МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

**И. В. Курбатова, А. В. Печкуров**

# ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

*Учебное пособие*

Ч а с т ь 1

## Основы синтаксиса и его применение в объектно-ориентированном программировании

Воронеж
Издательский дом ВГУ
2019

УДК 004.432.2
ББК 32.973.2
К93

Р е ц е н з е н т ы:
доктор физико-математических наук, профессор *И. П. Половинкин*,
доктор физико-математических наук, профессор *Ю. А. Юраков*

**Курбатова И. В.**

К93 Язык программирования Java : учебное пособие/ И. В. Курбатова, А. В. Печкуров ; Воронежский государственный университет — Воронеж : Издательский дом ВГУ, 2019.

ISBN 978-5-9273-2790-4

Ч. 1 : Основы синтаксиса и его применение в объектно-ориентированном программировании. — 78 с.

ISBN 978-5-9273-2791-1

Учебное пособие подготовлено на кафедре программного обеспечения и администрирования информационных систем факультета прикладной математики, механики и информатики Воронежского государственного университета.

Рекомендовано для студентов 3 курса факультета прикладной математики, механики и информатики Воронежского государственного университета.

УДК 004.432.2
ББК 32.973.2


ISBN 978-5-9273-2791-1 (ч.1)

ISBN 978-5-9273-2790-4

# Оглавление

# Введение

Сложно переоценить важность объектно-ориентированного программирования (ООП) для мира разработки программного обеспечения. Объекты и коммуникация, происходящая между ними, — это выразительный и интуитивно понятный способ моделирования систем и процессов. В их терминах можно описать программы любой сложности. Парадигма ООП сформировалась много десятилетий назад и существенно повлияла на развитие языков программирования. Такие языки, как C$^{++}$, C$^{\#}$ и Java, являются наиболее яркими представителями парадигмы ООП.

Данное пособие посвящено языку Java, появившемуся в далеком 1995 году и уже третье десятилетие остающемуся актуальным. Не даром этот язык находится в глобальной десятке наиболее популярных языков программирования. На Java написано огромное количество разнообразных приложений, от банковского ПО до встраиваемых систем. Благодаря лаконичному синтаксису и гибкости Java позволяет моделировать и решать разнообразные математические задачи (см., например, [3]). Богатейшая экосистема и производительная, стабильная платформа — это то, за что разработчики выбирают данный язык для своих проектов. Именно поэтому Java в качестве основного языка программирования — это отличный выбор для студентов и начинающих разработчиков.

Языку Java посвящено множество книг [1; 4 – 11] но, как правило, эти книги требуют достаточно высокого уровня подготовки и содержат избыточный материал. Экосистема Java включает великое множество библиотек и технологий, что осложняет задачу формирования необходимого набора знаний для начинающего программиста. В этом учебном курсе содержится такой необходимый набор знаний. Курс преследует задачу научить читателей основам языка Java и его стандартным библиотекам, а также наиболее популярным технологиям. Изложение дополнено множеством практических рекомендаций от разработчика с более чем десятилетним стажем. Подразумевается, что читатели уже знакомы с одним из процедурных языков программирования, например C или Pascal, и знают основные понятия этих языков, такие как оператор, выражение, блок, цикл и т. д.

Не следует также забывать, что содержательной стороной любого программирования являются данные и алгоритмы их обработки [2]. Оптимальный выбор структур данных и связанных с ними алгоритмов существенно влияет на такие качества программ, как производительность и

потребление памяти. Освещаемые в курсе широкие возможности стандартной библиотеки Java помогут читателю сделать такой выбор во многих случаях.

Данное пособие состоит из нескольких частей. В первой части описывается основной синтаксис языка Java, его ключевые слова и конструкции. Материал подается шаг за шагом, от простого (процедурные конструкции языка) к более сложному (основы ООП). Все понятия закрепляются на наглядных примерах с комментариями, которые студенты могут написать, запустить и осмыслить самостоятельно.

# Глава 1

# Основы синтаксиса

## 1 Историческая справка

Java — строго типизированный объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последующем приобретенной компанией Oracle). Приложения Java транслируются в специальный байт-код, поэтому они могут работать на любой компьютерной архитектуре с помощью виртуальной Java-машины. Дата официального выпуска — 23 мая 1995 года.

Изначально язык назывался Oak («Дуб»), разрабатывался Джеймсом Гослингом для программирования бытовых электронных устройств. Впоследствии он был переименован в Java и стал использоваться для написания клиентских приложений и серверного программного обеспечения. Назван в честь марки кофе Java, которая, в свою очередь, получила наименование от одноименного острова (Ява), поэтому на официальной эмблеме языка изображена чашка с горячим кофе. Простейший пример программы на языке Java выглядит так:

```
public class HelloWorld {

    public static void main(String... args) {
        System.out.print("Hello world!");
    }

}
```

Как уже говорилось, программы, написанные на Java, транслируются в байт-код, выполняемый виртуальной машиной Java (JVM) — программой,

обрабатывающей байтовый код и передающей инструкции процессору как интерпретатор.

Достоинством такого способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, на котором имеется соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

В качестве примера приведем текстовое представление байт-кода, создаваемого предыдущей программой:

```
// class version 52.0 (52)
// access flags 0x21
public class HelloWorld {

  // compiled from: HelloWorld.java

  // access flags 0x1
  public <init>()V
   L0
    LINENUMBER 4 L0
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init> ()V
    RETURN
   L1
    LOCALVARIABLE this LHelloWorld; L0 L1 0
    MAXSTACK = 1
    MAXLOCALS = 1

  // access flags 0x89
  public static varargs main([Ljava/lang/String;)V
   L0
    LINENUMBER 7 L0
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
```

```
    LDC "Hello world!"
    INVOKEVIRTUAL java/io/PrintStream.print
                  (Ljava/lang/String;)V
   L1
    LINENUMBER 8 L1
    RETURN
   L2
    LOCALVARIABLE args [Ljava/lang/String; L0 L2 0
    MAXSTACK = 2
    MAXLOCALS = 1
}
```

## 2 Версии

Основными версиями языка Java с точки зрения этого учебного курса являются следующие.

JDK 1.0. Официальная версия была выпущена только 21 января 1996 года, хотя разработка Java началась в 1990 году.

Java SE 6. Релиз версии состоялся 11 декабря 2006 года. Эта версия является до сих пор актуальной и используется во многих проектах.

Java SE 9. Релиз версии состоялся 21 сентября 2017 года. На текущий момент эта версия является последней.

В рамках нашего базового курса языка Java мы будем рассматривать в полном объеме синтаксис и возможности Java 6 и некоторые нововведения, добавленные в более поздних версиях.

## 3 Классификация платформ Java

Внутри Java имеется несколько основных семейств технологий.

Java SE — Java Standard Edition, основное издание Java, содержит компиляторы, API, Java Runtime Environment; подходит для создания пользовательских приложений, в первую очередь — для настольных систем.

Java EE — Java Enterprise Edition, представляет собой набор спецификаций для создания программного обеспечения уровня предприятия.

Java ME — Java Micro Edition, создана для использования в устройствах, ограниченных по вычислительной мощности, например в мобильных телефонах и встроенных системах.

Java Card — технология предоставляющая безопасную среду для приложений, работающих на смарт-картах и других устройствах с очень ограниченным объемом памяти и возможностями обработки.

Java VM или JVM — Java Virtual Machine, виртуальная машина Java, основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java интерпретирует и исполняет байт-код, предварительно созданный из исходного текста Java-программы компилятором.

# 4 Краткая Java-терминология

- Виртуальная машина (*virtual machine*) — абстрактное вычислительное устройство, которое может быть реализовано разными способами: аппаратно или программно. Компиляция в набор команд виртуальной машины происходит почти так же, как и компиляция в набор команд микропроцессора.

- Java-платформа (*Java Platform*) — виртуальная машина Java, к которой добавлены стандартные классы. Java-платформа предоставляет программам унифицированный интерфейс независимо от операционной системы, на которой они работают.

- Среда исполнения Java (*Java Development Kit*, JDK). Комплект разработчика приложений на языке Java, включающий в себя компилятор Java (javac), стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему Java (JRE).

- Среда исполнения Java (*Java Runtime Environment*, JRE). Подмножество Java Development Kit, предназначенное для конечных пользователей. JRE состоит из виртуальной машины Java (JVM), стандартных классов Java и вспомогательных файлов.

- Виртуальная машина Java (*Java Virtual Machine*, JVM), часть среды исполнения Java, отвечающая за интерпретацию Java байт-кода. JVM состоит из набора команд байт-кода, набора регистров, стека, сборщика мусора и пространства для хранения методов.

- Java байт-код (*Java bytecode*) — машинно-независимый код, который генерирует Java-компилятор. Байт-код выполняется Java-интерпре-

татором. Виртуальная машина Java полностью стековая; это приводит к тому, что не требуется сложная адресация ячеек памяти и большое количество регистров. Поэтому команды JVM короткие, большинство из них имеет длину 1 байт, по этой причине команды JVM называют байт-кодами, хотя имеются команды длиной 2 и 3 байта (средняя длина команды составляет 1,8 байта). Напомним, что программа, написанная на языке Java, переводится компилятором в байт-код. Байт-код записывается в одном или нескольких файлах, может храниться во внешней памяти или передаваться по сети. Это особенно удобно благодаря небольшому размеру файлов с байт-кодом. Полученный в результате компиляции байт-код можно выполнять на любом компьютере, имеющем систему, реализующую JVM (вне зависимости от типа конкретного процессора и архитектуры ПК). Так реализуется принцип Java: «Write once, run anywhere» («Пишется однажды, выполняется всюду»).

# 5 Основы синтаксиса

Опишем вкратце основные понятия и идеи синтаксиса языка Java. В последующих параграфах они будут обсуждаться подробнее.

*Пакет.* Механизм, позволяющий организовать Java-классы в пространстве имен. Обычно в пакеты объединяют классы одной и той же категории либо предоставляющие сходную функциональность. Каждый пакет предоставляет уникальное пространство имен для своего содержимого. Допустимы вложенные пакеты.

*Объект.* Объекты имеют состояние и поведение. Например: собака может иметь состояние — цвет, имя, а также поведение — бежать, лаять, есть. Объект является *экземпляром класса.*

*Класс* может быть определен как шаблон, который описывает поведение объекта. Классы будут подробно рассмотрены ниже.

*Метод* — именованный блок кода с входными параметрами и возвращаемым значением, описывающий поведение объекта. Класс может содержать несколько методов. Методы манипулируют данными (полями объекта и локальными переменными) и выполняют все остальные действия.

*Переменные экземпляра* (поля). Каждый объект имеет свой уникальный набор переменных экземпляра. Состояние объекта определяется значениями, присвоенными этим переменным экземпляра.

*Переменные метода* (локальные переменные). Объявленные в методе переменные. Доступны только в пределах метода.

Java-программа может быть определена как совокупность объектов, которые взаимодействуют путем вызова методов друг друга. В данной главе мы рассмотрим только часть базового синтаксиса Java, минимально необходимую для написания простых программ.

## 5.1 Правила именования

Приведем основные правила именования сущностей в Java.

- *Пакеты.* Разделенные точками (`"."`) слова на английском языке. Обычно в качестве имени пакета используют имя Интернет-домена автора (например, компании), расположенное в обратном порядке: для компании с доменом `example.com` все пакеты принято начинать с префикса `com.example`. Примеры: `ru.vsu.amm`, `com.example.util`, `java.lang`.

- *Классы.* Используются существительные на английском языке в смешанном регистре начиная с большой буквы. Примеры: `DogType`, `Rectangle`, `SimpleDocument`.

- *Методы.* Используются глаголы на английском языке в смешанном регистре начиная с маленькой буквы. Примеры: `drawOnCanvas()`, `run()`, `getDocument()`.

- *Переменные и поля.* Используются содержательные и краткие названия на английском языке в смешанном регистре начиная с маленькой буквы. Примеры: `legCount`, `title`, `width`.

- *Константы.* Используются содержательные и краткие названия на английском языке в верхнем регистре. Слова разделяются нижним подчеркиванием (`"_"`). Примеры: `MIN_HEIGHT`, `CORES_COUNT`, `MAX_INT`.

