

В. В. КОНКИНА, А. П. ПУХА, К. И. ПАТУТИН

РАЗРАБОТКА НА JAVA



**Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2026**

Министерство науки и высшего образования Российской Федерации

**Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тамбовский государственный технический университет»**

В. В. КОНКИНА, А. П. ПУХА, К. И. ПАТУТИН

РАЗРАБОТКА НА JAVA

Утверждено Ученым советом
ФГБОУ ВО «Тамбовский государственный технический университет»
в качестве учебного пособия для студентов 2 курса направлений подготовки
09.03.01 «Информатика и вычислительная техника»
и 09.03.03 «Прикладная информатика», изучающих дисциплину
«Программирование на языке высокого уровня»,
всех форм обучения

Учебное электронное издание



Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2026

УДК 004.4
ББК 16.23
К64

Рецензенты:

Кандидат технических наук, ведущий инженер-программист
АО НТПП «Комплексное развитие технологий» (г. Москва)
С. П. Стрыгин

Кандидат технических наук, доцент, доцент кафедры
«Информационные процессы и управление» ФГБОУ ВО «ТГТУ»
И. А. Елизаров

Конкина, В. В.

К64 Разработка на Java [Электронный ресурс] : учебное пособие /
В. В. Конкина, А. П. Пуха, К. И. Патутин. – Тамбов : Издательский центр
ФГБОУ ВО «ТГТУ», 2026. – 1 электрон. опт. диск (CD-ROM). –
Системные требования : ПК не ниже класса Pentium IV ; RAM 512 Mb ;
необходимое место на HDD 2,5 Mb ; Windows 7/8/10/11 ; дисковод
CD-ROM ; мышь. – Загл. с экрана.
ISBN 978-5-8265-3050-4

Описаны типы данных и арифметические выражения, структуры выбора, математические функции, символы и строки, циклы, методы, одномерные массивы, многомерные массивы, рекурсия. Представлены объекты и классы, объектно-ориентированное мышление, наследование и полиморфизм, обработка исключений, абстрактные классы и интерфейсы, дженерики.

Предназначено для студентов 2 курса направлений подготовки 09.03.01 «Информатика и вычислительная техника» и 09.03.03 «Прикладная информатика», изучающих дисциплину «Программирование на языке высокого уровня», всех форм обучения.

УДК 004.4
ББК 16.23

*Все права на размножение и распространение в любой форме остаются за разработчиком.
Нелегальное копирование и использование данного продукта запрещено.*

ISBN 978-5-8265-3050-4

© Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тамбовский государственный технический
университет» (ФГБОУ ВО «ТГТУ»), 2026

*«Программирование – это нечто большее,
чем практическое искусство.
Это также гигантское предприятие
в области основ знания»*

*Грейс Хоппер**

* Грейс Хоппер (9 декабря 1906 –1 января 1992) – американская ученая и коммодор флота США. Будучи первооткрывательницей в своей области, она была одной из первых, кто писал программы для гарвардского компьютера Марк I. Она разработала первый компилятор для компьютерного языка программирования, развила концепцию машинно-независимых языков программирования, что привело к созданию COBOL, одного из первых высокоуровневых языков программирования. Ей приписывается популяризация термина debugging для устранения сбоев в работе компьютера.

ВВЕДЕНИЕ

Язык Java был разработан в компании Sun Microsystems командой под руководством Джеймса Гослинга в 1995 году. Компания Sun Microsystems была приобретена Oracle в 2010 году. Первоначально называвшийся Oak язык Java с 1991 года использовался в чипах, встроенных в бытовые электронные устройства, а в 1995 году был переименован в Java и переработан для разработки веб-приложений.

Ранее Java-программы могли запускаться из веб-браузера, такие программы назывались апплетами. В настоящее время из-за проблем с безопасностью апплеты больше не запускаются из веб-браузера.

Java – это полнофункциональный язык программирования общего назначения, который можно использовать для разработки надежных приложений. В настоящее время язык Java достаточно популярен для разработки веб-серверных приложений, которые обрабатывают данные, выполняют вычисления.

1. ОСНОВЫ JAVA-ПРОГРАММИРОВАНИЯ

1.1. СТИЛЬ ПРОГРАММИРОВАНИЯ И ДОКУМЕНТИРОВАНИЯ

Хороший стиль программирования и документирования облегчает чтение программы и помогает избежать ошибок.

В начале программы рекомендуется размещать резюме с пояснением функционала программы. В сам текст следует включать комментарии, которые поясняют важные точки программы. Комментарии должны быть лаконичными, не перегружающими программу и не затрудняющими ее прочтение.

Java поддерживает однострочные комментарии (начинающиеся с `//`), блочные комментарии (начинающиеся с `/*`), а также комментарии специального типа, называемые комментариями javadoc. Комментарии javadoc начинаются с `/**` и заканчиваются `*/`. Их можно извлечь в HTML-файл с помощью JDK-команды `javadoc`.

Комментарии javadoc (`/** ... */`) используются для комментирования целого класса или целого метода. Эти комментарии должны предшествовать классу или заголовку метода. Для комментирования шагов внутри метода используются однострочные комментарии (`//`).

Согласованный стиль отступов делает программы понятными и простыми для чтения, отладки и сопровождения. Отступы используются для иллюстрации структурных связей между компонентами или предложениями программы. Java может прочитать программу, даже если все предложения находятся на одной-единственной длинной строке, но людям легче читать и поддерживать правильно выровненный код. Отступ рекомендуется делать для каждого подкомпонента или предложения хотя бы на два пробела больше, чем для конструкции, в которую он вложен.

С обеих сторон бинарных операторов должны быть добавлены единичные пробелы, как показано далее в коде слева, в отличие от кода справа.

```
System.out.println(3 + 4 * 4);
```

Хороший стиль

```
System.out.println(3+4*4);
```

Плохой стиль

Блок – это группа предложений, заключенная в фигурные скобки. Существует два популярных стиля: стиль следующей строки и стиль конца строки, как показано далее.

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Стили блока");
    }
}
```

Стиль следующей строки

```
public class Test {
    public static void main(String[] args) {
        System.out.println("Стили блока ");
    }
}
```

Стиль конца строки

Стиль следующей строки выравнивает фигурные скобки по вертикали и облегчает чтение программ, в то время как стиль конца строки экономит место

и может помочь избежать некоторых ошибок программирования. Оба стиля являются приемлемыми стилями блоков. Выбор зависит от личных или корпоративных предпочтений. Не рекомендуется смешивать стили блоков.

1.2. ОШИБКИ ПРОГРАММИРОВАНИЯ

Выделяют три типа ошибок программирования: синтаксические, ошибки во время выполнения и логические.

Синтаксические ошибки являются результатом ошибок в написании кода, например, опечатки в зарезервированных словах, пропуск необходимых знаков препинания или использование открывающей скобки без соответствующей ей закрывающей скобки. Эти ошибки обычно легко обнаружить, т.к. компилятор сообщает, где они находятся и что их вызвало.

Ошибки во время выполнения программы приводят к аварийному завершению программы, если обнаруживается операция, которую невозможно выполнить. Например, если программа ожидает ввод числа, а вместо этого пользователь вводит строку, возникает ошибка типов данных.

Логические ошибки возникают, когда программа работает не так, как предполагалось.

Распространенные ошибки:

1. Отсутствие скобки. Фигурные скобки используются для обозначения блока в программе. Каждая открывающая скобка должна соответствовать закрывающей. Чтобы избежать этой ошибки, рекомендуется набирать закрывающую скобку при каждом наборе открывающей скобки:

```
public class Welcome {  
} // <- сразу же наберите }, чтобы она соответствовала {
```

2. Отсутствие точки с запятой. Каждое предложение заканчивается символом точки с запятой (;). Частая ошибка – не поместить этот символ для последнего предложения в блоке:

```
public static void main(String[] args) {  
    System.out.println("Programming is fun!");  
    System.out.println("Fundamentals First");  
    System.out.println("Problem Driven") // <- отсутствует ;  
}
```

3. Отсутствие двойной кавычки. Строка должна быть помещена в двойные кавычки. Частая ошибка – не поместить двойную кавычку в конце строки:

```
System.out.println("Problem Driven"); // <- отсутствует "
```

4. Ошибка в имени. Java чувствителен к регистру. Например, main может быть написано с ошибкой как Main, а String – как string:

```
public class Test {  
    public static void Main(string[] args) {  
        System.out.println((10.5 + 2 * 3) / (45 - 3.5));  
    }  
}
```

Распространенные элементарные ошибки программирования часто включают в себя необъявленные и неинициализированные переменные, целочисленное переполнение, непреднамеренное целочисленное деление и ошибки округления.

1. Необъявленные/неинициализированные и неиспользуемые переменные.

Переменная должна быть объявлена с конкретным типом данных и ей должно быть присвоено значение перед ее использованием. Распространенная ошибка заключается в необъявленной или неинициализированной переменной. Рассмотрим следующий фрагмент кода:

```
double interestRate = 0.05;  
double interest = interestrate * 45;
```

Этот код содержит ошибку, потому что переменной `interestRate` присваивается значение `0.05`, а переменная `interestrate` не была объявлена и инициализирована. Java чувствителен к регистру, поэтому он принимает `interestRate` и `interestrate` за две различные переменные.

Если переменная объявлена, но не используется, это может быть потенциальной ошибкой программирования, поэтому неиспользуемую переменную необходимо удалить из программы. Например, в следующем фрагменте кода переменная `taxRate` нигде не используется, поэтому ее следует удалить.

```
double interestRate = 0.05;  
double taxRate = 0.05;  
double interest = interestRate * 45;  
System.out.println("Процент равен " + interest);
```

2. Целочисленное переполнение.

Числа в компьютере хранятся с ограниченным количеством знаков. Если переменной присвоено значение, слишком большое для сохранения, это приведет к переполнению. Например, следующее предложение приведет к переполнению, поскольку наибольшее значение, которое может быть сохранено в переменной типа `int`, является `2147483647`. Значение `2147483648` слишком большое для сохранения его в качестве значения типа `int`:

```
int value = 2147483647 + 1; // фактически значение будет равно -2147483648
```

Аналогично следующее предложение также приведет к переполнению, поскольку наименьшее значение, которое может быть сохранено в переменной типа `int`, равно `-2147483648`. Значение `-2147483649` слишком большое для сохранения его в качестве значения типа `int`:

```
int value = -2147483648 - 1; // фактически значение будет равно 2147483647
```

В Java не сообщается о предупреждениях или ошибках при переполнении, поэтому особое внимание следует уделять целым числам, близким к максимальному или минимальному значениям в диапазоне этого типа данных.

Если число с плавающей запятой слишком маленькое для сохранения (т.е. стремится к нулю), то это приведет к незаполнению (потере значащих разрядов). Java такое число приближает к нулю, поэтому обычно не нужно беспокоиться о незаполнении.

3. Ошибка округления.

Ошибка округления – это разница между вычисленным приближенным значением числа и его точным математическим значением. Например, $1/3$ приближенно равно 0.333, если сохранить три десятичных знака, и приближенно равно 0.3333333, если сохранить семь десятичных знаков. Поскольку количество цифр, которые могут храниться в переменной, ограничено, ошибки округления неизбежны. Вычисления, включающие числа с плавающей запятой, аппроксимируются, поскольку эти числа не сохраняются с абсолютной точностью. Например,

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);
```

отображает 0.50000000000000001, а не 0.5, и

```
System.out.println(1.0 - 0.9);
```

отображает 0.09999999999999998, а не 0.1. Целые числа сохраняются абсолютно точно, поэтому вычисления с целыми числами дают точный целочисленный результат.

4. Непреднамеренное целочисленное деление.

В Java используется один и тот же оператор деления, а именно `/`, как для выполнения деления целых чисел, так и чисел с плавающей точкой. Если два операнда являются целыми числами, то оператор `/` выполняет целочисленное деление. Результатом этой операции является целое число, а дробная часть отбрасывается. Чтобы поделить два целых числа не нацело, а как числа с плавающей запятой, необходимо сделать одно из целых чисел числом с плавающей точкой. Например, код слева отображает среднее значение как 1, а код справа отображает среднее значение как 1.5.

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) / 2;
System.out.println(average);
```

```
int number1 = 1;
int number2 = 2;
double average = (number1 + number2) /
2.0;
System.out.println(average);
```

5. Избыточные объекты типа Scanner.

Например, следующий код считывает целое и вещественное число:

```
Scanner input = new Scanner(System.in);
System.out.print("Введите целое число: ");
int v1 = input.nextInt();
Scanner input1 = new Scanner(System.in); // плохой код
System.out.print("Введите вещественное число: ");
double v2 = input1.nextDouble();
```

Здесь без необходимости создаются два входных объекта `input` и `input1`, что может привести к некоторым специфическим ошибкам, поэтому этот код следует переписать следующим образом:

```
Scanner input = new Scanner(System.in); // ХОРОШИЙ КОД
System.out.print("Введите целое число: ");
int v1 = input.nextInt();
System.out.print("Введите вещественное число: ");
double v2 = input.nextDouble();
```

Распространенные ошибки в структурах выбора – это пропущенные фигурные скобки, завершение предложения `if` в неподходящем месте, принятие `==` за `=` и висящие `else`, а также повторяющиеся предложения в предложениях `if-else` и проверка на равенство значений типа `double`.

1. Пропуск необходимых фигурных скобок.

Фигурные скобки могут быть опущены, если блок содержит одно предложение. Однако пропуск фигурных скобок, необходимых для группировки нескольких предложений, является обычной ошибкой программирования. При изменении кода посредством добавления новых предложений в предложение `if`, которое изначально было без фигурных скобок, необходимо вставить фигурные скобки. Например, следующий код некорректен.

```
if (radius >= 0) // НЕКОРРЕКТНЫЙ КОД
    area = PI * radius * radius;
    System.out.println("Площадь "
        + " равна " + area + ".");
```

Он должен быть написан с фигурными скобками для группировки нескольких предложений, как показано в следующем коде.

```
if (radius >= 0) { // КОРРЕКТНЫЙ КОД
    area = PI * radius * radius;
    System.out.println("Площадь "
        + " равна " + area + ".");
}
```

В некорректном коде предложение вывода в консоль не является частью предложения `if`. По сути, этот некорректный код совпадает со следующим кодом:

```
if (radius >= 0) // СОВПАДАЕТ С НЕКОРРЕКТНЫМ
    area = PI * radius * radius;
    System.out.println("Площадь "
        + " равна " + area + ".");
```

Таким образом, независимо от условия в предложении `if` предложение вывода в консоль всегда выполняется.

2. Недопустимая точка с запятой в конце строки с `if`.

Добавление точки с запятой в конце строки с `if`, как показано далее, является распространенной ошибкой.

```
if (radius >= 0); // Логическая ошибка
{
    area = PI * radius * radius;
    System.out.println("Площадь "
        + " равна " + area + ".");
}
```

Эту ошибку трудно найти, поскольку она не является ни ошибкой компиляции, ни ошибкой во время выполнения, а является логической ошибкой. Этот код с логической ошибкой эквивалентен следующему коду с пустым блоком.

```
if (radius >= 0) { }; // Пустой блок
{
    area = PI * radius * radius;
    System.out.println("Площадь "
        + " равна " + area + ".");
}
```

Эта ошибка часто возникает при использовании стиля блока следующей строки. Использование стиля блока конца строки может помочь предотвратить эту ошибку.

3. Избыточная проверка логических значений.

Чтобы проверить, имеет ли переменная типа `boolean` значение `true` или `false` в условии, необходимо использовать оператор проверки на равенство, как показано в следующем коде:

```
if (even == true)
    System.out.println("Четное.");
```

Вместо этого лучше напрямую проверить переменную типа `boolean`, как показано в следующем коде:

```
if (even)
    System.out.println("Четное.");
```

Другая веская причина для этого – избежать ошибок, которые трудно обнаружить. Использование оператора `=` вместо оператора `==` для проверки на равенство двух элементов в условии является распространенной ошибкой. Это может привести к следующему некорректному предложению:

```
if (even = true)
    System.out.println("Четное.");
```

Это предложение не содержит ошибок компиляции. В нем присваивается значение `true` переменной `even`, поэтому условие всегда является истинным.

4. Висящий двусмысленный `else`.

Далее код слева содержит два оператора `if` и один `else`. Какому из двух операторов `if` соответствует этот оператор `else`? Отступы указывают, что опера-

топ else соответствует первому оператору if, однако оператор else на самом деле соответствует второму оператору if. Эта ситуация известна как висящий двусмысленный else. Оператор else всегда соответствует последнему беспарному оператору if в том же блоке. Следовательно, код слева эквивалентен коду справа.

```
int i = 1, j = 2, k = 3;

if (i > j)
    if (i > k)
        System.out.println("A");
else
    System.out.println("B");
```

```
int i = 1, j = 2, k = 3;

if (i > j)
    if (i > k)
        System.out.println("A");
else
    System.out.println("B");
```

Поскольку условие $i > j$ является ложным, предложениями кода ни слева, ни справа ничего не отображается. Чтобы привести оператор else в соответствие с первым оператором if, необходимо добавить пару фигурных скобок:

```
int i = 1, j = 2, k = 3;

if (i > j) {
    if (i > k)
        System.out.println("A");
} else
    System.out.println("B");
```

Теперь этим предложением отображается B.

5. Проверка на равенство двух чисел с плавающей точкой.

Как уже обсуждалось в предыдущем разделе, числа с плавающей точкой имеют ограниченную точность в представлении и вычислении. Использование чисел с плавающей точкой может привести к ошибкам округления, поэтому проверка равенства двух значений с плавающей точкой не является надежной. Например, ожидается, что следующий код будет отображать значение true, но, что удивительно, он будет отображать значение false:

```
double x = 1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1;
System.out.println(x == 0.5);
```

Здесь x равен не точно 0.5, а 0.5000000000000001. Нельзя надежно проверить на равенство два значения с плавающей точкой, однако можно сравнить, достаточно ли они близки, проверив, меньше ли разница между этими двумя числами некоторой пороговой величины. Таким образом, два числа x и y очень близки, если $|x - y| < \varepsilon$, где ε — очень маленькая величина. Греческая буква ε , произносимая как «эпсилон», обычно используется для очень маленьких значений. Как правило, устанавливается равным 10^{-14} для сравнения двух значений типа double и 10^{-7} для сравнения двух значений типа float. Например, следующий код

```
final double EPSILON = 1E-14;
double x = 1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1;
if (Math.abs(x - 0.5) < EPSILON)
    System.out.println(x + " приблизительно равен 0.5.");
```

Отобразит

```
0.5000000000000001 приблизительно равен 0.5.
```

Также можно использовать метод `Math.abs(a)` для получения модуля значения `a`.

6. Упрощение присваивания значений логическим переменным.

Присвоение значения условия переменной типа `boolean` показано в следующем коде:

```
if (number % 2 == 0)
    even = true;
else
    even = false;
```

Это не является ошибкой, но лучше написать, как показано в следующем коде:

```
boolean even = number % 2 == 0;
```

7. Избегание дублирования кода в блоках структуры выбора.

Часто в блоках структуры выбора дублируется код, который следует объединять в одном месте. Например, предложение `System.out.println("Стоимость обучения равна " + tuition);` дублируется в следующем фрагменте кода:

```
if (inState) {
    tuition = 5000;
    System.out.println("Стоимость обучения равна" + tuition);
} else {
    tuition = 15000;
    System.out.println("Стоимость обучения равна" + tuition);
}
```

Это не является ошибкой, но лучше написать следующим образом:

```
if (inState) {
    tuition = 5000;
} else {
    tuition = 15000;
}
System.out.println("Стоимость обучения равна" + tuition);
```

В новом коде устранено дублирование, что упрощает сопровождение кода, поскольку его необходимо изменять только в одном месте при изменении предложения `print`.

1.3. РАЗРАБОТКА ПРОСТЫХ Java-ПРОГРАММ

В Java для ссылки на стандартное устройство вывода используется `System.out`, а для ссылки на стандартное устройство ввода – `System.in`. По умолчанию устройством вывода является дисплейный монитор, а устройством ввода – клавиатура. Для того, чтобы отобразить в консоли значение примитивного типа или строку, используется просто метод `println()`. Для того, чтобы выполнить консольный ввод, необходимо использовать класс `Scanner` для создания объекта для считывания входных данных из `System.in` следующим образом:

```
Scanner input = new Scanner(System.in);
```

Синтаксис `new Scanner(System.in)` создает объект типа `Scanner`. Синтаксис `Scanner input` объявляет, что входные данные являются переменной типа `Scanner`. Все предложение `Scanner input = new Scanner (System.in);` создает объект `Scanner` и присваивает его ссылке в качестве значения переменной `input`. Объект может вызывать свои методы. Чтобы вызвать метод объекта, необходимо попросить объект выполнить задачу. Для считывания значения типа `double` можно вызвать метод `nextDouble()` следующим образом:

```
double radius = input.nextDouble();
```

Класс `Scanner` находится в пакете `java.util`. Существует два типа предложений `import`: одиночный и групповой. При одиночном импорте указывается один класс из пакета. Например, в следующем предложении импортируется класс `Scanner` из пакета `java.util`.

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

При групповом импорте указываются все классы в пакете, используя символ звездочки в качестве подстановочного знака. Например, следующее предложение импортирует все классы из пакета `java.util`:

