
    private double width;
```

```

private double height;
private String name;
// Конструктор по умолчанию
public TwoDShape() {
    width = height = 0.0;
    name = "null";
}
// Конструктор с параметрами
public TwoDShape(double w, double h, String n) {
    width = w;
    height = h;
    name = n;
}
// Формирование объекта с одинаковыми значениями width и height
public TwoDShape(double x, String n) {
    width = height = x;
    name = n;
}
// Формирование объекта на основе другого объекта
public TwoDShape(TwoDShape ob) {
    width = ob.width;
    height = ob.height;
    name = ob.name;
}
// Методы для доступа к переменным width и height
public double getWidth() {
    return width;
}
public double getHeight() {
    return height;
}
public void setWidth(double w) {
    width = w;
}
public void setHeight(double h) {
    height = h;
}
public String getName() {
    return name;
}
public void showDim() {
    System.out.println("Width and height are "
        + width + " and " + height);
}
// Теперь метод area() абстрактный

```

```

    public abstract double area();
}
// Подкласс класса TwoDShape для представления треугольников
public class Triangle extends TwoDShape {

    private String style;
// Конструктор по умолчанию
    public Triangle() {
        super();
        style = "null";
    }
// Конструктор класса Triangle
    public Triangle(String s, double w, double h) {
        super(w, h, "triangle");
        style = s;
    }
// Формирование равнобедренного треугольника
    public Triangle(double x) {
        super(x, "triangle"); // Вызов конструктора суперкласса
        style = "isosceles";
    }
// Формирование объекта на основе другого объекта
    public Triangle(Triangle ob) {
        super(ob); // Передача объекта конструктору TwoDShape
        style = ob.style;
    }
    public double area() {
        return getWidth() * getHeight() / 2;
    }
    public void showStyle() {
        System.out.println("Triangle is " + style);
    }
}
// Подкласс класса TwoDShape для представления прямоугольников
public class Rectangle extends TwoDShape {
    // Конструктор по умолчанию
    public Rectangle() {
        super();
    }
// Конструктор класса Rectangle
    public Rectangle(double w, double h) {
        super(w, h, "rectangle"); // Вызов конструктора суперкласса
    }
// Формирование квадрата
    public Rectangle(double x) {

```

```

        super(x, "rectangle"); // Вызов конструктора суперкласса
    }
    // Формирование объекта на основе другого объекта
    public Rectangle(Rectangle ob) {
        super(ob); // Передача объекта конструктору TwoDShape
    }
    public boolean isSquare() {
        if (getWidth() == getHeight()) {
            return true;
        }
        return false;
    }
    public double area() {
        return getWidth() * getHeight();
    }
}

public class AbsShape {
    public static void main(String args[]) {
        TwoDShape shapes[] = new TwoDShape[4];
        shapes[0] = new Triangle("right", 8.0, 12.0);
        shapes[1] = new Rectangle(10);
        shapes[2] = new Rectangle(10, 4);
        shapes[3] = new Triangle(7.0);
        for (int i = 0; i < shapes.length; i++) {
            System.out.println("object is " + shapes[i].getName());
            System.out.println("Area is " + shapes[i].area());
            System.out.println();
        }
    }
}

```

Итак, в данной программе во всех подклассах класса TwoDShape должен б переопределен метод area(). Чтобы убедиться в этом, попробуйте создать класс, в котором определение метода area() будет отсутствовать. При попытке скомпилировать такую программу вы получите сообщение об ошибке. Несмотря на то что в классе TwoDShape отсутствует определение одного из методов, возможность создать ссылку на объект данного типа. Это также видно из текста программы. В отличие от предыдущих вариантов класса TwoDShape объект данного типа на этот раз сформировать невозможно. По этой причине число элементов массива shapes в методе main() сокращено до четырех; объект TwoDShape теперь не создается.

И еще одно замечание. Обратите внимание на то, что в классе TwoDShape по-прежнему присутствуют определения методов showDim () и getName () и перед ними нет модификатора abstract. Это допустимо; абстрактный класс имеет право содержать полностью определенные методы,

которые могут быть или не быть переопределены в подклассах. Обязательному переопределению подлежат лишь те методы, перед которыми указан модификатор `abstract`.

#### **Задание:**

Напишите класс Собака, который наследуется от класса Животные. В классы Собака и Кот добавьте метод голос. В классе Животные этот метод абстрактный.

### **Тема 3.12 Использование ключевого слова `final`**

Какие бы богатые возможности ни представляли наследование и переопределение методов, в некоторых случаях бывает необходимо запретить их. Предположим, например, что вы создаете класс, в котором инкапсулированы средства управления некоторым устройством. Помимо прочего, в этом классе содержится метод, инициализирующий данное устройство. Предположим также, что инициализационная процедура может быть составлена так, что пользователю будут доступны некоторые секретные данные, хранящиеся на устройстве. В этой ситуации вам потребуется запретить переопределение инициализационного метода. Язык Java предоставляет средства, позволяющие без труда запретить переопределение метода или наследование класса. Решает эту задачу ключевое слово `final`.

### **Тема 3.13 Предотвращение переопределения методов**

Для того чтобы предотвратить переопределение метода, надо при его объявлении указать модификатор `final`. Методы, определенные таким образом, нельзя переопределять. Ниже приведен фрагмент кода, иллюстрирующий использование ключевого слова `final`.

*Листинг 3.20*

```
public class A {
    public final void meth() {
        System.out.println("This is a final method.");
    }
}

public class B extends A {
    public void meth() { // Ошибка! Переопределить метод невозможно
        System.out.println("Illegal!");
    }
}
```

Поскольку метод `meth ()` объявлен как `final`, его нельзя переопределить в классе `B`. Если вы попытаетесь сделать это, то возникнет ошибка при компиляции программы.

### Тема 3.14 Предотвращение наследования

Предотвратить наследование класса можно, указав в определении класса ключевое слово `final`. При этом считается, что данное ключевое слово было применено ко всем методам класса. Очевидно, что не имеет никакого смысла применять `final` к абстрактным классам. Ведь абстрактный класс не завершен по определению, и объявленные в нем методы должны быть реализованы в подклассах.

Ниже приведен пример класса, подклассы которого создавать запрещено.

```
public final class A { . //
// Следующее определение класса недопустимо.
public class B extends A { // Ошибка! Подкласс класса A
// не может существовать.
// ...
```

Как видно из комментариев, недопустимо, чтобы класс `B` наследовал класс `A` определен как `final`.

### Тема 3.15 Класс Object

В языке Java определен специальный класс `Object`. По умолчанию он считается суперклассом всех остальных классов. Другими словами, все классы являются подклассами `Object`. Это означает, что переменная типа `Object` может ссылаться на экземпляр любого класса. Более того, поскольку массивы реализованы в виде классов, переменная типа `Object` может также ссылаться на любой массив.

В классе `Object` определены перечисленные ниже методы, которые доступны в любом объекте.

Таблица 3.1 – Методы класса `Object`

Метод	Назначение
<code>Object clone()</code>	Создает новый объект, аналогичный объекту предназначенному для клонирования
<code>boolean equals(Object объект)</code>	Определяет, эквивалентны ли объекты
<code>void finalize ()</code>	Вызывается перед тем, как неиспользованный объект будет удален процедурой "сборки мусора"
<code>Class&lt;? Extends Object&gt; getClass()</code>	Позволяет определить класс

	объекта в процессе выполнения программы
<code>int hashCode()</code>	Возвращает хэш-код, связанный с вызывающим объектом
<code>void notify()</code>	Возобновляет работу потока, ожидающего оповещения от объекта
<code>void notifyAll()</code>	Возобновляет работу всех потоков, ожидающих оповещения от объекта
<code>String toString()</code>	Возвращает строку, описывающую объект
<code>void wait()</code>	Ожидает выполнения другого потока
<code>void wait(long миллисекунды)</code>	
<code>void wait(long миллисекунды, int наносекунды)</code>	

Методы `getClass()`, `notify()`, `notifyAll()`, и `wait()` объявлены `final`, остальные можно переопределять в подклассах.

Если при создании класса предполагается проверка логической эквивалентности объектов, которая не выполнена в суперклассе, следует переопределить два метода: `equals(Object ob)` и `hashCode()`. Кроме того, переопределение этих методов необходимо, если логика приложения предусматривает использование элементов в коллекциях. Метод `equals()` при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь – в противном случае. При переопределении метода `equals()` должны выполняться соглашения, предусмотренные спецификацией языка Java, а именно:

- 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 не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;
- ненулевая ссылка при сравнении с литералом `null` всегда возвращает значение `false`.

При создании информационных классов также рекомендуется переопределять методы `hashCode()` и `toString()`, чтобы адаптировать их действия для создаваемого типа.

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.

Один из способов создания правильного метода `hashCode()`, гарантирующий выполнение соглашений, приведен ниже, в примере # 10.

Метод `toString()` следует переопределять таким образом, чтобы кроме стандартной информации о пакете (опционально), в котором находится класс, и самого имени класса (опционально), он возвращал значения полей объекта, вызвавшего этот метод (то есть всю полезную информацию объекта), вместо хэш-кода, как это делается в классе `Object`. Метод `toString()` класса `Object` возвращает строку с описанием объекта в виде:

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

Метод вызывается автоматически, когда объект выводится методами `println()`, `print()` и некоторыми другими.

Объекты в методы передаются по ссылке, в результате чего в метод передается ссылка на объект, находящийся вне метода. Поэтому если в методе изменить значение поля объекта, то это изменение коснется исходного объекта. Во избежание такой ситуации для защиты внешнего объекта следует создать клон (копию) объекта в методе. Класс `Object` содержит `protected`-метод `clone()`, осуществляющий побитовое копирование объекта производного класса. Однако сначала необходимо переопределить метод `clone()` как `public` для обеспечения возможности вызова из другого пакета. В переопределенном методе следует вызвать базовую версию метода `super.clone()`, которая и выполняет собственно клонирование. Чтобы окончательно сделать объект клонируемым, класс должен реализовать интерфейс `Cloneable`. Интерфейс `Cloneable` не содержит методов относится к помеченным (tagged) интерфейсам, а его реализация гарантирует, что метод `clone()` класса `Object` возвратит точную копию вызвавшего его объекта с воспроизведением значений всех его полей. В противном случае метод генерирует исключение `CloneNotSupportedException`. Следует отметить, что при использовании этого механизма объект создается без вызова конструктора. В языке C++ аналогичный механизм реализован с помощью конструктора копирования.

Так как объекты создаются динамически с помощью операции `new`, а уничтожаются автоматически, то желательно знать механизм ликвидации объектов и способ освобождения памяти. Автоматическое освобождение памяти, занимаемой объектом, выполняется с помощью механизма “сборки мусора”. Когда никаких ссылок на объект не существует, то есть все ссылки



на него вышли из области видимости программы, предполагается, что объект больше не нужен, и память, занятая объектом, может быть освобождена. “Сборка мусора” происходит нерегулярно во время выполнения программы. Форсировать “сборку мусора” невозможно, можно лишь “рекомендовать” ее выполнить вызовом метода `System.gc()` или `Runtime.getRuntime().gc()`, но виртуальная машина выполнит очистку памяти тогда, когда сама посчитает это удобным. Вызов метода `System.runFinalization()` приведет к запуску метода `finalize()` для объектов утративших все ссылки.

Иногда объекту нужно выполнять некоторые действия перед освобождением памяти. Например, освободить внешние ресурсы. Для обработки таких ситуаций могут применяться два способа: конструкция `try-finally` и механизм `finalization`. Конструкция `try-finally` является предпочтительной, абсолютно надежной и будет рассмотрена в девятой главе. Запуск механизма `finalization` определяется алгоритмом сборки мусора и до его непосредственного исполнения может пройти сколь угодно много времени. Из-за всего этого поведение метода `finalize()` может повлиять на корректную работу программы, особенно при смене JVM. Если существует возможность освободить ресурсы или выполнить другие подобные действия без привлечения этого механизма, то лучше без него обойтись. Виртуальная машина вызывает этот метод всегда, когда она собирается уничтожить объект данного класса. Внутри метода `finalize()`, вызываемого непосредственно перед освобождением памяти, следует определить действия, которые должны быть выполнены до уничтожения объекта.

Метод `finalize()` имеет следующую сигнатуру:

```
protected void finalize(){  
    // код завершения  
}
```

Ключевое слово `protected` запрещает доступ к `finalize()` коду, определенному вне этого класса. Метод `finalize()` вызывается только перед самой “сборкой мусора”, а не тогда, когда объект выходит из области видимости, то есть заранее невозможно определить, когда `finalize()` будет выполнен, и недоступный объект может занимать память довольно долго. В принципе этот метод может быть вообще не выполнен! Недопустимо в приложении доверять такому методу критические по времени действия по освобождению ресурсов.

В листинге 3.21 показан пример переопределения всех методов.

*Листинг 3.21*

```
public class Test implements Cloneable {  
    private int a;  
    private int b;  
    public Test() {  
    }  
    public Test(int a, int b) {
```

```

        this.a = a;
        this.b = b;
    }
    public int getA() {
        return a;
    }
    public int getB() {
        return b;
    }
    public void setA(int a) {
        this.a = a;
    }
    public void setB(int b) {
        this.b = b;
    }
    @Override
    public String toString() {
        return "Test{" + "a=" + a + ", b=" + b + '}';
    }
    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Test other = (Test) obj;
        if (this.a != other.a) {
            return false;
        }
        if (this.b != other.b) {
            return false;
        }
        return true;
    }
    @Override
    public int hashCode() {
        int hash = 7;
        hash = 97 * hash + this.a;
        hash = 97 * hash + this.b;
        return hash;
    }
    @Override
    protected Object clone() {

```

```

        try {
            return super.clone();
        } catch (CloneNotSupportedException ex) {
            System.out.println("Ошибка при клонировании объекта");
        }
        return null;
    }
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Объект будет удален: a=" + a + ", b=" + b);
    }
}

public class Main {
    public static void main(String[] args) {
        Test ob = new Test();
        ob.setA(1);
        ob.setB(2);
        Test ob2 = new Test(1, 2);
        if (ob.equals(ob2)) {
            System.out.println("объекты равны");
        }
        System.out.println(ob);
        System.out.println(ob.hashCode());
        System.out.println(ob2.hashCode());
        Test ob3 = (Test) ob.clone();
        System.out.println(ob3);
        System.out.println(ob3.hashCode());
        ob = null;
        System.gc(); // просьба выполнить "сборку мусора"
    }
}

```

В результате выполнения данной программы получим:

объекты равны

Test{a=1, b=2}

65962

65962

Test{a=1, b=2}

65962

Объект будет удален: a=1, b=2

### Тема 3.16 Интерфейсы

### 3.16.1 Объявление интерфейса.

В объектно-ориентированном программировании часто бывает полезно указать, что класс должен делать, но не определять, как это должно быть сделано. Пример такого подхода уже встречался вам при рассмотрении абстрактных методов. Абстрактный метод определяет сигнатуру метода, но не его реализацию. Подкласс должен реализовать абстрактные методы, присутствующие в суперклассе. Таким образом, абстрактный метод описывает интерфейс метода, но не дает его реализацию. Убедившись, что абстрактные классы и методы очень полезны, имеет смысл несколько развить эту тему. В языке Java имеется возможность полностью отделить интерфейс класса от его реализации. Для этой цели служит ключевое слово `interface`.

Синтаксически интерфейсы напоминают абстрактные классы. Однако в составе интерфейса не может быть определено тело ни одного из методов. Таким образом, интерфейс не содержит реализации ни одного метода. Он описывает, что должно быть сделано, но не поясняет, как. Если интерфейс определен, его можно реализовать в сколь угодно большом количестве классов. Справедливо и обратное: один класс может реализовать любое количество интерфейсов.

Для того чтобы реализовать интерфейс, класс должен определить методы, описанные в интерфейсе. Каждый класс может содержать собственную реализацию методов. Другими словами, два класса могут реализовать один и тот же интерфейс различными способами, но каждый из них должен поддерживать один и тот же набор Методов. Метод, имеющий сведения об интерфейсе, может использовать экземпляры класса, поскольку интерфейс позволяет выполнять с объектом все необходимые операции. Посредством ключевого слова `interface` Java позволяет полностью использовать принцип полиморфизма "один интерфейс – несколько методов".

Интерфейс представляется в следующем формате.