Отметим также, что исходный код в Java допускает хранение в кодировке UTF-8 (Unicode), что позволяет использовать любые символы в строковых литералах и даже в именах переменных, методов и т. д. (хотя последнее считается не очень хорошим тоном).

## 5.2 Комментарии

Комментарии в языке Java, как и в большинстве других языков программирования, игнорируются при выполнении программы. Таким образом, в программу можно добавлять столько комментариев, сколько потребуется, не опасаясь повлиять на байт-код.

В языке Java имеются два способа вставки комментариев в текст. Чаще всего используются две косые черты //, они означают, что комментарий начинается сразу за символами // и продолжается до конца строки.

Если требуются более длинные комментарии, можно каждую строку начинать символами //. Но удобнее ограничивать блоки комментариев разделителями /* и */. Например,

```
public class HelloWorld {

    public static void main(String... args) {
        System.out.print("Hello world!");
        // это однострочный комментарий
        /* а это комментарий
        на несколько
        строк */
    }
}
```

В Java имеется способ документирования кода через комментарии специального вида в формате Javadoc. Написанная документация генерируется в виде отдельного HTML-файла специальной одноименной утилитой. Комментарии в этом формате описывают публичную часть кода: классы, интерфейсы, методы и т. д.

Пример документированного метода:

```
/**
 * Returns an Image object that can then be painted on the
 * screen. The url argument must specify an absolute
 * {@link URL}. The name argument is a specifier that is
 * relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
```

```
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of
 *              the image
 * @param  name the location of the image, relative to the url
 *              argument
 * @return      the image at the specified URL
 * @see         Image
 */
 public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

### 5.3 Методы

Напомним, что метод — это именованный блок кода с входными параметрами (аргументами) и возвращаемым значением, описывающий поведение объекта. Для определения методов используется следующий формат:

```
возвращаемый-тип идентификатор-метода(параметры) {
    тело-метода
}
```

*Возвращаемый тип* определяет тип данных, которые возвращает метод при вызове. Если метод не возвращает никакого значения, то возвращаемый тип имеет значение `void` (пустой).

*Идентификатор метода* определяет имя метода, а *параметры* — список аргументов, которые необходимо передать методу при его вызове. Параметры в списке отделяются друг от друга запятыми. Если параметры отсутствуют, после идентификатора-метода указывают пустые круглые скобки. Совокупность имени метода и набора его параметров называют его *сигнатурой*. Обратите внимание, что возвращаемое значение не входит в сигнатуру метода.

*Тело метода* содержит операторы, реализующие действия, выполняемые данным методом. Если возвращаемый тип не `void`, в теле метода должен быть хотя бы один оператор `return` выражение; при этом тип выражения должен совпадать с типом возвращаемого значения. Этот оператор возвращает результат вычисления выражения в точку вызова метода. Если тип возвращаемого значения — `void`, возврат из метода выполняется либо после выполнения последнего оператора тела метода, либо в результате выполнения оператора `return;` (таких операторов в теле метода может быть несколько).

Приведем пример простого метода, считающего сумму двух целых чисел:

```
public int sum(int a, int b) {
    int sum = a + b;
    return sum;
}
```

Java позволяет группировать два и более операторов в блоки кода, называемые также *кодовыми блоками*. Это осуществляется путем помещения операторов в фигурные скобки { и }. Сразу после создания блок кода становится логическим модулем, который можно использовать так же, как и отдельный оператор. Например, блок может служить в качестве тела для операторов `if` и `for`, которые будут рассмотрены далее.

В данной главе мы будем в основном использовать статические методы классов. Их можно трактовать как аналог процедур и функций в таких языках программирования, как Pascal и C.

## 5.4 Пример простой консольной программы

В качестве точки входа каждой Java-программы служит статический метод `main()`. Он может быть объявлен в любом классе (и файле, соответственно), главное, чтобы его сигнатура была такой, как показано в следующем примере:

```
public class MainExample {

    public static void main(String[] args) {
        // это стартовая точка программы
```

```
        if (args.length > 0) {
            System.out.println("Первый аргумент=" + args[0]);
        }
    }
}
```

Скомпилировать и запустить такой файл можно двумя следующими командами:

```
javac MainExample.java
java MainExample
```

Обратите внимание на то, что у `main`-метода имеется входной аргумент `String[] args`. В него будет передан массив со значениями аргументов командной строки, с которыми была запущена программа. Например, запустим рассмотренный выше пример следующим образом:

```
javac MainExample.java
java MainExample "Test argument value"
```

Последняя команда выведет в консоль следующее:

```
Первый аргумент=Test argument value
```

Многие примеры из данного курса не содержат `main`-метода по соображениям краткости изложения, однако подразумевают, что в конечной программе, исполняющей данный код, `main`-метод будет присутствовать.

## 5.5 Переменные

*Переменная* — это именованная область памяти, адрес которой можно использовать для доступа к данным. Каждая переменная в Java имеет конкретный тип, который определяет размер и размещение ее в памяти, диапазон значений, которые разрешается принимать данной переменной, а также набор операций, которые могут быть применены к переменной.

Приведем пример объявления и инициализации переменных:

```
int a, b;
// Объявление двух целочисленных переменных
int c = 5;
// Объявление и инициализация целочисленной переменной
byte d = 22;
```

```
// Объявление и инициализация переменной типа byte
double pi = 3.14;
// Объявление и инициализация переменной типа double
char l = 'A';
// Объявление и инициализация переменной типа char
```

Локальные переменные объявляются в методах, конструкторах или блоках. Локальные переменные создаются, когда метод, конструктор или блок запускается, и уничтожаются после того, как завершается метод, конструктор или блок.

В Java локальным переменным значение по умолчанию (начальное) не присваивается. Тем не менее это значение должно быть присвоено до первого использования в явном виде.

# 6 Пакеты и организация Java-кода

Прежде чем перейти к обсуждению системы типов Java, рассмотрим пакеты и организацию кода.

Многие реализации платформ Java для управления файлами исходного кода используют иерархическую структуру файлов, несмотря на то, что спецификация языка Java этого не требует. Идея состоит в том, чтобы поместить исходный код класса в текстовый файл, имя которого совпадает с именем типа (класса, интерфейса, перечисления или аннотации) и имеет расширение `.java`. Например,

```
// файл Rectangle.java
package graphics;

public class Rectangle {
    ...
}
```

Данный файл должен находиться в директории, название которой совпадает с именем пакета, к которому относится класс:

```
...\graphics\Rectangle.java
```

Чтобы скомпилировать данный файл, следует запустить в директории `\graphics` следующую команду:

```
javac Rectangle.java
```

Для выполнения файла (при условии наличия в нем метода `main`) следует запустить команду

```
java Rectangle.java
```

Следует отметить, что каждый элемент имени пакета соответствует поддиректории. Поэтому для файла `Rectangle.java`, находящегося в пакете `com.example.graphics`, путь будет следующим:

```
...\com\example\graphics\Rectangle.java
```

Когда исходный код компилируется, компилятор создает по одному выходному файлу для каждого типа, описанного в исходном файле. Имя выходного файла, содержащего байт-код, совпадает с именем типа и имеет расширение `.class`. Например, для исходного кода

```
// файл Rectangle.java
package com.example.graphics;

public class Rectangle {
    ...
}

class Helper {
    ...
}
```

скомпилированные файлы будут находиться на следующих путях:

```
<путь к директории с выходными файлами>\com\example
\graphics\Rectangle.class
<путь к директории с выходными файлами>\com\example
\graphics\Helper.class
```

Важно подчеркнуть, что файлы с исходным кодом (`.java`) и файлы (`.class`), полученные в результате компиляции, могут находиться в разных родительских директориях. В нашем примере мы могли бы разделить директории так:

```
<путь_1>\sources\com\example\graphics\Rectangle.java

<путь_2>\classes\com\example\graphics\Rectangle.class
```

Это позволяет легко отделить файлы с исходным кодом от результата компиляции программы.

Кроме того, Java-пакеты могут содержаться в сжатом виде в JAR-файлах (сокращение от англ. “Java ARchive”). JAR-файл представляет собой ZIP-архив, в котором содержится вся или некоторая часть программы на языке Java.

Располагая результаты компиляции в общедоступной директории, вы можете дать доступ к вашим классам другим Java-программам на той же машине. Полный путь к директории `classes` называют “class path”. Он задается переменной окружения `CLASSPATH`. Эта переменная может содержать несколько путей к директориям, разделяемых специальным символом (`;` для Windows и `:` для Linux). По умолчанию компилятор и JVM осуществляют поиск в локальной директории, где запущена программа, и в JAR-файле, содержащем стандартную библиотеку Java, поэтому эти классы автоматически попадают в `CLASSPATH`.

Для того чтобы JAR-файл был исполняемым, он должен содержать файл `MANIFEST.MF` в каталоге `META-INF`, в котором, свою очередь, должен быть указан главный класс программы (такой класс должен содержать метод `main` и задаваться параметром `Main-Class`). Пример файла `MANIFEST.MF`:

```
Manifest-Version: 1.0
Created-By: 1.5.0_20-141 (Company Inc.)
Main-Class: com.example.graphics.Rectangle
```

Команда для запуска (для определенного выше манифест-файла запустится `main`-метод класса `com.example.graphics.Rectangle`):

```
java -jar имя_файла
```

Для запуска класса, содержащегося в JAR-файле, следует выполнить следующую команду:

```
java -cp имя_jar_файла имя_класса
```

Аналогичным образом можно запустить скомпилированные (`.class`) файлы, не находящиеся в архиве.

Для использования этого класса в каком-то другом пакете, нужно воспользоваться оператором `import`:

```
package com.example.ui;

// импорт конкретного класса:
import com.example.graphics.Rectangle;
// импорт всего содержимого пакета graphics:
import com.example.graphics.*;

...
```

Следует отметить, что в случае, если пакет в исходном коде не объявлен, компилятор считает, что класс находится в пакете по умолчанию или пакет без названия (“unnamed package”). Такие классы доступны только из других классов, относящихся к пакету по умолчанию. На практике пакет по умолчанию бывает полезным в учебных примерах и простейших программах.

Кроме того, в Java имеется специальный пакет `java.lang`, содержащий базовые классы Java, такие как `System`, `Object` и `String`. Содержимое этого пакета не требуется импортировать в явном виде.

Имеется также возможность сделать статический импорт. Конструкция статического импорта позволяет получить прямой доступ к статическим членам без необходимости наследования от того типа, который содержит эти статические члены.

Продемонстрируем статический импорт на примере:

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

class Example {

    public void main(String[] args) {
        double radius = 2;
        double circleLength = 2 * PI * radius;
        // нет нужды писать Math.PI
    }
}
```

Отметим, что на практике этой возможностью языка пользуются нечасто.

# 7 Примитивные типы

В Java имеется 8 *примитивных типов*, которые делят на 3 группы:

- целые числа — `byte`, `short`, `char`, `int`, `long`;
- числа с плавающей точкой (дробные) — `float`, `double`;
- логический — `boolean`.

Значения примитивных типов представляют собой исключительно значение, а не объект. Поэтому у примитивных типов нет методов и полей.

Для инициализации переменной примитивного типа в Java используются литералы. *Литерал* — это часть исходного кода, представляющая собой константу, т. е. фиксированное значение.

Как показано ниже, литерал можно использовать для инициализации переменной примитивного типа:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

## 7.1 Целочисленные типы

Перечислим целочисленные типы данных в Java:

| Тип | Размер (бит) | Диапазон |
|---|---|---|
| *byte* | 8 | от –128 до 127 |
| *short* | 16 | от –32768 до 32767 |
| *char* | 16 | беззнаковое целое число, символом UTF-16 (буквы и цифры) |
| *int* | 32 | от –2147483648 до 2147483647 |
| *long* | 64 | от –9223372036854775808 до 9223372036854775807 |

В Java целочисленные литералы представляют собой тип `long`, если заканчивается буквой `L` или `l`. Если такого постфикса нет, то это литерал для типа `int`. Приведем пример подобных литералов:

```
long longVal = 10L;
int intVal = 10;
```

Целочисленные значения типа `byte, short, int,` и `long` можно создать из `int`-литералов, однако иногда значения типа `long` могут выходить за допустимые границы типа `int`. В этом случае инициализацию можно произвести с помощью `long`-литералов.