```
import java.util.*;
```

Информация о классах в импортированном пакете не считывается во время компиляции или выполнения программы до тех пор, пока классы не используются. Предложение `import` просто сообщает компилятору, где их найти. Между объявлениями одиночного и группового импорта нет разницы в производительности.

Идентификаторы – это имена, которые однозначно определяют в программе такие элементы, как классы, методы и переменные.

Все идентификаторы должны соответствовать следующим правилам:

- идентификатор – это последовательность символов, состоящая из букв, цифр, символов подчеркивания (`_`) и символов доллара (`$`).
- идентификатор может начинаться с буквы, символа подчеркивания (`_`) или символа доллара (`$`). Идентификатор НЕ может начинаться с цифры.

- идентификатор НЕ может быть ключевым словом.
- идентификатор НЕ может иметь значение true, false или null.
- идентификатор может быть любой длины.

Переменные используются для представления значений, которые могут быть изменены в программе.

Переменные предназначены для представления данных определенного типа. Чтобы использовать переменную, необходимо объявить ее, указав компилятору ее имя, а также тип данных, которые она может хранить. Объявление переменной предписывает компилятору выделить соответствующее место в памяти для переменной на основе ее типа данных.

Синтаксис объявления переменной:

ТипДанных имяПеременной;

Если переменные одного типа, то они могут быть вместе объявлены следующим образом:

ТипДанных имяПеременной1, имяПеременной2, ..., имяПеременнойN;

Переменные часто имеют начальные значения. Можно объявить переменную и инициализировать ее, т.е. присвоить ей начальное значение, в одном предложении. Рассмотрим, например, следующий код:

```
int count = 1;
```

Это одно предложение эквивалентно следующим двум:

```
int count;
count = 1;
```

Можно также использовать сокращенную форму для объявления и инициализации переменных одного типа. Например,

```
int i = 1, j = 2;
```

Прежде чем присваивать переменной значение, ее необходимо объявить. Прежде чем использовать переменную, объявленную в методе, ей необходимо присвоить значение.

Хорошим стилем программирования считается объявление в начале метода всех используемых в нем переменных до присваивания им каких-либо значений.

Каждая переменная имеет область видимости. Область видимости переменной – это часть программы, в которой можно ссылаться на эту переменную.

Предложение с присваиванием назначает значение переменной. Предложение с присваиванием может использоваться в Java в качестве выражения.

После объявления переменной ей можно присвоить значение с помощью оператора присваивания. В Java в качестве оператора присваивания используется символ равенства (=).

Синтаксис предложения с присваиванием:

переменная = выражение.

Именованная константа – это идентификатор, представляющий постоянное значение.

Значение переменной может меняться во время выполнения программы, а именованная константа (или просто константа) представляет постоянное значение, которое никогда не меняется. Константа в Java также известна как `final` (результатирующая) переменная.

Синтаксис объявления константы:

`final ТипДанных ИМЯ_КОНСТАНТЫ = значение.`

Для переменных, констант, классов и методов в программе должны быть выбраны описательные и однозначно интерпретируемые имена. Имена чувствительны к регистру. Ниже представлено соглашение по именованию переменных, методов и классов:

- для именования переменных и методов используются строчные буквы, например, `radius` и `area` – имена переменных, `print()` – имя метода. Если имя состоит из нескольких слов, необходимо объединить их в одно, сделав первое слово строчными буквами, а каждое последующее слово начав с заглавной буквы, например, `numberOfStudents` – имя переменной. Этот стиль именования известен как `camelCase` (букв. «верблюжий регистр»), поскольку заглавные буквы в имени напоминают горбы верблюда.

- каждое слово в имени класса начинается с заглавной буквы, например, `ComputeArea` и `System` – имена классов.

- в имени константы каждая буква заглавная и используется подчеркивание между словами, например, `PI` и `MAX_VALUE` – имена констант.

Соглашение по именованию облегчает чтение программ.

1.4. ТИПЫ ДАННЫХ И АРИФМЕТИЧЕСКИЕ ВЫРАЖЕНИЯ

В Java имеется шесть числовых типов данных для целых и вещественных чисел с операторами `+`, `-`, `*`, `/` и `%`. Для целых чисел в Java используется четыре типа данных: `byte`, `short`, `int` и `long`. Для чисел с плавающей точкой в Java используется два типа данных: `float` и `double`. Диапазон значений типа данных `double` в два раза больше, чем `float`, поэтому значения типа `double` еще называются числами двойной точности, а `float` – одинарной точности. В следующей таблице перечислены методы для считывания чисел типов данных `byte`, `short`, `int`, `long` и `float`:

Метод	Описание
<code>nextByte()</code>	считывает целое число типа <code>byte</code>
<code>nextShort()</code>	считывает целое число типа <code>short</code>
<code>nextInt()</code>	считывает целое число типа <code>int</code>
<code>nextLong()</code>	считывает целое число типа <code>long</code>
<code>nextFloat()</code>	считывает число с плавающей точкой типа <code>float</code>
<code>nextDouble()</code>	считывает число с плавающей точкой типа <code>double</code>

К операторам числовых типов данных относятся стандартные арифметические операторы: сложения (+), вычитания (-), умножения (*), деления (/) и остатка от деления (%).

Операнды – это значения, обрабатываемые операторами.

Литерал – это постоянное значение, которое появляется непосредственно в программе. Например, 34 и 0.305 являются литералами в следующих предложениях:

```
int numberOfYears = 34;  
double weight = 0.305;
```

Целочисленный литерал может быть присвоен целочисленной переменной, только если он может в нее поместиться. Если литерал слишком велик для хранения в переменной, то произойдет ошибка компиляции. Например, предложение `byte b = 128` вызовет ошибку компиляции, поскольку 128 не может быть сохранено в переменной типа `byte` (диапазон значений типа данных `byte` находится в диапазоне от -128 до 127).

Как правило, целочисленный литерал имеет тип `int`, значение которого находится в диапазоне от -2^{31} (-2147483648) до $2^{31} - 1$ (2147483647). Для обозначения целочисленного литерала типа `long` к нему добавляется буква `L` или `l`. Например, для записи целого числа 2147483648 в Java-программе необходимо написать его как `2147483648L` или `2147483648l`, поскольку 2147483648 выходит за диапазон значений типа `int`. Обозначение `L` является предпочтительным, поскольку `l` (строчная буква) можно легко перепутать с 1 (цифра).

По умолчанию целочисленный литерал является десятичным целым числом. Для обозначения двоичного целочисленного литерала в его начало добавляется `0b` или `0B` (ноль B); для обозначения восьмеричного целочисленного литерала в его начало добавляется `0` (ноль); а для обозначения шестнадцатеричного целочисленного литерала в его начало добавляется `0x` или `0X` (ноль X). Например:

```
System.out.println(0B1111); // Отображает 15  
System.out.println(07777); // Отображает 4095  
System.out.println(0xFFFF); // Отображает 65535
```

В Java для улучшения читаемости разрешается использовать подчеркивания в числовом литерале между разрядами числа. Например, допустимы следующие литералы:

```
long ssn = 232_45_4519;  
long creditCardNumber = 2324_4545_4519_3415L;
```

Однако `45_` или `_45` будут недопустимыми литералами. Знак подчеркивания обязательно должен быть помещен между разрядами числа.

Литералы с плавающей точкой записываются в виде десятичных чисел с точкой, отделяющей целую часть числа от дробной. По умолчанию литерал

с плавающей точкой обрабатывается как значение типа `double`. Например, `5.0` считается значением типа `double`, а не `float`. Можно сделать число типа `float`, добавив в конец числа букву `f` или `F`, и сделать число типа `double`, добавив в конец числа букву `d` или `D`. Например, можно использовать `100.2f` или `100.2F` для числа типа `float`, а `100.2d` или `100.2D` для числа типа `double`.

Типы данных `float` и `double` используются для представления десятичных чисел с точкой. Их называют числами с плавающей точкой, потому что эти числа в компьютере хранятся в экспоненциальном представлении. Когда, например, число `50.534`, сохраняется в программе, допустим, `5.0534E + 1`, то его точка перемещается («плывет») на новую позицию.

Арифметические операторы `+`, `-`, `*`, `/` и `%` можно объединять с оператором присваивания для формирования составных операторов.

В следующей таблице показаны остальные составные операторы присваивания.

Оператор	Имя	Пример	Эквивалент
<code>+=</code>	Сложение с присваиванием	<code>i += 8</code>	<code>i = i + 8</code>
<code>-=</code>	Вычитание с присваиванием	<code>i -= 8</code>	<code>i = i - 8</code>
<code>*=</code>	Умножение с присваиванием	<code>i *= 8</code>	<code>i = i * 8</code>
<code>/=</code>	Деление с присваиванием	<code>i /= 8</code>	<code>i = i / 8</code>
<code>%=</code>	Остаток от деления с присваиванием	<code>i %= 8</code>	<code>i = i % 8</code>

В составных операторах присваивания отсутствуют пробелы. Например, `+ =` должно быть `+=`.

Аналогично оператору присваивания (`=`) операторы (`+=`, `-=`, `*=`, `/=` и `%=`) могут использоваться для формирования предложений с присваиванием, а также выражений. Например, в следующем коде `x += 2` в первой строчке является предложением, а во второй строчке – выражением:

```
x += 2; // Предложение
System.out.println(x += 2); // Выражение
```

Оператор инкремента (`++`) и оператор декремента (`--`) предназначены для увеличения и уменьшения значения переменной на 1. Эти операторы известны как постфиксный инкремент (или постинкремент) и постфиксный декремент (или постдекремент), поскольку операторы `++` и `--` размещаются после переменной, т.е. справа. Эти операторы также можно поместить перед переменной, т.е. слева. Такие операторы известны как префиксный инкремент (или преинкремент) и префиксный декремент (или предекремент).

В следующей таблице описаны их различия, а также приведены примеры их использования.

Оператор	Имя	Описание	Пример (при i = 1)
++var	преинкремент	var инкрементируется на 1 и в предложении используется новое значение var	int j = ++i; // j равно 2, i равно 2
var++	постинкремент	var инкрементируется на 1, но в предложении используется исходное значение var	int j = i++; // j равно 1, i равно 2
--var	предекремент	var декрементируется на 1 и в предложении используется новое значение var	int j = --i; // j равно 0, i равно 0
var--	постдекремент	var декрементируется на 1, но в предложении используется исходное значение var	int j = i--; // j равно 1, i равно 0

Использование операторов инкремента и декремента делает выражения более короткими, но также более сложными и трудными для прочтения.

Числа с плавающей точкой можно преобразовать в целые числа с помощью явного приведения типа.

Приведение типов – это унарная операция, которая преобразует значение одного типа данных в значение другого типа данных. Приведение типа с меньшим диапазоном значений к типу с большим диапазоном значений называется расширением типа. Приведение типа с большим диапазоном значений в тип с меньшим диапазоном значений называется сужением типа. В Java тип расширяется автоматически, но явно сузить его необходимо программисту самостоятельно.

Синтаксис приведения типов для переменной/значения:

(ТипДанных)имяПеременной
(ТипДанных)значение

Например, следующее предложение

```
System.out.println((int)1.7);
```

отображает 1. При приведении значения типа double к значению типа int дробная часть отбрасывается.

Следующее предложение

```
System.out.println((double)1 / 2);
```

отображает 0.5, так как 1 сначала приводится к 1.0, затем 1.0 делится на 2. Однако предложение

```
System.out.println(1 / 2);
```

отображает 0, поскольку 1 и 2 являются целыми числами, и, следовательно, результат целочисленного деления также должен быть целым числом.

Приведение типов не изменяет значение приводимой переменной. Например, в следующем коде значение переменной `d` не изменится после приведения ее значения к значению типа `int`:

```
double d = 4.5;
int i = (int)d; // i будет равно 4, а d останется равным 4.5
```

В Java, предложение с составным присваиванием

```
x1 op= x2
```

реализовано как

```
x1 = (T)(x1 op x2)
```

где `T` – тип данных `x1`, `op` – оператор. Таким образом, следующий код корректен:

```
int sum = 0;
sum += 4.5; // sum равно 4 после этого предложения
sum += 4.5 эквивалентно sum = (int)(sum + 4.5).
```

Чтобы присвоить переменную типа `int` переменной типа `short` или `byte`, необходимо использовать явное приведение типов. Например, в следующих предложениях произойдет ошибка компиляции:

```
int i = 1;
byte b = i; // Ошибка, так как требуется явное приведение типов
```

Однако до тех пор, пока целочисленный литерал находится в пределах допустимого диапазона целевой переменной (к которой происходит приведение), для присваивания целочисленного литерала переменной типа `short` или `byte` явного приведения типов не требуется.

1.5. СТРУКТУРЫ ВЫБОРА

Переменная, содержащая логическое значение, называется логической переменной. Логический тип данных `boolean` используется для объявления логических переменных. Переменная типа `boolean` может иметь одно из двух значений: `true` или `false`. Как и числа, `true` и `false` являются литералами. Они считаются зарезервированными словами и не могут использоваться в программе в качестве идентификаторов.

Предложение `if` – это структура выбора, позволяющая программе описывать альтернативные пути своего выполнения.

В Java имеется несколько типовых структур выбора: одновариантные предложения `if`, двухвариантные предложения `if-else`, вложенные предложения `if`, многовариантные предложения `if-else`, предложения `switch` и условные операторы.

Одновариантное предложение `if` выполняет действие тогда и только тогда, когда условие истинно (`true`).

Синтаксис одновариантного предложения `if`:

```
if (boolean-выражение) {  
    предложение(я);  
}
```

`boolean-выражение` заключается в круглые скобки. Например, код слева является недопустимым, поэтому его следует исправить, как показано справа.

```
if i > 0 { // НЕ ДОПУСТИМО  
    System.out.println("i - положителен");  
}
```

```
if (i > 0) { // ДОПУСТИМО  
    System.out.println("i - положителен");  
}
```

Фигурные скобки блоков могут быть опущены, если они содержат единственное предложение. Например, следующие предложения эквивалентны:

```
if (i > 0) {  
    System.out.println("i - положителен");  
}
```

```
if (i > 0)  
    System.out.println("i - положителен");
```

Отсутствие фигурных скобок делает код короче, но тогда он становится подвержен ошибкам.

Предложение `if-else` определяет путь выполнения программы на основе значения условия `true` или `false`.

Одновариантное предложение `if` выполняет действие, если условие истинное (`true`), а если условие ложное (`false`), то это действие не выполняется. Можно использовать двухвариантное предложение `if-else`. Действия, определяемые двухвариантным предложением `if-else`, отличаются в зависимости от того, является ли условие истинным или ложным.

Синтаксис двухвариантного предложения `if-else`:

```
if (boolean-выражение) {  
    предложение(я) в случае true;  
} else {  
    предложение(я) в случае false;  
}
```

Предложение `if` может находиться внутри другого предложения `if` для получения вложенного предложения `if`. Предложение внутри предложения `if` или `if-else` может быть любым допустимым Java-предложением, включая другое предложение `if` или `if-else`. Внутреннее предложение `if`, как говорят, вложено во внешнее предложение `if`. В свою очередь, внутреннее предложение `if` может содержать другое предложение `if`. На самом деле нет предела глубины вложенности.

Логические операторы `!`, `&&`, `||` и `^` используются для создания составного логического выражения. Выполняться ли предложению или нет, иногда определяется комбинацией нескольких условий. Для объединения этих условий в составное логическое выражение используются логические операторы. Логические операторы, также называемые булевыми операторами, используют логические значения для создания нового логического значения. В следующей таблице перечислены логические операторы.

Оператор	Имя	Описание
<code>!</code>	НЕ	Логическое отрицание
<code>&&</code>	И	Логическая конъюнкция
<code> </code>	ИЛИ	Логическая дизъюнкция
<code>^</code>	исключающее ИЛИ	Логическое исключение

Условный оператор вычисляет выражение на основе условия. Синтаксис условного оператора:

`boolean-выражение ? выражение1 : выражение2`

Если `boolean-выражение` имеет значение `true`, то результатом будет `выражение1`, иначе результатом будет `выражение2`.

Появляющиеся вместе символы `?` и `:` называются условным оператором, также известным как тернарный (трехместный) оператор, потому что он использует три операнда. Условный оператор является единственным тернарным оператором в Java. Он представлен в другом стиле, в отличие от явного `if` в предложении.

1.6. МАТЕМАТИЧЕСКИЕ ФУНКЦИИ, СИМВОЛЫ И СТРОКИ

В Java предоставляется много полезных методов в классе `Math` для выполнения распространенных математических функций.

Для выполнения тригонометрических функций в классе `Math` содержатся методы, перечисленные в следующей таблице:

Метод	Описание
<code>sin(radians)</code>	Возвращает тригонометрический синус угла в радианах
<code>cos(radians)</code>	Возвращает тригонометрический косинус угла в радианах
<code>tan(radians)</code>	Возвращает тригонометрический тангенс угла в радианах
<code>toRadians(degree)</code>	Преобразует геометрический угол из градусов в радианы
<code>toDegrees(radians)</code>	Преобразует геометрический угол из радианов в градусы
<code>asin(a)</code>	Возвращает угол в радианах для арксинуса
<code>acos(a)</code>	Возвращает угол в радианах для арккосинуса
<code>atan(a)</code>	Возвращает угол в радианах для арктангенса

В классе Math существует пять экспоненциальных методов, которые перечислены в следующей таблице.

Метод	Описание
exp(x)	Возвращает число e, возведенное в степень x (e^x)
log(x)	Возвращает натуральный логарифм x ($\ln(x) = \log_e(x)$)
log10(x)	Возвращает десятичный логарифм x ($\log_{10}(x)$)
pow(a, b)	Возвращает a, возведенное в степень b (a^b)
sqrt(x)	Возвращает квадратный корень x, для $x \geq 0$

Класс Math содержит четыре округляющих метода, перечисленных в следующей таблице:

Метод	Описание
ceil(x)	x округляется в большую сторону до ближайшего целого числа. Это целое число возвращается как значение типа double
floor(x)	x округляется в меньшую сторону до ближайшего целого числа. Это целое число возвращается как значение типа double
rint(x)	x округляется до ближайшего целого числа. Если x равноудален от двух целых чисел, то четное из них возвращается как значение типа double
round(x)	Если x типа float, то возвращается (int)Math.floor(x + 0.5). Если x типа double, то возвращается (long)Math.floor(x + 0.5)

Методы min() и max() возвращают минимальное и максимальное значение из двух чисел (типа int, long, float или double).

Метод random() генерирует случайное значение типа double, большее или равное 0.0 и меньшее 1.0 ($0 \leq \text{Math.random()} < 1.0$). Этот метод можно использовать в написании простого выражения для генерации случайных чисел в любом диапазоне.

Символьный тип данных представляет единичные символы.

Строковый литерал должен заключаться в двойные кавычки (" "). Символьный литерал – это единичный символ, который заключается в одинарные кавычки (' '), поэтому "А" – это строка, а 'А' – символ.

В Java поддерживается Юникод – стандарт кодирования для поддержки обмена, обработки и отображения письменных текстов на различных языках мира. 16-битный Юникод занимает два байта, представленные четырьмя шест-

надцатеричными цифрами, которым предшествует \u, начиная с \u0000 и заканчивая \uFFFF.

В Java используется специальная нотация для представления специальных символов, как показано в следующей таблице. Эта специальная нотация, называемая *escape-последовательностью*, состоит из обратного слеша (\), за которым следует символ или комбинация цифр. Например, \t является *escape-последовательностью* символа табуляции (Tab), а *escape-последовательность*, например, \u00b1, используется для представления Юникода. Символы *escape-последовательности* интерпретируются как единое целое, а не отдельно друг от друга, поэтому *escape-последовательность* считается *единичным символом*.

Escape-последовательность	Имя	Юникод	Десятичное значение
\b	Возврат (Backspace)	\u0008	8
\t	Табуляция (Tab)	\u0009	9
\n	Переход на новую строку	\u000A	10
\f	Переход на новую страницу	\u000C	12
\r	Возврат каретки	\u000D	13
\\	Обратный слеш	\u005C	92
\"	Двойные кавычки	\u0022	34

Обратный слеш \ называется *escape-символом*. Это специальный символ, поэтому для отображения обратного слеша необходимо использовать *escape-последовательность* \\.

Символ можно привести к любому числовому типу и наоборот. Когда целое число приводится к символу, используются только последние 16 битов его данных; остальная часть игнорируется. Например,

```
// Обратите внимание, что шестнадцатеричное целое число записывается  
// с помощью префикса 0x  
char ch = (char)0xAB0041; // Последние 16 битов шестнадцатеричного  
                        // кода присваиваются переменной ch  
System.out.println(ch); // ch равно символу A
```

При приведении значения с плавающей точкой к типу char значение с плавающей точкой сначала приводится к типу int, который затем приводится к типу char.

```
char ch = (char)65.25; // Десятичное значение 65 присваивается  
                    // переменной ch  
System.out.println(ch); // ch равно символу A
```

При приведении типа `char` к числовому типу Юникод символа приводится к указанному числовому типу.