```
тип_доступа interface имя {  
    тип_возвращаемого_значения имя-метода_1 (список параметров);  
    тип_возвращаемого_значения имя-метода_2 (список параметров);  
    переменная_1 = значение;  
    переменная_2 = значение;  
    // ...  
    тип_возвращаемого_значения имя-метода_N (список параметров);  
    переменная_N = значение;  
}
```

В данном случае модификатор типа доступа может быть либо `public`, либо не использоваться вовсе. Если спецификатор отсутствует, применяется правило предусмотренное по умолчанию, т.е. интерфейс оказывается доступным только членам своего пакета. Ключевое слово `public` указывает на то, что интерфейс может использоваться из любого другого пакета. (Код интерфейса, объявленного как `public`, должен храниться в файле, имя

которого совпадает с именем интерфейса.) Имя интерфейса должно представлять собой допустимый идентификатор.

При объявлении методов указываются их сигнатуры и типы возвращаемого значения. Эти методы являются абстрактными. Как уже было сказано ранее, реализация метода не может содержаться в составе интерфейса.

Каждый класс, в определении которого указан интерфейс, должен реализовывать все методы, объявленные в составе интерфейса. Методы, объявленные в составе интерфейса, считаются общедоступными.

Переменные, объявленные в составе интерфейса, не являются переменными экземпляра. Считается, что перед ними указаны ключевые слова `public`, `final` и `static`; и они обязательно подлежат инициализации. Другими словами, в составе интерфейса недопустимы переменные, а могут присутствовать только константы.

Ниже приведен пример определения интерфейса `Radio`. Предполагается, что данный интерфейс будет реализован классом `HomeRadio`.

*Листинг 3.22*

```
public interface Radio {  
    public void on();  
    public void off();  
    public void nextChannel();  
    public void previousChannel();  
    public void showChannel();  
}
```

Данный интерфейс объявлен как `public`, следовательно, он может быть реализован классом, принадлежащим любому пакету.

### 3.16.2 Реализация интерфейсов

Единожды определенный интерфейс может быть реализован одним или несколькими классами. Для того чтобы реализовать интерфейс, в определении класса надо включить ключевое слово `implements`, а затем определить методы объявленные в составе интерфейса. Определение класса, реализующего интерфейс, выглядит следующим образом:

```
class имя_класса extends суперкласс implements интерфейс {  
    // тело класса  
}
```

Если класс должен реализовать несколько интерфейсов, то имена интерфейсов указываются через запятую. Очевидно, что в данном случае ключевое слово `extends` и имя суперкласса могут отсутствовать.

Реализуемые методы интерфейса должны быть объявлены как `public`. Сигнатура реализованного метода должна полностью соответствовать сигнатуре, заявленной в составе интерфейса.

Ниже приведен пример класса HomeRadio реализующего интерфейс Radio.

*Листинг 3.23*

// Реализация интерфейса Radio

```
public class HomeRadio implements Radio {
    private int channel;
    @Override
    public void on() {
        System.out.println("Радио включено");
    }
    @Override
    public void off() {
        System.out.println("Радио выключено");
    }
    @Override
    public void nextChannel() {
        this.channel++;
    }
    @Override
    public void previousChannel() {
        this.channel--;
    }
    @Override
    public void showChannel() {
        System.out.println("Текущий канал "+channel);
    }
}
```

Обратите внимание на то, что все методы объявлены как public. Это необходимо, так как любой метод интерфейса считается общедоступным. Метод, к которому нельзя обратиться из других классов, можно попросту не включать в интерфейс. Ниже приведен код программы, демонстрирующей использование класса (листинг 3.24).

*Листинг 3.24*

```
public class Main {
    public static void main(String[] args) {
        HomeRadio radio = new HomeRadio();
        radio.on();
        radio.showChannel();
        radio.nextChannel();
        radio.showChannel();
        radio.nextChannel();
        radio.showChannel();
        radio.previousChannel();
        radio.showChannel();
        radio.off();
    }
}
```

```
}  
}
```

В результате выполнения данной программы получим:

Радио включено

Текущий канал 0

Текущий канал 1

Текущий канал 2

Текущий канал 1

Радио выключено

Класс, реализующий интерфейс, может содержать дополнительные переменные и методы. Это допустимо, более того, именно так в большинстве случаев ступают разработчики. Например, в приведенном ниже варианте класса `HomeRadio` добавлен конструктор, методы `setChannel()` и `getChannel()`.

*Листинг 3.25*

```
public class HomeRadio implements Radio {  
    private int channel;  
    public HomeRadio(int channel) {  
        this.channel = channel;  
    }  
    public void setChannel(int channel) {  
        this.channel = channel;  
    }  
    public int getChannel() {  
        return channel;  
    }  
    @Override  
    public void on() {  
        System.out.println("Радио включено");  
    }  
    @Override  
    public void off() {  
        System.out.println("Радио выключено");  
    }  
    @Override  
    public void nextChannel() {  
        this.channel++;  
    }  
    @Override  
    public void previousChannel() {  
        this.channel--;  
    }  
    @Override  
    public void showChannel() {  
        System.out.println("Текущий канал "+channel);  
    }  
}
```

}

### 3.16.3 Использование ссылок на интерфейсы

Возможно, вы удивитесь, узнав, что при написании программ разрешается создавать переменные, тип которых определяется именами интерфейсов. Другими словами, вы можете создать ссылку на интерфейс. Такая переменная может ссылаться на любой объект, реализующий интерфейс. Если вызвать метод посредством ссылки на интерфейс, управление получит вариант метода, реализованный объектом, на который в данный момент ссылается переменная. Этот процесс аналогичен использованию ссылки на суперкласс для доступа к экземпляру подкласса.

Ниже приведен пример, демонстрирующий использование ссылки на интерфейс. Посредством такой ссылки будет вызываться метод `area()` реализованный в классах `Circle` и `Square` (листинг 3.26).

*Листинг 3.26*

```
//Интерфейс Area
public interface Area {
    public void area();
}
// Класс Circle реализует метод интерфейса
public class Circle implements Area{
    private int radius;
    public Circle(int radius) {
        this.radius = radius;
    }
    @Override
    public void area() {
        System.out.println("Площадь круга "+Math.PI*Math.pow(radius, 2));
    }
}
// Класс Square реализует метод интерфейса
public class Square implements Area {
    private int a;
    public Square(int a) {
        this.a = a;
    }
    @Override
    public void area() {
        System.out.println("Площадь квадрата " + a * a);
    }
}
// Класс Main демонстрирует использование ссылок на интерфейс
public class Main {
```



```

public static void main(String[] args) {
    Area s = new Square(5);
    Area c = new Circle(10);
    s.area();
    c.area();
}

```

В результате работы данной программы получим:

Площадь квадрата 25

Площадь круга 314.1592653589793

В методе `main()` при объявлении переменной с указан тип `Area`, соответствующий имени интерфейса. Это означает, что в данной переменной может храниться ссылка на любой объект, реализующий `Area`. Переменная, ссылающаяся на интерфейс, имеет сведения только о методах, объявленных в этом интерфейсе. Таким образом, переменная с не может быть использована для доступа к переменным и методам, содержащимся в составе объекта, но не объявленным в интерфейсе.

Можно создать массив ссылок на интерфейс. Измененный класс `Main` показан в листинге 3.27.

*Листинг 3.27*

```

public class Main {
    public static void main(String[] args) {
        Area mas[] = new Area[4];
        mas[0] = new Circle(5);
        mas[1] = new Circle(10);
        mas[2] = new Square(5);
        mas[3] = new Square(10);
        for (Area area : mas) {
            area.area();
        }
    }
}

```

В результате работы данной программы получим:

Площадь круга 78.53981633974483

Площадь круга 314.1592653589793

Площадь квадрата 25

Площадь квадрата 100

### 3.16.4 Переменные в составе интерфейсов

Как было сказано ранее, в составе интерфейсов могут объявляться переменные, но они считаются `public`, `static` и `final`. На первый взгляд может показаться, что такие переменные найдут лишь ограниченное применение, но это не так. В больших программах часто используются константы,

описывающие размеры Массивов, граничные значения, специальные наборы битов и многие другие величины. Поскольку для больших программ обычно создается несколько исходных файлов, нужен удобный способ, позволяющий обеспечить доступ к константам любого файла. В языке Java решить эту задачу помогают интерфейсы.

Для того чтобы определить набор разделяемых констант, надо лишь создать интерфейс, в котором не описывались бы методы, а только содержались требуемые константы. Каждый класс, которому нужны данные константы, должен "реализовать" интерфейс. В результате константы становятся доступными. Ниже приведен простой пример использования подобного подхода.

### *Листинг 3.28*

```
// Интерфейс, определяющий константы
public interface ConstInterface {
    public int MIN=0;
    public int MAX=10;
    public String ERROR=" Вы вышли за границы
диапазона["+MIN+", "+MAX+"]!";
}
//Демонстрация использования констант
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            int x = (int) (Math.random() * 20 - 5);
            if (x < ConstInterface.MIN || x > ConstInterface.MAX) {
                System.out.println("x = " + x + ConstInterface.ERROR);
            } else {
                System.out.println("x = " + x);
            }
        }
    }
}
```

В результате работы данной программы получим:

```
x = -4 Вы вышли за границы диапазона[0,10]!
x = 10
x = 3
x = 9
x = 14 Вы вышли за границы диапазона[0,10]!
x = 6
x = -2 Вы вышли за границы диапазона[0,10]!
x = -1 Вы вышли за границы диапазона[0,10]!
x = 2
x = 4
```

### 3.16.5 Наследование интерфейсов

Интерфейс может наследовать другой интерфейс. Данная задача решается с помощью ключевого слова `extends`. При этом синтаксис не отличается от наследования классов. Если класс реализует интерфейс, наследующий другие интерфейсы, в нем надо полностью определить все методы, объявленные в интерфейсах по всей цепочке наследования. Ниже приведен пример наследования интерфейсов (листинг 3.29).

*Листинг 3.29*

```
// Наследование интерфейсов
public interface A {
    void meth1();
    void meth2();
}
// Интерфейс B содержит meth1() и meth2(),
// кроме того, в него добавляется meth3()
public interface B extends A { // B наследует A
    void meth3();
}
// Данный класс должен реализовать все методы,
// объявленные в интерфейсах A и B
public class MyClass implements B {
    public void meth1() {
        System.out.println("Implement meth1().");
    }
    public void meth2() {
        System.out.println("Implement meth2().");
    }
    public void meth3() {
        System.out.println("Implement meth3().");
    }
}
public class Main {
    public static void main(String[] args) {
        MyClass ob = new MyClass();
```

В некоторых случаях необходимо создать суперкласс, который лишь определяет общий формат, детали же уточняются лишь в его подклассах. В таком суперклассе нет необходимости реализовывать методы, — эта задача делегируй дочерним классам. Иногда подобная потребность возникает потому, что в суперклассе попросту невозможно создать осмысленную реализацию метода. Таким является класс `TwoDShape`, рассмотренный в предыдущих примерах. В нем метод `area()` представляет собой не более чем заглушку. Он не способен вычислить и отобразить площадь конкретного объекта.

```
        ob.meth1();
```

```

        ob.meth2();
        ob.meth3();
    }
}

```

В качестве эксперимента можно попробовать удалить из MyClass реализацию метода meth1(). В результате на этапе компиляции возникнет ошибка. Как было сказано ранее, в каждом классе, реализующем интерфейс, должны быть определены все методы, объявленные в интерфейсе, в том числе те, которые были унаследованы от других интерфейсов.

### Тема 3.17 Пакеты и ограничение доступа

Пакет (**package**) – это некий контейнер, который используется для того, чтобы изолировать имена классов. Например, вы можете создать класс List, заключить его в пакет и не думать после этого о возможных конфликтах, которые могли бы возникнуть, если бы кто-нибудь еще создал класс с именем List.

Все идентификаторы, которые мы до сих пор использовали в наших примерах, располагались в одном и том же пространстве имен (name space). Это означает, что нам во избежание конфликтных ситуаций приходилось заботиться о том, чтобы у каждого класса было свое уникальное имя. Пакеты – это механизм, который служит как для работы с пространством имен, так и для ограничения видимости. У каждого файла .java есть 4 одинаковых внутренних части, из которых мы до сих пор использовали только одну. Ниже приведена общая форма исходного файла Java.

*одинокый оператор package (необязателен)*

*любое количество операторов import (необязательны)*

*одинокое объявление открытого (public) класса*

*любое количество закрытых (private) классов пакета (необязательны)*

Первое, что может появиться в исходном файле Java – это оператор package, который сообщает транслятору, в каком пакете должны определяться содержащиеся в данном файле классы. Пакеты задают набор отдельных пространств имен, в которых хранятся имена классов. Если оператор package не указан, классы попадают в безымянное пространство имен, используемое по умолчанию. Если вы объявляете класс, как принадлежащий определенному пакету, например,

```
package java.awt.image;
```

то и исходный код этого класса должен храниться в каталоге java/awt/image.

При попытке поместить класс в пакет, вы сразу натолкнетесь на жесткое требование точного совпадения иерархии каталогов с иерархией пакетов. Вы не можете переименовать пакет, не переименовав каталог, в котором хранятся его классы. Эта трудность видна сразу, но есть и менее очевидная проблема.

Представьте себе, что вы написали класс с именем `PackTest` в пакете `test`. Вы создаете каталог `test`, помещаете в этот каталог файл `PackTest.java` и транслируете. Пока – все в порядке. Однако при попытке запустить его вы получаете от интерпретатора сообщение «`can't find class PackTest`» («Не могу найти класс `PackTest`»). Ваш новый класс теперь хранится в пакете с именем `test`, так что теперь надо указывать всю иерархию пакетов, разделяя их имена точками - `test.PackTest`.

После оператора `package`, но до любого определения классов в исходном Java-файле, может присутствовать список операторов `import`. Пакеты являются хорошим механизмом для отделения классов друг от друга, поэтому все встроенные в Java классы хранятся в пакетах. Общая форма оператора `import` такова:

```
import пакет1 [.пакет2].(имякласса); // один класс  
import пакет1 [.пакет2].*; // все классы пакета
```

Здесь *пакет1* – имя пакета верхнего уровня, *пакет2* – это необязательное имя пакета, вложенного в первый пакет и отделенное точкой. И, наконец, после указания пути в иерархии пакетов, указывается либо имя класса, либо метасимвол звездочка. Звездочка означает, что, если Java-транслятору потребуется какой-либо класс, для которого пакет не указан явно, он должен просмотреть все содержимое пакета со звездочкой вместо имени класса.

Но использовать без нужды форму записи оператора `import` с использованием звездочки не рекомендуется, т.к. это может значительно увеличить время трансляции кода (на скорость работы и размер программы это не влияет).

Все встроенные в Java классы, которые входят в комплект поставки, хранятся в пакете с именем `java`. Базовые функции языка хранятся во вложенном пакете `java.lang`. Весь этот пакет автоматически импортируется транслятором во все программы. Это эквивалентно размещению в начале каждой программы оператора

```
import java.lang.*;
```

Если в двух пакетах, подключаемых с помощью формы оператора `import` со звездочкой, есть классы с одинаковыми именами, однако вы их не используете, транслятор не отреагирует. А вот при попытке использовать такой класс, вы сразу получите сообщение об ошибке, и вам придется переписать операторы `import`, чтобы явно указать, класс какого пакета вы имеете в виду.

### **Ограничение доступа**

Java предоставляет несколько уровней защиты, обеспечивающих возможность тонкой настройки области видимости данных и методов. Из-за наличия пакетов Java должна уметь работать еще с четырьмя категориями видимости между элементами классов :

- Подклассы в том же пакете.
- Не подклассы в том же пакете.
- Подклассы в различных пакетах.

- Классы, которые не являются подклассами и не входят в тот же пакет.

В языке Java имеется три уровня доступа, определяемых ключевыми словами: `private` (закрытый), `public` (открытый) и `protected` (защищенный), которые употребляются в различных комбинациях. Содержимое ячеек таблицы определяет доступность переменной с данной комбинацией модификаторов (столбец) из указанного места (строка).

Таблица 3.2 – Уровни доступа

	<code>private</code>	модификатор отсутствует	<code>private protected</code>	<code>protected</code>	<code>public</code>
тот же класс	да	да	да	да	да
подкласс в том же пакете	нет	да	да	да	да
независимый класс в том же пакете	нет	да	нет	да	да
подкласс в другом пакете	нет	нет	да	да	да
независимый класс в другом пакете	нет	нет	нет	нет	да

На первый взгляд все это может показаться чрезмерно сложным, но есть несколько правил, которые помогут вам разобраться. Элемент, объявленный `public`, доступен из любого места. Все, что объявлено `private`, доступно только внутри класса, и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется в языке Java по умолчанию. Если же вы хотите, чтобы элемент был доступен извне пакета, но только подклассам того класса, которому он принадлежит, вам нужно объявить такой элемент `protected`. И наконец, если вы хотите, чтобы элемент был доступен только подклассам, причем независимо от того, находятся ли они в данном пакете или нет, то используйте комбинацию `private protected`.

### Тема 3.18 Внутренние классы

Классы могут взаимодействовать друг с другом не только посредством наследования и использования ссылок, но и посредством организации логической структуры с определением одного класса в теле другого.

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. С одной стороны, обоснованное использование в коде внутренних классов делает его более эффективным и понятным. С другой стороны, применение внутренних классов есть один из способов сокрытия кода, так как внутренний класс может быть абсолютно недоступен и не виден вне класса-владельца. Внутренние классы также используются в качестве блоков прослушивания событий (глава «События»). Одной из важнейших причин использования внутренних классов является возможность независимого наследования внутренними классами. Фактически при этом реализуется множественное наследование со своими преимуществами и проблемами.

В качестве примеров можно рассмотреть взаимосвязи классов «Корабль», «Двигатель» и «Шлюпка». Объект класса «Двигатель» расположен внутри (невидим извне) объекта «Корабль» и его деятельность приводит «Корабль» в движение. Оба этих объекта неразрывно связаны, то есть запустить «Двигатель» можно только посредством использования объекта «Корабль», например, из машинного отделения. Таким образом, перед инициализацией объекта внутреннего класса «Двигатель» должен быть создан объект внешнего класса «Корабль».

Класс «Шлюпка» также является логической частью класса «Корабль», однако ситуация с его объектами проще по причине того, что данные объекты могут быть использованы независимо от наличия объекта «Корабль». Объект класса «Шлюпка» только использует имя (на борту) своего внешнего класса. Такой внутренний класс следует определять как `static`. Если объект «Шлюпка» используется без привязки к какому-либо судну, то класс следует определять как обычный независимый класс.

Вложенные классы могут быть статическими, объявляемыми с модификатором `static`, и нестатическими. Статические классы могут обращаться к членам включающего класса не напрямую, а только через его объект. Нестатические внутренние классы имеют доступ ко всем переменным и методам своего внешнего класса-владельца.

### 3.18.1 Внутренние (inner) классы

Нестатические вложенные классы принято называть внутренними (inner) классами. Доступ к элементам внутреннего класса возможен из внешнего класса только через объект внутреннего класса, который должен быть создан в коде метода внешнего класса. Объект внутреннего класса всегда ассоциируется (скрыто хранит ссылку) с создавшим его объектом внешнего класса — так называемым внешним (enclosing) объектом. Внешний и внутренний классы приведены в листинге 3.30.

*Листинг 3.30*

```
public class Ship {  
    // поля и конструкторы  
    // abstract, final, private, protected - допустимы  
    public class Engine { // определение внутреннего класса
```

```

// поля и методы
    public void launch() {
        System.out.println("Запуск двигателя");
    }
} // конец объявления внутреннего класса
public void init() { // метод внешнего класса
// объявление объекта внутреннего класса
    Engine eng = new Engine();
    eng.launch();
}
}

```

При таком объявлении объекта внутреннего класса Engine в методе внешнего класса Ship нет реального отличия от использования какого-либо другого внешнего класса, кроме объявления внутри класса Ship. Использование объекта внутреннего класса вне своего внешнего класса возможно только при наличии доступа (видимости) и при объявлении ссылки в виде:

```
Ship.Engine obj = new Ship().new Engine();
```

Основное отличие от внешнего класса состоит в больших возможностях ограничения видимости внутреннего класса по сравнению с обычным внешним классом. Внутренний класс может быть объявлен как `private`, что обеспечивает его полную невидимость вне класса-владельца и надежное сокрытие реализации. В этом случае ссылку `obj`, приведенную выше, объявить было бы нельзя. Создать объект такого класса можно только в методах и логических блоках внешнего класса. Использование `protected` позволяет получить доступ к внутреннему классу для класса в другом пакете, являющегося суперклассом внешнего класса.

После компиляции объектный модуль, соответствующий внутреннему классу, получит имя `Ship$Engine.class`.

Методы внутреннего класса имеют прямой доступ ко всем полям и методам внешнего класса, в то же время внешний класс может получить доступ к содержимому внутреннего класса только после создания объекта внутреннего класса. Внутренние классы не могут содержать статические атрибуты и методы, кроме констант (`final static`). Внутренние классы имеют право наследовать другие классы, реализовывать интерфейсы и выступать в роли объектов наследования. Допустимо наследование следующего вида:

```

public class WarShip extends Ship {
    protected class SpecialEngine extends Engine {
    }
}

```

Если внутренний класс наследуется обычным образом другим классом (после `extends` указывается `ИмяВнешнегоКласса.ИмяВнутреннегоКласса`), то



он теряет доступ к полям своего внешнего класса, в котором он был объявлен.