Кроме описанных возможностей, целочисленные литералы могут быть выражены в десятичной, двоичной и шестнадцатеричной системах счисления. Для большинства задач достаточно только десятичной системы счисления. Однако в некоторых случаях вам могут потребоваться другие системы счисления. Для шестнадцатеричных литералов используется префикс `0x`, а для двоичных — `0b`. Ниже приведены примеры таких литералов:

```
// число 26 в шестнадцатеричном представлении
int hexVal = 0x1a;
// число 26 в двоичном представлении
int binVal = 0b11010;
```

## 7.2 Типы с плавающей точкой.

Перечислим типы данных с плавающей точкой (дробные числа) в Java:

| Тип | Размер (бит) | Диапазон |
|---|---|---|
| *float* | 32 | от –1.4e-45f до 3.4e+38f |
| *double* | 64 | от –4.9e-324 до 1.7e+308 |

Поскольку компьютерная память ограничена, мы не можем хранить бесконечные десятичные дроби, в какой-то момент “хвост” придется отрезать. С другой стороны, для нужд инженеров и проектировщиков различных процессоров зачастую необходима различная точность, и нельзя заранее сказать, что, скажем, 20 или 30 знаков после запятой удовлетворят всех, поскольку неизвестно, сколько цифр после запятой значимы. Особенно важна точность во время выполнения вычислений над дробными числами, поскольку потеря точности в промежуточных значениях может привести к серьезным расхождениям в результатах вычисления.

Одним из способов решения этой проблемы являются числа с плавающей точкой. Идея состоит в том, чтобы составить число из двух частей: мантиссы — часть, содержащая значимые цифры, и показателя степени — местоположения точки. Числа с плавающей точкой можно представить следующим образом:

```
знак * мантисса * основание ^ показатель
```

За счет изменения знака числа и знака показателя в формате с плавающей точкой можно хранить, как довольно малые, так и большие положительные и отрицательные числа.

В Java поддержка чисел с плавающей точкой и операций над ними соответствует стандарту IEEE-754. Практически все современные компьютеры поддерживают операции над числами, представленные в этом стандарте на аппаратном уровне. Согласно стандарту, основанием степени является 2. Для хранения знака числа отводится 1 бит, для которого 0 соответствует положительному числу, а 1 — отрицательному. Под показатель степени отводится 8 бит для типа `float` и 11 бит для типа `double`. Под мантиссу отводится последняя часть отведенной памяти — 23 бита для типа `float` и 52 бита для типа `double`. Особенности хранения и интерпретации каждой из частей числа с плавающей точкой выходят за рамки повествования. За более подробной информацией советуем обратиться к стандарту. Описание стандарта IEEE-754 и его полный текст можно найти по ссылке: https://ru.wikipedia.org/wiki/IEEE_754-2008

В Java литералы для типа `float` заканчиваются постфиксом `F` или `f`, а для типа `double` — `D` или `d`. Примеры инициализации переменных различными литералами показаны ниже:

```
float f1 = 42.0f;
double d1 = 123.456D;
double d2 = 123.456;
```

Следует обратить внимание, что при записи литерала без постфикса (последняя переменная из примера) компилятор подразумевает литерал типа `double`.

Допускается инициализировать переменные типа float и double с помощью литералов для int и long. Покажем это на примере:

```
float f1 = 42;
float f2 = 42L;
double d1 = 123;
double d2 = 123L;
```

Кроме такой формы записи, используется еще экспоненциальная форма. Экспоненциальная запись имеет вид `MEp`, где `M` — мантисса, `E` — буква `E` или `e`, означающая `*10^`, `p` — показатель. Примеры таких литералов показаны ниже:

```
double d1 = 123.4;
double d2 = 1.234e2;
// то же число в экспоненциальной записи
float f1 = 1.234E2F;
// постфикс "F" указывает компилятору на тип float
```

Как вы могли уже догадаться, хранение чисел с плавающей точкой имеет определенные минусы. Например, иррациональные числа нельзя представить в виде числа с плавающей точкой. Более того, поскольку основанием степени является 2, данный формат хранения не позволяет точно представить многие числа, имеющие периодические представления в двоичном виде, такие как `0.1`. Проиллюстрируем эту особенность чисел с плавающей точкой на следующем примере:

```
double ruble = 1.00;
double tenKopeykas = 0.10;
int number = 7;
double result = ruble - number * tenKopeykas;
System.out.println(
        "Рубль без " + number
                + " монет в 10 копеек равно "
                + result
);
```

Данный код выведет в консоль следующее:

```
Рубль без 7 монет в 10 копеек равно 0.29999999999999993
```

Таким образом, следует иметь в виду, что числа с плавающей точкой не вполне походят для ряда задач, например для финансовых вычислений. В этих случаях следует использовать другие виды хранения дробных чисел.

## 7.3 Примитивные и объектные типы

Помимо уже рассмотренных примитивных типов, в Java имеются еще объектные типы. В отличие от встроенных примитивных типов, добавлять к которым новые нельзя, объектные типы можно создавать и расширять.

Названия объектных типов принято начинать с большой буквы и писать в “camel case”. Например, `JustAnotherClass`. Названия переменных

и полей классов принято начинать с маленькой буквы и тоже писать в “camel case”. Пример названия переменной — `anotherVariable`.

На этом отличия примитивных и объектных типов не заканчиваются. У примитивных типов нет никаких полей и методов, а относиться к ним нужно, как к значениям, именованным через название переменных или полей. При передаче переменных в какой-то метод переменная примитивного типа копируется по значению, т. е. в аргументе метода будет доступно не оригинальное, а скопированное значение. Переменные объектных типов, наоборот, передаются по ссылке, т. е. в аргументе метода будет доступен объект-оригинал. Рассмотрим это на примере:

```
public static void main(String[] args) {
    int pCnt = 0;
    Counter oCnt = new Counter();
    Counter oCnt2 = oCnt;
    variableValuesTest(pCnt, oCnt);
    System.out.println("примитивный тип=" + pCnt);
    System.out.println("объектный тип=" + oCnt.cnt);
    System.out.println("еще объектный тип=" + oCnt2.cnt);
}

public static void variableValuesTest(int pCnt, Counter oCnt) {
    pCnt = 42;
    oCnt.cnt = 42;
}

public static class Counter {
    public int cnt = 0;
}
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
примитивный тип=0
объектный тип=42
еще объектный тип=42
```

Подробнее объектные типы будут рассмотрены в дальнейшем.

## 7.4 Приведение примитивных типов

Допустим, требуется перевести `short` в `int` или наоборот. Такая операция называется *приведением* (преобразованием) типов. Приведение ("casting") примитивных типов в Java бывает двух видов: неявное ("implicit") и явное ("explicit"). Приведем примеры этих способов:

```
int a = 2;
short b = 4;
a = b;            // неявное приведение
a = (int) b;      // явное приведение
b = a;            // так делать нельзя! ошибка компиляции
b = (short) a;    // явное приведение
```

Неявное преобразование для совместимых типов происходит при преобразовании значения типа с меньшим размером, например `int`, в тип с большим размером, например `long`. Такое преобразование еще можно назвать повышающим или расширением. Правило преобразования можно описать следующей цепочкой отношений между типами:

```
byte -> short -> int -> long -> float -> double
```

Пример неявного преобразования:

```
byte i = 50;
// здесь преобразование неявное
short j = i;
int k = j;
long l = k;
float m = l;
double n = m;

System.out.println("Переменная типа byte: " + i);
System.out.println("Переменная типа short: " + j);
System.out.println("Переменная типа int: " + k);
System.out.println("Переменная типа long: " + l);
System.out.println("Переменная типа float: " + m);
System.out.println("Переменная типа double: " + n);
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Переменная типа byte: 50
Переменная типа short: 50
Переменная типа int: 50
Переменная типа long: 50
Переменная типа float: 50.0
Переменная типа double: 50.0
```

Явное преобразование для числовых типов происходит при преобразовании значения типа с большим размером, например `float`, в тип с меньшим размером, например `byte`. Такое преобразование еще можно назвать понижающим (или сужением). Правило преобразования можно описать следующей цепочкой отношений между типами:

```
double -> float -> long -> int -> short -> byte
```

При этом в случае, если значение числа выходит за рамки диапазона значений итогового типа или дробное значение преобразуется в целое, происходит изменение значения.

Пример неявного преобразования:

```
double d = 175.0;
// Здесь необходимо явное преобразование
float f = (float) d;
long l = (long) f;
int i = (int) l;
short s = (short) i;
byte b = (byte) d;

System.out.println("Переменная типа double: " + d);
System.out.println("Переменная типа float: " + f);
System.out.println("Переменная типа long: " + l);
System.out.println("Переменная типа int: " + i);
System.out.println("Переменная типа short: " + s);
System.out.println("Переменная типа byte: " + b);
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Переменная типа double: 175.0
Переменная типа float: 175.0
```

```
Переменная типа long: 175
Переменная типа int: 175
Переменная типа short: 175
Переменная типа byte: -81
```

Подробные правила преобразования типов описаны в документации: https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.1.3

# 8 Основные операторы

## 8.1 Арифметические операторы

*Арифметические операторы* используются в математических выражениях таким же образом, как они используются в алгебре:

- `+` складывает значения по обе стороны от оператора;
- `-` вычитает правый операнд из левого операнда;
- `*` умножает значения по обе стороны от оператора;
- `/` делит левый операнд на правый операнд;
- `%` делит левый операнд на правый операнд и возвращает остаток ;
- `++` инкремент — увеличивает значение операнда на 1;
- `--` декремент — уменьшает значение операнда на 1.

Последние два оператора, инкремент и декремент, могут быть использованы как в инфиксной, так и в постфиксной записи. В первом случае вначале применяется операция, потом возвращается значение переменной. Во втором случае, наоборот, сначала возвращается текущее значение переменной, а потом к переменной применяется операция. Продемонстрируем это на примере:

```
int input1 = 0;
int output1 = input1++;
System.out.println("постфиксная запись=" + output1);

int input2 = 0;
int output2 = ++input2;
System.out.println("инфиксная запись=" + output2);
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
постфиксная запись=0
инфиксная запись=1
```

## 8.2 Операторы сравнения

В Java имеются следующие операторы сравнения значений:

- `==` проверяет, равны или нет значения двух операндов, если да, то результат становится истинным;
- `!=` проверяет, равны или нет значения двух операндов, если значения не равны, то результат становится истинным;
- `>` проверяет, является ли значение левого операнда больше, чем значение правого операнда, если да, то результат становится истинным;
- `<` проверяет, является ли значение левого операнда меньше, чем значение правого операнда, если да, то результат становится истинным;
- `>=` проверяет, является ли значение левого операнда больше или равно значению правого операнда, если да, то результат становится истинным;
- `<=` проверяет, является значение левого операнда меньшим или равным значению правого операнда, если да, то результат становится истинным.

Следует отметить, что для примитивных типов сравниваются значения переменных (или полей). Для объектных типов применимы первые два оператора (`==` и `!=`), но они сравнивают ссылки, а не сами объекты.

## 8.3 Логические операторы

Логические операторы в Java представлены следующими:

- `&&` — логический оператор “И”. Если оба операнда принимают значение `true`, то возвращаемое значение становится истинным. В противном случае — ложным;

- `||` — логический оператор “ИЛИ”. Если любой из двух операндов принимают значение `true`, то возвращаемое значение становится истинным;
- `!` — логический оператор “ОТРИЦАНИЕ”. Использование меняет логическое состояние своего операнда на противоположное. Если условие имеет значение `true`, то оператор логического “ОТРИЦАНИЯ” сделает все выражение равным `false`.

## 8.4 Побитовые логические операторы

Операторы `&`, `|`, `^` и `~` выполняют побитовые логические операции над целочисленными типами в Java. Операторы `&`, `|` и `^` принимают на вход два операнда, тогда как оператор `~` является унарным, т. е. применяется к одному операнду.