```
int i = (int)'A'; // Юникод символа A присваивается переменной i
System.out.println(i); // i равно 65
```

Два символа можно сравнивать с помощью операторов отношений точно так же, как числа.

Строка – это последовательность символов. Тип `char` представляет только единичные символы. Для представления строки символов используется тип данных, который называется `String`. Тип `String` не является примитивным типом, он известен как ссылочный тип данных. Любой Java-класс может использоваться переменной в качестве ссылочного типа. Переменная, объявленная со ссылочным типом, называется ссылочной переменной, т.е. переменной, которая ссылается на объект.

В следующей таблице перечислены методы `String` для получения длины строки, доступа к символам строки, конкатенации строк, преобразования строки в заглавные или строчные буквы, а также для усечения строки.

Метод	Описание
<code>length()</code>	Возвращает количество символов в строке
<code>charAt(index)</code>	Возвращает символ в указанном индексе (<code>index</code>) строки
<code>concat(s1)</code>	Возвращает новую строку, конкатенированную со строкой <code>s1</code>
<code>toUpperCase()</code>	Возвращает новую строку со всеми заглавными буквами
<code>toLowerCase()</code>	Возвращает новую строку со всеми строчными буквами
<code>trim()</code>	Возвращает новую строку с усеченными с обеих сторон пробелами

В Java строки являются объектами. Методы, перечисленные в предыдущей таблице, могут быть вызваны только конкретным строковым экземпляром (объектом). По этой причине эти методы называются методами экземпляров. Неэкземплярный метод называется статическим. Статический метод можно вызвать без использования объекта. Все методы, определенные в классе `Math`, являются статическими. Они не привязаны к конкретному экземпляру (объекту).

Синтаксис для вызова метода экземпляра:

ссылочнаяПеременная.имяМетода(аргументы)

Метод может иметь несколько аргументов или ни одного.

При использовании строки часто известно ее литеральное значение. Для удобства в Java разрешается использовать строковый литерал для прямого обращения к строкам без создания новых переменных. Таким образом, вызов

"Welcome to Java".length() является допустимым и возвращает 15. "" обозначает пустую строку и "".length() равно 0.

Попытка получить доступ к символам вне границ строки s является пространенной ошибкой программирования. Чтобы избежать этого, не используйте индексы за пределами s.length() – 1. Например, s.charAt (s.length()) вызовет исключение StringIndexOutOfBoundsException.

Для конкатенации двух строк можно использовать метод concat(). Поскольку в программировании широко используется конкатенация строк, в Java предоставляется удобный способ ее выполнения. Для конкатенации двух строк можно использовать оператор плюс (+).

Класс String содержит методы сравнения двух строк, перечисленные в следующей таблице.

Метод	Описание
equals(s1)	Возвращает true, если строка равна строке s1
equalsIgnoreCase(s1)	Возвращает true, если строка равна строке s1; не учитывает регистр
compareTo(s1)	Возвращает целое число, большее 0, равное 0 или меньше 0, для указания того, является ли строка больше, равной или меньше s1
compareToIgnoreCase(s1)	Аналогичен методу compareTo(), за исключением того, что сравнение не учитывает регистр
startsWith(prefix)	Возвращает true, если строка начинается с указанного префикса (prefix)
endsWith(suffix)	Возвращает true, если строка заканчивается с указанным суффиксом (suffix)
contains(s1)	Возвращает true, если s1 является подстрокой

Для сравнения можно попытаться использовать оператор ==. Однако оператор == проверяет только, ссылаются ли string1 и string2 на один и тот же объект; он не сообщает, имеют ли они одинаковое содержимое, поэтому недопустимо использовать оператор ==, чтобы выяснить, имеют ли две строковые переменные одинаковое содержимое. Вместо этого следует использовать метод equals().

При сравнении строк с помощью операторов отношений >, >=, < и <= происходят синтаксические ошибки. Вместо этого необходимо использовать s1.compareTo(s2).

Метод `equals()` возвращает `true`, если две строки равны, и `false`, если они не равны. Метод `compareTo()` возвращает 0, положительное целое число или отрицательное целое число, в зависимости от того, равны, больше или меньше строки относительно друг друга.

Можно извлечь один символ из строки с помощью метода `charAt()`. Также можно извлечь подстроку из строки с помощью подстрочных методов класса `String`, как показано в следующей таблице.

Метод	Описание
<code>substring(beginIndex)</code>	Возвращает подстроку, которая начинается с символа под индексом <code>beginIndex</code> и продолжается до конца строки, как показано на рисунке
<code>substring(beginIndex, endIndex)</code>	Возвращает подстроку, которая начинается с символа под индексом <code>beginIndex</code> и заканчивается символом под индексом <code>endIndex - 1</code> , как показано на рисунке. Обратите внимание, что символ под индексом <code>endIndex</code> не является частью подстроки

Если `beginIndex` равен `endIndex`, то метод `substring(beginIndex, endIndex)` возвращает пустую строку длиной 0. Если `beginIndex > endIndex`, то произойдет ошибка во время выполнения.

В классе `String` предоставляется несколько версий методов `indexOf()` и `lastIndexOf()` для поиска символа или подстроки в строке, как показано в следующей таблице.

Метод	Описание
<code>indexOf(ch)</code>	Возвращает индекс первого вхождения символа <code>ch</code> в строке. Возвращает -1, если не входит
<code>indexOf(ch, fromIndex)</code>	Возвращает индекс первого вхождения символа <code>ch</code> в строке после <code>fromIndex</code> . Возвращает -1, если не входит
<code>indexOf(s)</code>	Возвращает индекс первого вхождения заданной строки <code>s</code> в строке. Возвращает -1, если не входит
<code>indexOf(s, fromIndex)</code>	Возвращает индекс первого вхождения заданной строки <code>s</code> в строке после <code>fromIndex</code> . Возвращает -1, если не входит
<code>lastIndexOf(ch)</code>	Возвращает индекс последнего вхождения символа <code>ch</code> в строке. Возвращает -1, если не входит
<code>lastIndexOf(ch, fromIndex)</code>	Возвращает индекс последнего вхождения символа <code>ch</code> в строке перед <code>fromIndex</code> . Возвращает -1, если не входит
<code>lastIndexOf(s)</code>	Возвращает индекс последнего вхождения заданной строки <code>s</code> в строке. Возвращает -1, если не входит
<code>lastIndexOf(s, fromIndex)</code>	Возвращает индекс последнего вхождения заданной строки <code>s</code> в строке перед <code>fromIndex</code> . Возвращает -1, если не входит

Числовую строку можно преобразовать в число. Чтобы преобразовать строку в значение типа `int`, используется метод `Integer.parseInt()`. Чтобы преобразовать строку в значение типа `double`, используется метод `Double.parseDouble()`. Если строка не является числовой, то такое преобразование приведет к ошибке во время выполнения. Классы `Integer` и `Double` включены в пакет `java.lang` и, таким образом, импортируются автоматически.

1.7. ЦИКЛЫ

Управляющая структура, которая повторяет несколько шагов в программе, называется циклом.

Вид цикла	Когда используется	Предложение
Счетный цикл	Перед выполнением цикла можно точно определить, сколько повторений цикла будет необходимо для решения задачи	<code>while</code> <code>for</code>
Цикл с сигнальной меткой	Ввод списка данных любой длины, заканчивающийся специальным значением	<code>while</code> <code>for</code>
Цикл с проверкой правильности ввода	Повторяющийся интерактивный ввод данных до тех пор, пока не будет введено значение вне допустимого диапазона	<code>do-while</code>
Цикл с общим условием	Повторная обработка данных до тех пор, пока выполняется заданное условие	<code>while</code> <code>for</code>

Счетный цикл – это цикл, требуемое количество итераций которого может быть определено до начала его выполнения.

Цикл, показанный ниже в виде псевдокода, называется циклом, управляемым счетчиком или счетным циклом, поскольку его повторением руководит переменная управления циклом, значение которой представляет счет.

Счетный цикл `while` имеет следующую форму:

```

Присвоить переменной управления циклом начальное значение 0.
while значение переменной управления циклом < конечное значение
...
    Увеличить переменную управления циклом на 1.

```

Переменная управления циклом для правильного выполнения цикла должна быть инициализирована, проверена и обновлена.

Циклы в дополнение к телу цикла имеют три управляющих циклом компонента:

- инициализацию переменной управления циклом;
- проверку условия повторения цикла;
- изменение (обновление) переменной управления циклом.

В заголовке предложения `for` соединены сразу все три управляющих циклом шага – инициализации, проверки и обновления.

Несмотря на то что в Java допускается использовать числа с плавающей точкой в качестве переменных управления циклом, настоятельно не рекомендуется их использовать для проверки равенства в условии цикла. Поскольку значения с плавающей точкой являются приближенными для некоторых значений, их использование может привести к неточным значениям счетчика и, соответственно, ошибочным результатам.

Рассмотрим следующий код для вычисления $1 + 0.9 + 0.8 + \dots + 0.1$:

```
double item, sum;
sum = 0;
// Нет гарантии, что item будет равен 0
for (item = 1; item != 0; item -= 0.1)
    sum += item;
System.out.println(sum);
```

Переменная `item` начинается с 1 и уменьшается на 0.1 при каждом выполнении тела цикла. Когда значение переменной `item` становится равным 0, цикл должен завершиться, однако нет гарантии, что значение переменной `item` будет точно равно 0, потому что арифметические операции с числами с плавающей точкой выполняются с некоторым приближением. Внешне этот цикл кажется нормальным, но на самом деле он является бесконечным.

Во многих задачах программирования невозможно определить точное количество повторений цикла до начала его выполнения. Цикл `while` является примером цикла с общим условием, псевдокод которого показан ниже:

```
Инициализировать переменную управления циклом
while условие выхода из цикла не выполняется
    Продолжать обработку
```

Во многих программах с циклами один или несколько дополнительных элементов данных вводятся каждый раз во время повторения тела цикла. Зачастую неизвестно, сколько элементов данных должен обработать цикл, когда он начинает свое выполнение, поэтому необходимо найти какой-то способ сигнализировать программе прекратить считывание и обработку новых элементов данных.

Один из способов сделать это – дать указание пользователю ввести после последнего элемента данных уникальное значение, называемое *сигнальной меткой* (англ. *sentinel*). Условие повторения цикла проверяет каждый элемент данных и приводит к выходу из цикла при считывании сигнальной метки. Необходимо тщательно выбирать значение сигнальной метки, так как это должно быть значение, которое не встречается в элементах данных.

Выход из цикла с сигнальной меткой происходит в результате считывания конечного маркера, следующего за последним элементом входных данных.

Псевдокод цикла, который обрабатывает данные до тех пор, пока не будет введена сигнальная метка:

```
Получить элемент данных // инициализация
while не встретится сигнальная метка // проверка условия
    Обработать элемент данных
    Получить следующий элемент данных // обновление
```

Этот цикл имеет шаги инициализации, проверки условия и обновления. В результате инициализации цикл получает первый элемент данных для обработки; в результате обновления цикл получает все остальные элементы данных, а затем пытается получить еще один элемент, являющийся сигнальной меткой. Этой попыткой разрешается ввод, но не обработка, сигнальной меткой.

Цикл `do-while` аналогичен циклам `while` и `for`, за исключением того, что он в начале выполняет тело цикла, а в конце проверяет условие повторения цикла.

И предложение `for`, и предложение `while` вычисляют условие повторения цикла перед первым выполнением тела цикла. В большинстве задач эта предварительная проверка является желательной и предотвращает выполнение цикла в случаях, когда элементы данных для обработки отсутствуют или когда начальное значение переменной управления циклом выходит за границы допустимого диапазона. Существуют некоторые ситуации, обычно связанные с интерактивным вводом, когда заранее известно, что цикл должен выполняться хотя бы один раз.

Псевдокод цикла проверки правильности ввода можно записать следующим образом:

1. Получить значение.
2. Если значение находится вне допустимого диапазона, то вернуться к шагу 1.

Один цикл может быть вложен в другой цикл. Вложенные циклы состоят из одного внешнего цикла и одного или нескольких внутренних циклов. Каждый раз при проходе внешнего цикла происходит повторный вход во внутренние циклы, таким образом, они запускаются заново.

С помощью ключевых слов `break` и `continue` предоставляются дополнительные средства управления циклом. Два ключевых слова, `break` и `continue` могут использоваться в циклах для обеспечения дополнительных средств управления. С помощью `break` и `continue` в некоторых случаях можно упростить процесс программирования, однако их чрезмерное или неправильное использование может затруднить чтение и отладку программ.

Предложение `continue` всегда находится внутри тела цикла. В циклах `while` и `do-while` условие повторения цикла вычисляется сразу же после предложения `continue`. В цикле `for` сначала выполняется обновление переменной управления циклом, а затем, сразу после `continue`, вычисляется условие повторения цикла.

В некоторых языках программирования есть предложение `goto`. Предложение `goto` без разбора передает управление любому предложению в программе

и выполняет его. Это делает программу уязвимой с точки зрения ошибок. В Java предложения break и continue отличаются от предложения goto. Они работают только в цикле, при этом break выходит из самого цикла, а continue выходит из текущей итерации цикла.

Всегда можно написать программу, не используя при этом в цикле break или continue. В целом целесообразно использовать break и continue, только если это упрощает кодирование и облегчает чтение программ.

Предположим, вам нужно написать программу, которая находит наименьший делитель, отличный от 1, для целого числа n (пусть $n \geq 2$). С помощью предложения break можно написать простой и интуитивно понятный код следующим образом:

```
int factor = 2;
while (factor <= n) {
    if (n % factor == 0)
        break;
    factor++;
}
System.out.println("Наименьший делитель, отличный от 1, для "
    + n + " равен " + factor);
```

Этот код можно переписать без использования break следующим образом:

```
boolean found = false;
int factor = 2;
while (factor <= n && !found) {
    if (n % factor == 0)
        found = true;
    else
        factor++;
}
System.out.println("Наименьший делитель, отличный от 1, для "
    + n + " равен " + factor);
```

Очевидно, что в данном случае break упрощает эту программу и делает ее легче для чтения. Однако следует использовать break и continue с осторожностью. Слишком много предложений break и continue создадут цикл с несколькими точками выхода и затруднят чтение программы.

Программирование – это творческая деятельность. Существует множество различных способов написания кода. На самом деле наименьший делитель можно найти с помощью довольно простого кода следующим образом:

```
int factor = 2;
while (n % factor != 0)
    factor++;
```

или

```
for (int factor = 2; n % factor != 0; factor++);
```

Этот код находит наименьший делитель для целого числа n .
Разберем пример.

```
int balance = 10;
while (true) {
    if (balance < 9)
        break;
    balance = balance - 9;
}
System.out.println("Остаток равен "
    + balance);
```

```
int balance = 10;
while (true) {
    if (balance < 9)
        continue;
    balance = balance - 9;
}
System.out.println("Остаток равен "
    + balance);
```

В коде слева ключевое слово `break` используется для выхода из текущего цикла. Программа завершится с результатом Остаток равен 1. В коде справа ключевое слово `continue` приводит к пропуску оставшейся части тела цикла для текущей итерации. Цикл `while` не завершится.

Использование чисел с плавающей точкой в условии повторения цикла может привести к числовым ошибкам. Числовые ошибки, связанные с числами с плавающей точкой, неизбежны, поскольку числа с плавающей точкой представлены в компьютерах с некоторым приближением. В этом подразделе рассматривается на примерах, как минимизировать такие ошибки.

В примере суммируется последовательность чисел, которая начинается с 0.01 и заканчивается 1.0. Числа в последовательности увеличиваются на 0.01 следующим образом: $0.01 + 0.02 + 0.03$ и так далее.

```
public class TestSum {
    public static void main(String[] args) {
        // Инициализировать sum
        float sum, i;
        sum = 0;
        // Прибавить 0.01, 0.02, ..., 0.99, 1 к sum
        for (i = 0.01f; i <= 1.0f; i = i + 0.01f)
            sum += i;

        // Отобразить sum
        System.out.println("Сумма равна " + sum);
    }
}
```

Цикл `for` (строчки № 7 и 8) многократно прибавляет переменную управления циклом `i` к `sum`. Эта переменная, начинающаяся с 0.01, увеличивается на 0.01 после каждой итерации. Цикл завершится, когда `i` превысит 1.0.

Шагом инициализации в цикле `for` может быть любое предложение, но, как правило, он реализуется инициализацией переменной управления циклом. Из этого примера видно, что переменная управления циклом может быть типа `float`. Фактически, она может быть любого типа данных.

Сумма должна быть равна 50.50, но фактически отображается 50.499985. Результат неточен, поскольку в компьютерах используется конечное количество битов для представления чисел с плавающей точкой, и в них не могут быть точно представлены некоторые числа с плавающей точкой. Если в программе заменить float на double, то это должно, по идее, привести к незначительному повышению точности, поскольку переменная типа double содержит 64 бита, тогда как переменная типа float – 32 бита.

```
// Инициализировать sum
double sum = 0;
// Прибавить 0.01, 0.02, ..., 0.99, 1 к sum
for (double i = 0.01; i <= 1.0; i = i + 0.01)
    sum += i;
```

Однако, результат на самом деле становится равным 49.50000000000003. Если отобразить i для каждой итерации в цикле, будет видно, что последнее значение i немного больше 1, т.е. не точно равно 1. Это приводит к тому, что последнее значение i не добавляется в сумму. Основная проблема заключается в том, что числа с плавающей точкой представляются приближенно. Чтобы решить эту проблему, используйте целочисленный счетчик для прибавления всех чисел к sum. Новый цикл будет выглядеть следующим образом:

```
double currentValue = 0.01;
for (int count = 0; count < 100; count++) {
    sum += currentValue;
    currentValue += 0.01;
}
```

После этого цикла сумма будет равна 50.50000000000003. Этот цикл прибавляет числа от наименьшего до наибольшего. А что произойдет, если прибавлять числа от наибольшего к наименьшему (т.е. в обратной последовательности – 1.0, 0.99, 0.98, ..., 0.02, 0.01) следующим образом?

```
double currentValue = 1.0;
for (int count = 0; count < 100; count++) {
    sum += currentValue;
    currentValue -= 0.01;
}
```

После этого цикла сумма будет равна 50.49999999999995. Прибавление от наибольшего к наименьшему является менее точным, чем прибавление от наименьшего к наибольшему. Этот феномен присущ арифметике с ограниченной точностью. Сложение очень маленького числа с очень большим может не иметь результата, если сумма требует большей точности, чем переменная может обеспечить. Например, неточным результатом $100000000.0 +$

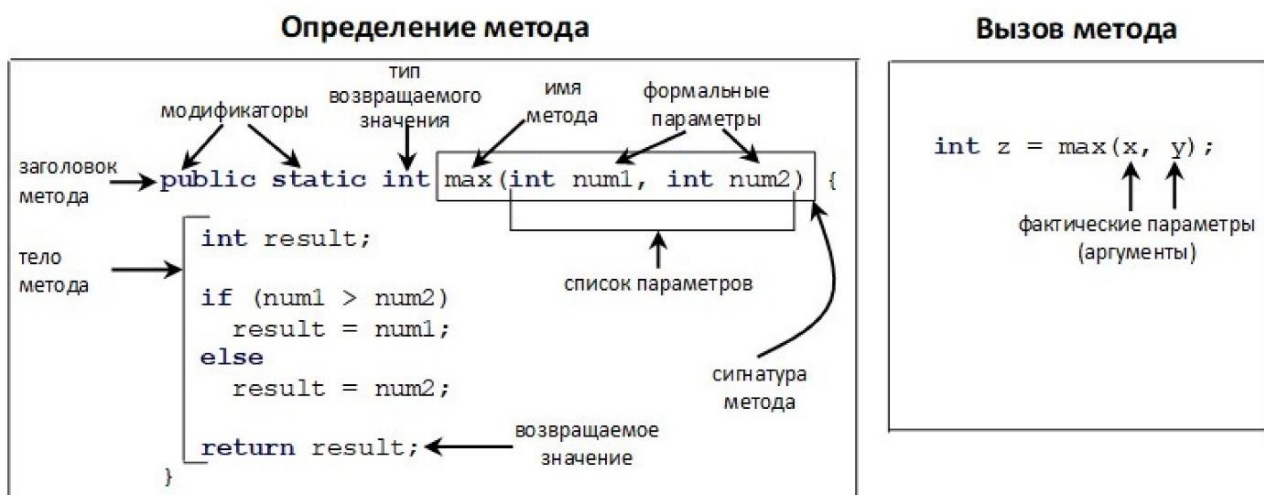
+ 0.000000001 является 100000000.0. Чтобы получать более точные результаты, тщательно выбирайте порядок вычислений. Сложение маленьких чисел до больших является одним из способов минимизации ошибок.

1.8. МЕТОДЫ

Методы используются для описания многократно используемого кода, а также для упрощения кодирования. Определение метода состоит из имени метода, параметров, типа возвращаемого значения и тела метода. Синтаксис определения метода:

```
/**
 * Что делает метод
 */
модификаторы ТипВозвращаемогоЗначения имяМетода(список параметров) {
    // Тело метода
}
```

Здесь в качестве необязательной, но рекомендуемой части метода, представлен заголовочный комментарий в стиле javadoc. Рассмотрим метод, определяемый для нахождения наибольшего значения из двух целых чисел. Этот метод с именем `max` имеет два параметра типа `int`, `num1` и `num2`, наибольшее значение из которых возвращается методом. На следующем рисунке показаны компоненты этого метода.



В заголовке метода определяются модификаторы, тип возвращаемого значения, имя метода и его параметры.

Метод может возвращать значение. ТипВозвращаемогоЗначения является типом данных значения, которое возвращает метод. Некоторые методы выполняют требуемые операции без возврата значения. В этом случае ТипВозвращаемогоЗначения является ключевым словом `void`. Например, в методе `main()`, а также в методах `System.exit()` и `System.out.println()` ТипВозвращаемогоЗначения

является `void`. Если метод возвращает значение, то он называется возвращающим значение методом; иначе он называется методом типа `void`.

Переменные, определяемые в заголовке метода, называются формальными параметрами, или просто параметрами. При вызове метода параметру передается значение. Это значение называется фактическим параметром, или аргументом. Список параметров содержит типы данных, последовательность и количество параметров метода. Имя метода и список параметров вместе составляют сигнатуру метода. Параметры являются необязательными компонентами, т.е. метод может не содержать ни одного параметра. Например, метод `Math.random()` не имеет ни одного параметра.

Тело метода содержит совокупность предложений, реализующих метод. В теле метода `max()` используется предложение `if` для определения того, какое из чисел наибольшее, и возврата значения этого числа. Для того, чтобы возвращающий значение метод возвращал результат, требуется предложение с ключевым словом `return`. Метод завершается после выполнения предложения `return`.

В некоторых языках программирования методы называются процедурами и функциями. В этих языках, как правило, возвращающий значение метод называется функцией, а метод типа `void` – процедурой.

В заголовке метода необходимо отдельно объявлять каждый параметр. Например, заголовок `max(int num1, int num2)` является допустимым, а `max(int num1, num2)` – недопустимым.

При вызове метода выполняется код внутри метода. В определении метода указывается, что он должен делать. Для выполнения метода необходимо его вызвать. Существует два способа вызова метода в зависимости от того, возвращает он значение или нет.

Если метод возвращает значение, то вызов метода обычно рассматривается как значение. Например, предложение

```
int larger = max(3, 4);
```

вызывает метод `max(3, 4)` и присваивает результат метода переменной `larger`. Другим примером вызова, который рассматривается как значение, является предложение

```
System.out.println(max(3, 4));
```

которое отображает значение, возвращаемое методом `max(3, 4)`.

Если метод ничего не возвращает, т.е. типа `void`, то вызов метода должен быть предложением. Например, метод `println()` ничего не возвращает. Следующий вызов является предложением:

```
System.out.println("Welcome to Java!");
```

Возвращающий значение метод также может быть вызван в Java в качестве предложения. В этом случае вызывающий метод просто игнорирует возвращаемое значение. Это делается не так часто, но является допустимым вызовом, если вызывающий метод не заинтересован в возвращаемом значении.

Когда происходит вызов метода, управление программой передается вызываемому методу. Вызываемый метод возвращает управление вызвавшему его методу при выполнении предложения `return` или при достижении закрывающей тело метода скобки.

Предложение `return` необходимо для возвращающего значения метода. Метод, показанный слева, является логически корректным, но имеет ошибку компиляции, поскольку Java-компилятор считает, что этот метод может не вернуть значение.