```
public class Motor extends Ship.Engine {  
    public Motor(Ship obj) {  
        obj.super();  
    }  
}
```

В данном случае конструктор класса Motor должен быть объявлен с параметром типа Ship, что позволит получить доступ к ссылке на внутренний класс Engine, наследуемый классом Motor.

Внутренние классы позволяют окончательно решить проблему множественного наследования, когда требуется наследовать свойства нескольких классов.

При объявлении внутреннего класса могут использоваться модификаторы final, abstract, private, protected, public.

### 3.18.2 Вложенные (nested) классы

Если не существует необходимости в связи объекта внутреннего класса с объектом внешнего класса, то есть смысл сделать такой класс статическим.

Вложенный класс логически связан с классом-владельцем, но может быть использован независимо от него.

При объявлении такого внутреннего класса присутствует служебное слово static, и такой класс называется вложенным (nested). Если класс вложен в интерфейс, то он становится статическим по умолчанию. Такой класс способен наследовать другие классы, реализовывать интерфейсы и являться объектом наследования для любого класса, обладающего необходимыми правами доступа. В то же время статический вложенный класс для доступа к нестатическим членам и методам внешнего класса должен создавать объект внешнего класса, а напрямую имеет доступ только к статическим полям и методам внешнего класса. Для создания объекта вложенного класса объект внешнего класса создавать нет необходимости. Подкласс вложенного класса не способен унаследовать возможность доступа к членам внешнего класса, которыми наделен его суперкласс.

*Листинг 3.31*

```
public class Ship {  
    private int id;  
    // abstract, final, private, protected - допустимы  
    public static class LifeBoat {  
        public static void down() {  
            System.out.println("шлюпки на воду!");  
        }  
        public void swim() {  
            System.out.println("отплытие шлюпки");  
        }  
    }  
}
```

```

    }
}
public class Main {
    public static void main(String[] args) {
// вызов статического метода
        Ship.LifeBoat.down();
// создание объекта статического класса
        Ship.LifeBoat lf = new Ship.LifeBoat();
// вызов обычного метода
        lf.swim();
    }
}

```

Статический метод вложенного класса вызывается при указании полного относительного пути к нему. Объект lf вложенного класса создается с использованием имени внешнего класса без вызова его конструктора.

Класс, вложенный в интерфейс, по умолчанию статический. На него не накладывается никаких особых ограничений, и он может содержать поля и методы как статические, так и нестатические.

### 3.18.3 Анонимные (anonymous) классы

Анонимные (безымянные) классы применяются для придания уникальной функциональности отдельно взятому объекту для обработки событий, реализации блоков прослушивания и т.д. Можно объявить анонимный класс, который будет расширять другой класс или реализовывать интерфейс при объявлении одного, единственного объекта, когда остальным объектам этого класса будет соответствовать реализация метода, определенная в самом классе. Объявление анонимного класса выполняется одновременно с созданием его объекта посредством оператора new.

Анонимные классы эффективно используются, как правило, для реализации (переопределения) нескольких методов и создания собственных методов объекта. Этот прием эффективен в случае, когда необходимо переопределение метода, но создавать новый класс нет необходимости из-за узкой области (или одноразового) применения метода.

Конструкторы анонимных классов нельзя определять и переопределять. Анонимные классы допускают вложенность друг в друга, что может сильно запутать код и сделать эти конструкции непонятными.

*Листинг 3.32*

```

public class TypeQuest {
    private int id = 1;
    public TypeQuest() {
    }
    public TypeQuest(int id) {

```

```

        this.id = id;
    }
    public void addNewType() {
        // реализация
        System.out.println("добавлен вопрос на соответствие");
    }
}
public class Main {
    public static void main(String[] args) {
        TypeQuest unique = new TypeQuest() { // анонимный класс #1
            public void addNewType() {
                // новая реализация метода
                System.out.println("добавлен вопрос со свободным ответом");
            }
        }; // конец объявления анонимного класса
        unique.addNewType();
        new TypeQuest(71) { // анонимный класс #2
            private String name = "Drag&Drop";
            public void addNewType() {
                // новая реализация метода #2
                System.out.println("добавлен " + getName());
            }
            String getName() {
                return name;
            }
        }.addNewType();
        TypeQuest standard = new TypeQuest(35);
        standard.addNewType();
    }
}

```

В результате будет выведено:

добавлен вопрос со свободным ответом

добавлен Drag&Drop

добавлен вопрос на соответствие

При запуске приложения происходит объявление объекта `unique` с применением анонимного класса, в котором переопределяется метод `addNewType()`. Вызов данного метода на объекте `unique` приводит к вызову версии метода из анонимного класса, который компилируется в объектный модуль с именем `RunnerAnonym$1`.

Процесс создания второго объекта с анонимным типом применяется в программировании значительно чаще, особенно при реализации классов-

адаптеров и реализации интерфейсов в блоках прослушивания. В этом же объявлении продемонстрирована возможность объявления в анонимном классе полей и методов, которые доступны объекту вне этого класса.

При помощи анонимных классов удобно реализовывать интерфейс. Пример реализации методов интерфейса приведен в листинге 3.33.

*Листинг 3.33*

```
// интерфейс OnOff
public interface OnOff {
    public void on();
    public void off();
}

public class Main {
    public static void main(String[] args) {
        // реализация методдов для объекта radio
        OnOff radio = new OnOff() {
            @Override
            public void on() {
                System.out.println("Радио включено");
            }

            @Override
            public void off() {
                System.out.println("Радио выключено");
            }
        };
        // реализация методдов для объекта tv
        OnOff tv = new OnOff() {
            @Override
            public void on() {
                System.out.println("Телевизор включен");
            }
            @Override
            public void off() {
                System.out.println("Телевизор выключен");
            }
        };
        radio.on();
        radio.off();
        tv.on();
        tv.off();
    }
}
```

В результате рабботы программы получим:  
Радио включено

Радио выключено  
Телевизор включен  
Телевизор выключен

## **Выводы к главе:**

- *Наследование – один из трех базовых принципов объектно-ориентированного программирования.*
- *В Java нет множественного наследования классов.*
- *С помощью наследования можно сформировать общий класс, определяющий характерные особенности некоторого понятия.*
- *В языке Java наследуемый класс принято называть суперклассом. Его дочерние классы называются подклассами.*
- *Подкласс – это специализированная версия суперкласса. Он наследует все переменные и методы, определенные в суперклассе, и дополняет их своими элементами.*
- *Абстрактный метод – метод имеющий заголовок, но не имеющий реализации(тела).*
- *Если в классе есть хотя бы один абстрактный метод, то класс становится абстрактным.*
- *Интерфейс не содержит реализации ни одного метода. Он описывает, что должно быть сделано, но не поясняет, как.*
- *Если интерфейс определен, его можно реализовать в сколь угодно большом количестве классов.*
- *Один класс может реализовать любое количество интерфейсов.*
- *Для того чтобы реализовать интерфейс, класс должен определить методы, описанные в интерфейсе. Каждый класс может содержать собственную реализацию методов.*
- *Пакет – это контейнер, позволяющий разделить пространство имен классов.*
- *В разных пакетах могут быть классы с одинаковыми именами, в одном – нет.*
- *Классы могут взаимодействовать друг с другом не только посредством наследования и использования ссылок, но и посредством организации логической структуры с определением одного класса в теле другого.*
- *Статический класс описанный в теле другого класса – вложенный.*
- *Нестатические вложенные классы принято называть внутренними (inner) классами.*
- *Анонимный класс расширяет другой класс или реализует интерфейс при объявлении одного, единственного объекта, когда остальным объектам этого класса будет соответствовать реализация методов, определенная в самом классе. Объявление анонимного класса выполняется одновременно с созданием его объекта посредством оператора new.*

### Задания к главе:

1). Разработать интерфейс Арифметика. Методы – сложение, сравнение. Класс Матрица (поле – двумерный массив) реализует интерфейс. Дополнительные методы:

создание матрицы (размерность и значения вводятся с консоли),  
заполнение матрицы случайными числами,  
вывод 1 значения из матрицы, по номеру ячейки,  
масштабирование матрицы в меньшую сторону (поэлементное деление на число).

Класс Строки (поле – массив символов char) реализует интерфейс. Дополнительные методы:

создание строки (ввод с консоли),  
вывод 1 значение из строки, по номеру.

Класс Вектор (как набор значений, поле – одномерный массив) реализует интерфейс. Дополнительные методы:

создание вектора (ввод с консоли),  
сравнение длин двух векторов.

2). Создать интерфейс с методом площадь и реализовать его в классах: прямоугольник, круг, прямоугольный треугольник, трапеция со своими.

Для проверки определить массив ссылок на интерфейс, которым присваиваются различные объекты.

Площадь трапеции:  $S=(a+b)h/2$

3). Создать интерфейс с методом норма и реализовать его в классах: комплексные числа, вектор из 10 элементов, матрица (2x2). Определить метод нормы:

для комплексных чисел – модуль в квадрате,  
для вектора – корень квадратный из суммы элементов по модулю,  
для матрицы – максимальное значение по модулю.

4). Реализовать интерфейсы, а также наследование и полиморфизм для следующих классов:

4.1). `interface Абитуриент <- abstract class Студент <- class Студент-Заочник.`

4.2). `interface Сотрудник <- class Инженер <-class Руководитель.`

4.3). `interface Здание <- abstract class Общественное Здание <- class Театр.`

4.4). `interface Корабль <- abstract class Военный Корабль <- class Авианосец.`

4.5). `interface Корабль <- class Грузовой Корабль <- class Танкер.`

4.6). `interface Техника <- abstract class Плеер <- class Видеоплеер.`

4.7). `interface Транспортное Средство <- abstract class Общественный Транспорт <- class Трамвай.`

## Глава 4 ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

### Тема 4.1 Исключения в Java

Исключение в Java – это объект, который описывает исключительную ситуацию, возникшую в каком-либо участке программного кода. Когда возникает исключительная ситуация, создается объект класса `Exception` или одного из его потомков. Этот объект попадает в блок `catch`, обрабатывающий данный тип исключительной ситуации. Исключения могут возбуждаться и «вручную» для того, чтобы сообщить о некоторых нештатных ситуациях.

К механизму обработки исключений в Java имеют отношение 5 ключевых слов: – `try`, `catch`, `throw`, `throws` и `finally`. Схема работы этого механизма следующая. Тот код, в котором необходимо отслеживать ошибки помещается в блок `try`, как только происходит ошибка, часть кода от строки с ошибкой до конца блока `try` пропускается и попадаем в блок `catch`, соответствующий данному типу ошибки. Тот код, который необходимо выполнить вне зависимости от того произошли ошибки или нет помещается в блок `finally`(сохранение данных, закрытие соединений с БД и т.д.). При помощи ключевого слова `throw` можно вручную сгенерировать исключение(повторная генерация исключений и создание своего собственного исключения – потомка класса `Exception`). Слово `throws` используется для указания типов исключений, которые могут возникнуть в данном методе.

Ниже приведена общая форма блока обработки исключений.

```
try {  
    // блок кода  
}  
catch (ТипИсключения1 e) {  
    // обработчик исключений типа ТипИсключения1  
}  
catch (ТипИсключения2 e) {  
    // обработчик исключений типа ТипИсключения2  
    throw(e) // повторное возбуждение исключения  
}  
finally {  
    //код, который выполняется всегда  
}
```

### Тема 4.2 Типы исключений

В вершине иерархии исключений стоит класс `Throwable`. Каждый из типов исключений является подклассом класса `Throwable`. Два непосредственных наследника класса `Throwable` делят иерархию подклассов



исключений на две различные ветви. Один из них – класс `Exception` – используется для описания исключительных ситуаций, которые возникают в ходе выполнения вашей программы. Другая ветвь дерева подклассов `Throwable` – класс `Error`, который предназначен для описания исключительных ситуаций, которые могут возникнуть из-за сбоев в виртуальной Java машине.

### Тема 4.3 Неперехваченные исключения

Объекты-исключения автоматически создаются исполняющей средой Java в результате возникновения определенных исключительных состояний. В листинге 4.1 приведен пример автоматического создания исключения в случае деления на ноль.

*Листинг 4.1*

```
public class Main {  
    public static void main(String[] args) {  
        int x = 10 / 0;  
    }  
}
```

В ходе выполнения данной программы получим:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at javaapplication1.Main.main(Main.java:4)
```

Обратите внимание на тот факт что типом возбужденного исключения был не `Exception` и не `Throwable`. Это подкласс класса `Exception`, а именно: `ArithmeticException`, поясняющий, какая ошибка возникла при выполнении программы. Вот другая версия того же класса, в которой возникает та же исключительная ситуация, но на этот раз не в программном коде метода `main`.

*Листинг 4.2*

```
public class Main {  
    public static void metod() {  
        int x = 10 / 0;  
    }  
    public static void main(String[] args) {  
        metod();  
    }  
}
```

Вывод этой программы показывает, как обработчик исключений исполняющей системы Java выводит содержимое всего стека вызовов.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at javaapplication1.Main.metod(Main.java:4)  
at javaapplication1.Main.main(Main.java:7)
```

Если создан объект-исключение и он не отловлен, то он попадает в список системных ошибок и программа останавливается.

#### Тема 4.4 Ключевые слова try и catch

Для задания блока программного кода, который требуется защитить от исключений, используется ключевое слово try. Сразу же после try-блока помещается блок catch, задающий тип исключения которое вы хотите обрабатывать.

*Листинг 4.3*

```
public class Main {  
    public static void main(String[] args) {  
        try {  
            int x = 10 / 0;  
            System.out.println("x=" + x);  
        } catch (ArithmeticException e) {  
            System.out.println("Ошибка! Деление на ноль!");  
        }  
    }  
}
```

В блок try помещается не только строка, в которой может быть ошибка, но и весь логически связанный с ней код. В нашем случае, если произошла ошибка, то выводить результат совершенно бессмысленно.

Целью большинства хорошо сконструированных catch-разделов должна быть обработка возникшей исключительной ситуации и приведение переменных программы в некоторое разумное состояние – такое, чтобы программу можно было продолжить так, будто никакой ошибки и не было (в нашем примере выводится предупреждение – Ошибка! Деление на ноль!).

#### **Задание:**

Написать метод деления двух чисел, учесть обработку в случае деления на ноль.

В некоторых случаях один и тот же блок программного кода может возбуждать исключения различных типов. Для того, чтобы обрабатывать подобные ситуации, Java позволяет использовать любое количество catch-разделов для try-блока. Наиболее специализированные классы исключений должны идти первыми, поскольку ни один подкласс не будет достигнут, если поставить его после суперкласса. Следующая программа перехватывает два различных типа исключений, причем за этими двумя специализированными обработчиками следует раздел catch общего назначения, перехватывающий все подклассы класса Throwable.

*Листинг 4.4*

```
public class Main {
```

```

public static void main(String[] args) {
    try {
        int a = args.length;
        System.out.println("a = " + a);
        int b = 42 / a;
        int c[] = {1};
        c[42] = 99;
    } catch (ArithmeticException e) {
        System.out.println("Деление на ноль " + e);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Выход за пределы массива " + e);
    } catch (Throwable e) {
        //исключения всех остальных типов попадают сюда
        System.out.println("Ошибка " + e);
    }
}
}

```

Этот пример, запущенный без параметров, вызывает возбуждение исключительной ситуации деления на ноль. Если же мы зададим в командной строке один или несколько параметров, тем самым установив `a` в значение больше нуля, наш пример переживет оператор деления, но в следующем операторе будет возбуждено исключение выхода индекса за границы массива `ArrayIndexOutOfBoundsException`. Ниже приведены результаты работы этой программы, запущенной и тем и другим способом.

`a = 0`

Деление на ноль `java.lang.ArithmeticException: / by zero`

`a = 1`

Выход за пределы массива `java.lang.ArrayIndexOutOfBoundsException:`

42

**В случае, если произойдет ошибка какого-либо другого типа, то она будет перехвачена последним блоком `catch (Throwable e)`.**

Задание:

Напишите программу деления двух массивов друг на друга, учесть обработку ошибок в случае деления на ноль и выход за пределы массива.

## Тема 4.5 Вложенные операторы `try`

Операторы `try` можно вкладывать друг в друга аналогично тому, как можно создавать вложенные области видимости переменных. Если у оператора `try` низкого уровня нет раздела `catch`, соответствующего возбужденному исключению, стек будет развернут на одну ступень выше, и в поисках подходящего обработчика будут проверены разделы `catch` внешнего

оператора try. Вот пример, в котором два оператора try вложены друг в друга посредством вызова метода.

*Листинг 4.5*