- `&` — оператор побитовое “И”. Записывает 1 в выходной бит, если оба входные бита равны 1, в противном случае он записывает 0.
- `|` — оператор побитовое “ИЛИ”. Записывает 1 в выходной бит, если хотя бы один входной бит равны 1, в противном случае, если оба входные бита равны 0, он записывает 0.
- `^` — оператор “ИСКЛЮЧАЮЩЕЕ ИЛИ”. Записывает 1 в выходной бит, если только один входной бит равен 1, в противном случае, если оба входные бита равны 0 или 1, он записывает 0.
- `~` — оператор побитовое “ОТРИЦАНИЕ”. На той позиции (входном бите), где в двоичном представлении операнда был 0, в результат записывает 1, и, наоборот, где была 1, записывает 0.

Продемонстрируем действие поразрядных операторов на примерах:

```
int i1 = 0b101 | 0b110; // => 0b111 == 7
System.out.println("i1=" + i1);
int i2 = 0b101 ^ 0b110; // => 0b011 == 3
System.out.println("i2=" + i2);
int i3 = 0b101 & 0b110; // => 0b100 == 4
System.out.println("i3=" + i3);
byte b1 = ~Byte.MAX_VALUE; // => -0b10000000
 == Byte.MIN_VALUE == -128
System.out.println("b1=" + b1);
```

После выполнения программы на экран будет выведено следующее:

```
i1=7
i2=3
i3=4
b1=-128
```

Заметим, что операторы `&` и `|` записываются с помощью тех же символов, что и логические `&&` и `||`, и важно научиться их различать, особенно учитывая то, что поразрядные операторы применимы к операндам типа `boolean`. Важное отличие поразрядных операторов от логических в том, что они приводят к вычислению значений обоих операндов. В то же время для вычисления результата логическому оператору иногда достаточно только первого (например, `true || condition` сразу принимается равным `true` без проверки `condition` и аналогично `false && condition = false`). Следующий пример это наглядно демонстрирует:

```
public class Main {

    public static void main(String[] args) {
        if (calc1() || calc2()) {
            System.out.println("Выполнено условие с ||");
        }
        if (calc1() | calc2()) {
            System.out.println("Выполнено условие с |");
        }
    }

    public static boolean calc1() {
        System.out.println("Вычисление 1");
        return true;
    }

    public static boolean calc2() {
        System.out.println("Вычисление 2");
        return true;
    }
}
```

После выполнения программы на экран выведется следующее:

```
Вычисление 1
Выполнено условие с ||
Вычисление 1
Вычисление 2
Выполнено условие с |
```

## 8.5 Операторы битового сдвига

В Java также имеются операторы битового сдвига влево (<<) и вправо (>>), применимые к целочисленными типам. Операторы битового сдвига в Java осуществляют так называемый арифметический сдвиг. Арифметические сдвиги влево и вправо часто используют для быстрого умножения и деления на 2.

При сдвиге значения битов копируются в соседние по направлению сдвига. При правом сдвиге старший бит сохраняет свое значение, поэтому числа знаковых типов сохраняют свой знак. А при сдвиге влево значение последнего бита по направлению сдвига теряется (копируясь в бит переноса), а первый приобретает нулевое значение. Следующий пример хорошо иллюстрирует действие операторов сдвига:

```
byte b1 = 0b01 << 1; // => 0b10 == 2
System.out.println("b1=" + b1);
byte b2 = 0b1 << 3; // => 0b1000 == 8
System.out.println("b2=" + b2);
byte b3 = 0b10 << 1; // => 0b100 == 4
System.out.println("b3=" + b3);
byte b4 = -0b100 >> 2; // => -0b001 == -1
System.out.println("b4=" + b4);
```

После выполнения программы на экран будет выведено следующее:

```
b1=2
b2=8
b3=4
b4=-1
```

## 8.6 Операторы присваивания

В Java имеются следующие вариации оператора присваивания:

- `=` — простой оператор присваивания, присваивает значение из правой стороны операндов левому операнду;
- `+=` — оператор присваивания «*Прибавление*», он прибавляет левому операнду значения правого: `b += a` эквивалентно `b = b + a`;
- `-=` — оператор присваивания «*Вычитание*», он вычитает из правого операнда левый операнд: `b -= a` эквивалентно `b = b - a`;
- `*=` — оператор присваивания «*Умножение*», он умножает правый операнд на левый операнд: `b * = a` эквивалентно `b = b * a`;
- `/=` — оператор присваивания «*Деление*», он делит левый операнд на правый операнд: `b /= a` эквивалентно `b = b / a`.

## 8.7 Тернарный оператор

Тернарный оператор — оператор, который состоит из трех операндов и используется для оценки выражений типа boolean. Тернарный оператор в Java также известен как *условный оператор*. Тернарный оператор возвращает то или иное значение в зависимости от входного выражения, возвращающего булевское значение. Запись этого оператора имеет следующий вид:

```
переменная = выражение ?
    выражение-для-случая-true :
    выражение-для-случая-false
```

Приведем конкретный пример:

```
int a = 10;
int b;
b = a == 1 ? 20 : 30;
System.out.println("b = " + b);
b = (a == 10) ? 20 : 30;
System.out.println("b = " + b);
```

После выполнения программы на экран будет выведено:

```
b = 30
b = 20
```

## 8.8 Операторы выбора

Java поддерживает два оператора выбора: `if` и `switch`.

Оператор `if` имеет следующий вид:

```
if (условие)
    выражение-1;
else
    выражение-2;
```

Оператор `if` работает так: если `условие` имеет значение `true`, то выполняется `выражение-1`, иначе выполняется `выражение-2` (если оно присутствует). Оба выражения (или блока) вместе не выполняются ни в коем случае.

Общую программную конструкцию, которая основана на последовательности вложенных `if`, называют многозвенным `if` ("(ladder) if-else-if"). Эта конструкция выглядит так:

```
if (условие-1)
    выражение-1;
else if (условие-2)
    выражение-2;
else if (условие-3)
    выражение-3;
// ...
```

Операторы `if-else-if` выполняются сверху вниз. Как только одно из условий, управляющих оператором `if`, становится `true`, выражение или блок, связанный с этим `if`, выполняется, а остальная часть многозвенной схемы пропускается.

Оператор `switch` — это Java-оператор множественного ветвления. Он переключает выполнение на различные части кода программы, основываясь на значении текущего условия, и часто обеспечивает более удобную альтернативу, чем длинный ряд операторов `if-else-if`.

Отличие `switch` от `if` в том, что первый оператор в качестве условия может проверять только равенство (значения входного выражения и `case`-меток), тогда как `if` принимает любое выражение, возвращающее булевское значение.

Рассмотрим конструкцию оператора `switch`:

```
switch (входное-выражение) {
    case значение-1:
        последовательность-операторов-1
        break;
    case значение-2:
        последовательность-операторов-2
        break;
    // ...
    default:
        последовательность-операторов-по-умолчанию
}
```

Здесь `входное-выражение` должно иметь определенный тип (в Java 6 это `byte`, `short`, `int`, `char` или `enum`, а в Java 7 стало возможным использовать `String`[1]); каждое значение, указанное в операторах `case`, должно иметь тип, совместимый с типом выражения. Каждое значение `case` должно быть уникальной константой (а не переменной). Дублирование значений `case` недопустимо.

Оператор `switch` работает следующим образом. Значение выражения последовательно сравнивается с каждым из указанных значений в `case`-операторах. Если соответствие найдено, то выполняется блок кода, следующий после этого оператора `case` (важное замечание: и всех `case` после него). Если ни одна из `case`-констант не соответствует значению выражения, то выполняется блок для `default`. Однако оператор `default` необязателен. Если совпадающих `case` нет, и `default` не присутствует, то никаких дальнейших действий не выполняется.

Оператор `break`, написанный после оператора `case`, прерывает выполнение оператора `switch` для случая, если входное выражение соответствует данному `case`. Продемонстрируем это на примере:

```
int a = 10;
switch (a) {
    case 0:
    case 1:
        System.out.println("Это 0 или 1.");
        break;
    case 10:
```

[1]Об объектных типах данных `enum` и `String` будет рассказано в следующей главе.

```
        System.out.println("Это 10.");
    case 20: {
        System.out.print("Это пример блока кода
        с несколькими операторами. ");
        System.out.println("Это точно 20?");
    }
    default:
        System.out.println("Совпадений нет.");
}
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Это 10.
Это пример блока кода с несколькими операторами. Это точно 20?
Совпадений нет.
```

Обратите внимание, что помимо блока для `case 10`, выполнился еще блок для `case 20`. Чтобы избежать такого поведения, следует добавить вызов оператора `break` в конце блока для `case 10` так, как это сделано для `case 1`.

## 8.9 Операторы цикла

Операторы цикла (итераций) Java — это `while`, `do while` и `for`.

Цикл `while` (цикл с предусловием) повторяет оператор или блок операторов, пока его управляющее условие имеет значение `true`. Цикл `while` имеет следующий вид:

```
while (условие) {
    тело-цикла
}
```

Здесь `условие` может быть любым булевским выражением. Тело цикла будет выполняться до тех пор, пока условное выражение имеет значение `true`. Продемонстрируем это на примере:

```
int i = 0;
while (i <= 2) {
    System.out.println("Текущее значение=" + i);
    i += 1;
}
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Текущее значение=0
Текущее значение=1
Текущее значение=2
```

Цикл `do while` (цикл с постусловием) всегда выполняет свое тело по крайней мере один раз, потому что его условие размещается в конце цикла. В цикле `do while` проверяется условие продолжения, а не окончания цикла. Этот цикл имеет следующий вид:

```
do {
    тело-цикла
} while (условие);
```

Главное отличие между `while` и `do while` в том, что инструкция в цикле `do while` всегда выполняется не менее одного раза, даже если вычисленное выражение ложное с самого начала. В цикле `while`, если условие ложное в первый раз, инструкция никогда не выполнится. На практике `do while` используется реже, чем `while`.

Продемонстрируем это на примере:

```
int i = 0;
do {
    System.out.println("Текущее значение=" + i);
    i += 1;
} while (i <= 1);
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Текущее значение=0
Текущее значение=1
```

Цикл `for` наиболее удобен в тех случаях, когда тело цикла должно быть выполнено определенное количество раз. Общая форма цикла `for`:

```
for (инициализация; условие; итерация) {
    тело-цикла
}
```

Цикл `for` работает следующим образом. В начале работы цикла выполняется выражение `инициализация`. В общем случае это выражение устанавливает значение переменной управления циклом, которая действует как счетчик. Важно, что выражение инициализации выполняется только один раз. Затем оценивается `условие`. Оно должно быть булевским выражением и обычно сравнивает переменную управления циклом с некоторым граничным значением. Если это выражение `true`, то отрабатывают операторы из тела цикла, если `false` — цикл заканчивается. Далее выполняется часть цикла `итерация`. Обычно это выражение, которое осуществляет инкрементные или декрементные операции с переменной управления циклом. Затем снова проверяется условие и т. д. Этот процесс повторяется до тех пор, пока условие не вернет `false`.

Продемонстрируем это на примере:

```
for (int i = 0; i < 2; i++) {
    System.out.println("Текущее значение=" + i);
}
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Текущее значение=0
Текущее значение=1
```

## 8.10 Операторы перехода

Оператор `break`, о котором мы говорили при рассмотрении оператора `switch`, в Java используется в трех случаях. Во-первых, он заканчивает последовательность операторов в ветвях оператора `switch`. Во-вторых, его можно использовать для выхода из цикла любого вида. В-третьих, он может использоваться совместно с метками.

В следующем примере оператор `break` можно использовать для выхода из цикла `while`, который иначе выполнялся бы до бесконечности:

```
int i = 0;
while (true) {
    i++;
    if (i > 10) break;
}
```

Помимо рассмотренного сценария использования оператора `break`, он может применяться в циклах совместно в метками. Метками при этом разрешается помечать операторы циклов.

