

1. К какому типу языков относится язык Джава

Java — язык программирования общего назначения. Относится к объектно-ориентированным языкам программирования, к языкам с строгой типизацией.

2. Особенности языка Джава

Создатели реализовали принцип WORA: write once, run anywhere или «пиши один раз, запускай везде». Это значит, что написанное на Java приложение можно запустить на любой платформе, если на ней установлена среда исполнения Java (JRE -Java Runtime Environment).

Эта задача решается благодаря компиляции написанного на Java кода в байт-код. Этот формат исполняет JVM или виртуальная машина Java. JVM — часть среды исполнения Java (JRE). Виртуальная машина не зависит от платформы.

В Java реализован механизм управления памятью, который называется сборщиком мусора или garbage collector. Разработчик создаёт объекты, а JRE с помощью сборщика мусора очищает память, когда объекты перестают использоваться. Объясняет эксперт Никита Липский: «Есть такое понятие — циклический мусор. Внутри цикла на все объекты есть ссылки, однако garbage collector в Java удалит его, если объекты не могут использоваться из программы».

Как отмечалось выше, синтаксис языка Java похож на синтаксис других си-подобных языков. Вот его некоторые особенности:

- чувствительность к регистру — идентификаторы **User** и **user** в Java представляют собой разные сущности;
- для именования методов используется lowerCamelCase. Если название метода состоит из одного слова, оно должно начинаться со строчной буквы.
Пример: **firstMethodName()**;
- для именования классов используется UpperCamelCase. Если название состоит из одного слова, оно должно начинаться с прописной буквы. Пример: **FirstClassName**.
- название файлов программы должно точно совпадать с названием класса с учётом чувствительности к регистру. Например, если класс называется **FirstClassName**, файл должен называться **FirstClassName.java**;
- идентификаторы всегда начинаются с буквы (**A-Z**, **a-z**), знака **\$** или нижнего подчёркивания **_**;

3. Класс Scanner и его использование для чтения стандартного потока ввода и 4. Класс Scanner, конструктор класса Scanner для чтения стандартного потока ввода (3-4 вместе)

Потоки ввода/вывода и строки в Java

Для ввода данных используется класс Scanner из библиотеки пакетов. В классе есть методы для чтения очередного символа заданного типа со стандартного потока ввода, а также для проверки существования такого символа.

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

```
import java.util.Scanner; // импортируем класс

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

То есть чтобы использовать сканер, необходимо сделать 3 шага:

Шаг №1: Сделать импорт сканера из пакета java.util

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

Шаг №2: Объявить сканер. Например: Scanner scan = new Scanner(System.in);

Шаг №3: Вызвать соответствующий метод для считывания с консоли. Например: int number = scan.nextInt();

Конструкторы класса Scanner

Конструктор public Scanner(Readable source)

Создает новый сканер, который создает значения, отсканированные из указанного источника.

Параметры: source — источник символов, реализующий интерфейс Readable

Не путайте с типом объекта, доступным для чтения в качестве параметра конструктора. Readable — это интерфейс, который был реализован с помощью BufferedReader, CharArrayReader, CharBuffer, FileReader, FilterReader, InputStreamReader, LineNumberReader, PipedReader, PushbackReader, Reader, StringReader.

Это означает, что мы можем использовать любой из этих классов в Java при создании экземпляра объекта Scanner.

```

package com.javatutorialhq.java.examples;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Scanner;
public class ScannerExample1 {
    public static void main(String[] args) {
        try {
            FileReader reader = new FileReader(new File("D:\\temp\\test.txt"));
            Scanner scan = new Scanner(reader);
            while(scan.hasNextLine()){
                System.out.println(scan.nextLine());
            }
            scan.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error Reading File");
            e.printStackTrace();
        }
    }
}

```

Конструктор public Scanner (InputStream source)

Создает новый сканер, который создает значения, отсканированные из указанного входного потока. Байты из потока преобразуются в символы с использованием кодировки по умолчанию базовой платформы.

Параметры: источник — входной поток для сканирования.

Пример:

Метод ввода этого конструктора — InputStream. Класс InputStream является одним из классов верхнего уровня в пакете java.io, и его использование будет проблемой.

Однако мы можем использовать подклассы InputStream, как показано ниже. Мы использовали FileInputStream, поскольку он является подклассом InputStream, при его включении проблем не возникнет.

```

package com.javatutorialhq.java.examples;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class ScannerExample2 {
    public static void main(String[] args) {
        try {
            File input = new File("D:\\temp\\test.txt");
            FileInputStream fis = new FileInputStream(input);
            Scanner scan = new Scanner(fis);
            while(scan.hasNextLine()){
                System.out.println(scan.nextLine());
            }
            scan.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error Reading File");
            e.printStackTrace();
        }
    }
}

```

Конструктор `public Scanner(File source)` выдает исключение `FileNotFoundException`

Байты из файла преобразуются в символы с кодировкой по умолчанию базовой платформы.

Параметры: источник — файл для сканирования

Этот конструктор очень прост. Просто требует источник файла. Единственной целью этого конструктора является создание экземпляра объекта `Scanner` для сканирования через файл.