```
public class Main {
    public static void metod() {
        try {
            int c[] = {1};
            c[42] = 99;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Выход за пределы массива " + e);
        }
    }
    public static void main(String[] args) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            metod();
        } catch (ArithmeticException e) {
            System.out.println("Деление на ноль " + e);
        }
    }
}
```

В данном случае обработка исключения выход за пределы массива сделана в пределах метода, поэтому отслеживать эту ошибку в методе main нет необходимости.

Блоки try могут быть вложены и в пределах одного метода.

*Листинг 4.6*

```
public class Main {
    public static void main(String[] args) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            try {
                int c[] = {1};
                c[42] = 99;
            } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Выход за пределы массива " + e);
            }
            System.out.println("Окончание внешнего блока try");
        } catch (ArithmeticException e) {
            System.out.println("Деление на ноль " + e);
        } catch (Throwable e) {
            //исключения всех остальных типов попадают сюда
        }
    }
}
```

```

        System.out.println("Ошибка " + e);
    }
}

```

Если ошибка произойдет во внешнем блоке try, то попадаем в блок catch(ArithmeticException e) и программа останавливается. Если ошибка происходит во внутреннем блоке try, то она обрабатывается его блоком catch(ArrayIndexOutOfBoundsException e), но внешний блок все равно выполниться до конца.

## Тема 4.6 Ключевое слово throw

Оператор throw используется для возбуждения исключения «вручную». Для того, чтобы сделать это, нужно иметь объект подкласса класса Throwable, который можно либо получить как параметр оператора catch, либо создать с помощью оператора new. Ниже приведена общая форма оператора throw.

*throw ОбъектTunaThrowable;*

При достижении этого оператора нормальное выполнение кода немедленно прекращается, так что следующий за ним оператор не выполняется. Ближайший окружающий блок try проверяется на наличие соответствующего возбужденному исключению обработчика catch. Если такой отыщется, управление передается ему. Если нет, проверяется следующий из вложенных операторов try, и так до тех пор пока либо не будет найден подходящий раздел catch, либо обработчик исключений исполняющей системы Java не остановит программу, выведя при этом состояние стека вызовов. В листинге 4.7 приведен пример повторной генерации исключения. Это применяется в случае, если ошибка обработана в методе, но за пределами этого метода необходимо знать про ошибку.

*Листинг 4.7*

```

public class Main {
    public static int metod1(int a, int b) {
        int c = 0;
        try {
            c = a / b;
        } catch (ArithmeticException e) {
            System.out.println("Произошла ошибка " + e);
            throw e;
        }
        return c;
    }
    public static void metod2(int c) {
        System.out.println("c=" + c);
    }
}

```

```

public static void main(String[] args) {
    int c = 0;
    try {
        c = metod1(5, 0);
        metod2(c);
    } catch (ArithmeticException e) {
        System.out.println("Ошибка при выполнении метода " + e);
    }
}

```

В этом примере обработка исключения проводится в два приема. Это позволяет избежать вызова второго метода, в случае ошибки в первом.

В результате выполнения программы получим:

Произошла ошибка java.lang.ArithmeticException: / by zero

Ошибка при выполнении метода java.lang.ArithmeticException: / by zero

## Тема 4.7 Ключевое слово throws

Если метод способен возбуждать исключения, которые он сам не обрабатывает, он должен объявить о таком поведении, чтобы вызывающие методы могли защитить себя от этих исключений. Для задания списка исключений, которые могут возбуждаться методом, используется ключевое слово `throws`. Если метод в явном виде (т.е. с помощью оператора `throw`) возбуждает исключение соответствующего класса, тип класса исключений должен быть указан в операторе `throws` в объявлении этого метода. С учетом этого наш прежний синтаксис определения метода должен быть расширен следующим образом:

```

тип_имя_метода(список_аргументов) throws список_исключений {
    //тело метода
}

```

Внесем изменения в листинг 4.7. Теперь в методе `main` заранее известно какого типа исключение необходимо отловить.

*Листинг 4.8*

```

public class Main {
    public static int metod1(int a, int b) throws ArithmeticException{
        int c = 0;
        try {
            c = a / b;
        } catch (ArithmeticException e) {
            System.out.println("Произошла ошибка " + e);
            throw e;
        }
        return c;
    }
}

```

```

    public static void metod2(int c) {
        System.out.println("c=" + c);
    }
    public static void main(String[] args) throws PinException,
NotEnoughMoneyException {
        int c = 0;
        try {
            c = metod1(5, 0);
            metod2(c);
        } catch (ArithmeticException e) {
            System.out.println("Ошибка при выполнении метода " + e);
        }
    }
}

```

#### Тема 4.8 Ключевое слово **finally**

Иногда требуется гарантировать, что определенный участок кода будет выполняться независимо от того, какие исключения были возбуждены и перехвачены. Для создания такого участка кода используется ключевое слово **finally**. Даже в тех случаях, когда в методе нет соответствующего возбужденному исключению раздела **catch**, блок **finally** будет выполнен до того, как управление перейдет к операторам, следующим за разделом **try**. У каждого раздела **try** должен быть по крайней мере или один раздел **catch** или блок **finally**. Блок **finally** очень удобен для закрытия файлов и освобождения любых других ресурсов, захваченных для временного использования в начале выполнения метода. Ниже приведен пример класса с двумя методами, завершение которых происходит по разным причинам, но в обоих перед выходом выполняется код раздела **finally**.

*Листинг 4.9*

```

public class Main {
    public static void metod1() throws ArithmeticException {
        try {
            System.out.println("Metod1 run");
            throw new ArithmeticException("Demo");
        } finally {
            System.out.println("Metod1 stop");
        }
    }
    public static void metod2() {
        try {
            System.out.println("Metod2 run");
            return;
        } finally {

```

```

        System.out.println("Metod2 stop");
    }
}
public static void main(String[] args) {
    try {
        metod1();
    } catch (ArithmeticException e) {
        System.out.println("Ошибка при выполнении метода1 " + e);
    }
    metod2();
}
}

```

В первом методе исключение сгенерировано вручную, но, не смотря на это, блок `finally` все равно выполнился. Во втором методе выход из метода осуществлен при помощи `return`, но блок `finally` все равно выполнился. В результате выполнения данной программы получили:

Metod1 run

**Metod1 stop**

**Ошибка при выполнении метода1 java.lang.ArithmeticException:**

**Demo**

Metod2 run

**Metod2 stop**

#### **Тема 4.9 Потомки Exception или написание своих классов ошибок**

Только подклассы класса `Throwable` могут быть возбуждены или перехвачены. Простые типы – `int`, `char` и т.п., а также классы, не являющиеся подклассами `Throwable`, например, `String` и `Object`, использоваться в качестве исключений не могут. Наиболее общий путь для использования исключений – создание своих собственных подклассов класса `Exception`. Ниже приведена программа, в которой объявлен новый подкласс класса `Exception`.

*Листинг 4.10*

```

public class OverrangeTenException extends Exception {
    private int x;
    public OverrangeTenException(int x) {
        this.x = x;
    }
    @Override
    public String toString() {
        return "Значение переменной превысило 10: x=" + x;
    }
}

public class Main {
    public static void main(String[] args) {
        int x = 0;
    }
}

```



```

for (int i = 0; i < 5; i++) {
    try {
        x = (int) (Math.random() * 20);
        if (x > 10) {
            throw new OverrangeTenException(x);
        }
        System.out.println("x=" + x);
    } catch (OverrangeTenException ex) {
        System.out.println(ex);
    }
}
}
}

```

В данном примере создан собственный класс ошибок, как потомок класса Exception. В случае, если значение переменной превышает значение 10, то выбрасываем исключение. Свое исключение нужно отловить блоком catch (OverrangeTenException ex), иначе программа остановится.

В ходе выполнения данной программы получили:

x=9

Значение переменной превысило 10: x=13

x=5

Значение переменной превысило 10: x=11

x=0

### **Задание:**

Доработайте первое задание к главе 3. Создайте свое исключение SizeMismatchException(несовпадение размеров) и обработайте его.

## **Выводы к главе:**

*Исключение – это объект одного из классов потомков `Throwable`*

*Исключение необходимо отловить, до того как оно попадет в список системных ошибок и программа остановится*

*Для обработки исключений в языке Java предусмотрено пять ключевых слов: `try`, `catch`, `throw`, `throws` и `finally`.*

*Код, в котором необходимо отслеживать ошибки помещается в блок `try`.*

```
try{
//код, в котором могут возникнуть ошибки
}
```

*Блок `try` нельзя написать без блока `catch` и/или `finally`.*

*Блок `catch` (если он есть) размещается после блока `try`*

```
try{
}
catch (Тип_ошибки переменная) {
//обработка ошибки
}
```

*Блок `finally`(если он есть) размещается после блока/блоков `catch`, если он есть, или сразу после блока `try`*

```
try{
}
// catch (Тип_ошибки1 переменная) {
// }
// catch (Тип_ошибки2 переменная) {
// }
finally {
//код, который выполняется всегда
}
```

*Блок `catch` или блок `finally` могут отсутствовать. Но не могут отсутствовать вместе!!!*

*Для генерации исключения используется `throw`*

*`throw new Конструктор_исключения();`*

*8. Сгенерированные исключения и исключения, возникшие в результате выполнения программы, обрабатываются совершенно одинаково.*

*9. Если метод не обрабатывает свои исключения, то используют ключевое слово `throws`, после которого через запятую указывают, какие исключения могут возникнуть в методе.*

```
тип имя_метода(список аргументов) throws список_исключений {
//тело метода
}
```

## Глава 5 УНИВЕРСАЛЬНЫЕ ТИПЫ. КОЛЛЕКЦИИ

В последнее время в языке Java было реализовано много новых возможностей. Все они очень полезны и расширяют сферу применения Java, но на одной из них следует остановиться особо, так как она оказывает огромное влияние на язык в целом. Речь идет об универсальных типах. Универсальные типы – это совершенно новая синтаксическая конструкция, появление которой вызвало существенные изменения во многих классах и методах базового API. Не будет преувеличением сказать, что введение универсальных типов изменило сам язык Java.

Универсальные типы – обширная тема. Каждый программист, использующий язык Java, должен иметь хотя бы общее представление об этих средствах. На первый взгляд синтаксис универсальных типов может показаться непонятным, но не следует опускать руки. Использовать их достаточно просто. К тому времени, как вы закончите изучение данной главы, основные понятия будут вам не только знакомы, но и привычны, и вы сможете успешно применять универсальные типы в своих программах.

Если вы используете версии Java, предшествующие J2SE 5, вы не сможете работать с универсальными типами.

### Тема 5.1 Общие сведения об универсальных типах

По сути, универсальные типы должны были бы называться параметризованными типами. Они очень важны, поскольку позволяют создавать классы, интерфейсы и методы, для которых типы обрабатываемых данных передаются в качестве параметра. Классы, интерфейсы или методы, которые обрабатывают типы, передаваемые посредством параметров, называются универсальными, например, можно говорить об универсальном классе или универсальном методе.

Главное преимущество универсального кода состоит в том, что он автоматически настраивается на работу с нужным типом данных. Многие алгоритмы выполняются одинаково, независимо от того, к информации какого типа данных они должны быть применены. Например, быстрая сортировка не зависит от типа данных, в качестве которого можно использовать Integer, String, Object и даже Thread. Используя универсальные типы, можно единожды реализовать алгоритм, а затем без труда применять его к любому типу данных.

Как вы уже знаете, язык Java всегда давал возможность создавать классы, интерфейсы и методы, пригодные для обработки любых типов данных. Это достигалось за счет использования класса Object. Поскольку Object является суперклассом для всех остальных классов, ссылка на Object может выступать в качестве ссылки на любой другой объект. Таким образом, до появления универсального кода для выполнения действий с любыми типами данных в классах, объектах и методах использовалась ссылка типа

Object. При этом возникала серьезная проблема, связанная с необходимостью явно преобразовывать Object в другой тип. Подобное преобразование часто становилось источником ошибок. Универсальные типы повышают уровень безопасности при работе с данными, поскольку при этом все преобразования типов происходят неявно. Таким образом, универсальные типы повышают пригодность кода к повторному использованию.

Синтаксис объявления универсального класса выглядит так:

```
class имя_класса<параметры_типа> { // ...
```

Синтаксис объявления ссылки на универсальный класс приведен ниже.

```
имя_класса<передаваемые_тип имя_переменной =
```

```
new имя_класса<передаваемые_типы>(передаваемые_значения);
```

В листинге 5.1 приведен простой пример использования универсальных типов.

#### *Листинг 5.1*

```
// Здесь T - это параметр, который заменяется реальным именем типа
```

```
// при создании объекта Gen
```

```
public class Gen<T> {
```

```
    private T ob; // Объявление объекта типа T
```

```
    // Конструктору передается ссылка на объект типа T
```

```
    public Gen(T o) {
```

```
        ob = o;
```

```
    }
```

```
    // Метод возвращает объект ob
```

```
    public T getob() {
```

```
        return ob;
```

```
    }
```

```
    // Отображение типа T
```

```
    public void showType() {
```

```
        System.out.println("Type of T is "+ ob.getClass().getName());
```

```
    }
```

```
}
```

```
// Демонстрация универсального класса
```

```
public class DemoGeneric {
```

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

```
        // Создание ссылки на объект типа Gen<Integer>
```

```
        Gen<Integer> iOb;
```

```
    // Создание объекта Gen<Integer> и присваивание ссылки
```

```
    // на него переменной iOb.
```

```
    // Обратите внимание на то, что при помещении* значения 8
```

```
    // в состав объекта Integer используется автоупаковка
```

```
        iOb = new Gen<Integer>(88);
```

```
    // Отображение типа данных, используемого iOb
```

```
        iOb.showType();
```

```
    // Получение значения iOb. Обратите внимание на то,
```

```
    // что приведение типов не требуется
```

```

        int v = iOb.getob();
        System.out.println("value: " + v);
        System.out.println();
    // Создание ссылки на объект типа Gen<String>
        Gen<String> strOb = new Gen<String>("Generics Test");
    // Отображение типа данных, используемого strOb
        strOb.showType();
    // Получение значения strOb. В данном случае приведение
    // типов также не требуется
        String str = strOb.getob();
        System.out.println("value: " + str);
    }
}

```

Рассмотрим код программы более подробно. В первую очередь обратите внимание на способ объявления Gen. Для этого используется следующая строка кода:

```
public class Gen<T> {
```

где T – это имя параметра типа, т.е. параметра, с помощью которого задается тип данных. Данное имя впоследствии будет заменено реальным именем типа, которое передается Gen при создании объекта. Таким образом, T используется в объекте Gen там, где необходим параметр типа. Объявляемый параметр типа указывается в угловых сколках. Поскольку в Gen используется параметр типа, Gen является универсальным классом.

Обратите внимание на имя T в объявлении класса Gen. Для этой цели может быть использован любой допустимый идентификатор, но традиционно программисты применяют имя T. Кроме того, настоятельно рекомендуется, чтобы параметр типа состоял из одной прописной буквы. Если при объявлении класса надо задать несколько параметров типа, то кроме T часто используют имена V и E.

В следующем выражении T используется для объявления объекта ob:

```
private T ob; // Объявление объекта типа T
```

Как было сказано ранее, идентификатор T предназначен для того, чтобы быть замещенным реальным типом при создании объекта Gen. Таким образом, в качестве типа объекта ob будет принят тип, переданный посредством T. Например, при создании объекта будет указан тип String, то объект ob будет также Принадлежать типу String.

Рассмотрим конструктор Gen.

```
public Gen (T o) {
    ob = o
}
```

Обратите внимание на то, что параметр o данного конструктора принадлежит T. Это означает, что реальный тип o определяется типом, переданным посредством T при создании объекта Gen. Поскольку параметр o и переменная ob принадлежат типу T, то после замены T они также будут принадлежать одному и тому же реальному типу.

Параметр типа `T` можно также использовать для того, чтобы задать тип значения, возвращаемого методом. Примером может служить метод `getob()`, код которого показан ниже.

```
public T getob() {  
    return ob;  
}
```

Поскольку типом переменной `ob` является `T`, она совместима с типом, возвращаемым методом `getob()`.

Метод `showType()` отображает тип `T`. Эта задача решается путем вызова метода `getName()` объекта `Class`, полученного в результате вызова метода `getClass()` объекта `ob`. До сих пор мы не использовали данные средства, поэтому рассмотрим их подробнее. В классе `Object` определен метод `getClass()`. Таким образом, данный метод является членом каждого класса. Он возвращает объект `Class`, соответствующий типу текущего объекта. Класс `Class` принадлежит пакету `java.lang` и инкапсулирует информацию о классе. В нем определено несколько методов, которые позволяют в процессе выполнения программы получать сведения о классе. К их числу принадлежит метод `getName()`, возвращающий строковое представление имени класса.

Класс `GenDemo` демонстрирует работу универсального класса `Gen`. В первую очередь в нем создается вариант объекта `Gen` для целых чисел.