Ниже показано, как оператор `break` во внутреннем цикле совместно с метками можно использовать для выхода из внешнего цикла:

```
outerLoopLabel: while (true) {
    for (int i = 0; i < 10; i++) {
        break outerLoopLabel;
    }
}
```

Эта форма применения оператора `break` может рассматриваться как некая разновидность формы оператора безусловного перехода `goto`. Заметим, что необходимость в использовании меток на практике встречается довольно редко.

Оператор `continue` позволяет закончить выполнение текущего тела цикла и перейти к следующей итерации. Рассмотрим пример:

```
int i = 0;
while (i <= 2) {
    i += 1;
    if (i % 2 == 0) {
        continue;
    }
    System.out.println("Текущее нечетное значение=" + i);
}
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Текущее нечетное значение=1
Текущее нечетное значение=3
```

Как и оператор `break`, оператор `continue` можно использовать совместно с метками.

# Глава 2

# Основы ООП

## 1 Классы и объекты

Java — объектно-ориентированный язык. Поэтому все действия, выполняемые программой, находятся в методах тех или иных классов.

Описание класса начинается с ключевого слова `class`, после которого указывается идентификатор — имя класса. Затем в простейшем случае в фигурных скобках перечисляются поля (переменные) и методы класса.

Приведем в качестве примера класс `Dog` (собака), имеющий два поля: `name` (кличка) и `age` (возраст) — и один метод — `voice()` (подать голос). Конечно, лаять по-настоящему собака не будет, она будет выводить в консоль текст “гав-гав”.

```
class Dog {

    int age = 1;
    String name = "Трезор";

    public void voice() {
        System.out.println("Собака " + name
                   + " залаяла: гав-гав!");
    }

}
```

Класс должен описывать некоторое законченное понятие (или сущность). Это может быть понятие из предметной области программы (сессия, документ, пользователь, роль) или понятие, необходимое для работы самой программы (очередь, список, строка, окно, кнопка).

Полями класса должны быть данные, характеризующие это понятие. Для собаки это возраст, кличка, порода и т. д., а для сессии — дата начала, продолжительность и т. д.

Методы класса, как правило, работают с данными этого класса. Например, метод `voice()` в нашем примере обращается к полю `name`. Подчеркнем, что в классе не может быть двух методов с одинаковыми сигнатурами.

После того как какой-то класс описан, могут создаваться (конструироваться) объекты этого класса (экземпляры), затем с ними можно работать, вызывая их методы: кормить собаку, выгуливать, просить ее лаять, т. е. делать все то, что позволяет поведение класса (совокупность его публичных полей и методов).

Для работы с классом `Dog` необходимо создать переменную типа `Dog`[1]:

```
Dog dog;
```

В этой ситуации переменная `Dog` является ссылочной, она не хранит данные (как переменные простых типов int, char и т. д.), а указывает на место в памяти, где эти данные хранятся. Данными, на которые указывает только что описанная переменная `dog`, может быть объект класса `Dog` (или одного из классов-наследников). Прежде чем работать с переменной `dog`, необходимо предварительно сконструировать экземпляр класса `Dog`, использовав ключевое слово `new`:

```
dog = new Dog();
```

Ниже мы также рассмотрим конструкторы, наследование и многие другие детали, без которых невозможно создание экземпляра класса.

Теперь переменная `dog` указывает на некий объект класса `Dog`, хранящий в памяти свои данные. Кроме того, эту собаку можно заставить лаять, вызвав соответствующий метод командой `dog.voice();`.

Обратите внимание, “залаяла” именно та собака, на которую “указывала” переменная `dog`, т. е. метод `voice()` был вызван именно у конкретного экземпляра класса `Dog`. Поэтому вызов `voice();` не имеет никакого смысла, если употребляется вне класса `Dog`. Указание на конкретный объект (экземпляр класса), с которым производится действие, обязательно[2].

[1] Вообще говоря, это не совсем так: еще до того, как был создан первый экземпляр класса `Dog`, можно работать с его статическими полями, но об этом — позже.

[2] Опять же, кроме случая статических полей.

Кроме того, следует отметить, что переменные объектных типов передаются по ссылке. Таким образом, в аргументе метода доступен сам объект-оригинал. Подробнее это обсуждалось в предыдущей главе.

В случае когда поле объектного типа было объявлено, но не инициализировано, это поле будет иметь особое значение `null`, которое означает отсутствие ссылки на объект. Это же ключевое слово можно использовать для присваивания (фактически стирания ссылки) полям и переменным объектных типов и проверки их на непустоту. Приведем соответствующий пример:

```
public static void main(String[] args) {
        Dog dog = null;
        if (dog == null) {
            System.out.println("Переменная dog не
                                инициализирована");
            dog.voice();
            // ошибка выполнения (NullPointerException)
        }
}
```

## 2 Модификаторы видимости

Доступ извне к любому элементу класса (полю, конструктору или методу) можно ограничить, применив *модификатор видимости* (доступа).

Для максимального ограничения доступа перед объявлением элемента необходимо поставить ключевое слово `private`. Этот модификатор означает, что к этому члену класса можно обращаться только внутри его класса и нельзя обратиться из методов других классов.

Ключевое слово `public` может употребляться в тех же случаях, но имеет противоположный смысл. Этот модификатор означает, что данный член класса доступен из методов других классов: если это поле, его можно использовать в выражениях или изменять при помощи присваивания, а если метод — то его можно вызывать. Часто говорят, что совокупность публичных методов и полей образует “интерфейс” класса (не следует путать “интерфейс” в данном контексте с отдельной сущностью Интерфейс, которая будет рассмотрена ниже).

Ключевое слово `protected` означает, что доступ к полю или методу имеет сам класс и все его потомки.

Если при объявлении члена класса не указан ни один из перечисленных выше трех модификаторов, используется модификатор по умолчанию (“default”). Он означает, что доступ к члену класса имеют все классы, объявленные в том же пакете.

Перепишем класс `Dog` следующим образом:

```
public class Dog {

    private int age;
    private String name;

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void voice() {
        System.out.println("Собака " + name
                    + " залаяла: гав-гав!");
    }

}
```

Поля `age` и `name` имеют модификатор `private`; это означает, что они скрыты. Поэтому мы не можем изменять их (или считывать их значение) где-либо за пределами класса. А именно мы *не сможем* в методе `main()` создать объект класса `Dog`, а затем присвоить его полю `age` или `name` новое значение, как в следующем примере:

```
public static void main(String[] args) {
    Dog dog = new Dog("Тузик", 4);
    dog.age = 10; // ошибка компиляции: поле age скрыто
    dog.name = "Жучка"; // ошибка компиляции:
    // переименовать собаку тоже нельзя, поле name скрыто
    dog.voice(); // это можно, метод voice() открытый
}
```

Обратите внимание, что модификаторы доступа нельзя использовать для локальных переменных. Они являются видимыми только в пределах объявленного метода, конструктора или блока.

Кроме того, для сущностей верхнего уровня (классы, интерфейсы) в Java тоже можно менять область видимости. Однако для них разрешается использовать только модификатор `public` или модификатор по умолчанию. Это ограничение можно обойти, используя внутренние классы, которые будут рассмотрены ниже.

Возможность скрывать поля и методы класса используется для того, чтобы уберечь программиста от возможных ошибок, сделать классы понятнее и проще в использовании. При этом реализуется принцип "инкапсуляции".

### Инкапсуляция

*Инкапсуляция* означает сокрытие деталей реализации класса. Класс разделяется на две части: внутреннюю и внешнюю. Внешняя часть (интерфейс класса) описывает поведение класса и тщательно продумывается исходя из того, каким образом могут взаимодействовать с объектами данного класса другие объекты программы. Внутренняя часть закрыта от посторонних, она нужна только самому классу для обеспечения правильной работы открытых методов.

Ярким и простым примером сокрытия реализации является сокрытие всех полей: поля имеют модификатор доступа `private` (или `protected`), а если нужно дать доступ к чтению и/или записи, то это делается через так называемые методы доступа ("getter"- и "setter"-методы):

```
public class Dog {

    private int age;
    private String name;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    ...
```

```
}
```

В данном примере в “setter”-методе используется ключевое слово `this`. Через него можно обращаться к самому объекту, его полям, методам и конструкторам. В случае когда имя аргумента метода совпадает с именем поля класса, эту коллизию можно разрешить, используя `this`.

Обратите внимание, что “getter”-метод не должен изменять состояние объекта (по крайней мере, внешнюю его часть). В противном случае, поскольку другой программист не будет ожидать от этого метода такого поведения, это приведет к неизбежным ошибкам.

Возникает разумный вопрос: зачем нужны методы доступа? Если поле можно изменить с помощью метода доступа, то почему бы не объявить поле `public` и сделать это напрямую? Методы доступа нужны для лучшего контроля, а также разделения публичного интерфейса и реализации класса. Допустим, в вашем поле должно храниться обязательно четное число, или же обязательно положительное, или число из определенного диапазона, или оно должно удовлетворять какому-либо другому более сложному условию. В этом случае в методе доступа (или конструкторе) можно поместить проверку на это условие, а уже потом, в случае его выполнения, менять поле:

```
public void setAge(int age) throws DogException {
   if (age >= 0) {
       this.age = age;
   } else {
       throw new DogException(
            "Некорректное значение возраста");
   }
}
```

В данном примере для обработки ошибочной ситуации использовано исключение. Об исключениях мы подробно поговорим ниже.

Еще одним хорошим примером преимуществ методов доступа перед прямым доступом к полям является то, что если требуется запретить изменение какого-то поля извне, то можно не реализовывать “setter”-метод.

Подводя итог, еще раз подчеркнем, что стандартом (или как минимум хорошим тоном) для написания кода на Java считается использование “getter”- и “setter”-методов для предоставления доступа к публичным (в смысле интерфейса класса) полям.

# 3 Конструкторы классов

Конструктор — это особый метод класса, который вызывается автоматически в момент создания объектов этого класса, т. е. при использовании ключевого слова `new`. Имя конструктора совпадает с именем класса. В отличие от обычных методов, для конструктора не требуется указывать выходной тип, так как он всегда неявным образом возвращает сконструированный объект данного класса. При этом, очевидно, для вызова конструктора не нужен существующий экземпляр класса. В этом смысле конструкторы являются статическими методами, о которых мы поговорим подробнее ниже.

Для нашего класса `Dog` удобен конструктор с двумя параметрами, который при создании новой собаки позволяет сразу задать ее кличку и возраст. Реализуем этот конструктор:

```
public Dog(String name, int age) {
    this.name = name;
    this.age = age;
}
```

Конструктор вызывается после ключевого слова `new` в момент создания объекта:

```
Dog dog = new Dog("Тузик", 2);
```

В результате переменная `dog` будет указывать на “собаку” по кличке Тузик, в возрасте 2 лет.

У класса может быть несколько конструкторов с разными сигнатурами. Например, в нашем случае можно не задавать сразу возраст и кличку собаки и, реализовав конструктор без аргументов, оставить эти поля не инициализированными. Однако в нашем примере сразу задать кличку и возраст собаки более естественно, чем каждый раз порождать безымянного щенка, а затем давать ему имя и быстро выращивать до нужного возраста. Зачастую в класс добавляются конструкторы с различными входными данными, если в момент создания объекта нужно выполнить какие-то действия (начальную настройку) с его полями.

Ключевое слово `this`, про которое мы говорили ранее, также можно использовать для вызова другого конструктора в теле данного конструктора. Например, помимо только что описанного конструктора, мы могли бы добавить конструктор, который ожидает только кличку собаки, а возраст

заполняет каким-то значением по умолчанию. Вместо повторения логики инициализации предыдущего конструктора новый конструктор может просто вызывать предыдущий. Продемонстрируем это:

```
public Dog(String name) {
    this(name, 1);
}
```

Важно отметить, что имеется ограничение на вызов других конструкторов. Вызов другого конструктора должен стоять на первом месте в теле конструктора.

Еще стоит отметить, что конструктор без параметров класса-предка будет автоматически подставлен в скомпилированный код класса-наследника в начало конструктора в случае, если он не вызывается явно.