```
public static int sign(int n) {  
    if (n > 0)  
        return 1;  
    else if (n == 0)  
        return 0;  
    else if (n < 0)  
        return -1;  
}
```

```
public static int sign(int n) {  
    if (n > 0)  
        return 1;  
    else if (n == 0)  
        return 0;  
    else  
        return -1;  
}
```

Чтобы решить эту проблему, необходимо удалить слева `if (n < 0)`, таким образом компилятор увидит предложение `return`, которое должно быть достигнуто независимо от того, как вычисляется предложение `if`.

Методы позволяют делиться кодом и повторно его использовать. Каждый раз при вызове метода система создает запись активации (также называемую фреймом активации), которая сохраняет параметры и переменные метода, и помещает ее в область памяти, называемую стеком вызовов. Стек вызовов также называют стеком выполнения, машинным стеком, или часто просто укорачивают до стека. Когда один метод вызывает другой метод, запись активации вызывающего метода сохраняется неизменной, а для вызванного метода создается новая запись активации. Когда этот метод завершает свою работу и возвращается к вызывающему методу, его запись активации удаляется из стека вызовов.

Стек вызовов сохраняет записи активации по принципу «первым пришел – последним обслужен» (FILO – сокр. от First In Last Out), т.е. запись активации метода, который вызывается первым, удаляется из стека последней.

Следующая программа `TestMax` используется для проверки метода `max()`.

```
public class TestMax {  
    /**  
     * Метод main  
     */  
    public static void main(String[] args) {  
        int i, j, k;  
        i = 5;  
        j = 2;  
        k = max(i, j); // вызов метода max  
        System.out.println("Наибольшее из " + i +  
            " и " + j + " равно " + k);  
    }  
}
```

```

/**
 * Возвращает наибольшее из двух чисел
 */
public static int max(int num1, int num2) {
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result; // возврат result
}

```

Таблица трассировки программы TestMax:

№ строчки	i	j	k	num1	num2	result
7	5					
8		2				
17				5	2	
18						?
21						5
9			5			

Программа TestMax содержит метод main() и метод max(). Метод main() аналогичен любому другому методу, за исключением того, что он вызывается JVM для запуска программы.

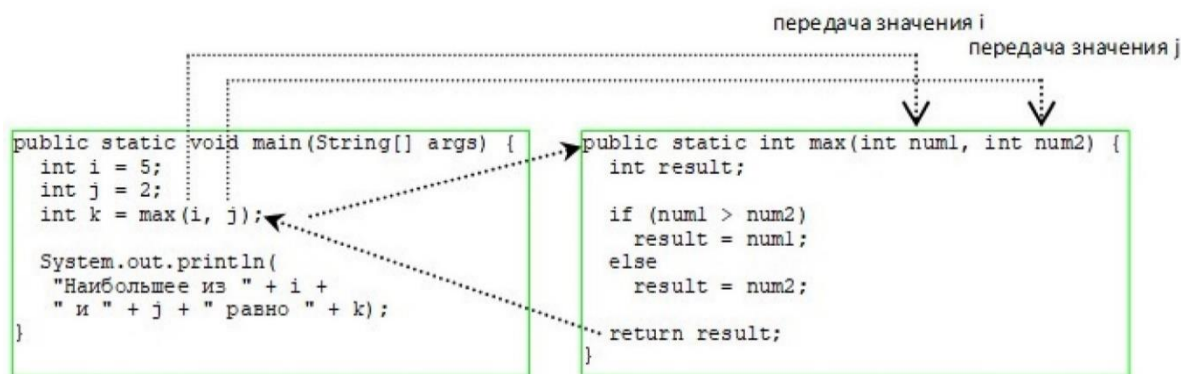
Заголовок метода main() всегда один и тот же. Как и в программе TestMax, он включает модификаторы public и static, тип возвращаемого значения void, имя метода main и параметр типа String[]. String[] указывает на то, что параметр представляет собой массив строк.

Предложения в методе main() могут вызывать другие методы, которые определены в классе, содержащем метод main(), или в других классах. В программе TestMax метод main() вызывает метод max(i, j), который определяется в том же классе, что и метод main().

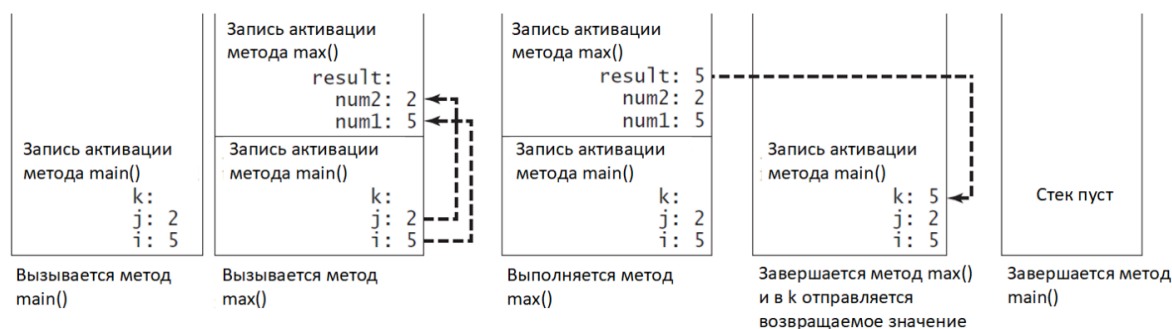
При вызове метода max() (строчка № 9) значение 5 переменной i передается параметру num1, а значение 2 переменной j передается параметру num2 метода max(). Поток управления переходит к методу max() и выполняется метод max(). При выполнении предложения return метода max() он возвращает управление вызвавшему его методу main(). Этот процесс показан на рисунке, показанном ниже.

Предположим, что метод m1() вызывает метод m2(), а метод m2() вызывает метод m3(). Система сначала помещает в стек запись активации метода m1(), затем – m2() и, наконец, – m3(). После завершения метода m3() его запись

активации удаляется из стека. После завершения метода m2() его запись активации удаляется из стека. После завершения метода m1() его запись активации удаляется из стека.



Рассмотрение стеков вызовов помогает понять, каким образом методы вызываются. Переменными, определенными в методе main() программы TestMax, являются i, j и k. Переменными, определенными в методе max(), являются num1, num2 и result. Переменные num1 и num2 определены в сигнатуре метода и являются параметрами метода max(). Их значения передаются посредством вызова метода. На следующем рисунке показаны записи активации вызываемых методов в стеке.



Метод типа void не возвращает значение. Методу типа void не требуется предложение return, но его можно использовать для завершения метода типа void и возврата к вызывающему его методу. Синтаксис в таком случае следующий:

```
return;
```

Это делается не часто в методах типа void, но иногда это полезно для обхода стандартного потока управления.

Значения аргументов передаются параметрам при вызове метода. При вызове метода необходимо указать аргументы, которые должны быть заданы в том же порядке, что и соответствующие им параметры в сигнатуре метода. Это называется сопоставлением списка параметров. Аргументы должны соответствовать параметрам в части последовательности, количества и совместимости типа, как определено в сигнатуре метода. Совместимость типа означает, что можно передать аргумент параметру без явного приведения типа, например, передать аргумент типа int параметру типа double.

При вызове метода с аргументом значение аргумента передается параметру. Это называется передачей по значению. Если аргумент является переменной, а не литеральным значением, то параметру передается значение переменной. Независимо от изменений значения параметра внутри метода, значение переменной не меняется.

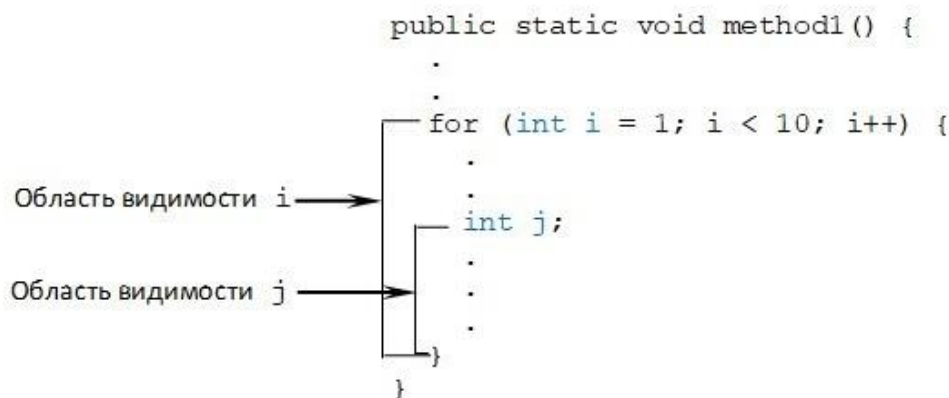
Параметр является переменной внутри метода с собственной областью памяти. Переменной выделяется память при вызове метода и перераспределяется при его завершении.

Для простоты Java-разработчики часто говорят «*x* передается *y*», что фактически означает «значение аргумента *x* передается параметру *y*».

Перегрузка методов позволяет определять несколько методов с одинаковыми именами, если их списки параметров различаются. Перегрузка методов может сделать программы более понятными и удобочитаемыми. Методы, которые выполняют одну и ту же функцию с различными типами параметров, должны иметь одно и то же имя. Перегруженные методы должны различаться, по крайней мере, списком параметров. Невозможно перегрузить методы, имеющие только различные типы возвращаемых значений. Невозможно перегрузить методы, имеющие различные модификаторы. Иногда существует два или более возможных определений для вызова метода, но компилятор не может определить из них наиболее подходящий. Это называется неоднозначным вызовом. Неоднозначный вызов приводит к ошибке компиляции.

Область видимости переменной – это та часть программы, в которой можно ссылаться на эту переменную. Переменная, объявленная внутри метода, называется локальной переменной. Область видимости локальной переменной начинается с ее объявления и продолжается до конца блока, содержащего эту переменную. Перед использованием локальной переменной необходимо ее объявить и присвоить ей значение.

Параметр фактически является локальной переменной. Область видимости параметра метода охватывает весь метод. Переменная, объявленная в заголовке цикла `for`, имеет область видимости внутри всего цикла. Однако переменная, объявленная внутри тела цикла `for`, имеет область видимости, ограниченную в теле цикла от ее объявления до конца блока, который содержит эту переменную, как показано на следующем рисунке.



Внутри метода можно объявить в различных блоках локальную переменную с одним и тем же именем, но нельзя объявить локальную переменную дважды в одном и том же блоке или во вложенных друг в друга блоках, как показано на следующем рисунке.



Распространенной ошибкой является объявление переменной внутри цикла `for`, а затем попытка использовать ее вне цикла. Как показано в следующем коде, переменная `i` объявляется внутри цикла `for`, а доступ к ней осуществляется извне цикла `for`, что приводит к синтаксической ошибке.

```
for (int i = 0; i < 10; i++) {
}
System.out.println(i); // Приводит к синтаксической ошибке
```

1.9. ОДНОМЕРНЫЕ МАССИВЫ

При создании массива фиксируется его размер. Для доступа к элементу массива используется индексированная переменная. Массив используется для хранения коллекции данных, но чаще рассматривается как набор переменных одного типа. Вместо объявления отдельных переменных, например, `number0`, `number1`, ..., `number99`, объявляется одна переменная массива, например, `numbers`, и используются индексированные переменные, например, `numbers[0]`, `numbers[1]`, ..., `numbers[99]` для доступа к отдельным элементам массива.

Чтобы использовать в программе массив, необходимо объявить переменную для хранения ссылки на массив и указать тип элемента массива.

Синтаксис объявления переменной массива:

ТипЭлемента[] ссылкаНаМассив;

или

ТипЭлемента ссылкаНаМассив[]; // Допускается, но менее предпочтителен.

В отличие от объявления переменных примитивного типа данных, при объявлении переменной массива выделяется память не для всего массива,

а только для ссылки на массив. Если переменная не содержит ссылку на массив, то значение этой переменной равно `null`. До тех пор, пока массив не создан, нельзя присвоить значения его элементам. После объявления переменной массива можно создать массив с помощью оператора `new` и присвоить переменной ссылку на него с помощью следующего синтаксиса:

```
ссылкаНаМассив = new ТипЭлемента[размерМассива];
```

Это предложение, во-первых, создает массив, используя `new` ТипЭлемента [размерМассива], и, во-вторых, присваивает переменной `ссылкаНаМассив` ссылку на только что созданный массив.

Объявление переменной массива, создание массива и присваивание переменной ссылки на массив можно объединить в одном предложении следующим образом:

```
ТипЭлемента[] ссылкаНаМассив = new ТипЭлемента[размерМассива];
```

или

```
ТипЭлемента ссылкаНаМассив[] = new ТипЭлемента[размерМассива];
```

Для присваивания значений элементам массива используется следующий синтаксис:

```
ссылкаНаМассив[индекс] = значение;
```

Переменная массива, которая, казалось бы, содержит сам массив, фактически содержит ссылку на этот массив. Строго говоря, переменная массива и массив – это разные понятия, но большую часть времени это различие можно игнорировать.

При выделении памяти для массива необходимо задать размер массива, указав количество элементов, которые могут быть в нем сохранены. После создания массива его размер не может быть изменен. Размер можно получить с помощью `ссылкаНаМассив.length`.

При создании массива его элементам по умолчанию присваивается значение `0` для примитивных числовых типов данных, `\u0000` – для символьного типа и `false` – для типа `boolean`.

Доступ к элементу массива осуществляется через индекс. Индексы массива начинаются с `0`, т.е. они находятся в диапазоне от `0` до `ссылкаНаМассив.length - 1`. На предыдущем рисунке массив `myList` содержит 10 значений типа `double`, а индексы массива находятся в диапазоне от `0` до `9` включительно.

Каждый элемент массива представляется с помощью следующего синтаксиса, называемого индексированной переменной:

```
ссылкаНаМассив[индекс];
```

Язык Java имеет краткую нотацию, называемую инициализатором массива, которая сочетает объявление, создание и инициализацию массива в одном предложении с помощью следующего синтаксиса:

```
ТипЭлемента[] ссылкаНаМассив = {значение0, значение1, ..., значениек};
```

В синтаксисе инициализатора массива не используется оператор `new`. С помощью инициализатора массива можно объявить, создать и инициализировать массив только в одном-единственном предложении. Разделение этих операций может привести к синтаксической ошибке. Таким образом, следующее предложение является недопустимым:

```
double[] myList;  
myList = {1.9, 2.9, 3.4, 3.5}; // Не допустимо
```

При обработке элементов массива часто используется цикл `for` по следующим причинам:

- Все элементы массива одного типа данных, поэтому они могут быть последовательно многократно обработаны единым способом с помощью цикла.
- Так как известен размер массива, логично использовать счетный цикл, например, цикл `for`.

В Java поддерживается удобный цикл `for`, называемый циклом `foreach`, который позволяет последовательно проходить массив без использования индексированной переменной. Например, следующий код отображает все элементы массива `myList`:

```
for (double e: myList) {  
    System.out.println(e);  
}
```

Этот код можно прочесть как «для каждого элемента `e` в `myList` выполните следующие действия». Переменная `e` должна быть объявлена с тем же типом данных, что элементы `myList`.

Как правило, синтаксис цикла `foreach` следующий:

```
for (ТипЭлемента элемент: ссылкаНаМассив) {  
    // Обработать элемент  
}
```

Если требуется пройти по массиву в другой последовательности или изменить элементы массива, то необходимо использовать индексированную переменную.

Попытка доступа к массиву вне его границ – это распространенная ошибка программирования, которая выбрасывает исключение во время выполнения `ArrayIndexOutOfBoundsException`. Чтобы избежать этой ошибки, не следует обращаться к индексам после `arrayRefVar.length – 1` или можно использовать цикл `foreach`, если это возможно.

Программисты часто ошибочно ссылаются на первый элемент в массиве с помощью индекса 1, а не 0. Это называется ошибкой завышения на единицу. Другой распространенной ошибкой в цикле является использование `<=`, где должно использоваться `<`. Например, следующий цикл является некорректным:

```
for (int i = 0; i <= list.length; i++)  
    System.out.print(list[i] + " ");
```

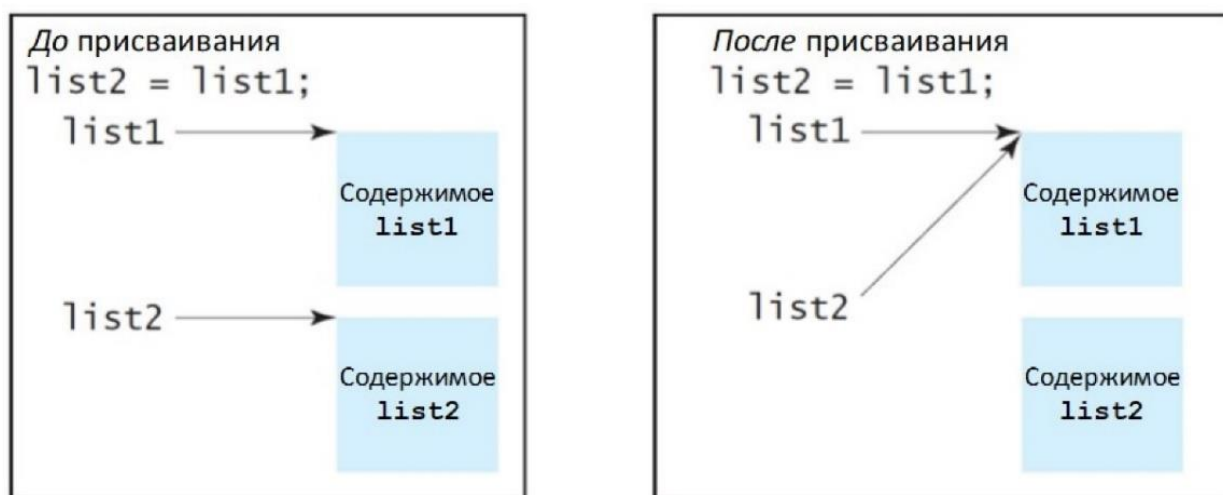
Здесь оператор `<=` следует заменить на `<`. В этом случае можно избежать ошибки завышения на единицу с помощью цикла `foreach`.

Чтобы скопировать содержимое одного массива в другой, необходимо копировать отдельные элементы.

Часто в программе необходимо продублировать массив целиком или его часть. В таких случаях можно попытаться использовать оператор присваивания (`=`) следующим образом:

```
list2 = list1;
```

Однако это предложение копирует в `list2` не содержимое массива, на который ссылается `list1`, а ссылку на массив. После этого предложения `list1` и `list2` ссылаются на один и тот же массив, как показано на следующем рисунке. Массив, на который ранее ссылался массив `list2`, больше не используется и он становится мусором, который автоматически собирается JVM. Этот процесс называется сборкой мусора.



В Java для копирования переменных примитивного типа данных можно использовать операторы присваивания, но они не работают в случае копирования массивов. Присваивание одной переменной массива другой переменной массива фактически копирует одну ссылку в другую и делает обе переменные указывающими на одну и ту же ячейку памяти.

Существует три способа копирования массива:

1. Цикл для копирования отдельных элементов друг за другом.
2. Статический метод `arraycopy()` из класса `System`.
3. Метод `clone()` для копирования массивов.

При передаче массива методу передается ссылка на массив. В Java для передачи аргументов методу используется передача по значению. Существуют важные различия между передачей значений переменных примитивных типов данных и передачей массивов.

- Для аргумента примитивного типа передается его значение.
- Для массива-аргумента значение аргумента является ссылкой на массив; это ссылочное значение и передается методу. Это называется передачей по ссылке, т.е. и передаваемый массив, и массив внутри метода, – один и тот же.

Таким образом, при изменении массива внутри метода, это изменение будет видно и вне метода.

Если массив создается с помощью следующего синтаксиса:

```
new типЭлемента[] {значение0, значение 1, ..., значениек};
```

такой массив называется анонимным, т.к. здесь нет явной ссылки на массив.

В Java массивы являются объектами. JVM хранит объекты в области памяти, называемой кучей, которая используется для динамического распределения памяти.

При возврате массива из метода, возвращается ссылка на массив.

Класс `java.util.Arrays` содержит полезные методы для распространенных операций с массивами, таких как сортировка и поиск.

1.10. МНОГОМЕРНЫЕ МАССИВЫ

Доступ к элементу в двумерном массиве осуществляется через индекс строки и индекс столбца.

Синтаксис объявления двумерного массива:

```
ТипЭлемента[][] ссылкаНаМассив;
```

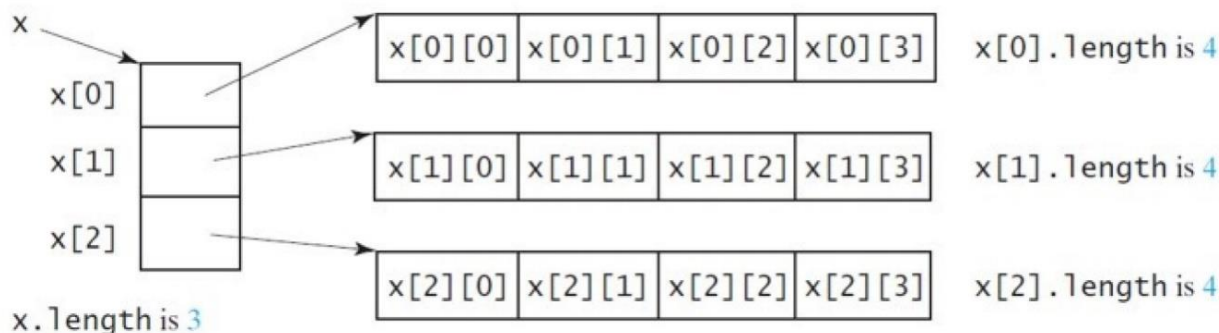
или

```
ТипЭлемента ссылкаНаМассив[][]; // Допускается, но менее предпочтителен
```

Распространенной ошибкой является использование синтаксиса `matrix[2, 1]` для доступа к элементу в строке 2 и столбце 1. В Java каждый индекс должен быть заключен в квадратные скобки.

Двумерный массив фактически является массивом, в котором каждый элемент является одномерным массивом. Длина массива `x` — это количество элементов в массиве, которое можно получить с помощью `x.length`. `x[0]`, `x[1]`, ..., `x[x.length - 1]` — это массивы, поэтому их длины могут быть получены с помощью `x[0].length`, `x[1].length`, ..., `x[x.length - 1].length`.

Пусть `x = new int[3][4]`, `x[0]`, `x[1]` и `x[2]` — одномерные массивы, каждый из которых содержит четыре элемента, как показано на следующем рисунке. `x.length` равно 3, а `x[0].length`, `x[1].length` и `x[2].length` равны 4.



Для обработки двумерных массивов часто используют вложенные циклы `for`. При передаче двумерного массива методу передается ссылка на массив.

Двумерный массив – это массив одномерных массивов, а трехмерный массив – это массив двумерных массивов. Иногда необходимо представить n -мерные структуры данных. В Java можно создавать n -мерные массивы, где n – положительное целое число.

1.11. РЕКУРСИЯ

Рекурсивный метод – это метод, который непосредственно или косвенно сам себя вызывает.

Рекурсивный вызов может привести к еще большему количеству рекурсивных вызовов, поскольку этот подход продолжает разбивать текущую подзадачу на новые подзадачи. Для завершения рекурсивного метода подзадача должна быть в конечном итоге сведена к простому случаю, внутри которого метод возвращает результат вызвавшему его методу. Затем этот метод выполняет вычисления и возвращает результат вызвавшему его методу. Этот процесс продолжается до тех пор, пока результат не будет передан изначально вызвавшему методу.

Если рекурсия не уменьшает задачу таким образом, который позволяет ей в итоге прийти к простому случаю или простой случай не указан, то может произойти бесконечная рекурсия. Например, предположим, что ошибочно был написан метод `factorial()` следующим образом:

```
public static long factorial(int n) {  
    return n * factorial(n - 1);  
}
```

Этот метод вызывается бесконечно и приводит к ошибке `StackOverflowError`.

Все рекурсивные методы имеют следующие характеристики:

- Метод реализуется с помощью предложения `if-else`, которое приводит к различным случаям.
- Один или несколько простых случаев используются для завершения рекурсии.
- Каждый рекурсивный вызов уменьшает исходную задачу, приближая ее к простому случаю до тех пор, пока она сама не станет этим случаем.

Как правило, для решения задачи с помощью рекурсии ее разбивают на подзадачи. Каждая подзадача совпадает с исходной задачей, но меньше ее по размеру. Для рекурсивного решения каждой подзадачи можно применить один и тот же подход. Таким образом, решение исходной задачи размера n сводится к решению $n + 1$ задач размера 1.

Рекурсия является альтернативной формой управления программой. Она, по сути, повторяет предложения программы, не используя циклы.

При использовании цикла определяется его тело. Повторение тела цикла контролируется структурой управления циклом. В рекурсии метод сам себя

многократно вызывает. Для управления рекурсивным вызовом метода должна использоваться структура выбора.

Рекурсия несет существенные издержки. Каждый раз, когда программа вызывает метод, операционная система должна выделять память для всех локальных переменных и параметров этого метода, что может потреблять значительную часть памяти и требует дополнительного времени для ее управления.

Любая задача, которая может быть решена рекурсивно, также может быть решена нерекурсивно с помощью итераций. Рекурсия имеет несколько негативных аспектов: она требует слишком много времени и слишком много памяти. В некоторых случаях использование рекурсии позволяет написать естественное и простое решение рекурсивной по своей сути задачи, которое иначе было бы трудно получить. Примером является задача о ханойских башнях, которую довольно сложно решить без рекурсии.

Решение о том, использовать ли рекурсию или итерацию, должно основываться на характере и понимании задачи. Как правило, используется тот подход, который поможет наилучшим способом разработать интуитивное решение, естественно отражающее задачу. Зачастую итеративное решение более эффективно, чем рекурсивный подход.

Рекурсивным программам может не хватить памяти, это приводит к ошибке `StackOverflowError`.

Рекурсивный метод А считается методом с хвостовой рекурсией, если нет ожидающих операций, которые должны быть выполнены после возврата из рекурсивного вызова, как показано слева в следующей таблице. Однако метод В в таблице справа не является методом с хвостовой рекурсией, так как после возврата из рекурсивного вызова метода ожидаются операции.

Рекурсивный метод А Вызвать метод А рекурсивно	Рекурсивный метод В Вызвать метод В рекурсивно
<i>Хвостовая рекурсия</i>	<i>Не хвостовая рекурсия</i>

Хвостовая рекурсия желательна, поскольку метод заканчивается, когда заканчивается последний рекурсивный вызов, и нет необходимости хранить промежуточные вызовы в стеке. Компиляторы могут оптимизировать хвостовую рекурсию для уменьшения размера стека.

Нехвостовая рекурсия часто может быть преобразована в хвостовую с помощью дополнительных параметров. Эти параметры используются для того, чтобы содержать результат. Идея заключается в том, чтобы встроить ожидающие операции во вспомогательные параметры таким образом, чтобы у рекурсивного вызова не было ожидающей операции. Можно определить новый рекурсивный метод с дополнительными параметрами. Этот метод может перегрузить исходный метод с тем же именем, но другой сигнатурой.

2. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ Java-ПРОГРАММИРОВАНИЕ

2.1. ОБЪЕКТЫ И КЛАССЫ

Класс определяет свойства и поведение объектов. Объектно-ориентированное программирование (ООП) включает в себя программирование с помощью объектов. Объект представляет сущность реального мира, которая может быть четко идентифицирована. Например, слушатель, стол, круг, кнопка и даже кредит могут рассматриваться как объекты. Объект имеет уникальный идентификатор, состояние и поведение.

- Состояние объекта (также известное как его свойства или атрибуты) представлено полями данных с их текущими значениями. У объекта круга, например, есть поле данных `radius`, которое является свойством, характеризующим круг. У объекта прямоугольника, например, есть поля данных `width` и `height`, являющиеся свойствами, характеризующими прямоугольник.

- Поведение объекта (также известное как его действия) определяется методами. Для вызова метода объекта необходимо попросить объект выполнить действие. Например, можно определить методы `getArea()` и `getPerimeter()` для объектов круга. У объекта круга можно вызвать метод `getArea()` для возврата площади, и `getPerimeter()` для возврата периметра. Можно также определить метод `setRadius(radius)`. У объекта круга можно вызвать этот метод для изменения радиуса.

Объекты одного типа определяются с помощью общего класса. Класс — это шаблон, образец или проект, определяющий поля данных и методы объекта. Объект является экземпляром класса. Можно создать несколько экземпляров одного класса. Создание экземпляра называется инстанцированием. Термины «объект» и «экземпляр класса» часто являются взаимозаменяемыми. Отношения между классом и объектами аналогичны отношениям между рецептом яблочного пирога и самим яблочным пирогом: по одному рецепту можно приготовить сколько угодно яблочных пирогов. На следующем рисунке показан класс `Circle` и три его объекта.



Java-класс использует переменные для определения полей данных, а методы – для определения действий. Кроме того, класс предоставляет методы специального типа, известные как конструкторы, которые вызываются для создания нового объекта. Конструктор может выполнять любое действие, но предназначен он для выполнения инициализирующих действий, таких как инициализация полей данных объектов. В следующем фрагменте кода показан пример определения класса для объектов круга.

```

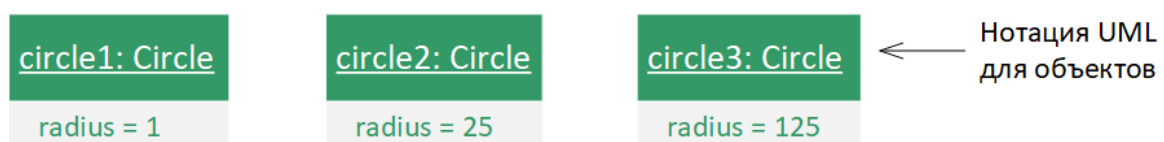
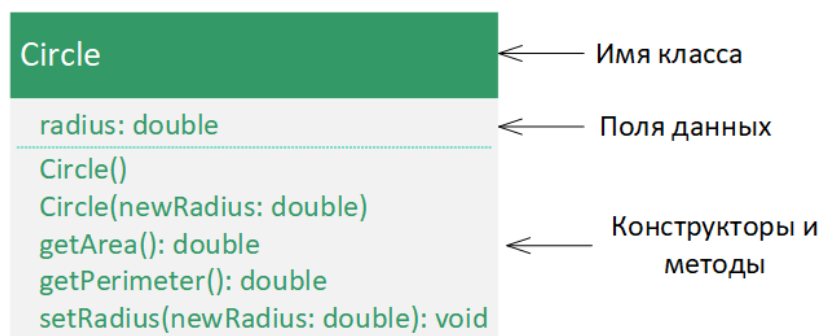
1 public class Circle {
2     /** Радиус круга */
3     double radius = 1; // <--- Поле данных
4     /** Создает объект круга *///-----
5     Circle() { //
6     } //
7     /** Создает объект круга с указанным радиусом */// <--- Конструкторы
8     Circle(double newRadius) { //
9         radius = newRadius; //
10    } //-----
11    /** Возвращает площадь этого круга *///-----
12    double getArea() { //
13        return radius * radius * Math.PI; //
14    } //
15    /** Возвращает периметр этого круга *///
16    double getPerimeter() { // <--- Методы
17        return 2 * radius * Math.PI; //
18    } //
19    /** Присваивает новый радиус этому кругу *///
20    void setRadius(double newRadius) { //
21        radius = newRadius; //
22    } //-----
23 }

```

У класса Circle нет метода main() и поэтому он не может быть запущен, но он является определением для объектов круга. Класс, который содержит метод main(), далее будем называть основным классом.

Изображение классов и объектов можно стандартизировать с помощью нотации Unified Modeling Language (UML, унифицированный язык моделирования). Эта нотация, как показано на следующем рисунке, называется UML-диаграммой классов или просто диаграммой классов.

UML-диаграмма
классов



На диаграмме классов поле данных обозначается как

имяПоляДанных: типПоляДанных

Конструктор обозначается как

ИмяКласса(имяПараметра: типПараметра)

Метод обозначается как

имяМетода(имяПараметра: типПараметра): типВозвращаемогоЗначения

С помощью оператора `new` вызывается конструктор для создания объекта.

Конструктор – это особый вид метода. У него есть три особенности:

- Конструктор должен иметь то же имя, что и класс.
- Конструктор не имеет типа возвращаемого значения – даже типа `void`.
- Конструктор вызывается при создании объекта с помощью оператора `new`, таким образом он играет роль инициализатора объектов.

Конструктор имеет точно такое же имя, как и определяющий его класс. Как и обычные методы, конструкторы могут быть перегружены (т.е. несколько конструкторов могут иметь одно и то же имя, но разные сигнатуры), что упрощает создание объектов с разными начальными значениями полей данных.

Распространенной ошибкой является помещение ключевого слова `void` в начало конструктора. Например,

```
public void Circle() {  
}
```

В этом случае `Circle()` является методом, а не конструктором.

Конструкторы используются для создания объектов. Чтобы создать объект класса, вызовите конструктор класса с помощью оператора `new` следующим образом:

```
new ИмяКласса(аргументы);
```

Например, `new Circle(25)` создает объект, используя второй конструктор, определенный в классе `Circle`.

Класс обычно предоставляет конструктор без аргументов или безаргументный конструктор (например, `Circle()`).

Класс можно определить и без конструкторов. В таком случае в классе неявно определяется общедоступный конструктор без аргументов с пустым телом. Этот конструктор, называемый заданным по умолчанию конструктором, предоставляется автоматически, только если в классе нет явно определенных конструкторов.

Доступ к полям и методам объекта можно получить с помощью ссылочной переменной, используя оператор точки (`.`).

Для вновь созданных объектов выделяется память. Доступ к объектам можно получить с помощью ссылочных переменных.

Доступ к объектам осуществляется с помощью ссылочных переменных, т.е. переменных, которые содержат ссылки на объекты. Такие переменные объявляются с помощью следующего синтаксиса:

ИмяКласса ссылочнаяПеременнаяОбъекта;

Класс по существу является типом, определяемым программистом. Класс является ссылочным типом, что означает, что переменная типа класса может ссылаться на экземпляр класса. В следующем предложении объявляется переменная `myCircle` типа `Circle`:

`Circle myCircle;`

Переменная `myCircle` может ссылаться на объект типа `Circle`. В следующем предложении создается объект и переменной `myCircle` присваивается ссылка на него:

`myCircle = new Circle();`

Можно написать одно-единственное предложение, объединяющее объявление ссылочной переменной, создание объекта и присваивание этой переменной ссылки на этот объект со следующим синтаксисом:

ИмяКласса ссылочнаяПеременнаяОбъекта = new ИмяКласса();

Пример:

`Circle myCircle = new Circle();`

Переменная `myCircle` содержит ссылку на объект типа `Circle`.

Ссылочная переменная объекта, которая, казалось бы, содержит этот объект, фактически содержит ссылку на этот объект. Строго говоря, ссылочная переменная объекта и сам объект различаются, но большую часть времени это различие можно игнорировать, поэтому для простоты можно говорить, что `myCircle` является объектом типа `Circle`, и не использовать пространное описание, что `myCircle` является переменной, которая содержит ссылку на объект типа `Circle`.

В Java массивы обрабатываются так же, как объекты. Они создаются с помощью оператора `new`. Переменная массива фактически является переменной, содержащей ссылку на массив.

В терминологии ООП к элементам объекта относятся его поля данных и методы. После создания объекта можно получить доступ к его полям и вызвать его методы с помощью оператора точки (`.`), также известного как оператор доступа к элементам объекта:

- `ссылочнаяПеременнаяОбъекта.полеДанных` ссылается на поле данных объекта;
- `ссылочнаяПеременнаяОбъекта.метод(аргументы)` вызывает метод объекта.

Например, `myCircle.radius` ссылается на поле `radius` объекта `myCircle`, а `myCircle.getArea()` вызывает метод `getArea()` объекта `myCircle`.

Поле данных `radius` называется переменной экземпляра, так как его значение зависит от конкретного экземпляра класса `Circle`. По этой же причине метод `getArea()` называется методом экземпляра, так как его можно вызвать только у конкретного экземпляра класса `Circle`.

Для вызова метода класса `Math` используется `Math.имяМетода(аргументы)` (например, `Math.pow(3, 2.5)`). Вызвать `getArea()` с помощью `Circle.getArea()` нельзя. Все методы класса `Math` являются статическими методами, которые определяются с помощью ключевого слова `static`. Однако `getArea()` является методом экземпляра и, таким образом, не статическим. Он должен быть вызван из объекта с помощью `ссылочнаяПеременнаяОбъекта.имяМетода(аргументы)` (например, `myCircle.getArea()`).

Как правило, сначала создается объект, который присваивается переменной, а затем она уже используется для ссылки на этот объект. Но не всегда есть необходимость ссылаться на объект. В таком случае можно создать объект без явного присваивания его переменной с помощью синтаксиса:

```
new Circle();
```

или

```
System.out.println("Площадь равна" + new Circle(5).getArea());
```

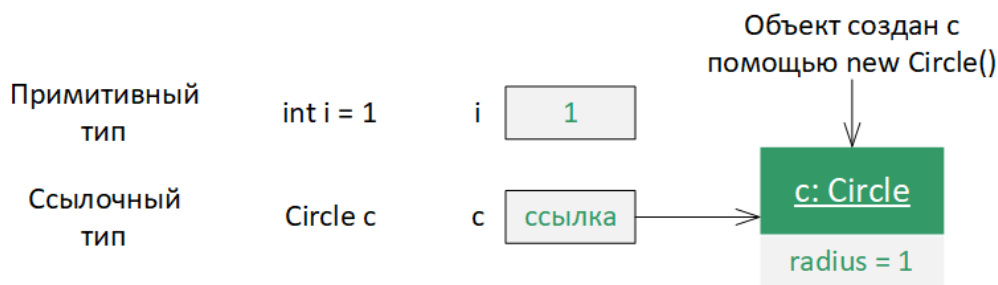
В первом предложении создается объект `Circle`, в последнем – создается объект `Circle` и вызывается его метод `getArea()` для возврата площади. Созданный таким образом объект называется анонимным объектом.

Поля данных могут быть ссылочного типа. Если поле данных ссылочного типа не ссылается ни на один объект, то это поле данных содержит специальное Java-значение, равное `null`. Аналогично `true` или `false`, `null` является литералом, но, в то время как `true` и `false` являются логическими литералами, `null` является литералом ссылочного типа.

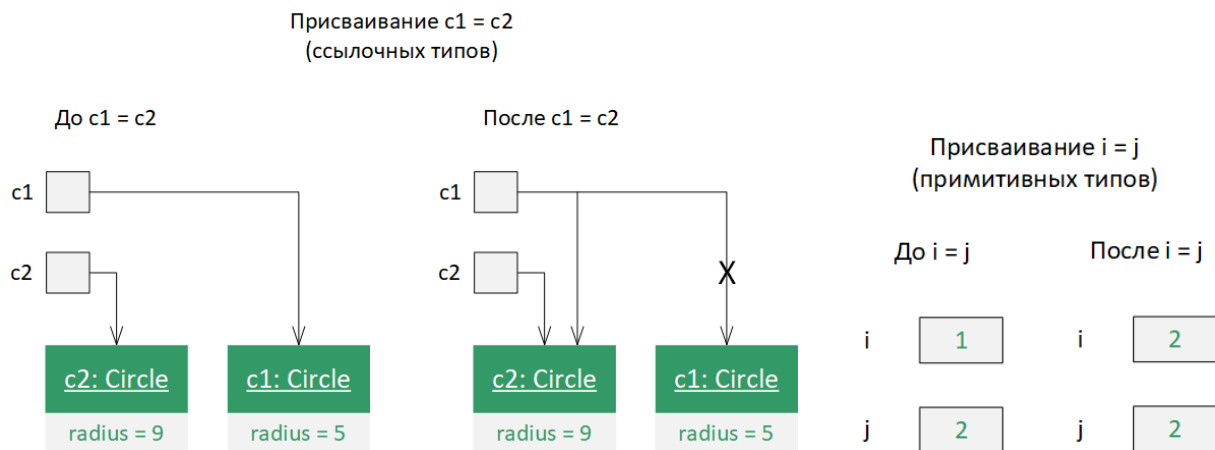
Значение поля данных по умолчанию равно `null` для ссылочного типа, `0` – для числового типа, `false` – для типа `boolean` и `\u0000` – для типа `char`. В Java, однако, не присваивается значение по умолчанию локальной переменной внутри метода.

`NullPointerException` – это распространенная ошибка во время выполнения. Она происходит при вызове метода у ссылочной переменной со значением `null`, поэтому перед вызовом метода с помощью ссылочной переменной необходимо убедиться, что этой переменной присвоена ссылка на объект.

Каждая переменная представляет ячейку памяти, содержащую значение. Объявляя переменную, вы сообщаете компилятору, значение какого типа может содержать эта переменная. Для переменной примитивного типа значение имеет примитивный тип. Для переменной ссылочного типа значение является ссылкой на местоположение объекта в памяти. Например, как показано на следующем рисунке, значение переменной `i` типа `int` равно значению `1` типа `int`, а значение объекта с типа `Circle` содержит ссылку на место в памяти, где хранится содержимое объекта типа `Circle`.



При присваивании одной переменной другой последней присваивается то же значение. Для переменной примитивного типа вещественное значение одной переменной присваивается другой переменной. Для переменной ссылочного типа ссылка одной переменной присваивается другой переменной. Как показано на рисунке справа, предложение с присваиванием $i = j$ копирует содержимое j в i для примитивных переменных.



Как показано на рисунке слева, предложение с присваиванием $c1 = c2$ копирует ссылку $c2$ в $c1$ для ссылочных переменных. После присваивания переменные $c1$ и $c2$ ссылаются на один и тот же объект.

Как показано на предыдущем рисунке слева, после присваивания $c1 = c2$, $c1$ ссылается на тот же объект, на который ссылается $c2$. Объект, на который ранее ссылался $c1$, больше не используется и поэтому теперь считается мусором. Мусор занимает место в памяти, поэтому JRE обнаруживает его и автоматически «восстанавливает» занимаемое им место для дальнейшего использования. Этот процесс называется сборкой мусора.

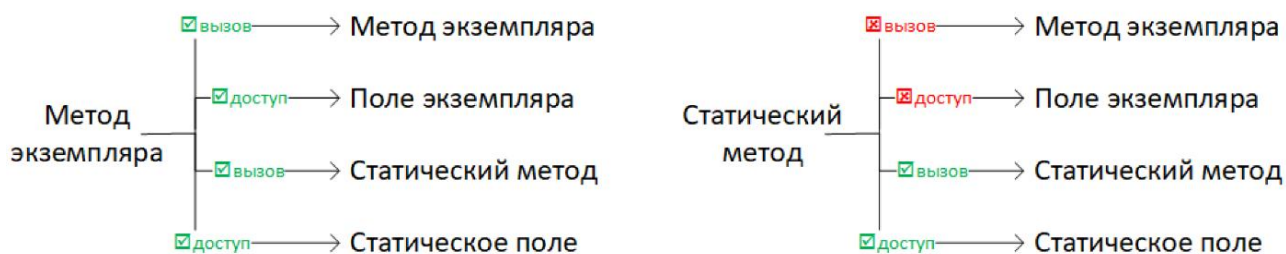
Если известно, что объект больше не нужен, то можно явно присвоить значение `null` переменной, которая ссылается на этот объект. JVM будет автоматически собирать этот мусор, если на него не ссылается ни одна переменная.

Java API содержит широкий набор классов для разработки Java-программ. Например, можно создавать случайные числа с помощью класса `Random`, а также с помощью класса `java.security.SecureRandom`. Случайные числа, сгенерированные из класса `Random`, являются детерминированными и могут быть предсказаны хакерами. Случайные числа, сгенерированные из класса `SecureRandom`, являются недетерминированными и поэтому более защищенными.

Статическая переменная является общей для всех экземпляров класса. Статический метод не может получить доступ к элементам экземпляра класса (т.е. к полям данных и методам конкретного объекта).

Поле данных `radius` в классе `Circle` называется переменной экземпляра. Переменная экземпляра привязана к конкретному объекту, т.е. она не является общей для объектов этого класса. Если требуется, чтобы все экземпляры класса имели общий доступ к полям данных, то используйте статические переменные, также называемые переменными класса. Статические переменные хранят значения переменных в общей ячейке памяти. Благодаря такому совместному использованию, если один объект изменяет значение статической переменной, то затрагиваются все экземпляры этого класса. Java поддерживает как статические методы, так и статические переменные. Статические методы можно вызывать без создания объектов.

Метод экземпляра может вызвать метод экземпляра или статический метод и получить доступ к полю данных экземпляра или статическому полю данных соответственно. Статический метод может вызвать статический метод и получить доступ к статическому полю данных, но он не может вызвать метод экземпляра или получить доступ к полю данных экземпляра, так как статические методы и статические переменные не принадлежат конкретному экземпляру. Взаимосвязь между статическими элементами класса и элементами экземпляра класса показана на следующем рисунке:



Переменная или метод, зависящие от конкретного объекта, должны быть переменной или методом экземпляра. Переменная или метод, не зависящие от конкретного объекта, должны быть статическими. Например, каждый круг имеет свой собственный радиус, следовательно, значение радиуса зависит от конкретного круга, поэтому `radius` является переменной экземпляра класса `Circle`. Поскольку метод `getArea()` зависит от конкретного круга, то он является методом экземпляра. Ни один из методов в классе `Math`, из таких как `random()`, `pow()`, `sin()` и `cos()`, не зависит от конкретного объекта, поэтому эти методы являются статическими. Метод `main()` является статическим и может быть вызван непосредственно из класса.

Распространенной ошибкой проектирования является определение метода как метода экземпляра, а не как статического метода. Например, метод `factorial(int n)` должен быть определен как статический, как показано далее, так как он не зависит от какого-либо конкретного объекта.

Доступ к классу и его элементам можно задавать с помощью модификаторов доступа. Чтобы обозначить, что доступ к классам, методам и полям данных возможен из любых других классов, можно использовать модификатор доступа `public`. Если этот модификатор доступа не используется, то классы, методы и поля данных по умолчанию доступны любому классу этого же пакета.

Пакеты можно использовать для организации классов. Для этого необходимо добавить следующую строчку в качестве первого предложения в программе:

```
package имяПакета;
```

Если класс определен без предложения `package`, то он помещается в заданный по умолчанию пакет.

В Java рекомендуется помещать классы в пакеты, а не использовать пакет, заданный по умолчанию.

В дополнение к модификаторам доступа `public` и заданному по умолчанию, Java предоставляет модификаторы `private` и `protected` для элементов класса. Если класс не определен как `public`, то доступ к нему возможен только внутри этого же пакета.

Модификатор доступа определяет, можно ли получить доступ к полям данных и методам класса за пределами этого класса. Не существует ограничений на доступ к полям данных и методам внутри класса. Как показано справа в следующей таблице, объект с типа `C` не может получить доступ к своим `private`-элементам, поскольку он находится внутри класса `Test`. Как показано слева в следующей таблице, объект с типа `C` может получить доступ к своим `private`-элементам, поскольку он определен внутри своего собственного класса.

<pre>public class C { private boolean x; public static void main(string[] args){ C c = new C(); system.out.println(c.x); system.out.println(c.convert()); } private int convert() { return x ? 1 : -1; } }</pre>	<pre>public class Test { public static void main(string[] args){ C c = new C(); system.out.println(c.x); system.out.println(c.convert()); } }</pre>
Это допустимо, т.к. объект <code>c</code> используется внутри класса <code>C</code>.	Это не допустимо, так как <code>x</code> и <code>convert</code> имеют модификатор <code>private</code> внутри класса <code>C</code>.

Модификатор `private` применяется только к элементам класса. Модификатор `public` может применяться к классу или элементам класса. Использование модификаторов `public` и `private` для локальных переменных может привести к ошибке компиляции.

В большинстве случаев конструктор должен иметь модификатор `public`. Однако, если требуется запретить пользователю создавать экземпляр класса, то используется конструктор с модификатором доступа `private`. Например, нет причин создавать экземпляр класса `Math`, так как все его поля данных и методы являются статическими. Чтобы запретить пользователю создавать объекты типа `Math`, конструктор в `java.lang.Math` определяется следующим образом:

```
private Math() {  
}
```

Создание `private`-полей защищает данные и упрощает сопровождение класса.

Для предотвращения непосредственного (прямого) изменения значений полей данных необходимо объявить их скрытыми с помощью модификатора `private`. Это называется инкапсуляцией полей данных.

Объект не может получить доступ к `private`-полю вне класса, определяющего это поле. Однако, часто требуется получить или изменить значение поля данных. Чтобы сделать `private`-поле доступным для получения, необходимо предусмотреть `getter`-метод для возврата его значения. Чтобы изменить значение `private`-поля, необходимо предусмотреть `setter`-метод для присваивания ему нового значения. `Getter`-метод также называется получающим методом (`accessor`), а `setter`-метод – модифицирующим методом (`mutator`).

`Getter`-метод имеет следующую сигнатуру:

```
public типВозвращаемогоЗначения getИмяПоля()
```

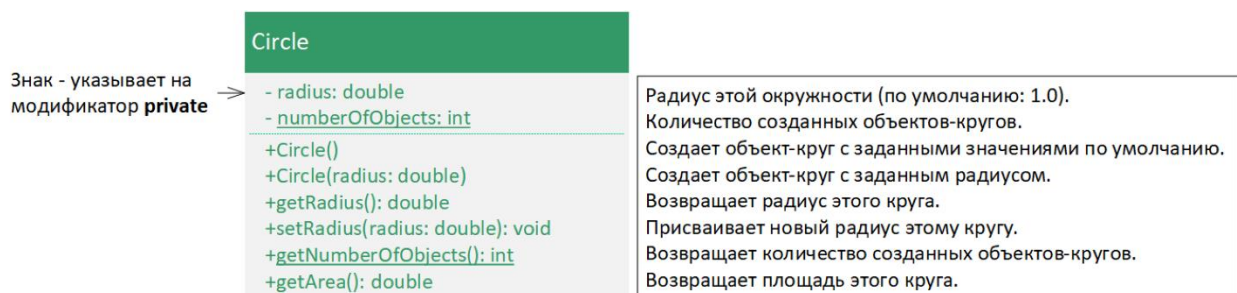
Если `типВозвращаемогоЗначения` является `boolean`, то `getter`-метод по соглашению об именовании должен быть определен следующим образом:

```
public boolean isИмяПоля()
```

`Setter`-метод имеет следующую сигнатуру:

```
public void setИмяПоля(ТипДанных значениеПоля)
```

Далее для примера определен новый класс `Circle` с `private`-полем `radius` и связанными с ним методами `getter` и `setter`. UML-диаграмма классов представлена на следующем рисунке.



Передача объекта методу заключается в передаче ссылки на этот объект.

Для создания неизменяемых объектов можно определить неизменяемые классы. Содержимое неизменяемых объектов не может быть изменено.

Как правило, создается объект, и его содержимое позднее изменяется. Однако иногда требуется создать объект, содержимое которого нельзя менять после создания этого объекта. Такой объект называется неизменяемым объектом, а его класс — неизменяемым классом.

Чтобы класс был неизменяемым, он должен удовлетворять следующим требованиям:

- все поля данных должны иметь модификатор доступа `private`;
- ни у каких полей данных не должно быть `setter`-методов;
- никакие `getter`-методы не должны возвращать ссылку на изменяемое поле данных.

Областью видимости экземпляра класса и статических переменных является весь класс, независимо от того, где объявлены эти переменные.

Переменные экземпляра и статические переменные класса называются переменными класса или полями данных класса. Переменная, объявленная внутри метода, называется локальной переменной. Областью видимости переменных класса является весь этот класс, независимо от того, где объявлены переменные. Переменные и методы класса могут появляться внутри класса в любом порядке, как показано в следующей таблице слева. Исключением является случай, когда поле данных инициализируется на основе ссылки на другое поле данных. В таких случаях сначала должно быть объявлено другое поле данных, как показано в следующей таблице справа.

<pre>public class Circle { public double getArea() { return radius * radius * Math.PI; } private double radius = 1; }</pre>	<pre>public class F { private int i; private int j = i + 1; }</pre>
Переменная radius и метод getArea() могут быть объявлены в любом порядке.	Переменная i должна быть объявлена перед j , так как начальное значение j зависит от i .

С помощью ключевого слова `this` объект ссылается сам на себя. Также можно использовать `this` внутри конструктора для вызова другого конструктора этого же класса.

Ключевое слово `this` является именем ссылки, которую объект может использовать для ссылки на себя. Ключевое слово `this` можно использовать для ссылки на элементы экземпляра класса. Например, в следующей таблице код слева использует `this` для ссылки на радиус объекта и явно вызывает его метод `getArea()`. Ссылка `this` обычно для краткости опускается, как показано в следующей таблице справа. Однако ссылка `this` необходима для ссылки на поле данных, скрытое методом или параметром конструктора, или для вызова перегруженного конструктора.

```

public class Circle {
    private double radius;
    ...
    public double getArea() {
        return this.radius * this.radius * Math.PI;
    }
    public String toString() {
        return "радиус: " + this.radius
            + "площадь: " + this.getArea();
    }
}

```

```

public class Circle {
    private double radius;
    ...
    public double getArea() {
        return radius * radius * Math.PI;
    }
    public String toString() {
        return "радиус: " + radius
            + "площадь: " + getArea();
    }
}

```

Рекомендуется использовать поле данных в качестве имени параметра в setter-методе или конструкторе, чтобы облегчить чтение кода и избежать создания ненужных имен. В этом случае необходимо использовать ключевое слово `this` для ссылки на поле данных в setter-методе. Например, метод `setRadius()` может быть реализован, как показано в следующем фрагменте кода.

```

private double radius;
public void setRadius(double radius) {
    this.radius = radius; // здесь this.radius ссылается на поле radius
}

```

И было бы неправильно, если бы этот метод был реализован, как показано далее.

```

private double radius = 1;
public void setRadius(double radius) {
    radius = radius; // здесь radius является параметром метода
}

```

Здесь поле данных `radius` скрыто в setter-методе параметром `radius`. Необходимо ссылаться на имя поля данных внутри метода с помощью синтаксиса `this.radius`. Доступ к скрытой статической переменной можно получить с помощью ссылки `ИмяКласса.имяСтатическойПеременной`. Доступ к скрытой переменной экземпляра можно получить с помощью ключевого слова `this`, как показано в следующей таблице слева.

```

public class F {
    private int i = 5;
    private static double k = 0;
    public void setI(int i) {
        this.i = i;
    }
    public static void setK(double k) {
        F.k = k;
    }
    // остальные методы опущены
}

```

Пусть `f1` и `f2` – объекты типа `F`.

Вызов `f1.setI(10)` выполняет `this.i = 10`, где `this` ссылается на `f1`.

Вызов `f2.setI(45)` выполняет `this.i = 45`, где `this` ссылается на `f2`.

Вызов `F.setK(33)` выполняет `F.k = 33`. `setK()` – статический метод.

Здесь ключевое слово `this` предоставляет способ сослаться на объект, который вызывает метод экземпляра. Для вызова `f1.setI(10)` выполняется предложение `this.i = i`, которое присваивает значение параметра `i` полю данных `i` этого вызывающего объекта `f1`. Ключевое слово `this` ссылается на объект, который вызывает метод экземпляра `setI()`, как показано в предыдущей таблице справа. Строчка `F.k = k` означает, что значение параметра `k` присваивается статическому полю `k` этого класса, которое совместно используется всеми объектами класса.

Ключевое слово `this` можно использовать для вызова другого конструктора этого же класса. Например, класс `Circle` можно переписать следующим образом:

```
public class Circle {
    private double radius;
    public Circle(double radius) {
        this.radius = radius; // здесь this используется для ссылки на поле
                             // данных radius только что созданного объекта
    }
    public Circle() {
        this(1.0); // здесь this используется для вызова другого конструктора
    }
}
```

Строчка `this(1.0)` во втором конструкторе вызывает первый конструктор со значением аргумента типа `double`.

В Java требуется, чтобы предложение `this(список-аргументов)` первым появлялось в конструкторе перед любыми другими исполняемыми предложениями.

Если у класса несколько конструкторов, то лучше реализовать их (насколько это возможно) с помощью `this(список-аргументов)`. Как правило, конструктор без или с небольшим количеством аргументов может вызвать конструктор с большим количеством аргументов, используя `this(список-аргументов)`. Этот синтаксис часто упрощает кодирование и облегчает чтение и сопровождение класса.

2.2. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МЫШЛЕНИЕ

Абстракция класса – это отделение реализации класса от его использования. Детали реализации инкапсулируются и скрываются от пользователя. Это называется инкапсуляцией класса.

Java обеспечивает несколько уровней абстракции, в частности, *абстракция класса* отделяет реализацию класса от его использования. Создатель класса описывает его функциональность и разрешает пользователю узнать, как можно его использовать. Совокупность конструкторов, методов и полей, которые доступны вне класса, а также описание их поведения служат так называемым «*контрактом*», т.е. взаимодействие с классом и его объектами основано на строго определенных соглашениях. Пользователю класса не нужно знать,

как он реализуется. Детали реализации инкапсулируются и скрываются от пользователя. Это называется *инкапсуляцией класса*. Например, можно создать объект типа `Circle` и найти площадь круга, не зная, как именно вычисляется его площадь. По этой причине класс также называют абстрактным типом данных (ADT – abstract data type).

Абстракция и инкапсуляция класса – это две стороны одной монеты. Многие примеры из реальной жизни иллюстрируют понятие абстракции классов. Рассмотрим, например, сборку компьютерной системы. У персонального компьютера есть несколько компонентов – центральный процессор, память, жесткий диск, системная плата, вентилятор и т.д. Каждый компонент можно рассматривать как объект, у которого есть свойства (поля данных) и методы. Для совместной работы компонентов необходимо только знать, как используется каждый компонент и как он взаимодействует с другими компонентами. Не нужно знать, как компоненты работают, поэтому их внутренняя реализация инкапсулирована и скрыта. Таким образом, компьютер можно собрать, не зная, как реализуется каждый из его компонентов.

Аналогия с компьютерной системой очень точно отражает объектно-ориентированный подход. Каждый компонент может рассматриваться как объект класса. Например, у вас может быть класс, моделирующий все виды вентиляторов с такими свойствами, как размер и скорость, и такими методами, как запуск и остановка. Конкретный вентилятор является экземпляром этого класса с конкретными значениями свойств.

Парадигма процедурного программирования фокусируется на проектировании методов. Парадигма объектно-ориентированного программирования соединяет данные и методы в объекте. Проектирование ПО с использованием объектно-ориентированной парадигмы фокусируется на объектах и операциях над ними.

Процедурная парадигма фокусируется на проектировании методов. Объектно-ориентированная парадигма соединяет данные и методы внутри объекта. Проектирование с использованием объектно-ориентированной парадигмы фокусируется на объектах и операциях над ними. Объектно-ориентированный подход сочетает в себе возможности процедурной парадигмы с дополнительным аспектом, который соединяет данные с операциями внутри объекта.

В процедурном программировании данные и операции над ними отделены друг от друга, и этот подход требует передачи данных методам. Объектно-ориентированное программирование помещает данные и относящиеся к ним операции в объект. Этот подход решает присущие процедурному программированию проблемы. Объектно-ориентированный подход к программированию создает программы таким образом, что все объекты связаны как со своими свойствами, так и с действиями. Использование объектов улучшает и упрощает разработку и сопровождение программ. Объектно-ориентированное Java-программирование заставляет мыслить с точки зрения объектов, а Java-программа рассматривается как совокупность взаимодействующих объектов.

Значение примитивного типа можно автоматически преобразовать в объект с помощью класса-обертки, и наоборот, в зависимости от контекста.

Преобразование значения примитивного типа данных в объект-обертку называется *упаковкой* (*boxing*). Обратное преобразование называется *распаковкой* (*unboxing*). В Java могут автоматически преобразовываться друг в друга примитивные типы и классы-обертки. Java-компилятор автоматически упаковывает примитивное значение, которое появляется в контексте, требующем объекта, а также распаковывает объект, который появляется в контексте, требующем примитивного значения. Это называется *автоупаковкой* (*autoboxing*) и *автораспаковкой* (*autounboxing*).

Например, следующее предложение слева можно упростить, как показано справа, используя автоупаковку.