```
Gen<Integer> iOb;
```

Приведенная выше строка кода заслуживает пристального внимания. Во-первых, заметьте, что тип `Integer` указывается после имени `Gen` в угловых скобках. В данном случае `Integer` – это передаваемый тип, который заменяет в классе `Gen` параметр типа `T`. Таким образом создается вариант `Gen`, в котором ссылки на `T` преобразуются в ссылки на `Integer`. В объекте, созданном при выполнении приведенного выражения, переменная `ob` и возвращаемое значение метода `getob()` будут принадлежать типу `Integer`.

Перед тем как продолжить рассмотрение универсальных типов, необходимо принять во внимание то, что компилятор `Java` реально не создает различные версии `Gen` или другого универсального класса. В принципе удобно было бы считать, что это происходит, но на самом деле все обстоит по-другому. Вместо создан разных версий компилятор удаляет всю информацию об универсальном типе и использует вместо нее приведение типа. В результате полученный объект ведет себя так, как будто в программе была создана конкретная версия класса `Gen`. Таким образом, в программе реально существует лишь одна версия `Gen`. Процесс удаления информации об универсальном типе называется стиранием (*erasure*). Этот механизм мы рассмотрим более подробно в конце данного модуля.

В следующей строке кода переменной `iOb` присваивается ссылка на экземпляра `Integer`-варианта класса `Gen`:

```
iOb = new Gen<Integer>(88);
```

Обратите внимание на то, что при вызове конструктора `Gen` задается передаваемый тип `Integer`. Это необходимо, поскольку тип объекта, на который указывает ссылка (в данном случае это `iOb`), должен

соответствовать `Gen<Integer>`. Если тип ссылки, возвращаемой оператором `new`, будет отличаться от `Gen<Integer>`, возникнет ошибка компиляции. Например, сообщение о такой ошибке будет сгенерировано при попытке скомпилировать выражение

```
iOb = new Gen<Double>(88.0); // Ошибка!
```

Поскольку переменная `iOb` принадлежит типу `Gen<Integer>`, ее нельзя использовать для хранения ссылки, например на объект, `Gen<Double>`. Возможность проверки на соответствие типов – одно из основных преимуществ универсальных типов.

Продолжим рассмотрение выражения; для удобства повторим его еще раз.

```
iOb = new Gen<Integer>(88);
```

При его выполнении осуществляется автоупаковка целочисленного значения 88 в объект `Integer`. Дело в том, что конструктору `Gen<Integer>` должен передаваться параметр типа `Integer`. При необходимости преобразование типов может быть выполнено явно, как в следующем примере:

```
iOb = new Gen<Integer>(new Integer(88));
```

Однако в данном случае длинная строка кода не дает никаких преимуществ по сравнению с предыдущим, более компактным выражением.

Далее в программе отображается тип переменной `ob`, принадлежащей объекту `iOb` (в данном случае это тип `Integer`). Для получения значения `ob` используется следующее выражение:

```
int v = iOb.getob();
```

Поскольку метод `getob()` возвращает значение типа `T`, которое заменяется `Integer`, возвращаемым типом `getob()` будет `Integer`. Это значение перед присвоением переменной `v` (типа `int`) будет подвергнуто автораспаковке.

И наконец, в классе `GenDemo` объявляется объект типа `Gen<String>`.

```
Gen<String> strOb = new Gen<String>("Generics Test");
```

Поскольку параметр типа – это `String`, данный тип заменяет `T` в составе `Gen`. Таким способом создается `String`-вариант `Gen`, работу которого демонстрируют оставшиеся строки кода.

При определении экземпляра универсального класса передаваемый тип, который заменяет параметр типа, должен быть именем класса. Для этой цели нельзя использовать простой тип, например `int` или `char`. Например, в случае класса `Gen` можно передать в качестве `T` любой класс, но не простой тип. Другими словами, следующее выражение недопустимо:

```
Gen<int> strOb = new Gen<int>(53); // Ошибка.
```

```
// Простой тип в данном случае использовать нельзя.
```

Понятно, что запрет на использование простых типов не является серьезным ограничением, так как всегда можно использовать класс оболочки, инкапсулировав в нем требуемое значение (именно так мы поступили в предыдущем примере). Поддержка автоупаковки и автораспаковки еще больше упрощает данный подход.

Чтобы лучше понять универсальные типы, необходимо заметить, что ссылка на конкретный вариант одного универсального типа несовместима с объектом другого универсального типа. Например, если бы в рассмотренной ранее программе присутствовала Приведенная ниже строка кода, компилятор сгенерировал бы сообщение об ошибке и не сформировал файл класса.

```
iOb = strOb; // Ошибка!
```

Несмотря на то что и iOb и strOb принадлежат типу Gen<T>, они являются ссылками на различные типы. Такое несоответствие возникает вследствие различия передаваемых типов. Эта особенность универсальных типов предотвращает ошибки при написании программ.

## Тема 5.2 Универсальный класс с двумя параметрами типа

В универсальном классе можно задать несколько параметров типа. В этом случае параметры разделяются запятыми. В листинге 5.2 приведен ниже класс TwoGen(модификацией класса Gen) и в нем определены два параметра типа.

### Листинг 5.2

```
// Простой универсальный класс с двумя параметрами типа: T и V
public class TwoGen<T, V> {
    private T ob1;
    private V ob2;
    // Конструктору класса передаются ссылки
    // на объекты типов T и V
    public TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    // Отображение типов T и V
    public void showTypes() {
        System.out.println("Type of T is " + ob1.getClass().getName());
        System.out.println("Type of V is " + ob2.getClass().getName());
    }
    public T getob1() {
        return ob1;
    }
    public V getob2() {
        return ob2;
    }
}
// Демонстрация работы класса TwoGen
public class SimpGen {
    public static void main(String args[]) {
        // В данном случае параметр T заменяется типом Integer,
```



```
// а параметр V - типом String
    TwoGen<Integer, String> tgObj =
        new TwoGen<Integer, String>(88, "Generics");
// Отображение типов
    tgObj.showTypes();
// Получение и отображение переменных
    int v = tgObj.getob1();
    System.out.println("value: " + v);
    String str = tgObj.getob2();
    System.out.println("value: " + str);
}
}
```

Обратите внимание на объявление класса TwoGen.

```
public class TwoGen<T, V> {
```

Здесь определяются два параметра типа: T и V; они разделяются запятыми. Поскольку в классе используются два параметра типа, при создании объекта надо указывать два передаваемых типа.

```
    TwoGen<Integer, String> tgObj =
    new TwoGen<Integer, String>(88, "Generics");
```

В данном случае Integer заменяет параметр типа T, а String заменяет V. И хотя в данном примере передаваемые типы различаются, в принципе они могут совпадать. Например, следующая строка кода допустима:

```
TwoGen<String, String> x = new TwoGen<String, String>("A", "B");
```

В данном случае T и V заменяются типом String. Конечно, если передаваемые типы всегда совпадают, определять два параметра типа нет никакой необходимости.

### Тема 5.3 Ограниченные типы

В предыдущих примерах параметры типа могли заменяться любым классом. Такое решение подходит для многих целей, но иногда полезно ограничить допустимый набор передаваемых типов. Предположим, например, что вы хотите создать универсальный класс, который хранил бы числовое значение и мог выполнять различные математические функции, например, вычислять обратное значение или извлекать дробную часть. Предположим, что мы пошли дальше и хотим использовать данный класс для работы с любыми числовыми типами: целочисленными и с плавающей точкой.

Для подобных ситуаций в языке Java предусмотрены ограниченные типы. При объявлении параметра типа вы можете указать так называемую верхнюю границу, т.е. задать суперкласс, который должны наследовать все передаваемые типы. С этой целью было введено выражение *extends*, определяющее параметр типа так, как показано ниже.

```
<T extends суперкласс>
```

С помощью данного выражения компилятору предоставляется информация о том, что Т может быть заменен только суперклассом или его подклассами. Таким образом, суперкласс определяет верхнюю границу в иерархии классов в Java.

В листинге 5.3 приведен пример использования ограниченного параметра

*Листинг 5.3*

```
public class NumericFns<T extends Number> {
    private T num;
    public NumericFns(T n) {
        num = n;
    }
    public double reciprocal() {
        return 1 / num.doubleValue();
    }
    public double fraction() {
        return num.doubleValue() - num.intValue();
    }
}
// Демонстрация работы NumericFns
public class BoundsDemo {
    public static void main(String args[]) {
        // Применение Integer допустимо, поскольку
        // данный класс является подклассом Number
        NumericFns<Integer> iOb = new NumericFns<Integer>(5);
        System.out.println("Reciprocal of iOb is " + iOb.reciprocal());
        System.out.println("Fractional component of iOb is " + iOb.fraction());
        System.out.println();
        // Применение Double также допустимо
        NumericFns<Double> dOb = new NumericFns<Double>(5.25);
        System.out.println("Reciprocal of dOb is " + dOb.reciprocal());
        System.out.println("Fractional component of dOb is " + dOb.fraction());
        // Следующая строка кода не будет компилироваться,
        // поскольку String не является подклассом Number
        // NumericFns<String> strOb = new NumericFns<String>("Error");
    }
}
```

Заметьте, что теперь для объявления NumericFns используется следующая строка кода:

```
public class NumericFns<T extends Number> {
```

Поскольку тип Т теперь ограничен классом Number, компилятор Java знает, что все объекты типа Т содержат метод doubleValue(), а также другие методы определенные в Number. Это само по себе является огромным преимуществом. Но это еще не все. Данный механизм предотвращает создание объектов NumericFns типов, отличных от числовых. Например, если

вы попытаетесь удалить комментарии из строк, расположенных в конце программы, а затем попытаетесь повторно скомпилировать код, вы получите сообщение об ошибке, поскольку String не является подклассом Number.

### **Задание:**

Напишите параметризированный класс с двумя параметрами, ограниченными классом Number и разработайте метод вычисления суммы двух чисел, любых типов, метод сравнивающий поэлементно два массива разных типов, метод определяющий наибольший и наименьший элементы в массивах.

## **Тема 5.4 Использование групповых параметров**

Рассмотренные выше варианты универсальных типов удобны в работе, но в ряде случаев приходится отдавать предпочтение другим конструкциям. Предположим, что нам надо реализовать метод `absEqual()`, который возвращал бы значение `true` в случае, если два объекта `NumericFns` (этот класс был рассмотрен ранее) содержат одинаковые значения. Предположим также, что нам необходимо, чтобы этот метод работал с любыми типами данных, которые могут храниться в сравниваемых объектах. Например, если один объект содержит значение 1,25 типа `Double`, а второй - значение -1,25 типа `Float`, метод `absEqual()` должен возвращать значение `true`. Один из способов реализации `absEqual()` состоит в том, чтобы передавать Методу параметр типа `NumericFns`, а затем сравнивать его абсолютное значение с абсолютным значением текущего объекта и возвращать `true`, если эти значения совпадают. Например, вызов `absEqual()` может выглядеть следующим образом:

```
NumericFns<Double> dOb = new NumericFns<Double>(1.25);
NumericFns<Float> fOb = new NumericFns<Float>(-1.25);
if(dOb.absEqual(fOb))
    System.out.println("Absolute values are the same.");
else
```

```
    System.out.println("Absolute values differ.");
```

На первый взгляд кажется, что при работе с `absEqual()` не возникают никакие проблемы. Однако это не так. Неприятности начнутся сразу же, как только вы попытаетесь объявить параметр типа `NumericFns`. Каким он должен быть? Подходящим кажется решение наподобие следующего, где в качестве параметра типа задается `T`:

```
// Программа работает некорректно!
// Определение равенства абсолютных значений двух объектов,
public boolean absEqual(NumericFns<T> ob) {
    if(Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue()))
        return true;
    else
```

```

return false;
}

```

В данном случае для определения абсолютного значения каждого числа используется стандартный метод `Math.abs()`. Полученные значения сравниваются. Проблема состоит в том, что описанный подход будет действовать только тогда, когда объект `NumericFns`, передаваемый в качестве параметра, имеет тот же тип, что и текущий объект. Например, если текущий объект принадлежит типу `NumericFns<Integer>`, параметр `ob` также должен быть типа `NumericFns<Integer>`. Сравнить текущий объект с объектом типа `NumericFns<Double>` невозможно. Таким образом, решение не является универсальным.

Для того чтобы создать метод `absEqual()`, применимый во всех случаях, вам надо использовать еще одно средство, связанное с универсальными типами, – групповой параметр. В качестве такого параметра используется символ `?`, который представляет неизвестный тип. Используя данное средство, метод `absEqual()` можно переписать так:

```

// Определение равенства абсолютных значений двух объектов,
public boolean absEqual(NumericFns<?> ob) {
    if(Math.abs(num.doubleValue()) == Math.abs(ob.num.doubleValue()))
        return true;
    else
        return false;
}

```

Здесь выражение `NumericFns<?>` соответствует любому типу `NumericFns` и позволяет сравнивать абсолютные значения двух произвольных объектов `NumericFns`.

#### *Листинг 5.4*

```

// Использование группового параметра
public class NumericF<T extends Number> {
    private T num;
    public NumericF(T n) {
        num = n;
    }
    public double reciprocal() {
        return 1 / num.doubleValue();
    }
    public double fraction() {
        return num.doubleValue() - num.intValue();
    }
    public boolean absEqual(NumericF<?> ob) {
        if (Math.abs(num.doubleValue())
            == Math.abs(ob.num.doubleValue())) {
            return true;
        } else {
            return false;
        }
    }
}

```

```

    }
}
}
// Демонстрация использования группового параметра
public class WildcardDemo {
    public static void main(String args[]) {
        NumericF<Integer> iOb = new NumericF<Integer>(6);
        NumericF<Double> dOb = new NumericF<Double>(-6.0);
        NumericF<Long> lOb = new NumericF<Long>(5L);
        System.out.println("Testing iOb and dOb.");
        if (iOb.absEqual(dOb)) // групповой тип соответствует Double
        {
            System.out.println("Absolute values are equal.");
        } else {
            System.out.println("Absolute values differ.");
        }
        System.out.println();
        System.out.println("Testing iOb and lOb.");
        if (iOb.absEqual(lOb)) // групповой тип соответствует Long
        {
            System.out.println("Absolute values are equal.");
        } else {
            System.out.println("Absolute values differ.");
        }
    }
}

```

Обратите внимание на следующие два вызова `absEqual()`:

```
if (iOb.absEqual(dOb))
```

```
if(iOb.absEqual(lOb))
```

В первом выражении `iOb` – это объект типа `NumericF< Integer>`, а `dOb` - объект типа `NumericF<Double>`. Благодаря использованию группового параметра становится возможным передать `dOb` методу `absEqual()`. Подобным образом формируется и другой вызов, в котором методу передается объект типа `NumericF<Long>`.

Важно понимать, что групповые параметры не влияют на то, какой тип объекта `NumericF` может быть создан. Этим управляет выражение `extends` в объявлении `NumericF`. Групповой параметр лишь указывает на соответствие любому объекту `NumericF`.

## Тема 5.5 Универсальные методы

Как было показано в предыдущих примерах, методы в составе универсальных классов могут использовать параметры типов, заданные для классов. Следовательно, такие методы автоматически становятся

универсальными относительно параметров типов. Можно, однако, объявить универсальный метод, использующий один или несколько собственных параметров типов. Более того, такой метод может принадлежать обычному, не универсальному классу.

Ниже приведен код программы, в котором объявлен класс `GenericMethodDemo`, не являющийся универсальным. В этом классе содержится статический универсальный метод `arraysEqual()`, который определяет, содержатся ли в двух массивах одинаковые элементы, расположенные в том же порядке. Такой метод можно использовать для сравнения двух массивов одинаковых или совместимых между собой типов.

*Листинг 5.5*

```
public class GenericMethodDemo {
    public static <T, V extends T> boolean arraysEqual(T[] x, V[] y) {
        if (x.length != y.length) {
            return false;
        }
        for (int i = 0; i < x.length; i++) {
            if (!x[i].equals(y[i])) {
                return false; // Массивы различаются
            }
        }
        return true; // Содержимое массивов совпадает
    }
    public static void main(String args[]) {
        Integer nums[] = { 1, 2, 3, 4, 5 };
        Integer nums2[] = { 1, 2, 3, 4, 5 };
        Integer nums3[] = { 1, 2, 7, 4, 5 };
        Integer nums4[] = { 1, 2, 7, 4, 5, 6 };
        // Параметры типов T и V неявно определяются
        // при вызове метода
        if (arraysEqual(nums, nums)) {
            System.out.println("nums equals nums");
        }
        if (arraysEqual(nums, nums2)) {
            System.out.println("nums equals nums2");
        }
        if (arraysEqual(nums, nums3)) {
            System.out.println("nums equals nums3");
        }
        if (arraysEqual(nums, nums4)) {
            System.out.println("nums equals nums4");
        }
        // Создание массива типа Double
        Double dvals[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        // Следующая строка не будет скомпилирована, поскольку
```

```
// типы nums и dvals не совпадают
// if(arrayEqual(nums, dvals))
//   System.out.println("nums equals dvals");
// }
}
```

Рассмотрим подробнее метод `arrayEqual()`. В первую очередь обратите внимание на то, как он объявляется.

```
public static <T, V extends T> boolean arrayEqual(T[] x, V[] y) {
```

Параметры типа указаны перед типом значения, возвращаемого методом. Далее, заметьте, что верхней границей `V` является `T`. Таким образом, `V` должен принадлежать или тому же типу, что и `T`, или быть его подклассом. Такая связь гарантирует, что при вызове метода `arrayEqual()` могут быть указаны только параметры, совместимые друг с другом. Также обратите внимание на тот факт, что Метод `arrayEqual()` объявлен как `static`, т.е. его можно вызывать, не создавая предварительно объект. Однако универсальные методы не обязательно должны быть статическими. В этом смысле на них не накладываются ограничения. Теперь рассмотрим обращение к `arrayEqual()` из тела метода `main()`. Для этого используется привычный всем синтаксис, и параметры типа не указываются. Такое решение становится возможным потому, что параметры автоматически распознаются и типы `T` и `V` настраиваются соответствующим образом. Рассмотрим в качестве примера первый вызов.

```
if (arrayEqual(nums, nums))
```

В данном случае базовый тип первого параметра – `Integer`; этот тип заменяет `T`. Таким же является базовый тип второго параметра, следовательно, `V` также заменяется типом `Integer`. Таким образом, выражение для вызова `arrayEqual()` составлено корректно и два массива можно сравнить между собой.

Теперь обратите внимание на закомментированные строки.