## 3.1 Конструктор по умолчанию

Если в классе не описан ни один конструктор, для него при компиляции автоматически создается конструктор по умолчанию (исходный код при этом, конечно, не изменяется). Этот конструктор не имеет параметров, все что он делает — вызывает конструктор без параметров класса-предка. В Java предок имеется у любого класса. Если предок явно не указан, то предком является класс `Object` из пакета `java.lang`. О наследовании мы поговорим далее.

Поэтому мы и смогли создать объект класса `Dog` в самом первом примере в этой главе, хотя не описывали в классе никаких конструкторов.

## 3.2 Инициализация полей

В нашем примере можно реализовать только один конструктор без полей и с пустым телом (что эквивалентно конструктору по умолчанию). Сделаем это:

```
public class Dog {

    private int age;
    private String name = "Трезор";

    public Dog() {
    }
```

```
    ...
}
```

Обратите внимание, что поле `name` инициализируется значением “Трезор” при объявлении, а поле `age` — нет. Это приводит к тому, что после конструирования объекта поле `age` будет неявным образом инициализировано значением по умолчанию, так как это поле примитивного типа.

Значения по умолчанию для разных типов данных, использующиеся при неявной инициализации, приведены ниже:

| Тип | Значение |
|---|---|
| *boolean* | *false* |
| *char* | `'\u0000'`(пустой символ) |
| *byte*, *short*, *int*, *long* | 0 |
| *float*, *double* | 0.0 |
| объектные типы (ссылка) | *null* |

Как мы уже видели, при конструировании объекта поля можно инициализировать двумя способами: при объявлении и в конструкторе. Важно отметить, что к моменту выполнения тела конструктора поля́ объекта будут инициализированы (явным или неявным образом), причем порядок инициализации определяется порядком объявления полей в классе. Продемонстрируем это на примере:

```
public class Dog {

    int age;

    public Dog() {
        System.out.println("Возраст в начале конструктора="
        + age);
        age = 1;
    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        System.out.println("Возраст после конструктора="
        + dog.age);
    }
```

```
}
```

В результате выполнения данного кода в консоль будет выведено следующее:

```
Возраст в начале конструктора=0
Возраст после конструктора=1
```

# 4 Перегрузка

В одном и том же классе можно создать несколько методов с одним и тем же именем, различающихся по типам своих аргументов. Этот прием называется перегрузкой методов (“overloading”). Когда один из этих методов вызывается где-то в коде, компилятор производит сопоставление переданных ему аргументов (их количества и типов) с параметрами всех методов класса с таким именем и выбирает из них подходящий. Существенная деталь: каждый перегруженный метод должен иметь уникальный список типов аргументов.

Обратите внимание на то, что в Java перегруженные методы не допускается различать по возвращаемым типам, т. е. не допускается описывать два метода, которые различаются только возвращаемыми типами.

Например, опишем класс для точек на целочисленной плоскости $\mathbb{Z}^2$ и операцию сложения двух точек:

```
class Point {

    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point plus(int x, int y) {
        System.out.println("Метод plus(int, int)");
        return new Point(this.x + x, this.y + y);
    }

    public Point plus(Point anotherPoint) {
```

```
        System.out.println("Метод plus(Point)");
        return new Point(this.x + anotherPoint.x,
            this.y + anotherPoint.y);
    }

    public Point plus(byte x, byte y) {
        System.out.println("Метод plus(byte, byte)");
        return new Point(this.x + x, this.y + y);
    }

    public static void main(String[] args) {
        Point p1 = new Point(1, 1);
        p1.plus(1, 1);
        p1.plus(new Point(1, 1));
        byte x = 1, y = 1;
        p1.plus(x, y);
    }

    ...
}
```

В результате выполнения этого кода в консоль будет выведено следующее:

```
Метод plus(int, int)
Метод plus(Point)
Метод plus(byte, byte)
```

Таким образом, перегрузка в первую очередь предназначена для использования одного и то же названия для методов со схожим поведением, но разными типами аргументов. Напомним, что конструкторы тоже можно перегружать, что обсуждалось ранее.

# 5 Наследование

Наследование — это отношение между классами, при котором один класс перенимает поведение и расширяет функциональность другого. Это значит, что он автоматически перенимает интерфейс родительского класса (публичные поля и методы), а также добавляет некоторые свои.

Наследование обычно используют для того, чтобы все объекты одного класса одновременно являлись объектами другого класса (отношение общее/частное). Например, все объекты класса `Студент` являются объектами класса `Человек`. В этом случае говорят, что класс `Студент` наследует от класса `Человек`. Аналогично класс `Собака` может наследовать от класса `Животное`, а класс `Овчарка` — от класса `Собака`. Класс, который наследует, называется *потомком* или дочерним (производным) классом, а класс, от которого наследуют, называется *предком* или суперклассом (ключевое слово `super`).

Заметим, что имеет место транзитивность: если класс `B` является потомком класса `A`, а класс `C` является потомком класса `B`, то класс `C` является также потомком класса `A`.

В Java отсутствует множественное наследование, иными словами, у класса может быть только один класс-предок. В некоторой степени отсутствие множественного наследования компенсируется за счет возможности для классов реализовывать несколько интерфейсов. Но об этом мы поговорим ниже.

Наследование избавляет программиста от лишней работы. Например, если в программе необходимо ввести новый класс `Овчарка`, его можно создать на основе уже существующего класса `Собака`, не программируя заново все поля и методы, а лишь добавив те, которых хватает в родительском классе.

Для того чтобы один класс был потомком другого, необходимо при его объявлении после имени класса указать ключевое слово `extends` и название класса-предка. Приведем пример:

```
public class Wolfhound extends Dog {

    private String color;

    public Wolfhound(String name, int age, String color) {
        super(name, age);
        this.color = color;
    }

    public void howl() {
        System.out.println("Волкодав "
            + this.getName() + " издает вой.");
```

```
    }

    ...

}
```

Обратите внимание, что в методах дочернего класса имеется доступ к полям и методам родителя (с подходящим уровнем видимости, конечно). Кроме того, имеется возможность вызвать один из конструкторов родительского класса, используя ключевое слово `super` и конструкцию вида `super(...)`. Ключевое слово `super` означает класс-предок.

Важно отметить, что если ключевое слово `extends` не указано, считается, что класс унаследован от базового класса `java.lang.Object`. Таким образом, в Java класс `Object` является родительским классом верхнего уровня в любой иерархии наследования, т. е. является предком любого класса.

### Особенности конструирования классов при наследовании

В предыдущем примере мы вызываем конструктор суперкласса `Dog` в теле конструктора `Wolfhound`:

```
public Wolfhound(String name, int age, String color) {
    super(name, age);
    this.color = color;
}
```

При этом в родительский конструктор мы передаем два параметра (`String` и `int`). Следовательно, из всех конструкторов суперкласса будет выбран именно тот, который нас интересует. При этом вызов конструктора класса-предка (`super(...)`) исключает возможность вызвать другой конструктор данного класса (`this(...)`), и наоборот. В любом случае вызов другого конструктора должен стоять на первом месте в теле конструктора.

В конструкторе собаки `Wolfhound` в предыдущем примере использовался конструктор класса `Dog`. Но если бы мы попытались использовать в дочернем классе этот же родительский конструктор через ключевое слово `this`, у нас бы ничего не получилось:

```
public Wolfhound(String name, int age, String color) {
```

```
    this(name, age);
    // ошибка компиляции: такого конструктора в классе нет
    this.color = color;
}
```

Дело в том, что конструкторы не считаются членами класса и, в отличие от других методов, не наследуются. Однако, как мы уже показывали ранее, их можно явно вызывать.

Также напомним, что вместо вызова конструктора суперкласса можно вызвать один из конструкторов того же самого класса. Это делается с помощью конструкции вида `this(...)`.

Если в начале конструктора нет вызова ни `this(...)`, ни `super(...)`, то автоматически происходит обращение к конструктору суперкласса без аргументов. Если у базового класса нет конструктора без аргументов, компилятор выдаст ошибку.Однако, если у суперкласса нет конструктора вообще, то компилятор создаст конструктор по умолчанию и ошибки не произойдет.

Порядок инициализации объектов (вызов конструкторов и инициализации полей) строго определен. Приведем более сложный пример на данную тему:

```
class Point {

    Point() {
        System.out.println("Конструктор Point");
    }
}

class Figure {

    Figure() {
        System.out.println("Конструктор Figure");
    }
}

public class Circle extends Figure {

    Point center = new Point();
```

```
    public Circle() {
        System.out.println("Конструктор Circle");
    }

    public static void main(String[] args) {
        Circle circle = new Circle();
    }

}
```

В результате выполнения данного кода в консоль будет выведено следующее:

```
Конструктор Figure
Конструктор Point
Конструктор Circle
```

В самом начале был проинициализирован базовый класс `Figure`, затем произошла инициализация поля `center` и только после этого был вызван конструктор класса `Circle`.

Таким образом, порядок инициализации сложного объекта можно описать следующей последовательностью действий.

- Сначала идет вызов конструктора базового класса. Этот шаг рекурсивный: сначала конструируется самый базовый класс иерархии, после него — его ближайший наследник и так далее до тех пор, пока не будет достигнут текущий класс.
- Производится инициализация полей класса в порядке их объявления[3].
- Вызывается тело конструктора текущего класса.

# 6 Полиморфизм

В общем виде полиморфизм — это предоставление единого интерфейса для сущностей разных видов. Применительно к классам полиморфизм означает возможность класса выступать в программе в роли любого из

[3]Напомним, что поля всегда инициализируются до входа в тело конструктора, поэтому в рассмотренном выше примере это правило применяется ко всей цепочке наследования.

своих предков, несмотря на то, что в нем может быть изменено поведение, т. е. переопределена реализация методов предка.

Изменить работу любого из методов, унаследованных от родительского класса, в классе-потомке можно, описав новый метод с точно таким же именем и параметрами. Это называется *переопределением*. При вызове такого метода для объекта класса-потомка будет выполнена новая реализация. Очевидно, что переопределение относится к полиморфизму[4].

Давайте расширим наш класс `Dog` классом `BigDog` для того, чтобы особым образом моделировать поведение больших собак. В частности, большие собаки лают громче и дольше. Поэтому переопределим метод `voice()`:

```
class BigDog extends Dog {
    ...

    @Override
    public void voice() {
        System.out.print("Собака " + name + " залаяла: ");
        for (int i = 1; i <= 29; i++) {
            System.out.print("ГАВ-");
        }
        System.out.print("ГАВ!");
    }

}
```

Обратите внимание, что перед переопределенным методом `voice()` стоит конструкция `@Override`. Это аннотация, особая форма метаданных, которая может быть добавлена в исходный код. Подробно об аннотациях мы поговорим ниже.

Аннотация `@Override` информирует компилятор о том, что вновь определяемый метод переопределяет метод с тем же именем в суперклассе. Использование этой аннотации необязательно, но на практике возможны случаи, когда вызываемый метод класса-наследника не может найти метод класса-предка либо из-за неправильно расставленных модификаторов доступа, либо из-за неправильного расположения классов в пакетах. Если

[4]Стоит отметить, что перегрузку тоже иногда относят к разновидностям полиморфизма, так называемому “ad hoc” полиморфизму.

не использовать аннотацию `@Override`, то "неправильный" код скомпилируется без ошибок, но компилятор будет считать, что метод в классе-наследнике не переопределяет соответствующий метод класса-предка. В такой ситуации найти источник проблемы бывает проблематично, поэтому лучше взять за правило всегда использовать аннотацию `@Override` для переопределенных методов.

Заметим, что для переопределенных методов в Java допускается расширять область видимости, т. е. если родительский метод был, например, `protected`, допускается сделать переопределенный метод `public`. При этом порядок областей от самой узкой к самой широкой считается следующим: область видимости по умолчанию, затем — `protected`, затем — `public`.

Кроме того, для переопределенных методов в Java работает ковариантность возвращаемых типов. Это означает, что переопределенный метод может быть объявлен с возвращаемым типом, производным от типа, возвращаемого методом класса-предка.