```
Integer intObject = new Integer(2);
```

```
Integer intObject = 2;
```

Следующее предложение слева является таким же, как справа, из-за автоматической автораспаковки.

```
int i = 1;
```

```
int i = new Integer(1);
```

2.3. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Объектно-ориентированное программирование позволяет определять новые классы из уже существующих, т.е. можно определить более общий класс (т.е. суперкласс) и позже породить от него более конкретные (т.е. подклассы). Это называется наследованием.

Класс используется для моделирования объектов одного типа. У разных классов могут быть некоторые общие свойства и поведение, которые можно обобщить в одном классе. Этот общий класс могут совместно использовать другие классы. Можно определить более конкретный класс, порожденный от более общего. Таким образом, конкретные классы будут *наследовать* свойства и методы общего класса.

Особенности наследования в Java.

- Подкласс не является подмножеством своего суперкласса.

В отличие от традиционной интерпретации, подкласс не является подмножеством своего суперкласса. На самом деле, подкласс обычно содержит больше информации и методов, чем его суперкласс.

- Скрытые поля данных суперкласса нельзя использовать в подклассе напрямую.

Скрытые (`private`) поля данных суперкласса недоступны вне этого класса, поэтому их нельзя использовать в подклассе напрямую. Однако к ним можно обратиться/изменить с помощью общедоступных (`public`) `getter`- и `setter`-методов, если они определены в суперклассе.

- Наследование используется для моделирования отношения *is-a*.

Не стоит слепо порождать класс только ради многократного использования методов. Например, классу `Tree` нет смысла порождать класс `Person`, даже если они имеют общие свойства, такие как рост и вес. Подкласс и его суперкласс должны иметь отношение *is-a*.

- Не все отношения is-a должны моделироваться с помощью наследования.

Например, квадрат является прямоугольником, но не следует порождать класс Square от класса Rectangle, так как свойства width и height не подходят для квадрата. Вместо этого необходимо определить класс Square как порожденный от класса GeometricObject и свойство side для стороны квадрата.

- Java-класс может напрямую наследоваться только от одного суперкласса.

Некоторые языки программирования позволяют порождать подкласс из нескольких классов. Эта способность называется множественным наследованием. Однако в Java не допускается множественное наследование. Java-класс может напрямую порождаться только от одного суперкласса. Это ограничение называется единичным наследованием. Если для определения подкласса используется ключевое слово extends, то оно допускает только один родительский класс. Тем не менее, множественное наследование можно реализовать с помощью интерфейсов, которые будут представлены в следующих разделах.

Ключевое слово super ссылается на суперкласс, и его можно использовать для вызова методов и конструкторов суперкласса.

Ключевое слово this используется для ссылки на вызванный объект. Ключевое слово super ссылается на суперкласс класса, в котором это слово появляется. Его можно использовать двумя способами: для вызова конструктора суперкласса и для вызова метода суперкласса.

Конструктор используется для создания экземпляра класса. В отличие от свойств и методов, конструкторы суперкласса не наследуются подклассом. Их можно вызвать из конструкторов подклассов только с помощью ключевого слова super.

Синтаксис вызова конструктора суперкласса:

```
super();
```

или

```
super(аргументы);
```

Предложение super() вызывает безаргументный конструктор своего суперкласса, а предложение super(аргументы) вызывает конструктор суперкласса, который соответствует этим аргументам. Предложение super() или super(аргументы) должно быть первым предложением внутри конструктора подкласса, и это единственный способ явного вызова конструктора суперкласса.

Чтобы переопределить метод, он должен быть определен в подклассе с помощью той же сигнатуры, что и в суперклассе.

Подкласс наследует методы суперкласса. Иногда необходимо, чтобы подкласс изменил реализацию метода, определенного в суперклассе. Это называется переопределением метода.

Переопределяющий метод должен иметь ту же сигнатуру, что и переопределенный метод, и тот же или совместимый тип возвращаемого значения. Совместимость означает, что тип возвращаемого значения переопределяющего

метода является подтипом типа возвращаемого значения переопределенного метода.

Метод экземпляра можно переопределить, только если он доступен. Таким образом, private-метод нельзя переопределить, так как он недоступен вне собственного класса. Если метод, определенный в подклассе, является скрытым (private) в суперклассе, то эти два метода совершенно не связаны между собой.

Как и метод экземпляра, статический метод может наследоваться. Однако нельзя переопределить статический метод. Если статический метод, определенный в суперклассе, переопределен в подклассе, то метод, определенный в суперклассе, скрыт. Скрытые статические методы можно вызвать с помощью синтаксиса `ИмяСуперКласса.имяСтатическогоМетода()`.

Перегрузка означает определение нескольких методов с одинаковыми именами, но разными сигнатурами. Переопределение означает обеспечение новой реализацией метода в подклассе.

Переопределенные методы находятся в разных классах, связанных наследованием; перегруженные методы могут находиться либо в одном классе, либо в разных классах, связанных наследованием.

Переопределенные методы имеют одинаковую сигнатуру; перегруженные методы имеют одно и то же имя, но разные списки параметров.

Пример, показывающий разницу между переопределением и перегрузкой: слева метод `p(double i)` класса `A` переопределяет этот же метод, определенный в классе `B`; справа класс `A` имеет два перегруженных метода: `p(double i)` и `p(int i)`, метод `p(double i)` наследуется от `B`.

```
public class TestOverriding {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}
class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}
class A extends B {
    // Этот метод переопределяет
    // аналогичный метод класса B
    public void p(double i) {
        System.out.println(i);
    }
}
```

```
public class TestOverloading {
    public static void main(String[] args) {
        A a = new A();
        a.p(10);
        a.p(10.0);
    }
}
class B {
    public void p(double i) {
        System.out.println(i * 2);
    }
}
class A extends B {
    // Этот метод перегружает
    // аналогичный метод класса B
    public void p(int i) {
        System.out.println(i);
    }
}
```

Слева при запуске класса `TestOverriding`, и метод `a.p(10)`, и метод `a.p(10.0)` вызывают метод `p(double i)`, определенный в классе `A`, для отображения 10.0. Справа при запуске класса `TestOverloading`, `a.p(10)` вызывает метод `p(int i)`, определенный в классе `A`, для отображения 10, а `a.p(10.0)` вызывает метод `p(double i)`, определенный в классе `B`, для отображения 20.0.

Каждый Java-класс является потомком класса `java.lang.Object`.

Если при определении класса не указывается наследование, то суперклассом этого класса по умолчанию является `Object`.

Полиморфизм означает, что переменная супертипа может ссылаться на объект подтипа.

Тремя основными принципами объектно-ориентированного программирования являются инкапсуляция, наследование и полиморфизм.

Отношение наследования позволяет подклассу наследовать элементы своего суперкласса, а также добавлять новые элементы.

Подкласс – это конкретизация суперкласса; каждый экземпляр подкласса также является экземпляром суперкласса, но не наоборот. Например, каждый круг является геометрическим объектом, но не каждый геометрический объект является кругом. Следовательно, всегда можно передать экземпляр подкласса параметру типа суперкласса.

Доступ к защищенному (`protected`) элементу класса можно получить из подкласса.

До настоящего момента, чтобы указать, можно ли получить доступ к полям данных и методам за пределами класса, использовались ключевые слова `private` и `public`. `Private`-элементы могут быть доступны только внутри класса, а `public`-элементы – из любых других классов.

Как правило, желательно разрешать подклассам обращаться к полям данных и методам, определенным в суперклассе, но не разрешать неподклассам из других пакетов обращаться к этим полям данных и методам. Для этого можно использовать ключевое слово `protected`. Таким образом, можно получить доступ к защищенным полям данных и методам суперкласса из его подклассов.

Модификаторы `private`, `protected` и `public` называются модификаторами доступа (или видимости), так как они специфицируют степень доступности классов и их элементов. Видимость этих модификаторов увеличивается в следующем порядке (по убыванию): `private`, по умолчанию (модификатор отсутствует), `protected`, `public`.

В следующей таблице показана информация о видимости элементов класса.

Модификатор элементов класса	Доступ из того же класса	Доступ из того же пакета	Доступ из подкласса в другом пакете	Доступ из другого пакета
<code>public</code>	V	V	V	V
<code>protected</code>	V	V	V	x
по умолчанию (модификатор отсутствует)	V	V	x	x
<code>private</code>	V	x	x	x

Модификатор `private` необходимо использовать для полного скрытия элементов класса, чтобы к ним нельзя было получить прямой доступ вне класса.

Чтобы разрешить прямой доступ к элементам класса из любого другого класса этого же пакета, но не из других пакетов, необходимо не использовать никакие модификаторы.

Модификатор `protected` необходимо использовать, чтобы разрешить элементам класса доступ к подклассам любого пакета или классам этого же пакета.

Модификатор `public` необходимо использовать, чтобы разрешить доступ к элементам класса из любого другого класса.

Класс можно использовать двумя способами: (1) для создания экземпляров этого класса и (2) для определения подклассов, наследуемых от этого класса.

Элементы необходимо делать скрытыми, если они не предназначены для использования вне класса.

Элементы необходимо делать общедоступными, если они предназначены для пользователей класса.

Необходимо обеспечить защиту полей и методов, если они предназначены для наследников класса, но не для пользователей этого класса.

Модификаторы `private` и `protected` могут использовать только элементы класса. Модификатор `public` и модификатор, заданный по умолчанию (т.е. когда его нет), могут использовать как элементы класса, так и сам класс. Класс без модификатора (т.е. не `public`-класс) недоступен для классов из других пакетов.

Подкласс может переопределить `protected`-метод, определенный в его суперклассе, и изменить его видимость на `public`. Однако подкласс не может уменьшить степень доступности метода, который определен в суперклассе. Например, если метод определен в суперклассе как `public`, то он должен быть определен в подклассе как `public`.

От класса с модификатором `final` нельзя породить новый класс. Метод с модификатором `final` нельзя переопределить в подклассе. Поле данных с модификатором `final` является константой.

Иногда может потребоваться предотвратить наследование классов. В таких случаях нужно использовать модификатор `final`, чтобы указать, что класс является конечным и не может быть родительским. Класс `Math` является конечным классом. Классы `String`, `StringBuilder` и `StringBuffer`, а также все классы-обертки для примитивных типов данных также являются конечными классами. Например, следующий класс `A` является конечным и не может породить ни один класс:

```
public final class A {  
    // Поля данных, конструкторы и методы опущены  
}
```

Также можно определить метод, который будет конечным; `final`-метод не может быть переопределен своими подклассами. Например, следующий метод `m()` является конечным и не может быть переопределен:

```
public class Test {
    // Поля данных, конструкторы и методы опущены
    public final void m() {
        // Сделать что-нибудь
    }
}
```

Модификаторы `public`, `protected`, `private`, `static`, `abstract` и `final` используются для классов и элементов класса (данные и методы), за исключением того, что модификатор `final` также можно использовать для локальных переменных внутри метода. Локальная переменная с модификатором `final` является константой внутри метода.

2.4. ИСКЛЮЧЕНИЯ

Исключениями называются ошибки во время выполнения программы. Обработка исключений позволяет программе управлять ошибками во время выполнения и продолжать нормальную работу.

Ошибки во время выполнения программы происходят, если JVM обнаруживает невыполнимую операцию. Например, при доступе к массиву с помощью индекса, выходящего за границы, произойдет ошибка во время выполнения `ArrayIndexOutOfBoundsException`. При вводе значения типа `double` вместо целого числа произойдет ошибка во время выполнения `InputMismatchException`.

В Java ошибки во время выполнения выбрасываются в виде исключений.

Исключение — это объект, который представляет ошибку или условие, препятствующее нормальному выполнению программы.

Если исключение не обработать, то программа нормально не завершит свою работу.

Исключения выбрасываются из метода. Перехватить и обработать исключение может вызывающая метод сторона.

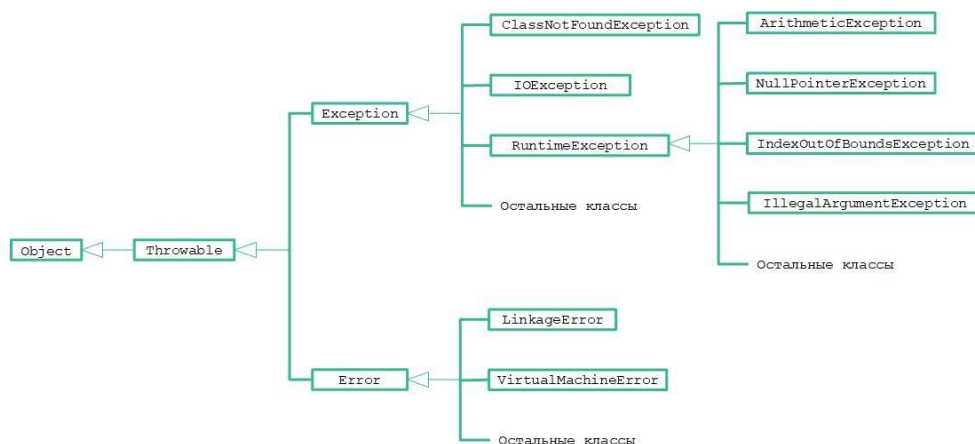
Шаблон для блока `try-throw-catch` может выглядеть следующим образом:

```
try {
    Код для выполнения;
    Предложение или метод, которые могут выбросить исключение;
    Код для выполнения;
}
catch (type ex) {
    Код для обработки исключения;
}
```

Преимущество использования обработки исключений: оно позволяет методу перебросить исключение вызывающей стороне, позволяя ей обработать это исключение. Без этой возможности вызываемый метод сам должен обработать исключение или завершить выполнение программы. Часто вызываемый метод не знает, что делать в случае ошибки. Обычно это относится к библиотечным методам. Библиотечный метод может обнаружить ошибку, но только

вызывающая сторона знает, что необходимо сделать при возникновении ошибки. Ключевым преимуществом обработки исключений является отделение обнаружения ошибки (в вызываемом методе) от обработки ошибки (в вызывающем методе).

В Java API существует множество predefined классов исключений. На следующем рисунке показаны некоторые из них.



Программа, обнаруживающая ошибку, может создать экземпляр соответствующего типа исключения и выбросить его. Это называется выбрасыванием исключения. При выбрасывании исключения его можно перехватить и обработать в блоке try-catch следующим образом:

```
try {  
    предложения; // Предложения, которые могут выбросить исключения  
}  
catch (Exception1 exVar1) {  
    обработчик для exception1;  
}  
catch (Exception2 exVar2) {  
    обработчик для exception2;  
}  
...  
catch (ExceptionN exVarN) {  
    обработчик для exceptionN;  
}
```

Если во время выполнения блока try не возникает никаких исключений, то блоки catch пропускаются.

Если одно из предложений в блоке try выбрасывает исключение, то Java пропускает оставшиеся предложения в блоке try и начинает процесс поиска кода для обработки этого исключения. Код, который обрабатывает исключение, называется обработчиком исключения; он обнаруживается с помощью передачи назад исключения по цепочке вызовов методов, начиная с текущего. По очереди, от первого до последнего, каждый блок catch проверяется на предмет того, является ли тип объекта исключения в блоке catch экземпляром класса исключений. Если является, то объект исключения присваивается объявленной

переменной и выполняется код в блоке `catch`. Если обработчик не найден, то Java выходит из этого метода, передает исключение вызывающему методу и продолжает этот же процесс для поиска обработчика. Если обработчик не найден в цепочке вызываемых методов, то программа завершает работу и отображает в консоли сообщение об ошибке. Процесс поиска обработчика называется перехватом исключения.

Блок `finally` выполняется независимо от того, произошло исключение или нет.

Иногда необходимо выполнить определенный код независимо от того, происходит или перехватывается исключение. В Java есть блок `finally`, который можно использовать для достижения этой цели. Синтаксис блока `finally` выглядит следующим образом:

```
try {  
    предложения;  
}  
catch (TheException ex) {  
    обработка ex;  
}  
finally {  
    заключительныеПредложения;  
}
```

Код в блоке `finally` выполняется в любом случае, независимо от того, происходит в блоке `try` или перехватывается исключение. Рассмотрим три возможных случая:

1. Если в блоке `try` не происходит никаких исключений, то выполняются `заклучительныеПредложения` и предложение, следующее после выполнения предложения `try`.

2. Если одно из предложений в блоке `try` приводит к исключению, которое перехватывается в блоке `catch`, то остальные предложения в блоке `try` пропускаются, выполняется блок `catch` и блок `finally`, а также предложение, следующее после предложения `try`.

3. Если одно из предложений в блоке `try` приводит к исключению, которое не перехватывается ни в одном из блоков `catch`, то остальные предложения в блоке `try` пропускаются, выполняется блок `finally`, и исключение передается вызывающей стороне.

Блок `finally` выполняется, даже если перед ним есть предложение `return`. Можно опустить блок `catch` при наличии блока `finally`.

Метод должен выбросить исключение, если ошибку необходимо обработать с вызывающей стороны.

Блок `try` содержит код, который выполняется в обычных случаях. Блок `catch` содержит код, который выполняется в исключительных случаях. Обработка исключений отделяет код для обработки ошибок от обычных задач программирования, облегчая таким образом чтение и изменение программ. Однако следует иметь в виду, что обработка исключений обычно требует больше вре-

мени и ресурсов, так как необходимо создать экземпляр для нового исключения, откатиться назад в стеке вызовов и передать исключение по цепочке вызовов методов для поиска обработчика.

Исключение происходит в методе. Если требуется, чтобы исключение обрабатывалось вызывающей стороной, то следует создать объект исключения и выбросить его. Если вы можете обработать исключение в методе, где оно происходит, то нет необходимости выбрасывать или использовать исключения.

Как правило, часто встречающиеся исключения, которые могут возникать в нескольких классах одного проекта, являются кандидатами для создания классов исключений. Простые ошибки, которые могут происходить в отдельных методах, лучше всего обрабатывать без выбрасывания исключений. Это можно сделать с помощью предложения `if` для проверки на такие ошибки.

Когда следует использовать в коде блок `try-catch`? Используйте его при возникновении непредвиденных ошибок. Не используйте блок `try-catch` для обработки ожидаемых простых ситуаций. Например, в следующем коде:

```
try {
    System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
    System.out.println("refVar равно null");
}
```

лучше заменить на

```
if (refVar != null)
    System.out.println(refVar.toString());
else
    System.out.println("refVar равно null");
```

Иногда бывает трудно определить, какие ситуации являются непредвиденными, а какие ожидаемыми. Речь идет о том, чтобы не злоупотреблять обработкой исключений в качестве способа решения простых логических задач.

Java позволяет обработчику перебросить исключение, если он не может его обработать или просто хочет, чтобы вызывающая его сторона была уведомлена об этом исключении.

Синтаксис для перебрасывания исключения выглядит следующим образом:

```
try {
    предложения;
}
catch (TheException ex) {
    выполнить операции перед выходом;
    throw ex;
}
```

Предложение `throw ex` перебрасывает исключение вызывающей стороне, так что другие обработчики получают возможность обработать исключение `ex`.

Выбрасывание одного исключения вместе с другим образует в результате сцепленное исключение.

Иногда необходимо выбрасывать новое исключение (с дополнительной информацией) вместе с исходным исключением. Это называется сцепленными исключениями.

В Java можно определить пользовательский класс исключений, породив его от класса `java.lang.Exception`.

2.5. АБСТРАКТНЫЕ КЛАССЫ И ИНТЕРФЕЙСЫ

Абстрактный класс нельзя использовать для создания объектов. Абстрактный класс содержит абстрактные методы, которые реализуются в конкретных подклассах.

В иерархии наследования с каждым новым подклассом классы становятся более специализированными и конкретными. При движении из подкласса обратно в суперкласс классы становятся менее специализированными и более общими. Проект класса должен обеспечивать наличие в суперклассе общих признаков его подклассов. Иногда суперкласс настолько абстрактен, что его нельзя использовать для создания каких-либо конкретных экземпляров класса. Такой класс называется абстрактным классом.

Абстрактные классы похожи на обычные, за исключением того, что нельзя создавать экземпляры абстрактных классов с помощью оператора `new`. Абстрактный метод определяется без реализации. Его реализация обеспечивается подклассами. Класс, содержащий абстрактные методы, должен быть определен как абстрактный.

Конструктор в абстрактном классе определен с модификатором `protected`, поскольку он используется только подклассами. При создании экземпляра конкретного подкласса вызывается конструктор его суперкласса для инициализации полей данных, определенных в суперклассе.

Интерфейс – это аналогичная классу конструкция для определения общих операций над объектами. Во многих отношениях интерфейс аналогичен абстрактному классу, но его целью является определение общего поведения для объектов связанных или несвязанных классов.

Чтобы отличить интерфейс от класса, Java использует следующий синтаксис для определения интерфейса:

```
модификатор interface ИмяИнтерфейса {  
    /** Объявление констант */  
    /** Сигнатуры абстрактных методов */  
}
```

Интерфейс в Java рассматривается как специальный класс. Каждый интерфейс компилируется в отдельный файл с байт-кодом, аналогично обычному классу. Интерфейс можно использовать более или менее таким же образом, как абстрактный класс. Например, можно использовать интерфейс в качестве типа данных для ссылочной переменной, в качестве результата приведения

типов и т.д. Как и в случае с абстрактным классом, нельзя создать экземпляр интерфейса с помощью оператора new.

В интерфейсе могут быть опущены модификаторы public static final для полей данных и модификаторы public abstract для методов. Следовательно, следующие определения интерфейсов эквивалентны:

```
public interface T {  
    public static final int K = 1;  
    public abstract void p();  
}
```

```
public interface T {  
    int K = 1;  
    void p();  
}
```

Несмотря на то, что модификатор public можно опустить для метода, определенного в интерфейсе, этот метод при реализации в подклассе должен быть определен как public.

В Java 8 с помощью ключевого слова default представлены заданные по умолчанию методы интерфейса. Заданный по умолчанию метод предоставляет заданную по умолчанию реализацию этого метода в интерфейсе. Класс, реализующий интерфейс, может просто использовать заданную по умолчанию реализацию этого метода или переопределить метод с новой реализацией. Эта возможность позволяет добавлять новый метод к существующему интерфейсу с заданной по умолчанию реализацией без необходимости переписывать код для существующих классов, которые реализуют этот интерфейс.

Начиная с Java 8, в интерфейсе можно использовать методы с модификатором public static, при этом их можно использовать так же, как методы с модификатором public static в классе.

Начиная с Java 9, в интерфейсе можно использовать методы с модификатором private. Эти методы используются для реализации заданных по умолчанию методов и методов с модификатором public static.

Ниже приведены примеры определения всех этих методов в интерфейсе:

```
public interface Java89Interface {  
    /** заданный по умолчанию метод в Java 8 */  
    public default void doSomething() {  
        System.out.println("Сделать что-нибудь");  
    }  
    /** статический метод в Java 8 */  
    public static int getAValue() {  
        return 0;  
    }  
    /** метод с модификатором private static в Java 9 */  
    private static int getAStaticValue() {  
        return 0;  
    }  
    /** метод экземпляра с модификатором private в Java 9 */  
    private void performPrivateAction() {  
    }  
}
```

Интерфейс Comparable определяет метод compareTo() для сравнения объектов.

Предположим, требуется спроектировать обобщенный метод для упорядочения двух объектов одного типа, например, таких как два студента, две даты, два круга, два прямоугольника или два квадрата. Для этого эти два объекта должны быть сравнимы, следовательно, общее поведение этих объектов должно быть сравнимым (comparable). С этой целью Java предоставляет интерфейс. Интерфейс определяется следующим образом:

```
// Интерфейс для сравнения объектов в java.lang  
package java.lang;  
public interface Comparable<E> {  
    public int compareTo(E o);  
}
```

Метод compareTo() определяет упорядоченность этого объекта с указанным o и возвращает отрицательное целое число, ноль или положительное целое число, если этот объект меньше, равен или больше o.

Интерфейс Comparable является обобщенным (generic interface). При реализации этого интерфейса обобщенный тип E заменяется конкретным типом. Многие классы в библиотеке Java реализуют интерфейс Comparable для определения естественного упорядочивания объектов (Byte, Short, Integer, Long, Float, Double, Character, BigInteger, BigDecimal, Calendar, String, Date).

Интерфейс Cloneable указывает, что объект можно клонировать.

Часто бывает необходимо создать копию объекта. Для этого требуется использовать метод clone() и понимать интерфейс Cloneable.

Обычный интерфейс содержит константы и абстрактные методы, но интерфейс Cloneable является особенным. Интерфейс Cloneable в пакете java.lang определяется следующим образом:

```
package java.lang;  
public interface Cloneable {  
}
```

Этот интерфейс пуст. Интерфейс с пустым телом называется маркер-интерфейсом (marker interface). Маркер-интерфейс используется для обозначения того, что класс обладает некоторыми желательными свойствами. Класс, реализующий интерфейс Cloneable, помечается как клонируемый, а его объекты можно клонировать с помощью метода clone(), определенного в классе Object.

Многие классы библиотеки Java (например, Date, Calendar и ArrayList) реализуют интерфейс Cloneable. Таким образом, экземпляры этих классов могут быть клонированы.

Класс может реализовать несколько интерфейсов, но наследоваться только от одного суперкласса.

Интерфейс можно использовать практически таким же образом, как абстрактный класс, но определение интерфейса отличается от определения абстрактного класса. В следующей таблице резюмируются эти отличия.

	Переменные	Конструкторы	Методы
Абстрактный класс	Без ограничений	Конструкторы вызываются подклассами по цепочке конструкторов. Абстрактный класс нельзя инстанцировать с помощью оператора new	Без ограничений
Интерфейс	Все переменные должны быть public static final	Конструкторы отсутствуют. Интерфейс нельзя инстанцировать с помощью оператора new	Может содержать public abstract методы экземпляра, public default и public static методы

В Java допускается одно единственное наследование от класса, но несколько наследований от интерфейсов. Например,

```
public class NewClass extends BaseClass
    implements Interface1, ... , InterfaceN {
    ...
}
```

Интерфейс может наследоваться от других интерфейсов с помощью ключевого слова extends. Такой интерфейс называется подинтерфейсом. Например, NewInterface в следующем коде является подинтерфейсом Interface1, ..., InterfaceN.

```
public interface NewInterface extends Interface1, ... , InterfaceN {
    // константы и абстрактные методы
}
```

Класс, реализующий NewInterface, должен реализовать абстрактные методы, определенные в NewInterface, Interface1, ..., InterfaceN. Интерфейс может наследоваться от других интерфейсов, но не от классов. Класс может наследоваться от своего суперкласса и реализовывать несколько интерфейсов.

У всех классов есть единый корень, класс Object, а у интерфейсов нет единого корня. Аналогично классу, интерфейс также определяет тип. Переменная интерфейсного типа может ссылаться на любой экземпляр класса, реализующий этот интерфейс. Если класс реализует интерфейс, то этот интерфейс аналогичен суперклассу для класса. Можно использовать интерфейс в качестве типа данных и привести переменную интерфейсного типа к его подклассу и наоборот.

Имена классов – это существительные. Имена интерфейсов могут быть как прилагательными, так и существительными.

Абстрактные классы и интерфейсы можно использовать для определения общего поведения объектов. Как правило, сильное отношение is-a, которое четко описывает связь «родитель-потомок», должно быть смоделировано с помощью классов. Например, григорианский календарь является календарем, поэтому отношение между классом `java.util.GregorianCalendar` и `java.util.Calendar` смоделировано с помощью наследования класса. Слабое отношение is-a, также известное как отношение is-kind-of, указывает на то, что объект обладает определенным свойством. Слабую связь is-a можно смоделировать с помощью интерфейсов. Например, все строки сравнимы, поэтому класс `String` реализует интерфейс `Comparable`.

Как правило, интерфейсы предпочтительнее абстрактных классов, так как интерфейс может определять общий супертип для несвязанных классов. Интерфейсы более гибкие, чем классы.

2.6. ДЖЕНЕРИКИ

Дженерики (generics – в пер. с англ. «обобщения») позволяют обнаруживать ошибки уже во время компиляции программы, а не во время ее выполнения.

Дженерики позволяют параметризовать типы, т.е. сделать типы зависимыми от параметров. Эта возможность позволяет определить класс или метод с обобщенными типами, которые компилятор может заменить на конкретные.

Java позволяет определять обобщенные классы, интерфейсы и методы, начиная с JDK 1.5. Несколько интерфейсов и классов в Java API были изменены с помощью дженериков. Например, до JDK 1.5 интерфейс `java.lang.Comparable` был определен, как показано слева, а начиная с JDK 1.5 он был изменен, как показано справа.

<pre>package java.lang; public interface Comparable { public int compareTo(Object o) }</pre>	<pre>package java.lang; public interface Comparable<T> { public int compareTo(T o) }</pre>
До JDK 1.5	JDK 1.5

Здесь `<T>` формальный обобщенный тип, который можно заменить на фактический конкретный тип. Замена обобщенного типа называется обобщенным инстанцированием. В соответствии с соглашением, для обозначения формального обобщенного типа используется заглавная буква, такая как E или T.

Чтобы увидеть преимущества использования дженериков, рассмотрим код в следующей таблице. В предложении слева объявляется, что `s` является ссылочной переменной типа `Comparable` и вызывается метод `compareTo()`

для сравнения объекта типа Date со строкой. Код скомпилируется нормально, но произойдет ошибка во время выполнения, так как строку нельзя сравнивать с датой.

В предложении справа объявляется, что c является ссылочной переменной типа Comparable<Date> и вызывается метод compareTo() для сравнения объекта типа Date со строкой. У этого кода произойдет ошибка компиляции, так как аргумент, переданный методу compareTo(), должен быть типа Date. Поскольку эти ошибки можно обнаружить уже во время компиляции, а не во время выполнения, обобщенный тип делает программу более надежной.

<pre>Comparable c = new Date(); System.out.println(c.compareTo("красный"));</pre>	<pre>Comparable<Date> c = new Date(); System.out.println(c.compareTo("красный"));</pre>
До JDK 1.5	JDK 1.5

Обобщенный тип можно определить для класса или интерфейса. Конкретный тип должен быть указан при использовании класса для создания объекта или при использовании класса или интерфейса для объявления ссылочной переменной.

Для создания стека строк используется new GenericStack<String>() или new GenericStack<>(). Это может ввести вас в заблуждение, исходя из того, что конструктор класса GenericStack должен быть определен как

```
public GenericStack<E>()
```

Это недопустимо. Он должен быть определен как

```
public GenericStack()
```

Иногда обобщенный класс может иметь несколько параметров. В этом случае поместите вместе параметры, разделенные запятыми, внутри скобок, например,

```
<E1, E2, E3>
```

Класс или интерфейс можно определить как подтип обобщенного класса или интерфейса. Например, класс java.lang.String определяется для реализации интерфейса Comparable в API Java следующим образом:

```
public class String implements Comparable<String>
```

Обобщенный класс или интерфейс, используемый без указания конкретного типа, называется «сырым» типом и обеспечивает обратную совместимость с более ранними версиями Java.

Обобщенный класс можно использовать без указания конкретного типа, например:

```
GenericStack stack = new GenericStack(); // «сырой» тип
```

Это приблизительно эквивалентно

```
GenericStack<Object> stack = new GenericStack<Object>();
```

Для указания диапазона обобщенного типа можно использовать unbounded wildcards, bounded wildcards и lower bound wildcards.

Информация о дженериках используется компилятором, но не доступна во время выполнения. Это называется стиранием типа.

Дженерики реализуются с помощью подхода, который называется стиранием типа, т.е. компилятор использует информацию о дженерике для компиляции кода, но впоследствии ее стирает, таким образом, информация о дженерике недоступна во время выполнения. Этот подход обеспечивает обратную совместимость дженерик-кода с legacy-кодом, который, в свою очередь, использует «сырые» типы.

Дженерики существуют только во время компиляции. Как только компилятор подтверждает, что обобщенный тип безопасно используется, он преобразует его в «сырой» тип. Например, компилятор проверяет, правильно ли следующий код слева использует дженерики, затем переводит его в эквивалентный код справа, чтобы использовать его во время выполнения. Код справа использует уже «сырой» тип.

```
ArrayList<String> list = new ArrayList<>();  
list.add("Осло");  
String state = list.get(0);
```

```
ArrayList list = new ArrayList();  
list.add("Осло");  
String state = (String)(list.get(0));
```

При компиляции обобщенных классов, интерфейсов и методов компилятор заменяет обобщенный тип на тип Object. Например, компилятор преобразует следующий метод слева в метод справа.

```
public static <E> void print(E[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

```
public static void print(Object[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

Если обобщенный тип ограничен, то компилятор заменит его ограниченным типом. Например, компилятор преобразует следующий метод слева в метод справа.

```
public static <E extends GeometricObject>  
    boolean equalArea(  
        E object1,  
        E object2) {  
    return object1.getArea() ==  
        object2.getArea();  
}
```

```
public static  
    boolean equalArea(  
        GeometricObject object1,  
        GeometricObject object2) {  
    return object1.getArea() ==  
        object2.getArea();  
}
```

Поскольку обобщенные типы стираются во время выполнения, существуют определенные ограничения на их использование. Вот некоторые из них:

1. Нельзя использовать `new E()`. Нельзя создать экземпляр с помощью параметра обобщенного типа. Например, следующее предложение недопустимо:

```
E object = new E();
```

Причина заключается в том, что `new E()` выполняется во время выполнения, а обобщенный тип `E` недоступен во время выполнения.

2. Нельзя использовать `new E[]`. Нельзя создать массив с помощью параметра обобщенного типа. Например, следующее предложение недопустимо:

```
E[] elements = new E[capacity];
```

Можно обойти это ограничение, создав массив типа `Object`, а затем приведя его к `E[]` следующим образом:

```
E[] elements = (E[])new Object[capacity];
```

Однако приведение к `(E[])` выбросит непроверяемое предупреждение во время компиляции. Предупреждение произойдет из-за того, что компилятор не будет уверен в успешности этого приведения во время выполнения. Например, если `E` типа `String`, а `new Object[]` – массив объектов типа `Integer`, то `(String[])(new Object[])` выбросит `ClassCastException`. Этот тип предупреждения во время компиляции является ограничением при использовании `Java-дженериков` и его нельзя избежать.

3. Параметр обобщенного типа для класса в статическом контексте не допускается. Поскольку во время выполнения у всех экземпляров обобщенного класса один и тот же класс, статические переменные и методы обобщенного класса совместно используются всеми его экземплярами. Поэтому недопустимо ссылаться на параметр обобщенного типа для класса в статическом методе, поле или предложении с инициализацией. Например, следующий код недопустим:

```
public class Test<E> {  
    public static void m(E o1) { // не допустимо  
    }  
    public static E o1; // не допустимо  
    static {  
        E o2; // не допустимо  
    }  
}
```

4. Классы исключений не могут быть обобщенными.

Обобщенный класс не может наследоваться от `java.lang.Throwable`, поэтому следующее объявление класса недопустимо:

```
public class MyException<T> extends Exception {  
}
```

Если бы оно было допустимо, то для `MyException<T>` было бы доступно предложение `catch`, как показано далее:

```
try {  
    ...  
}  
catch (MyException<T> ex) {  
    ...  
}
```

JVM должна проверить выбрасываемое из блока `try` исключение, чтобы увидеть, соответствует ли оно типу, указанному в блоке `catch`. Это невозможно, так как информация о типе отсутствует во время выполнения.

ЗАКЛЮЧЕНИЕ

В настоящее время язык Java чрезвычайно популярен. Его быстрый рост и широкое применение можно проследить по его проектным характеристикам, язык Java:

1. Простой. Язык Java частично построен на основе языка C++, но значительно упрощен и усовершенствован. Некоторые называют язык Java "C++--" потому что он похож на C++, но имеет большую функциональность и меньшее количество негативных аспектов.

2. Объектно-ориентированный. Язык Java по своей сути является объектно-ориентированным. Несмотря на то, что многие объектно-ориентированные языки начинались строго как процедурные, язык Java с самого начала проектировался как объектно-ориентированный. Объектно-ориентированное программирование (ООП) – популярный подход к программированию, заменяющий традиционную парадигму процедурного программирования.

3. Интерпретируемый. Для запуска Java-программ необходим интерпретатор. Программы компилируются в код виртуальной машины Java (Java Virtual Machine – JVM), который называется байт-кодом. Байт-код является машинно-независимым и может выполняться на любом компьютере, имеющем Java-интерпретатор, который является частью JVM.

4. Безопасный. В Java реализуется несколько механизмов безопасности для защиты системы от вредоносных программ.

5. Кроссплатформенный. С помощью JVM можно написать программу, которая будет запускаться на любой платформе.

6. Высокопроизводительный. Производительность Java иногда подвергается критике. Выполнение байт-кода никогда не происходит так же быстро, как в случае с компилируемым языком, например, таким, как C++. Поскольку язык Java – интерпретируемый, байт-код непосредственно выполняется не системой, а через интерпретатор. Однако его скорость более чем достаточная для большинства интерактивных приложений, где центральный процессор зачастую простаивает, ожидая входных данных с клавиатуры или из других источников.

7. Многопоточный. Многопоточное программирование интегрировано в Java. В других языках для обеспечения многопоточности необходимо вызывать процедуры, специфичные для операционной системы.

8. Динамичный. Язык Java был разработан для адаптации к развивающемуся окружению.

СПИСОК ЛИТЕРАТУРЫ

1. Евдокимов, А. А. Программирование на языке Java (web-формат) [Электронный ресурс. Мультимедиа] : методические указания / А. А. Евдокимов, Н. В. Майстренко. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2020 – URL : <https://www.tstu.ru/book/book/elib3/mm/2020/Evdakimov/> – ЭБС ТГТУ.
2. Вязовик, Н. А. Программирование на Java : учебное пособие / Н. А. Вязовик // Цифровой образовательный ресурс IPR SMART : [сайт]. – 3-е изд. – М. : Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021. – 601 с. – URL : <https://www.iprbookshop.ru/102048.html>
3. Монахов, В. В. Язык программирования Java и среда NetBeans : учебное пособие / В. В. Монахов // Цифровой образовательный ресурс IPR SMART : [сайт]. – 3-е изд. – М. : Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021. – 450 с. – URL : <https://www.iprbookshop.ru/102078.html>
4. Блох, Дж. Java. Эффективное программирование / Дж. Блох ; пер. В. Стрельцов ; под ред. Р. Усманов // Цифровой образовательный ресурс IPR SMART : [сайт]. – 2-е изд. – Саратов : Профобразование, 2019. – 310 с. – URL : <https://www.iprbookshop.ru/89870.html>
5. Наир, В. Предметно-ориентированное проектирование в Enterprise Java : руководство / В. Наир ; пер. с англ. А. В. Снастина // Лань : электронно-библиотечная система. – М. : ДМК Пресс, 2020. – 306 с. – URL : <https://e.lanbook.com/book/179503>
6. Коузен, К. Современный Java: рецепты программирования / К. Коузен // Лань : электронно-библиотечная система. – М. : ДМК Пресс, 2018. – 275 с. – URL : <https://e.lanbook.com/book/116121>.
7. Кишори, Ш. Java 9. Полный обзор нововведений. Для быстрого ознакомления и миграции / Ш. Кишори ; пер. с англ. А. А. Слинкин // Лань : электронно-библиотечная система. – М. : ДМК Пресс, 2018. – 544 с. – URL : <https://e.lanbook.com/book/108130>
8. Сеттер, Р. В. Изучаем Java на примерах и задачах / Р. В. Сеттер // Цифровой образовательный ресурс IPR SMART : [сайт]. – СПб. : Наука и Техника, 2016. – 240 с. – URL : <https://www.iprbookshop.ru/44025.html>
9. Летняя цифровая школа Сбера [Электронный ресурс]. – URL : <https://sberstudent.ru/digitalsummer/>

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	4
1. ОСНОВЫ JAVA-ПРОГРАММИРОВАНИЯ	5
1.1. СТИЛЬ ПРОГРАММИРОВАНИЯ И ДОКУМЕНТИРОВАНИЯ	5
1.2. ОШИБКИ ПРОГРАММИРОВАНИЯ	6
1.3. РАЗРАБОТКА ПРОСТЫХ Java-ПРОГРАММ	13
1.4. ТИПЫ ДАННЫХ И АРИФМЕТИЧЕСКИЕ ВЫРАЖЕНИЯ.....	15
1.5. СТРУКТУРЫ ВЫБОРА	19
1.6. МАТЕМАТИЧЕСКИЕ ФУНКЦИИ, СИМВОЛЫ И СТРОКИ.....	21
1.7. ЦИКЛЫ	27
1.8. МЕТОДЫ	33
1.9. ОДНОМЕРНЫЕ МАССИВЫ	39
1.10. МНОГОМЕРНЫЕ МАССИВЫ	43
1.11. РЕКУРСИЯ	44
2. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ Java-ПРОГРАММИРОВАНИЕ	46
2.1. ОБЪЕКТЫ И КЛАССЫ	46
2.2. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МЫШЛЕНИЕ.....	57
2.3. НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ.....	59
2.4. ИСКЛЮЧЕНИЯ	64
2.5. АБСТРАКТНЫЕ КЛАССЫ И ИНТЕРФЕЙСЫ	68
2.6. ДЖЕНЕРИКИ.....	72
ЗАКЛЮЧЕНИЕ	77
СПИСОК ЛИТЕРАТУРЫ	78

Учебное электронное издание

КОНКИНА Виктория Викторовна
ПУХА Анастасия Павловна
ПАТУТИН Кирилл Игоревич

РАЗРАБОТКА НА JAVA

Учебное пособие

Редактирование Е. С. Мордасовой
Графический и мультимедийный дизайнер Н. И. Кужильная
Обложка, упаковка, тиражирование Е. С. Мордасовой

ISBN 978-5-8265-3050-4



Подписано к использованию 24.08.2026.
Тираж 50 шт. Заказ № 87

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская,
д. 106/5, пом. 2, к. 14
Телефон 8(4752)63-81-08.
E-mail: izdatelstvo@tstu.ru