```
// if(arrayEqual(nums, dvals))
// System.out.println("nums equals dvals");
```

Если вы удалите символы комментариев и попытаетесь скомпилировать программу, компилятор отобразит сообщение об ошибке. Причина в том, что верхней границей параметра `V` является `T`; этот тип указан после ключевого слова `extends`. Это означает, что `V` может либо совпадать с `T`, либо представлять собой один из его подклассов. В данном случае первый параметр принадлежит типу `Integer`, им заменяется параметр типа `T`, однако второй параметр имеет тип `Double`, который не является подклассом `Integer`. Таким образом, обращение `arrayEqual()` становится недопустимым и возникает ошибка на этапе компиляции.

Ниже приведен общий формат определения универсального метода.

```
<параметры типа> возвращаемый_тип имя_метода (параметры) { //...
```

Параметры, как и при вызове обычного метода, разделяются запятыми. Список параметров типа предшествует возвращаемому типу.

Конструктор может быть универсальным, даже если сам класс не является таковым. В листинге 5.6 класс Summation не универсальный, но в нем используется универсальный конструктор.

*Листинг 5.6*

// Использование универсального конструктора

```
public class Summation {
    private int sum;
    // Универсальный конструктор
    public <T extends Number> Summation(T arg) {
        sum = 0;
        for (int i = 0; i <= arg.intValue(); i++) {
            sum += i;
        }
    }
    public int getSum() {
        return sum;
    }
}

public class GenConsDemo {
    public static void main(String args[]) {
        Summation ob = new Summation(4.0);
        System.out.println("Summation of 4.0 is " + ob.getSum());
    }
}
```

Класс Summation вычисляет и инкапсулирует сумму всех чисел от 0 до N. Значение N передается конструктору. Поскольку для конструктора Summation() указан параметр типа, ограниченный сверху классом Number, объект Summation может быть создан с использованием любого числового типа, в том числе Integer, Float, Double. Независимо от используемого числового типа, соответствующее значение преобразуется в тип Integer путем вызова intValue() и вычисляется требуемая сумма. Таким образом, совсем не обязательно объявлять класс Summation универсальным; достаточно, если универсальным будет лишь его конструктор.

## Тема 5.6 Универсальные интерфейсы

Наряду с универсальными классами и методами существуют также универсальные интерфейсы. Универсальные интерфейсы определяются точно так же, как и универсальные классы. Их использование иллюстрирует следующий пример. В нем создается интерфейс Containment, который может быть реализован классами, хранящими одно или несколько значений. Кроме того, в нем объявлен метод contains(), который определяет, содержится ли указанное значение в текущем объекте.

*Листинг 5.7*



```

// Данный интерфейс подразумевает, что
// реализующий его класс содержит одно или несколько значений
public interface Containment<T> {
// Метод contains() проверяет, содержится ли
// некоторый элемент в составе объекта, реализующего Containment

    public boolean contains(T o);
}
// Реализация интерфейса Containment с использованием
// массива, предназначенного для хранения значений.
// Любой класс, реализующий универсальный интерфейс,
// также должен быть универсальным
public class MyClass<T> implements Containment<T> {
    private T[] arrayRef;
    public MyClass(T[] o) {
        arrayRef = o;
    }
    public boolean contains(T o) {
        for (T x : arrayRef) {
            if (x.equals(o)) {
                return true;
            }
        }
        return false;
    }
}
public class GenIFDemo {
    public static void main(String args[]) {
        Integer x[] = {1, 2, 3};
        MyClass<Integer> ob = new MyClass<Integer>(x);
        if (ob.contains(2)) {
            System.out.println("2 is in ob");
        } else {
            System.out.println("2 is NOT in ob");
        }
        if (ob.contains(5)) {
            System.out.println("5 is in ob");
        } else {
            System.out.println("5 is NOT in ob");
        }
    }
}
// Следующие строки кода недопустимы, так как ob
// представляет собой вариант Integer
// реализации интерфейса Containment, а 9.25 —
// это значение Double
// if(ob.contains(9.25)) // Illegal!

```

```
// System.out.println("9.25 is in ob");
}
}
```

В основном элементы этой программы просты для восприятия, однако на некоторых ее особенностях следует остановиться. Прежде всего обратите внимание на то, как определен интерфейс `Containment`,

```
public interface Containment<T> {
```

Универсальные интерфейсы объявляются так же, как и универсальные классы. В данном случае параметр типа `T` задает тип включаемого объекта. Интерфейс `Containment` реализуется классом `MyClass`. Определение этого класса выглядит следующим образом:

```
public class MyClass<T> implements Containment<T> {
```

Если класс реализует универсальный интерфейс, то он также должен быть универсальным. В нем должен быть объявлен как минимум тот параметр типа, который указан для интерфейса. Например, приведенный ниже вариант объявления класса `MyClass` недопустим.

```
public class MyClass implements Containment<T> { // Ошибка!
```

В данном случае ошибка заключается в том, что в `MyClass` не объявлен параметр типа, а это значит, что нет возможности передать параметр типа интерфейсу `Containment`. При этом идентификатор `T` неизвестен и компилятор генерирует сообщение об ошибке. Класс, реализующий универсальный интерфейс, может не быть универсальным только в одном случае: если при объявлении класса для интерфейса указывается конкретный тип. Подобный вариант объявления класса приведен ниже.

```
public class MyClass implements Containment<Double> { // ОК
```

Наверное, вы не удивитесь, узнав, что один или несколько параметров типа для универсального интерфейса могут быть ограничены. Это позволяет указывать, какие типы данных допустимы для интерфейса. Например, если вы хотите запретить передачу `Containment` значений, не являющихся числовыми, вы можете использовать объявление

```
public interface Containment<T extends Number> {
```

Теперь любой класс, реализующий `Containment`, должен передавать интерфейсу значение типа, удовлетворяющее указанным ограничениям. Например, класс `MyClass`, реализующий рассмотренный интерфейс, должен объявляться следующим образом:

```
public class MyClass<T extends Number> implements Containment<T> {
```

Обратите особое внимание на то, как параметр типа `T` объявляется в классе `MyClass`, а затем передается интерфейсу `Containment`. Поскольку на этот раз интерфейсу `Containment` требуется тип, расширяющий `Number`, в классе, реализующем интерфейс (в данном случае это `MyClass`), должны быть определены соответствующие ограничения. Если верхняя граница задана в определении класса, то нет необходимости еще раз указывать ее в выражении `implements`. Более того, если вы попытаетесь сделать это, получите сообщение об ошибке. Например, следующее выражение некорректно и не будет компилироваться:

*// Ошибка!*

```
public class MyClass<T extends Number> implements Containment<T extends Number> {
```

Если параметр типа задан в определении класса, он лишь передается интерфейсу без дальнейшей модификации.

Формат объявления универсального интерфейса выглядит следующим образом:

```
public interface имя_интерфейса<параметры_типа> { // ...
```

Здесь параметры типа в составе списка разделяются запятыми. При реализации интерфейса имя класса также должно сопровождаться параметрами типа. Выражение, с помощью которого определяется класс, реализующий интерфейс, показано ниже.

```
class имя_класса<параметры_типа>  
implements имя_интерфейса<параметры_типа> {
```

### **Задание:**

Разработать очередь в которой можно хранить объекты любых типов. Методами `get()` и `set()` можно извлечь и добавить объект. Предусмотреть обработку исключений в случае переполнения и пустой очереди.

## **Тема 5.7 Ошибки неоднозначности**

Появление универсальных типов стало причиной возникновения новых ошибок, связанных с неоднозначностью. Ошибки неоднозначности возникают тогда, когда в результате стирания два, на первый взгляд различающихся универсальных объявления преобразуются в один тип, вызывая тем самым конфликтную ситуацию.

### *Листинг 5.8*

```
//Неоднозначность, вызванная стиранием  
//перегруженных методов  
public class MyGenClass<T, V> {  
    private T ob1;  
    private V ob2;  
    // Два следующих метода конфликтуют друг с другом  
    // поэтому код не компилируется  
    public void set(T o) {  
        ob1 = o;  
    }  
    public void set(V o) {  
        ob2 = o;  
    }  
}
```

В классе `MyGenClass` объявлены два универсальных типа: `T` и `V`. В классе `MyGenClass` сделана попытка перегрузки метода `set()`. Перегруженные

методы отличаются типами параметров, в данном случае это T и V. Кажется, что ошибки не должны возникать, так как T и V – различные типы. Однако мы сталкиваемся здесь с двумя проблемами, связанными с неоднозначностью. Во-первых, в классе MyGenClass нет никакой гарантии, что T и V действительно будут различаться. Так, например, не является ошибкой создание объекта MyGenClass с помощью выражения

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

В данном случае T и V заменяются String. В результате оба варианта метода set() становятся одинаковыми, что безусловно является ошибкой.

Вторая, более серьезная проблема состоит в том, что в результате стирания оба варианта метода set() преобразуются в следующий вид:

```
void set(Object o) { // ...
```

Таким образом, попытка перегрузки в классе MyGenClass в принципе приводит к неоднозначности определений метода set(). Решением данной проблемы является отказ от перегрузки и использование двух различных имен методов.

## Тема 5.8 Ограничения универсальных типов

Существует ряд ограничений, которые необходимо учитывать при работе Универсальными типами. К ним относятся создание объектов, тип которых является параметром типа, использование статических членов, генерация исключений и работа с массивами. Рассмотрим их подробнее.

Экземпляр параметра типа создать невозможно.

*Листинг 5.9*

```
//Экземпляр T создать невозможно
```

```
public class Gen<T> {  
    private T ob;  
    public Gen() {  
        ob = new T(); // Illegal!!!  
    }  
}
```

В данном случае попытка создания экземпляра класса T приводит к ошибке. Причину этого понять несложно. Поскольку при выполнении программы t не существует, компилятор не знает о том, какого типа объект должен быть сформирован. Как вы помните, в результате стирания все параметры типа удаляются на этапе компиляции.

Статические члены не могут использовать параметры типа, объявленные в содержащем их классе. Все объявления статических членов в приведенном ниже классе недопустимы.

*Листинг 5.10*

```
public class Wrong<T> {  
    // Статическую переменную типа T создать невозможно  
    private static T ob;
```

```

// Статический метод не может использовать T
public static T getob() {
    return ob;
}
// Статический метод не может обращаться к объекту
// типа T
public static void showob() {
    System.out.println(ob);
}
}

```

Несмотря на наличие описанного выше ограничения, допустимо объявление статические универсальные методы, в которых используются собственные параметры типа. Примеры таких объявлений приводились ранее.

Чтобы при работе с универсальными типами использовать массивы, необходимо учитывать два существенных ограничения. Во-первых, невозможно создать экземпляр массива, базовый тип которого определяется параметром типа. Во-вторых, нельзя создать массив универсальных ссылок, тип которых уточнен. Листинг демонстрирует оба случая.

#### *Листинг 5.11*

```

// Универсальные типы и массивы
public class Gen<T extends Number> {
    private T ob;
    private T vals[]; // Допустимо
    public Gen(T o, T[] nums) {
        ob = o;
        // Следующее выражение недопустимо
// vals = new T[10]; // Невозможно создать массив типа T
        // Следующее выражение корректно
        vals = nums; // Переменной можно присваивать ссылку на
// существующий массив
    }
}

public class GenArray {
    public static void main(String args[]) {
        Integer n[] = {1, 2, 3, 4, 5};
        Gen<Integer> iOb = new Gen<Integer>(50, n);
        // Невозможно создать массив универсальных ссылок
        // Gen<Integer> gens[] = new Gen<Integer>[10]; // Ошибка!
        Gen<?> gens[] = new Gen<?>[10]; // Выражение корректно
    }
}

```

Как видно из кода программы, допустимо создать ссылку на массив типа T. Демонстрирует следующая строка кода:

```
T vals[ ]; // Допустимо.
```

Однако сам массив `T` сформировать нельзя. По этой причине приведенная ниже строка закомментирована.

```
// vals = new T[ 10]; // Невозможно создать массив типа T.
```

Причина данного ограничения состоит в том, что тип `T` не существует при выполнении программы, поэтому компилятор не знает, какого типа массив надо в действительности создать.

Однако вы можете передать ссылку на массив конкретного типа конструктору `Gen()` при создании объекта, а также присвоить это значение переменной `vals`. Примером может служить следующая строка:

```
vals = nums; // Допускается присвоение переменной ссылки  
// на существующий массив.
```

Данное решение корректно, поскольку тип массива, передаваемого `Gen`, известен. При создании объекта его тип совпадает с `T`.

В теле метода `main()` содержится выражение, иллюстрирующее невозможность объявить массив ссылок на уточненный универсальный тип. Строка, приведенная ниже, не будет скомпилирована.