Теперь создадим в методе `main()` двух разных собак: обычную и большую — и попросим полаять.

```
Dog dog = new Dog("Тузик", 2);
dog.voice();
BigDog bigdog = new BigDog();
bigdog.voice();
```

Объект дочернего класса всегда будет одновременно являться объектом любого из своих суперклассов, т. е. с ним можно работать, как с любым другим экземпляром суперкласса. Поэтому в том же примере мы могли бы обойтись и одной переменной:

```
Dog dog = new Dog("Тузик", 2);
dog.voice();
dog = new BigDog();
dog.voice();
```

То есть переменная `dog` имеет тип `Dog`, но в третьей строке она начинает указывать на объект класса `BigDog`, т.е. на большую собаку, которая при вызове метода `voice()` будет громко лаять. Метод, который нужно вызвать, определяется во время исполнения кода и достигается за счет так называемого *позднего связывания* ("late binding"), также называемого динамическим.

В целом, вызов метода в Java означает, что этот метод привязывается к конкретному коду или в момент компиляции, или во время выполнения программы. Первый вариант называют *статическим связыванием*, второй — *динамическим*.

Статическое связывание имеет неизменный характер, так как происходит во время компиляции, т. е. в байт-коде описано, какой метод вызывать. Так как компиляция — это этап первой стадии жизненного цикла программы, то применяют также термин "раннее связывание" ("early binding").

Динамическое, или позднее связывание, которое мы видели в примере, происходит во время выполнения, после запуска кода виртуальной машиной Java (JVM). В этом случае вызываемый метод определяется на основании конкретного объекта, так что в момент компиляции эта информация недоступна.

Динамическое связывание в ООП языках, в частности в Java[5], обычно реализуется за счет таблицы виртуальных методов. Поэтому иногда методы, которые могут быть переопределены в классах-наследниках, называют виртуальными.

Раннее связывание используется в Java для разрешения `static`, `final`[6] и `private` методов, в то время как динамическое связывание в Java используется для всех прочих случаев.

У внимательного и любопытного читателя может возникнуть еще один вопрос относительно переопределения: что будет, если переопределить какой-либо метод класса-предка, который в свою очередь использовался где-то еще в классе-предке? Давайте немного изменим наш пример с собаками для ответа на этот вопрос:

```
class Dog {

    private String name;

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

    public void voice() {
```

[5]С некоторыми оговорками относительно JIT-компиляции.

[6]Эти модификаторы будут рассмотрены ниже.

```
        System.out.println("Собака " + name
        + " залаяла: гав-гав!");
    }

    public void greetOwner() {
        System.out.println("Собака " + name
        + " видит хозяина:");
        voice();
    }

    public String getName() {
        return name;
    }

}

class Wolfhound extends Dog {

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

    @Override
    public void voice() {
        System.out.println("Волкодав "
                + this.getName() + " издает вой.");
    }

    public static void main(String[] args) {
        Wolfhound wolfhound = new Wolfhound("Рэкс");
        wolfhound.greetOwner();
    }

}
```

В результате выполнения данного кода в консоль будет выведено следующее:

```
Собака Рэкс видит хозяина:
```

```
Волкодав Рэкс издает вой.
```

Пример демонстрирует, что метод `voice()`, переопределенный в классе-наследнике, будет вызван в теле метода `greetOwner()` класса-предка. Заметьте, что класс-предок может находиться в отдельной библиотеке, во время компиляции которой компилятор никак не может определить, какой именно метод `voice()` нужно будет вызывать в дальнейшем. Корректный вызов методов осуществляется за счет уже рассмотренного позднего связывания.

### Приведение объектных типов

Продолжим рассмотрение полиморфизма в Java. Как мы уже видели, объект класса-потомка можно присвоить переменной типа класса-предка. При этом Java производит автоматическое (неявное) преобразование типа, называемое расширением. *Расширение* — это переход от более конкретного типа к менее конкретному. Напомним, что повышающее преобразование примитивных типов, например переход от `byte` к `int`, — это тоже расширение.

У нас имеется хорошо знакомый класс `Dog`. Добавим в этот класс метод `call(String name)`, который возвращает `true`, если переданная в метод кличка совпадает с кличкой, скрытой в полях класса, т. е. собака отзывается на свою кличку:

```
public class Dog {
    ...

    public boolean call(String name) {
        return this.name.equals(name);
    }

    ...
}
```

Обратите внимание на то, что в этом примере значения строк сравниваются не через логический оператор `==`, а через метод `equals`. О сравнении объектов мы подробно поговорим ниже.

Мы уже унаследовали от класса `Dog` класс `BigDog`, теперь унаследуем еще и класс `SmallDog`, маленькая собака. Маленькую собаку можно носить в переноске. Создадим в этом классе булевское поле `inBag`, равное

`true`, если собака сидит в переноске, и `false` — иначе. Определим также методы `putInBag()` и `pullOutBag`, помещающие в переноску и вынимающие из нее, соответственно:

```
public class SmallDog extends Dog {
    ...

    private boolean inBag;

    public void putInBag() {
        inBag = true;
    }

    public void pullOutBag() {
        inBag = false;
    }

    ...
}
```

Создадим массив `Dog[]`[7], содержащий всех собак, которых мы рассматриваем (и больших, и маленьких). Элементы этого массива имеют тип `Dog`, но мы можем присваивать им ссылки на объекты как класса `BigDog`, так и класса `SmallDog`. В этот момент и будет происходить расширение типа. Продемонстрируем это:

```
Dog[] dogs = new Dog[3];

BigDog bigDog = new BigDog("Полкан", 4);
dogs[0] = bigDog;
// Java проводит автоматическое преобразование
// типа BigDog к типу Dog, чтобы поместить переменную bigDog
// в массив dogs

// дальнейшая инициализация массива
...

String name = "Моська";
```

[7] О массивах мы поговорим подробно ниже.

```
Dog current = null;
// теперь найдем собаку, откликающуюся
// на нужную нам кличку
for (int i = 0; i < dogs.length; i++) {
    if (dogs[i].call(name)) {
        current = dogs[i];
    }
}
```

Несмотря на то, что все объекты, добавленные в массив, сохраняют свой "настоящий" класс, программа работает с ними как с объектами класса `Dog`. Этого вполне достаточно, чтобы можно было найти нужную собаку по кличке и присвоить ее адрес в памяти переменной `current` типа `Dog`.

Предположим, нам известно, что переменная `current` сейчас ссылается на объект класса `SmallDog` и мы хотим поместить эту собаку в переноску. Необходимо вызвать метод `putInBag()`, но у нас не получится сделать это напрямую: при выполнении команды

```
current.putInBag();
```

появится сообщение об ошибке компилятора.

Дело в том, что в классе `Dog` просто нет метода `putInBag()`. Однако мы можем осуществить явное преобразование переменной `current` к типу `SmallDog`. Такое преобразование, т. е. переход от менее конкретного типа к более конкретному, называется сужением. По аналогии с примитивными типами явное преобразование делается с помощью оператора, представляющего собой имя целевого типа в скобках:

```
SmallDog smallDog = (SmallDog) current;
smallDog.putInBag();
```

В этом примере мы, прежде чем вызвать метод `putInBag()`, преобразовали `current` к типу `SmallDog`. У нас это получилось, поскольку `SmallDog` является потомком класса `Dog`. Однако, если бы во время выполнения программы оказалось, что на самом деле переменная `current` не ссылалась на объект класса `SmallDog` (а например, на `BigDog`), во время выполнения последней программы снова возникла бы ошибка.

Чтобы уточнить, соответствует ли текущее значение переменной конкретному типу, используется оператор `instanceof`, который принимает

два операнда — проверяемые объект и класс. Проверим, не является ли рассматриваемая собака действительно маленькой:

```
if (current instanceof SmallDog) {
    SmallDog smallDog = (SmallDog) current;
    smallDog.putInBag();
}
```

Этот код уже заведомо не приведет к ошибке. Однако на практике сужение используют только в крайних случаях, предпочитая ему работу через более общие типы, в качестве которых зачастую используют абстрактные классы или интерфейсы.

# 7 Абстрактные классы

Кроме обычных классов, в Java имеются так называемые абстрактные классы. Абстрактный класс похож на обычный тем, что в нем также можно определить любые поля и методы. Но в то же время нельзя создать объект (экземпляр) абстрактного класса. Абстрактные классы призваны описывать базовое поведение классов-наследников, т. е. частично реализовывать их функции и описывать их интерфейс. А производные классы абстрактного класса уже реализуют эти функции полностью.

При определении абстрактных классов используется ключевое слово `abstract` перед словом `class`:

```
public abstract class Human {

    private String name;

    public String getName() { return name; }

}
```

Мы не можем использовать конструктор абстрактного класса для создания его объекта. Например, так нельзя:

```
Human h = new Human();
```

Кроме обычных методов, абстрактный класс может также содержать абстрактные методы. Такие методы определяются с помощью ключевого

слова `abstract`, они описывают метод, т. е. его сигнатуру и возвращаемый тип, но не имеют тела. Приведем пример:

```
public abstract void display();
```

Фактически это только заголовок — своеобразное описание того, что производный класс переопределит и реализует. Очевидно, что абстрактные методы из-за своей сути не могут иметь модификатор `private`. Еще стоит отметить, что в других методах абстрактного класса могут вызываться его же абстрактные методы.

Заметим также, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

Зачем нужны абстрактные классы? Вернемся к рассмотренному ранее примеру с собаками. Допустим, у нас имеется питомник для собак (или ветеринарная клиника), для которого нужна база данных. У нас имеется класс `Dog`, а также много разных классов с породами собак (`Dalmatian` — далматин, `Husky` — хаски, `Bulldog` — бульдог и т. д.), производными от класса `Dog`. Очевидно, что в этом случае для каждой собаки мы будем создавать объект класса, соответствующего ее породе[8], а объектов класса `Dog` создавать не будем. Имеет смысл сделать класс `Dog` абстрактным и прописать в нем все общее для всех пород не абстрактными полями и методами (кличка и возраст), а то, что зависит от породы — абстрактными (тип лая, наличие пятен, умение вилять хвостом, необходимость купировать уши и т. д.). Рассмотрим это на примере:

```
public abstract class Dog {

    private String name;
    private int age;

    public Dog(String name, int age) {
        this.name=name;
        this.age=age;
    }

    public abstract void voice();
```

[8]Если уж быть совсем аккуратными, можно завести несколько специфических “пород” типа “маленькая дворняжка” и “помесь овчарки и питбуля”.

```
    public void greetOwner() {
        System.out.println("Собака " + name
        + " видит хозяина:");
        voice();
    }

    // getter'ы и т.~д.
    ...
}

public class Dalmatian extends Dog {

    int numOfSpots;

    public Dalmatian(String name, int age, int numOfSpots) {
        super(name, age);
        this.numOfSpots = numOfSpots;
    }

    public void voice() {
        System.out.println("Далматин по кличке "
            + this.getName()
            + " протяжно залаял: ``Гав-гав-гав!''");
    }

    ...
}

public class Husky extends Dog {

    private boolean blueEyes;

    public Husky(String name, int age, boolean blueEyes) {
        super(name,age);
        this.blueEyes = blueEyes;
    }
```

```
    public void voice() {
        System.out.println("Собака хаски по кличке "
            + getName() + " залаяла: ‘‘Гав-гав!’’");
    }

    ...
}

public class Main {

    public static void main(String[] args) {
        Dog marshall = new Dalmatian("Marshall", 2, 100);
        zeus.voice();
        Dog everest  = new Husky("Everest", 5, true);
        rocky.voice();
    }

}
```

Распространенным примером абстрактного класса является класс геометрических фигур. В реальности просто геометрической фигуры не существует, так как это абстрактное понятие. Имеются частные случаи — круг, прямоугольник, квадрат, но просто фигуры нет. Поэтому фигуры удобно описать как абстрактный класс, а круги квадраты и прочее — как дочерние классы. Приведем простой пример реализации этой идеи:

```
public abstract class SymmetricalShape {

    private float x; // x-координата центра
    private float y; // y-координата центра

    public SymmetricalShape(float x, float y) {
        this.x = x;
        this.y = y;
    }

    public abstract float getPerimeter();

    public abstract float getArea();
```

```
    // getter'ы и т.~д.
    ...
}

// производный класс - прямоугольник
public class Rectangle extends SymmetricalShape {

    private float width;
    private float height;

    public Rectangle(float x, float y,
                     float width, float height) {
        super(x, y);
        this.width = width;
        this.height = height;
    }

    @Override
    public float getPerimeter() {
        return width * 2 + height * 2;
    }

    @Override
    public float getArea() {
        return width * height;
    }

    ...
}
```

# 8 Интерфейсы

Мы уже неоднократно употребляли понятие "интерфейс" в основном в смысле некого протокола поведения класса. Но в Java интерфейсом называется отдельная сущность, о которой мы сейчас поговорим.

Ключевое слово `interface` используется для создания интерфейсов,

описывающих поведение реализующих их классов. Интерфейс описывает абстрактные методы и константы. В некотором отношении интерфейсы можно рассматривать как полностью абстрактные классы. Как и для абстрактных классов, невозможно создать экземпляр интерфейса через ключевое слово `new`. Например, мы могли бы описать интерфейс для предыдущего примера про фигуры:

```
public interface SymmetricalShape {

    float PI = 3.14; // компилятор автоматически подставляет
    // public final для констант

    float getPerimeter(); // компилятор автоматически
    // подставляет public abstract для методов

    float getArea();

}
```

Любой код, использующий какой-то интерфейс, знает только то, какие методы и константы описывает этот интерфейс, и то, что эти методы можно вызвать у текущего объекта, но не более того. Класс, который использует определенный интерфейс, содержит в заголовке ключевое слово `implements`.

Изменим знакомый нам пример с классом для прямоугольника так, чтобы он реализовал интерфейс:

```
public class Rectangle implements SymmetricalShape {

    private float width;
    private float height;

    // конструктор, getter-методы и т.~д.
    ...

    @Override
    public float getPerimeter() {
        return width * 2 + height * 2;
    }
```

```
    @Override
    public float getArea() {
        return width * height;
    }

    public static void main(String[] args) {
        SymmetricalShape shape = new SymmetricalShape();
        // ошибка компиляции
        SymmetricalShape rectangle = new Rectangle();
        // корректный код
        float circleArea = SymmetricalShape.PI * 2 * 2;
        // использование констант
    }

}
```

Интерфейс, как и обычный класс, может быть публичным (с модификатором доступа `public` в заголовке) или с доступом по умолчанию (без модификатора), если он используется только в пределах своего пакета. Интерфейс может содержать поля-константы, которые автоматически являются статическими (`static`) и неизменными (`final`)[9]. Все методы интерфейса неявно объявляются как `public` и `abstract`.

Если класс содержит интерфейс, но реализует не все определенные им методы, он должен быть объявлен как абстрактный (с ключевым словом `abstract`).

Как мы уже говорили, в Java нет множественного наследования, но классы могут реализовывать несколько интерфейсов, что частично компенсирует это ограничение. Если интерфейсов у класса несколько, то все они перечисляются за ключевым словом `implements` и разделяются запятыми.

Интерфейс может расширять другой интерфейс через ключевое слово `extends`. Приведем соответствующий пример:

```
public interface Dog {
    void voice();
}
```

[9] О ключевых словах `static` и `final` речь пойдет во второй части пособия.

```
public interface BigDog extends Dog {
    void bite(String target);
}

public class Wolfhound implements BigDog {
    ...

    @Override
    public void voice() {
        ...
    }

    @Override
    public void bite(String target) {
        ...
    }

}
```

Приведем пример класса, реализующего несколько интерфейсов:

```
public interface Printable {
    void print();
}

public interface Transferable {
    void transfer(String destination);
}

public class Book implements Printable, Transferable {

    private String name;
    private String author;

    public Book(String name, String author) {
        this.name = name;
        this.author = author;
    }
```

```
    @Override
    public void print() {
        System.out.prinln("Книга " + name + " - " + author);
    }

    @Override
    public void transfer(String destination) {
        System.out.prinln("Книга " + name + " - "
            + author + " была отправлена по адресу "
            + destination);
    }

    ...
}

public class Main {

    public static void main(String[] args) {
        Book b1 = new Book("Java. Complete Reference",
            "H. Shildt");
        b1.print();
        b1.transfer("Сосед по парте");

        Printable b2 =
            new Book("Thinking In Java. 4th Edition",
                "B. Eckel");
        b2.print();

        Transferable b3 =
            new Book("The Java Language Specification, "
                + "Java SE. 10 Edition", "J. Gosling");
        b3.transfer("Соседка по парте");
    }

}
```

В рассмотренном примере класс `Book` реализует интерфейсы `Printable`

и `Transferable`. Напомним, что в этом случае класс должен реализовать все методы интерфейсов, что в нашем примере означает методы `print()` и `transfer()`. В дальнейшем работать с объектом класса `Book` можно как через переменную самого класса, так и через переменную типа одного из интерфейсов. Во втором случае можно вызывать у объекта только методы, объявленные в интерфейсе.

Внимательный читатель мог задать себе вопрос, что будет в случае, если в разных интерфейсах имеются методы с одинаковым именем. Приведем такой пример:

```
public interface A {
    void sum(int value);
    int multi(int value);
}

public interface B {
    void sum(float value);
    int multi(int value);
}

public class C implements A, B {

    @Override
    public void sum(int value) { }

    @Override
    public void sum(float value) { }

    @Override
    public int multi(int value) { return 0; }

}
```

Как видно из примера, никаких проблем такая ситуация не создает. Компилятор считает, что метод `multi()` является реализацией одноименных методов из обоих интерфейсов; проблем не возникает, поскольку в обоих интерфейсах методы означают одно и то же. Что касается методов `sum()`, то поскольку они имеют различающиеся сигнатуры, это разные методы,

требующие отдельных реализаций — в зависимости от типа аргумента обращение будет происходить к соответствующему методу.

Давайте теперь расширим наш пример про книги. В дополнение к классу `Book` определим еще один класс, который будет реализовывать интерфейс `Printable`:

```
public class Magazine implements Printable {

    private String name;

    public String getName() {
        return name;
    }

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

    @Override
    public void print() {
        System.out.println("Журнал " + name);
    }

}
```

Здесь оба класса `Book` и `Journal` реализуют один и тот же интерфейс `Printable`. Поэтому мы динамически можем создавать в программе экземпляры обоих классов и обрабатывать их через интерфейс `Printable`. Продемонстрируем это:

```
public class Main {

    public static void main(String[] args) {
        Printable p1 = new Book("Le tour du monde en "
            + "quatre-vingts jours", "Jules Vern");
        p1.print();

        Printable p2 = new Magazine("National Geographic. "
            + "Issue Jan 2017");
```

```
        p2.print();
    }

}
```

# 9 Отношения: композиция, агрегация, ассоциация, делегирование

*Композиция* — механизм построения нового класса из объектов уже имеющихся классов. При этом между новым классом и имеющимися явно прослеживается отношение "часть – целое" ("has a"). Иными словами, сложный объект состоит из более простых и делегирует им какую-то работу.

Например, автомобиль состоит из двигателя, КПП, кузова, колес и т. д. Давайте в общих чертах опишем соответствующий пример:

```
public class Engine {
    ...
}

public class Gearbox {
    ...
}

...

public class Car {

    private Engine engine;
    private Gearbox gearbox;
    ...

    public void go() {
        engine.start();
        gearbox.selectDriveMode();
        engine.increaseThrottle(2000);
        ...
    }
```

```
}
```

Заметим, что при композиции мы, в отличие от наследования, не можем повлиять на функциональность используемых классов, например, переопределить метод или заменить значение по умолчанию для некоторого поля.

*Делегирование* — это отношение между классами, когда новый класс передает выполнение запрошенного действия связанному с ним объекту другого класса. В реальном мире мы встречаем подобное очень часто: руководитель, получив новую задачу, делегирует ее одному из своих подчиненных. Приведем пример делегирования:

```
public class Task {
    ...
}

public class WorkerProcess {
    public void processTask(Task task);
}

public class ProcessManager {

    private WorkerProcess worker;
    ...

    public void processTask(Task task) {
        worker.processTask(task);
    }

}
```

Обычно делегированию противопоставляют реализацию чего-либо в том же самом классе. Например, если в программе имеется класс, выполняющий много функций, желательно разделить его на несколько более простых классов, каждый из которых будет иметь свою зону ответственности. Главный класс будет осуществлять диспетчеризацию вызовов обращений к этими классами.

Композиции же обычно противопоставляют агрегацию и ассоциацию. *Агрегация* похожа на композицию тем, что это тоже отношение “часть –

целое", но разница в том, что между объектами нет отношения вложения. Например, "группа студентов": студент — часть группы, но студент может существовать и вне группы. *Ассоциация* — это более общее понятие, которое выражает любое отношение между объектами, имеющими возможность вызывать методы друг друга.

На самом деле, делегирование можно устроить, имея как отношение ассоциации, так и отношение композиции между объектами.

В заключение нельзя не упомянуть, что композиции противопоставляют наследование. В этом случае композиция и делегирование — синонимы. В случае с наследованием мы выносим общие методы в класс-предок, а различные их реализации — в классы-наследники. Создавая экземпляр одного из классов-наследников, мы получаем необходимую функциональность объекта. На практике иерархии наследования более 2–3 уровней глубины оказываются сложны в поддержке и рискованы с точки зрения возникновения ошибок при модификациях кода. Таким многоуровневым иерархиям стоит предпочесть рассмотренные выше отношения.

# Предметный указатель

# Библиографический список


[1] *Блох Дж.* Java. Эффективное программирование / Дж. Блох. — 3-е изд. — М. : Диалектика, 2019. — 464 с.

[2] *Вирт Н.* Алгоритмы и структуры данных / Н. Вирт. — М. : ДМК Пресс, 2016. — 272 с. — (Классика программирования).

[3] *Курбатова И. В.* Решение комбинаторных задач на языке программирования Java : учеб.-метод. пособие / И. В. Курбатова, М. А. Артемов, Е. С. Барановский. — Воронеж : Издательский дом ВГУ, 2018. — 42 с.

[4] *Шильдт Г.* Java 8. Руководство для начинающих / Г. Шильдт. — М. : Вильямс, 2018. — 720 с.

[5] *Эккель Б.* Философия Java / Б. Эккель. — 4-е изд. — СПб. : Питер, 2009. — 640 с. — (Библиотека программиста).

[6] *Arnold K.* The Java programming language / K. Arnold, J. Gosling, D. Holmes. — Forth edition. — Boston, MA : Addison Wesley Professional, 2005. — 928 p.

[7] *Hill E. F.* Jess in action : Java rule-based systems / E. F. Hill. — Greenwich, CT : Manning Publications Co., 2003. — xxxii+443 p.

[8] Java Concurrency in Practice / D. Lea, D. Holmes, J. Bowbeer [et al.]. — First edition. — Boston, MA : Addison-Wesley Professional, 2006. — xx+404 p.

[9] The Java language specification / J. Gosling, B. Joy, G. Steele [et al.]. — Eighth edition. — Boston, MA : Pearson Education and Addison-Wesley Professional, 2015. — xx+768 p.

[10] The Java virtual machine specification / T. Lindholm, F. Yellin, G. Bracha, A. Buckley. — Eighth edition. — Boston, MA : Addison-Wesley Professional, 2014. — xvi+581 p.

[11] *Oaks S.* Java Performance: The Definitive Guide / S. Oaks. — First edition. — Beijing–Cambridge : O'Reilly Media, Inc., 2014. — xiv+409 p.

У ч е б н о е  и з д а н и е

**Курбатова Ирина Витальевна,**
**Печкуров Андрей Викторович**

# ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

*Учебное пособие*

Ч а с т ь 1

**Основы синтаксиса и его применение**
**в объектно-ориентированном программировании**

Редактор *В. Г. Холина*
Компьютерная верстка *И. В. Курбатовой*

Подписано в печать 17.04.2019. Формат 60 × 84/16.
Уч.-изд. л. 4,9. Усл. печ. л. 4,7. Тираж 50 экз. Заказ 99.

Издательский дом ВГУ
394018 Воронеж, пл. Ленина, 10

Отпечатано с готового оригинал-макета
в типографии Издательского дома ВГУ
394018 Воронеж, ул. Пушкинская, 3