```
// Gen<Integer> gens[] = new Gen<Integer>[ 10] ; // Ошибка!
```

Универсальный класс не может расширять класс `Throwable`. Это означает, что создать универсальный класс исключения невозможно.

## Тема 5.9 Краткий обзор коллекций

Пакет `java.util` содержит одно из наиболее захватывающих расширений Java 2 – коллекции.

*Коллекция* – это группа объектов. Добавление коллекций вызвало фундаментальные изменения в структуре и архитектуре многих элементов пакета `java.util`. Они также расширили круг задач, к которым может применяться пакет. Коллекции – современная технология, которая заслуживает внимания всех программистов Java.

В дополнение к коллекциям, `java.util` содержит большой выбор классов и интерфейсов, которые поддерживают широкий диапазон функциональных возможностей. Эти классы и интерфейсы используются повсюду в пакетах ядра Java, а также доступны для применения в пользовательских программах. Подобные приложения включают генерацию псевдослучайных чисел, манипулирование датой и временем, наблюдение за событиями, манипулирование наборами битов и синтаксический анализ строк. Из-за обширного набора свойств `java.util` – один из наиболее широко используемых пакетов Java.

Структура коллекций (`collections framework`) Java стандартизирует способ, с помощью которого ваши программы обрабатывают группы объектов. В прошлых версиях Java обеспечивал специальные классы типа `Dictionary`, `Vector`, `Stack` и `Properties` для хранения и манипулирования группами объектов. Хотя перечисленные классы были весьма полезны, они

испытывали недостаток в централизованном, унифицирующем подходе при работе с группами объектов. Таким образом, способ использования класса `vector`, например, отличался от способа применения класса `Properties`. Кроме того, предыдущий специальный (*ad hoc*) подход не был предназначен для легкого расширения и адаптации. Коллекции как раз явились ответом на эти (и другие) проблемы.

Структура коллекций была разработана для нескольких целей. Во-первых, структура должна была быть высокоэффективной. Действительно, реализации фундаментальных коллекций (динамических массивов, связанных списков, деревьев и хеш-таблиц) в структуре коллекций Java 2 высокоэффективны. Во-вторых, структура коллекций должна была позволить различным типам коллекций работать похожим друг на друга образом и с высокой степенью способности к взаимодействию. В-третьих, расширение и/или адаптация коллекции должна была быть простой.

Для удобства созданы различные реализации специального назначения, а также некоторые частичные реализации, которые упрощают создание собственного коллекционного класса. Наконец, были добавлены механизмы, которые позволяют интегрировать в структуру коллекций стандартные массивы.

*Алгоритмы* – другая важная часть механизма коллекций. Алгоритмы работают на коллекциях и определены как статические методы в классе `Collections`. Таким образом, они доступны для всех коллекций. Каждый коллекционный класс не нуждается в реализации своих собственных версий. Алгоритмы обеспечивают стандартные средства манипулирования коллекциями..

Другой элемент, созданный структурой коллекций, – это интерфейс `Iterator`. *Итератор* обеспечивает универсальный, стандартизированный способ доступа к элементам коллекции – по одному. Таким образом, итератор обеспечивает средства *перечисления содержимого коллекции*. Поскольку каждая коллекция интерфейс `Iterator`, к элементам любого коллекционного класса можно обращаться через методы этого интерфейса. Поэтому (и только с мелкими изменениями) код, который циклически проходит, скажем, через *набор*, может также использоваться для циклического прохода *списка*, например.

И последнее, если вы знакомы с C++, то полезно знать, что технология коллекций Java подобна (по духу) Стандартной Библиотеке Шаблонов (STL. `Standard Template Library`), определенной в C++. То, что C++ называет контейнером, Java называет коллекцией.

На рисунке 5.1 показана общая структура коллекций.

В структуре коллекций определено несколько интерфейсов. Начать с интерфейсов коллекции необходимо потому, что они определяют фундаментальный характер коллекционных классов. Иначе, конкретные классы просто являются различными реализациями стандартных интерфейсов. Интерфейсы, которые подкрепляют коллекции, кратко описаны в таблице 5.1.

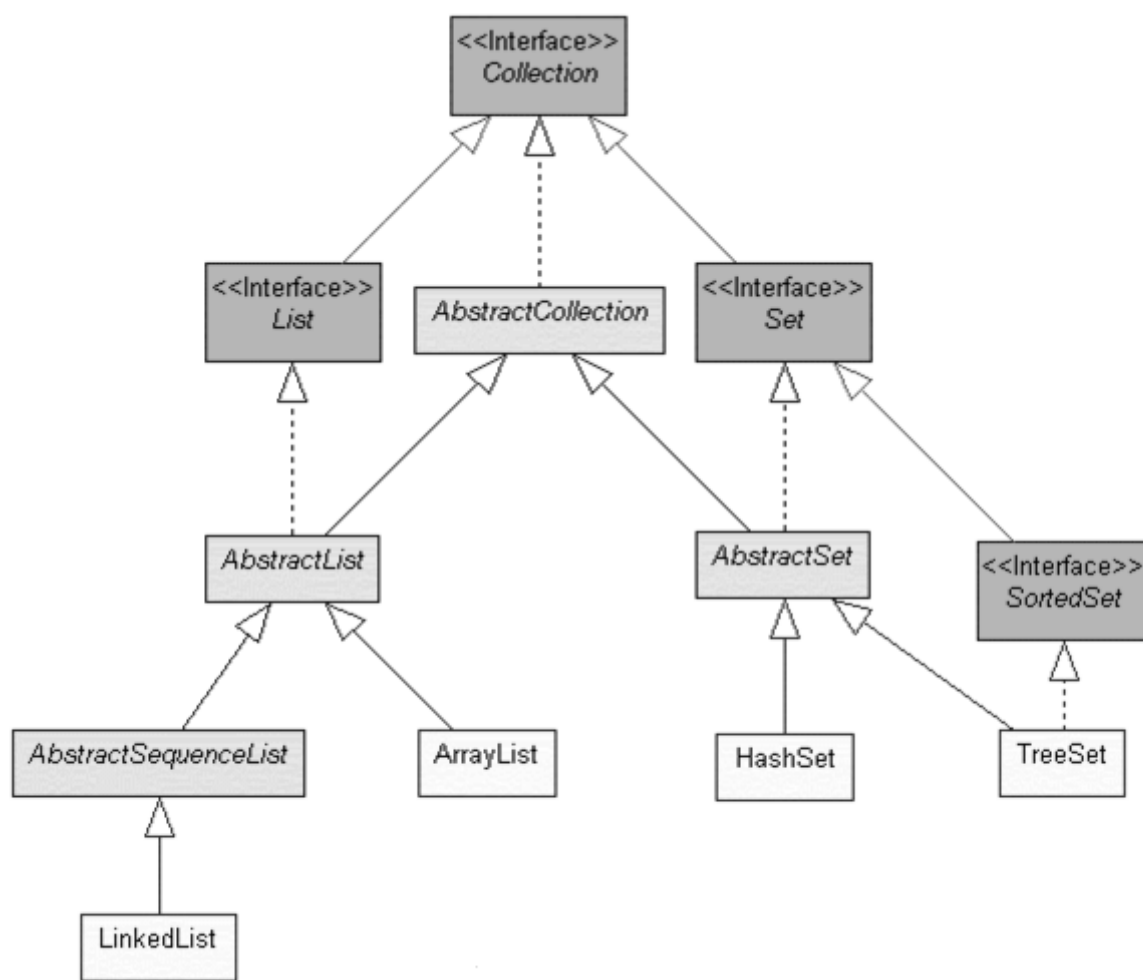


Рисунок 5.1 – Структура коллекций

Таблица 5.1 Интерфейсы коллекций

Интерфейс	Описание
Collection	Дает возможность работать с группами объектов; он находится наверху всей иерархии коллекций
List	Расширяет Collection для обработки последовательностей (списков) объектов
Set	Расширяет Collection для обработки наборов (sets), которые должны содержать уникальные элементы
SortedSet	Расширяет Set для обработки сортированных наборов (sorted sets)

В дополнение к этим интерфейсам, коллекции используют также интерфейсы Comparator, Iterator и ListIterator.

Для обеспечения максимальной гибкости в использовании, некоторые методы в интерфейсах коллекций могут быть необязательными. Необязательные методы дают возможность изменить содержимое коллекции.



Коллекции, которые поддерживают эти методы, называются *изменяемыми* (modifiable). Коллекции, которые не допускают изменения их содержимого, называются *неизменяемыми* (unmodifiable). Если сделана попытка использования одного из необязательных методов на неизменяемой коллекции, выбрасывается исключение `UnsupportedOperationException`. Все встроенные коллекции являются изменяемыми.

Интерфейс *Collection* – это основа, на которой сформирована структура коллекций. В нем объявляются основные методы, которые будут наследоваться всеми коллекциями. Эти методы описаны в таблице 5.2. Поскольку все коллекции реализуют интерфейс *Collection*, знакомство с его методами необходимо для четкого понимания структуры коллекций. Некоторые из методов могут выбрасывать исключение `UnsupportedOperationException`. Это происходит, если коллекция не может быть изменена. Исключение `ClassCastException` генерируется тогда, когда один объект несовместим с другим, т. е. когда осуществляется попытка добавить к коллекции несовместимый объект.

Таблица 5.2. Методы интерфейса *Collection*

Метод	Описание
<code>boolean add(Object obj)</code>	Прибавляет <code>obj</code> к вызывающей коллекции. Возвращает <code>true</code> , если <code>obj</code> был добавлен к коллекции, и <code>false</code> , если <code>obj</code> уже является элементом коллекции или если коллекция не допускает дубликатов
<code>boolean addAll(Collection c)</code>	Добавляет все элементы <code>c</code> к вызывающей коллекции. Возвращает <code>true</code> , если операция закончилась успешно (т. е. все элементы были добавлены). Иначе возвращает <code>false</code>
<code>void clear()</code>	Удаляет все элементы из вызывающей коллекции
<code>boolean contains(Object obj)</code>	Возвращает <code>true</code> , если вызывающая коллекция содержит элемент <code>obj</code> . Иначе – <code>false</code>
<code>boolean containsAll(Collection c)</code>	Возвращает <code>true</code> , если вызывающая коллекция содержит все элементы <code>c</code> . Иначе – <code>false</code>
<code>boolean equals(Object obj)</code>	Возвращает <code>true</code> , если объект вызывающей коллекции и объект <code>obj</code> равны. Иначе – <code>false</code>
<code>int hashCode()</code>	Возвращает хэш-код вызывающей коллекции

Метод	Описание
<code>boolean isEmpty()</code>	Возвращает <code>true</code> , если вызывающая коллекция пуста. Иначе – <code>false</code>
<code>Iterator iterator()</code>	Возвращает итератор ( <code>iterator</code> ) для вызывающей коллекции
<code>boolean remove(object obj)</code>	Удаляет один экземпляр <code>obj</code> из вызывающей коллекции. Возвращает <code>true</code> , если элемент был удален. Иначе – <code>false</code>
<code>boolean removeAll(Collection c)</code>	Удаляет все элементы <code>c</code> из вызывающей коллекции. Возвращает <code>true</code> , если коллекция изменена (т. е. элементы были удалены). Иначе возвращает <code>false</code>
<code>boolean retainAll(Collection c)</code>	Удаляет все элементы из вызывающей коллекции, кроме элементов <code>c</code> . Возвращает <code>true</code> , если коллекция изменена (т. е. элементы были удалены). Иначе возвращает <code>false</code>
<code>int size()</code>	Возвращает число элементов, содержащихся в вызывающей коллекции
<code>Object[ ] toArray()</code>	Возвращает массив, который содержит все элементы, хранящиеся в вызывающей коллекции. Элементы массива являются копиями элементов коллекции
<code>Object[ ] toArray(Object array[ ])</code>	<p>Возвращает массив, содержащий только те элементы коллекции, чей тип согласуется с типом элементов массива <code>array</code>. Элементы этого массива являются копиями элементов коллекции</p> <p>Если размер массива <code>array</code> равен количеству согласованных элементов, они возвращаются прямо в массиве <code>array</code>. Если размер <code>array</code> меньше, чем количество согласованных элементов, то распределяется и возвращается новый массив необходимого размера. Если размер <code>array</code> больше, чем количество согласованных элементов, элемент массива, следующий за последним элементом коллекции, устанавливается в</p>

Метод	Описание
	<i>null</i> . Если любой элемент коллекции имеет тип, который не является подтипом массива <i>array</i> , то выбрасывается исключение <i>ArrayStoreException</i> .

Интерфейс *List* расширяет *Collection* и объявляет поведение коллекции, которая хранит последовательность элементов. Элементы могут быть вставлены или извлечены с помощью их позиций в списке через отсчитываемый от нуля индекс. Список может содержать дублированные элементы.

В дополнение к методам, определенным в *Collection*, интерфейс *List* определяет собственные методы, которые показаны в таблице 5.3. Некоторые из этих методов выбрасывают исключения *UnsupportedOperationException*, если коллекция не может изменяться, и *ClassCastException*, когда один объект несовместим с другим, например, когда делается попытка добавить к коллекции несовместимый объект.

Таблица 5.3 Методы интерфейса *List*

Метод	Описание
<code>void add(int index, Object obj)</code>	Вставляет <i>obj</i> в вызывающий список в позицию с индексом <i>index</i> . Любые элементы, существовавшие ранее в точке вставки или за ней, сдвигаются (т. е. никакие элементы не перезаписываются поверх уже существующих)
<code>boolean addAll (int index, Collection c)</code>	Вставляет все элементы <i>c</i> в вызывающий список с позиции (индекса) <i>index</i> . Любые элементы, существовавшие ранее в точке вставки или за ней, сдвигаются (т. е. никакие элементы не перезаписываются поверх уже существующих). Возвращает <i>true</i> , если вызывающий список изменяется, и <i>false</i> – иначе
<code>Object get(int index)</code>	Возвращает объект, хранящийся в индексной позиции <i>index</i> вызывающей коллекции
<code>indexOf (Object obj)</code>	Возвращает индекс (номер) первого экземпляра объекта <i>obj</i> в вызывающем списке. Если <i>obj</i> – не элемент списка, то возвращается -1
<code>int lastIndexOf (Object obj)</code>	Возвращает индекс последнего экземпляра <i>obj</i> в вызывающем списке. Если <i>obj</i> - не элемент списка, то возвращается -1
<code>ListIterator listIterator()</code>	Возвращает итератор, установленный к началу вызывающего списка
<code>ListIterator listIterator (int index)</code>	Возвращает итератор, установленный к позиции <i>index</i> вызывающего списка

Метод	Описание
Object remove(int index)	Удаляет элемент в позиции index из вызывающего списка и возвращает удаленный элемент. Результирующий список уплотняется (т. е. индексы последующих элементов уменьшаются на 1)
Object set(int index, Object obj)	Устанавливает объект obj S позицию, указанную в index, в вызывающем списке
List subList(int start, int end)	Возвращает список, который включает элементы от номера start до end-1 вызывающего списка. После этого вызывающий объект будет ссылаться на элементы возвращенного (а не исходного) списка

Вызывая метод subList(), можно получить подсписок списка, указывая индексы (номера позиций) начала и окончания подсписка (в исходном списке). Использование этого метода обеспечивает очень удобную обработку списков.

Интерфейс Set расширяет интерфейс Collection и объявляет поведение коллекции, не допускающей дублирования элементов. Поэтому метод add() возвращает false, если осуществляется попытка добавить в набор дублирующие элементы. В Set не определяется никаких дополнительных собственных методов.

Интерфейс SortedSet расширяет Set и объявляет поведение набора, отсортированного в возрастающем порядке. В дополнение к методам, определенным в Set, интерфейс SortedSet объявляет методы, представленные в таблице 5.4.

Некоторые методы выбрасывают исключение NoSuchElementException, когда элементы не содержатся в вызывающем наборе. Исключение ClassCastException выбрасывается, когда объект несовместим с элементами набора, а исключение NullPointerException – если сделана попытка использовать null-объект (null-объекты недопустимы в наборе).

Таблица 5.4 Методы интерфейса SortedSet

Метод	Описание
Comparator comparator()	Возвращает компаратор вызывающего отсортированного набора. Если для этого набора используется естественное упорядочение, то возвращается null (пустая ссылка)
Object first ()	Возвращает первый элемент вызывающего отсортированного набора
SortedSet headSet(Object end)	Возвращает SortedSet-объект, содержащий те элементы вызывающего отсортированного набора, которые меньше, чем end. На эти элементы ссылается также

Метод	Описание
	и вызывающий объект
Object last()	Возвращает последний элемент вызывающего отсортированного набора
SortedSet subset(Object start, Object end)	Возвращает объект SortedSet, который содержит элементы, находящиеся между объектами start и end-1 вызывающего отсортированного списка. На эти элементы ссылается также и вызывающий объект
SortedSet tailSet(Object start)	Возвращает объект SortedSet, который содержит элементы отсортированного набора, больше чем или равные start-объекту. На элементы возвращенного набора ссылается также и вызывающий объект

После ознакомления с интерфейсами коллекций рассмотрим стандартные классы, которые их реализуют. Для некоторых классов обеспечены полные реализации, которые можно использовать непосредственно (т. е. сразу же приступая к созданию их экземпляров). Другие классы являются абстрактными и обеспечивают только схематические реализации, которые служат отправными точками для создания конкретных коллекций. Ни один из коллекционных классов не синхронизирован, но можно получить и синхронные версии. Стандартные классы коллекций перечислены в таблице 5.5

Таблица 5.5 Стандартные классы коллекций

Класс	Описание
AbstractCollection	Реализует большую часть интерфейса Collection
AbstractList	Расширяет AbstractCollection и реализует большую часть интерфейса List
AbstractSequentialList	Расширяет класс AbstractList для реализации коллекции, которая использует последовательный, а не произвольный доступ к ее элементам
LinkedList	Реализует связный список, расширяя класс AbstractSequentialList
ArrayList	Реализует динамический массив, расширяя класс AbstractList
AbstractSet	Расширяет класс AbstractCollection и реализует большую часть интерфейса Set
HashSet	Расширяет класс AbstractSet для реализации хеш-таблиц

Класс	Описание
TreeSet	Реализует набор, хранящийся в виде дерева (древовидный набор). Расширяет класс AbstractSet

### 5.9.1 Класс *ArrayList*

Класс *ArrayList* расширяет *AbstractList* и реализует интерфейс *List*. *ArrayList* поддерживает динамические массивы, которые могут расти по мере необходимости. В Java стандартные массивы имеют фиксированную длину. После того как массивы созданы, они не могут расширяться или сжиматься, что означает, что вы должны знать заранее, сколько элементов массив будет содержать. Но, иногда, вы не можете знать до момента выполнения точного размера массива. Для обработки подобных ситуаций в структуре коллекции определен класс *ArrayList*. По существу, он является массивом объектных ссылок переменной длины. То есть *ArrayList* можно динамически увеличивать или уменьшать в размере. Списки массивов создаются с некоторым начальным размером. Когда этот размер превышает, коллекция автоматически расширяется. Когда объекты удаляются, массив может быть сокращен.

*ArrayList* имеет следующие конструкторы:

*ArrayList()*

*ArrayList (Collection c)*

*ArrayList (int capacity)*

Первый конструктор строит пустой список массива. Второй – список массива, который инициализирован элементами коллекции *c*. Третий формирует список с указанной начальной емкостью (*capacity*). Емкость – это размер (т.е. количество элементов) рассматриваемого массива, который служит для хранения указанного (в параметре *capacity*) количества элементов. Когда элементы добавляются к списку массива, емкость растет автоматически.

*Листинг 5.12*

```
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        ArrayList a = new ArrayList();
        System.out.println("Size " + a.size());
        a.add("123");
        a.add("hello");
        a.add(10);
        System.out.println("Size " + a.size());
        System.out.println(a);
        a.add(1, 100);
        System.out.println("Size " + a.size());
        System.out.println(a);
        a.remove("123");
        System.out.println("Size " + a.size());
    }
}
```

```

        System.out.println(a);
    }
}

```

Обратите внимание, что объект *a* создается пустым и растет по мере добавления в него элементов. При удалении элементов его размер уменьшается. В один *ArrayList* можно помещать объекты разных типов.

В результате выполнения данной программы получим:

```

Size 0
Size 3
[123, hello, 10]
Size 4
[123, 100, hello, 10]
Size 3
[100, hello, 10]

```

### **Задание:**

Напишите методы пересечения и объединения двух коллекций.

## **5.9.2 Класс *LinkedList***

Класс *LinkedList* расширяет *AbstractSequentialList* и реализует интерфейс *List*. Он обеспечивает структуру данных связного списка и имеет два следующих конструктора:

```

LinkedList ()
LinkedList (Collection c)

```

Первый конструктор строит пустой связный список, а второй создает связный список, инициализированный элементами коллекции *c*.

В дополнение к методам, которые он наследует, класс *LinkedList* определяет некоторые собственные полезные методы для манипулирования и доступа к спискам. Чтобы *добавить* Элемент в начало списка, используйте метод *addFirst()*, а для добавления элементов в конец списка вызывайте метод *addLast()*:

```

void addFirst (Object obj)
void addLast (Object obj)

```

Здесь *obj* – добавляемый элемент.

Чтобы *получить* первый элемент, вызовите метод *getFirst ()*, а для извлечения последнего элемента – метод *getLast()*:

```

Object getFirst()
Object getLast()

```

Для удаления первого элемента служит метод *removeFirst()*, а для удаления последнего – *removeLast ()*:

```

Object removeFirst()
Object removeLast()

```

### 5.9.3 Класс *HashSet*

Класс `HashSet` расширяет `AbstractSet` и реализует интерфейс `Set`. Он создает коллекцию, которая использует хеш-таблицу для хранения коллекций. Хеш-таблица хранит информацию, используя механизм называемый хешированием. При *хешировании* информационное содержание ключа используется, чтобы определить уникальное значение, называемое его хэш-кодом. Затем хэш-код применяется как индекс (номер) элемента, в котором хранятся данные, связанные с ключом. Преобразование ключа в его хэш-код выполняется автоматически – вы никогда не видите сам хэш-код. Ваш код не может также прямо индексировать хеш-таблицу. Преимущество хеширования состоит в том, что оно сохраняет постоянным время выполнения основных операций, таких как `add()`, `contains()`, `remove()` и `size()` даже для больших наборов.

В классе определены следующие конструкторы:

`HashSet()`

`HashSet(Collection c)`

`HashSet(int capacity)`

`HashSet (int capacity, float fillRatio)`

Первая форма организует заданный по умолчанию хэш-набор. Вторая инициализирует хэш-набор, используя элементы `c`. Третья создает емкость хэш-набора величиной `capacity`. Четвертая форма инициализирует как емкость (`capacity`), так и отношение (коэффициент) заполнения (`fillRatio`), называемое также емкостью загрузки хэш-набора. Коэффициент заполнения должен быть между 0.0 и 1.0, и он определяет, насколько полным может быть хэш-набор, прежде чем он увеличивает свой размер. Когда число элементов больше, чем емкость хэш-набора, умноженная на его коэффициент заполнения, хэш-набор расширяется. Для конструкторов, которые не принимают коэффициент заполнения, используется множитель 0.75.

`HashSet` не определяет никаких дополнительных методов, кроме тех, что обеспечены его суперклассами и интерфейсами.

Важно обратить внимание, что хэш-набор не гарантирует упорядочивания его элементов, потому что процесс хеширования не предназначен для создания сортируемых наборов. Если вам нужна сортируемая память, то лучше выбрать другую коллекцию, такую как `TreeSet`.

*Листинг 5.13*

```
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet a = new HashSet();
        a.add("123");
        a.add("hello");
        a.add(10);
        a.add(5);
        a.add("A");
    }
}
```



```

        System.out.println(a);
    }
}

```

В результате выполнения данной программы получим:  
[hello, A, 5, 123, 10]

#### 5.9.4 Класс *TreeSet*

Класс *TreeSet* обеспечивает реализацию интерфейса *Set* и использует иерархическую (древовидную) структуру для хранения данных. Объекты хранятся в отсортированном по возрастанию порядке. Время доступа и поиска в такой структуре не велико, что делает *TreeSet* превосходным выбором для хранения больших количеств отсортированной информации, которая должна быть быстро найдена.

В классе определены следующие конструкторы:

```

TreeSet ()
TreeSet(Collection c)
TreeSet(Comparator comp)
TreeSet(SortedSet ss)

```

Первая форма создает пустой древовидный набор, который будет сортироваться в восходящем порядке согласно естественному порядку его элементов. Вторая – формирует древовидный набор, который содержит элементы коллекции *c*. Третья форма создает пустой древовидный набор, который будет сортироваться в соответствии с компаратором *comp*. Четвертая – строит древовидный набор, который содержит элементы отсортированного набора *ss*.

*Листинг 5.14*

```

import java.util.TreeSet;
public class Main {
    public static void main(String[] args) {
        TreeSet a = new TreeSet();
        a.add("123");
        a.add("hello");
        a.add(10);
        a.add(5);
        a.add("A");
        System.out.println(a);
    }
}

```

Если выполнить данный код, то получим:

```

Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to java.lang.Integer
    at java.lang.Integer.compareTo(Integer.java:52)
    at java.util.TreeMap.put(TreeMap.java:560)

```

```
at java.util.TreeSet.add(TreeSet.java:255)
at javaapplication2.Main.main(Main.java:9)
```

Данная ошибка возникла из-за того что невозможно отсортировать объекты разных классов, т.е. в TreeSet нужно помещать все объекты одного класса.

*Листинг 5.15*

```
import java.util.TreeSet;
public class Main {
    public static void main(String[] args) {
        TreeSet a = new TreeSet();
        a.add("123");
        a.add("hello");
        a.add("people");
        a.add("cat");
        a.add("10");
        TreeSet b = new TreeSet();
        b.add(10);
        b.add(5);
        b.add(20);
        b.add(1);
        System.out.println("a: "+a);
        System.out.println("b: "+b);
    }
}
```

В ходе выполнения данной программы получим:

a: [10, 123, cat, hello, people]

b: [1, 5, 10, 20]

Видим, что объекты отсортировались в, так называемом, естественном порядке: строки – по алфавиту, числа – по возрастанию.

Естественный порядок сортировки можно задать для любого класса. Для этого достаточно реализовать параметризованный интерфейс Comparable и его метод `public int compareTo(Object o)`. Данный метод возвращает число, но само число роли не играет. Важен знак этого числа. Минус соответствует знаку больше, плюс – меньше и 0 – равно.

*Листинг 5.16*

```
public class Circle implements Comparable<Circle> {
    private int radius;
    public Circle(int radius) {
        this.radius = radius;
    }
    @Override
    public String toString() {
        return "Circle{" + "radius=" + radius + '}';
    }
    @Override
```

```

    public int compareTo(Circle o) {
        return this.radius - o.radius;
    }
}

import java.util.TreeSet;
public class Main {
    public static void main(String[] args) {
        TreeSet<Circle> a = new TreeSet<Circle>();
        a.add(new Circle(5));
        a.add(new Circle(3));
        a.add(new Circle(10));
        a.add(new Circle(15));
        a.add(new Circle(2));
        System.out.println("a: " + a);
    }
}

```

В результате выполнения данной программы получили:

a: [Circle{radius=2}, Circle{radius=3}, Circle{radius=5}, Circle{radius=10}, Circle{radius=15}]

Т.е. все объекты автоматически отсортировались по возрастанию радиуса.

### 5.9.5 Доступ к коллекции через итератор

Часто нужно циклически проходить элементы в коллекции, например, для отображения их на экране. Самый простой способ сделать это – использовать *итератор* (iterator), т.е. специальный объект, который реализует один из интерфейсов – либо `Iterator`, либо `ListIterator`.

Интерфейс `Iterator` дает возможность циклически пройти через коллекцию, получая или удаляя ее элементы. Интерфейс `ListIterator` расширяет `Iterator`, обеспечивая двунаправленный обход списка, и модификацию элементов. В интерфейсе `Iterator` объявляются методы, перечисленные в таблице 5.6. Методы, объявленные в `ListIterator`, показаны в таблице 5.7.

Таблица 5.6 Методы интерфейса `Iterator`

Метод	Описание
<code>boolean hasNext()</code>	Возвращает <code>true</code> , если в коллекции присутствует следующий элемент. Иначе возвращает <code>false</code>
<code>Object next()</code>	Возвращает следующий элемент. Выбрасывает исключение типа <code>NoSuchElementException</code> , если следующего элемента нет
<code>void remove()</code>	Удаляет текущий элемент. Выбрасывает исключение типа <code>IllegalStateException</code> , если сделана попытка вызвать метод <code>remove()</code> , которой не

Метод	Описание
	предшествует вызов next()

Таблица 5.7 Методы интерфейса ListIterator

Метод	Описание
void add (Object obj)	Вставляет obj в список перед элементом, который будет возвращен следующим вызовом next()
boolean hasNext()	Возвращает true, если в коллекции присутствует следующий элемент. Иначе возвращает false
boolean hasPrevious()	Возвращает true, если существует предыдущий элемент. Иначе возвращает false
Object next()	Возвращает следующий элемент. Выбрасывает исключение типа NoSuchElementException, если следующего элемента нет
int nextIndex()	Возвращает индекс следующего элемента. Если следующего элемента нет, возвращает размер списка
Object previous()	Возвращает предыдущий элемент. Выбрасывается исключение типа NoSuchElementException, если предыдущего элемента нет
int previousIndex()	Возвращает индекс предыдущего элемента. Если предыдущего элемента нет, возвращает -1
void remove()	Удаляет текущий элемент из списка. Выбрасывается исключение типа IllegalStateException, если метод remove() вызывается, прежде чем вызван метод next() или previous()
void set (Object obj)	Назначает obj на текущий элемент. Это последний элемент, возвращенный вызовом метода next() или previous()

Перед обращением к коллекции через итератор вы должны получить его. В каждом из коллекционных классов определен метод iterator(), который возвращает итератор к началу коллекции. Используя этот итерационный объект, вы можете обращаться к каждому элементу в коллекции поодиночке. Вообще, чтобы применять итератор для циклического прохода содержимого коллекции, выполните следующие действия:

1. Вызывая коллекционный метод iterator(), получите итератор, чтобы стартовать коллекцию.

2. Установите цикл, который делает обращение к методу hasNext(). Повторяйте итерации, пока hasNext() возвращает true.

3. Внутри цикла, получайте каждый элемент коллекции, вызывая метод next().

Для коллекций, которые реализуют интерфейс List, можно таким же способом получать итератор, вызывая методы интерфейса ListIterator. Списковый итератор дает возможность обращаться к коллекции в прямом или в обратном направлении, а также позволяет изменять элемент коллекции. Иначе говоря, ListIterator применяется точно также, как Iterator.

Ниже показан пример, который реализует описанные выше действия и демонстрирует интерфейсы Iterator и ListIterator. Он использует объект типа ArrayList, но общие принципы применимы к коллекции любого типа. Конечно, интерфейс ListIterator доступен только тем коллекциям, которые реализуют интерфейс List.

*Листинг 5.17*

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;
public class Main {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        // использовать итератор для показа содержимого объекта al
        System.out.print("Исходное содержимое al: ");
        Iterator itr = al.iterator();
        while (itr.hasNext()) {
            Object element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();
        // модифицировать итерируемые объекты
        ListIterator litr = al.listIterator();
        while (litr.hasNext()) {
            Object element = litr.next();
            litr.set(element + "+");
        }
        System.out.print("Модифицированное содержимое al: ");
        itr = al.iterator();
        while (itr.hasNext()) {
            Object element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();
        // теперь показать список в обратном порядке
```

```

System.out.print("Модифицированный (обратный) список: ");
while (litr.hasPrevious()) {
    Object element = litr.previous();
    System.out.print(element + " ");
}
System.out.println();
}
}

```

В результате выполнения данной программы получим:

Исходное содержимое al: C A E B D F

Модифицированное содержимое al: C+ A+ E+ B+ D+ F+

Модифицированный (обратный) список: F+ D+ B+ E+ A+ C+

### 5.9.6 Алгоритмы коллекций

Структура коллекций содержит несколько алгоритмов, которые могут применяться к коллекциям. Эти алгоритмы определены как статические методы в классе `collections` и описаны в таблице 5.8.

Таблица 5.8 Алгоритмы класса `Collections`

Метод	Описание
<code>static int binarySearch(List list, Object value, Comparator c)</code>	Отыскивает объект <code>value</code> из списка <code>list</code> , упорядоченного с помощью компаратора <code>c</code> . Возвращает позицию <code>value</code> в списке <code>list</code> , или <code>-1</code> , если значение не найдено
<code>static int binarySearch(List list, Object value)</code>	Отыскивает объект <code>value</code> в списке <code>list</code> . Список должен быть отсортированным. Возвращает позицию <code>value</code> в списке <code>list</code> , или <code>-1</code> , если значение не найдено
<code>static void copy(List list1, List list2)</code>	Копирует элементы <code>list2</code> в <code>list1</code>
<code>static Enumeration enumeration(Collection c)</code>	Возвращает перечисление коллекции <code>c</code>
<code>static void fill(List list, Object obj)</code>	Присваивает объект <code>obj</code> каждому элементу списка <code>list</code>
<code>static Object max(Collection c, Comparator comp)</code>	Возвращает максимальный элемент коллекции <code>c</code> , использующей компаратор <code>comp</code>
<code>static Object max(Collection c)</code>	Возвращает максимальный элемент коллекции <code>c</code> , использующей естественное упорядочение. Коллекция может быть несортированной
<code>static Object min(Collection c, Comparator comp)</code>	Возвращает минимальный элемент коллекции <code>c</code> , использующей компаратор <code>comp</code> . Коллекция может быть

Метод	Описание
	несортированной
static Object min (Collection c)	Возвращает минимальный элемент коллекции c, использующей естественное упорядочение
static List nCopies(int num, Object obj)	Возвращает num копий объекта obj в форме неизменяемого списка, num должен быть больше или равен нулю
static void reverse (List list)	Реверсирует последовательность списка list
static Comparator reverseOrder()	Возвращает обратный компаратор (компаратор, который реверсирует вывод результата сравнения двух элементов)
static void shuffle(List list, Random r)	Перетасовывает (т. е. рандомизирует) элементы в списке list, используя r как источник случайных чисел
static void shuffle(List list)	Перетасовывает (т. е. рандомизирует) элементы в списке list (с рандомизатором по умолчанию)
static Set singleton(Object obj)	Возвращает obj как неизменяемый набор. Это простой способ преобразования одиночного объекта в набор
static void sort(List list, Comparator comp)	Сортирует элементы списка list по правилам компаратора comp
static void sort(List list)	Сортирует элементы списка list в естественном порядке
static Collection synchronizedCollection (Collection c)	Возвращает синхронизированную (поточно-безопасную) коллекцию, поддерживаемую коллекцией c
static List synchronizadList (List list)	Возвращает синхронизированный (поточно-безопасный) список, поддерживаемый списком list
static Map synchronorizedMap(Map m)	Возвращает синхронизированную (поточно-безопасную карту) отображений, поддерживаемую картой m
static Set synchronizedSet(Set s)	Возвращает синхронизированный (поточно-безопасный) набор, поддерживаемый набором s
static SortedMap synchronizedSortedMap (SortedMap sm)	Возвращает синхронизированную (поточно-безопасную) отсортированную карту, поддерживаемую отсортированной картой sm

Метод	Описание
static SortedSet synchxonizedSortedSat(SortedSet ss)	Возвращает синхронизированный (поточно-безопасный) отсортированный набор, поддерживаемый отсортированным набором ss
static Collection unmodifiableCollection (Collection c)	Возвращает неизменяемую коллекцию, поддерживаемую коллекцией c
static List unmodifiableList (List list)	Возвращает неизменяемый список, поддерживаемый списком list
static Map unmodifiableMap (Map m)	Возвращает неизменяемую карту отображений, поддерживаемую картой m
static Set unmodifiableSet (Set a)	Возвращает неизменяемый набор, поддерживаемый набором s
static SortedMap unmodifiableSortedMap (SortedMap sm)	Возвращает неизменяемую отсортированную карту отображений, поддерживаемую отсортированной картой sm
static SortedSet unmodifiableSortedSet(SortedSet ss)	Возвращает неизменяемый отсортированный набор, поддерживаемый отсортированным набором ss

Некоторые из методов могут выбрасывать исключения `ClassCastException`, которые происходят при попытке сравнения несовместимых типов, или исключения `UnsupportedOperationException`, возникающие во время изменения неизменяемой коллекции.

Коллекции представляют собой обширную группу классов дающих много возможностей для хранения и обработки объектов. В рамках данного методического пособия была рассмотрена лишь малая часть этих классов. Рекомендуем продолжить изучение классов коллекций по документации.



## **Выводы к главе:**

- *Универсальные типы – это совершенно новая синтаксическая конструкция, появление которой вызвало существенные изменения во многих классах и методах базового API.*
- *Универсальные типы позволяют создавать классы, интерфейсы и методы, для которых типы обрабатываемых данных передаются в качестве параметра.*
- *Классы, интерфейсы или методы, которые обрабатывают типы, передаваемые посредством параметров, называются универсальными.*
- *При определении экземпляра универсального класса передаваемый тип, который заменяет параметр типа, должен быть именем класса. Для этой цели нельзя использовать простой тип, например `int` или `char`.*
- *Групповой параметр можно ограничить при помощи ключевого слова `extends`.*
- *Класс может содержать несколько параметров одновременно.*
- *Не параметризованный класс может содержать один или несколько параметризованных методов. Конструктор может быть параметризован.*
- *В параметризованном классе не может быть статических полей и методов использующих параметр.*
- *Универсальный класс не может расширять класс `Throwable`. Это означает, что создать универсальный класс исключения невозможно.*
- *Коллекция – это группа объектов.*
- *Интерфейс `Collection` – это основа, на которой сформирована структура коллекций. В нем объявляются основные методы, которые будут наследоваться всеми коллекциями.*
- *Алгоритмы – важная часть механизма коллекций. Алгоритмы работают на коллекциях и определены как статические методы в классе `Collections`.*

### Задания к главе:

1). Напишите класс `Student`, предоставляющий информацию об имени студента методом `getName()` и о его курсе методом `getCourse()`. Напишите метод `printStudents(LinkedList students, int course)`, который получает список студентов и номер курса и печатает в консоль имена тех студентов из списка, которые обучаются на данном курсе. Протестируйте ваш метод (для этого предварительно придется создать десяток объектов класса `Student` и поместить их в список). Напишите методы `union(LinkedList set1, LinkedList set2)` и `intersect(LinkedList set1, LinkedList set2)`, реализующих операции объединения и пересечения двух множеств. Протестируйте работу этих методов на двух предварительно заполненных множествах. (Вам понадобится написать вспомогательный метод, выводящий все элементы множества на консоль.) Реализуйте интерфейс `Comparable` так, чтобы студенты сортировались по номеру курса. Проверить работу, используя класс `TreeSet`.

2). Не используя вспомогательных объектов, переставить отрицательные элементы данного списка в конец, а положительные - в начало этого списка.

3). В кругу стоят  $N$  человек, пронумерованных от 1 до  $N$ . При ведении счета по кругу вычеркивается каждый второй человек, пока не останется один. Составить две программы, моделирующие процесс. Одна из программ должна использовать класс `ArrayList`, а вторая - `LinkedList`.

4). Задан список целых чисел и число  $X$ . Не используя вспомогательных объектов и не изменяя размера списка, переставить элементы списка так, чтобы сначала шли числа, не превосходящие  $X$ , а затем числа, большие  $X$ .

5). Выполнить попарное суммирование произвольного конечного ряда чисел по следующим правилам: на первом этапе суммируются попарно рядом стоящие числа, на втором этапе суммируются результаты первого этапа и т.д. до тех пор, пока не останется одно число.

6). На базе коллекций реализовать структуру хранения чисел с поддержкой следующих операций:

- добавление/удаление числа;
- поиск числа, наиболее близкого к заданному (т.е. модуль разницы минимален)

## ЗАКЛЮЧЕНИЕ

В настоящее время Java один из наиболее популярных и перспективных языков, область применения которого с каждым днем становится все шире. Изучив данное методическое пособие, Вы получили лишь базовые знания по JavaSE – сделали первый шаг на пути изучения этого широко востребованного языка. Изучив основы языка, у Вас есть два варианта продолжения Вашего обучения и развития:

- изучить JavaEE на «Продвинутом курсе по Java» и заниматься разработкой под web;

- изучить JavaME на «Программировании под Android» и заниматься разработкой приложений под платформу Android.

Каждый выбирает то, что ближе ему. Главное не останавливаться в развитии и продолжить обучение, набраться опыта и стать успешным Java-программистом.

Если у Вас возникли какие-либо вопросы, замечания или предложения по данному методическому пособию, то присылайте их на адрес [KarsekaEV@tut.by](mailto:KarsekaEV@tut.by).

С уважением, Елена Карсека

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1) Г.Шилдт. Полный справочник по Java. - Изд. дом “Вильямс”,2007
- 2) Б.Эккель. Философия Java. – Изд-во. Питер, 2009
- 3) К.С.Хорстманн, Г.Корнелл. Java 2. Том 1. Основы. - Изд. дом “Вильямс”,2003
- 4) К.С.Хорстманн, Г.Корнелл. Java 2. Том 2. Тонкости программирования. - Изд. дом “Вильямс”,2003
- 5) И.Н.Блинов, В.С.Романчик. Промышленное программирование: практич. Пособие –Изд. УниверсалПресс, Минск, 2007