```
package com.javatutorialhq.java.examples;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class ScannerExample3 {
    public static void main(String[] args) {
        try {
            File input = new File("D:\\temp\\test.txt");
            Scanner scanner = new Scanner(input);
            while(scanner.hasNextLine()){
                System.out.println(scanner.nextLine());
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error Reading File");
            e.printStackTrace();
        }
    }
}
```

Конструктор `public Scanner(Path source)` throws `IOException`

источник — путь к файлу для сканирования. Для параметра конструктора требуется источник `Path`, который используется редко.

```
package com.javatutorialhq.java.examples;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.util.Scanner;
public class ScannerExample4 {
    public static void main(String[] args) {
        try {
            Path path = FileSystems.getDefault().getPath("access.log");
            System.out.println(path.toUri());
            Scanner scanner = new Scanner(path);
            while(scanner.hasNextLine()){
                System.out.println(scanner.nextLine());
            }
            scanner.close();
        } catch (IOException e) {
            System.out.println("Error Reading the path");
            e.printStackTrace();
        }
    }
}
```

Конструктор `public Scanner(String source)`

Создает новый сканер, который выдает значения, отсканированные из указанной строки. Источник — строка для сканирования.

Этот конструктор может быть самым простым на практике, поскольку для него требуется только строковый параметр, и он строго используется для сканирования ввода строки.

```
package com.javatutorialhq.java.examples;
import java.util.Scanner;
public class ScannerExample5 {
    public static void main(String[] args) {
        String source = "This is a test";
        Scanner scannner = new Scanner(source);
        while (scannner.hasNext()) {
            System.out.println(scannner.next());
        }
        scannner.close();
    }
}
```

Scanner в Java для чтения файлов

Считать файл очень легко, используя класс Scanner. Нам просто нужно объявить это с помощью конструктора Scanner, например:

```
Scanner scan = new Scanner(new File("C:/test.txt"));
scan.useLocale(Locale.ENGLISH);
```

Хитрость в итерации по токenu Scanner состоит в том, чтобы применять те методы, которые начинаются с hasNext, hasNextInt и т.д. Давайте сначала остановимся на чтении файла построчно.

```
while(scan.hasNextLine()){
    System.out.println(scan.nextLine());
}
scan.close();
```

В приведенном выше фрагменте кода мы использовали флаг scan.hasNextLine() как средство проверки наличия токена, который в этом примере доступен на входе сканера. Метод nextLine() возвращает текущий токен и переходит к следующему.

Комбинации hasNextLine() и nextLine() широко используются для получения всех токенов на входе сканера. После этого мы вызываем метод close(), чтобы закрыть объект и тем самым избежать утечки памяти.

Считать строку из консоли ввода, используя Scanner Class

Класс Scanner принимает также InputStream для одного из своих конструкторов. Таким образом, ввод можно сделать с помощью:

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

После помещения в наш объект сканера, у нас теперь есть доступ к читаемому и конвертируемому потоку, что означает, что мы можем манипулировать или разбивать входные данные на токены. Используя богатый набор API сканера, мы можем читать токены в зависимости от разделителя, который мы хотим использовать в конкретном типе данных.

Список методов java.util.Scanner

Ниже приведен список методов `java.util.Scanner`, которые мы можем использовать для сложного анализа ввода.

return	метод	Описание
void	<code>close()</code>	Закрывает объект сканера.
Pattern	<code>delimiter()</code>	Возвращает шаблон, который объект <code>Scanner</code> в настоящее время использует для сопоставления разделителей.
String	<code>findInLine(Pattern pattern)</code>	Этот метод возвращает объект <code>String</code> , который удовлетворяет объекту <code>Pattern</code> , указанному в качестве аргумента метода.
String	<code>findInLine(String pattern)</code>	Пытается найти следующее вхождение шаблона, созданного из указанной строки, игнорируя разделители.
String	<code>findWithinHorizon(Pattern pattern, int horizon)</code>	Ищет следующее вхождение указанного шаблона.
String	<code>findWithinHorizon(String pattern, int horizon)</code>	Ищет следующее вхождение шаблона ввода, игнорируя разделитель
boolean	<code>hasNext()</code>	Возвращает <code>true</code> , если у этого сканера есть другой токен на входе.
boolean	<code>hasNext(Pattern pattern)</code>	Возвращает <code>true</code> , если следующий полный токен соответствует указанному шаблону.
boolean	<code>hasNext(String pattern)</code>	Возвращает <code>true</code> , если следующий токен соответствует шаблону, созданному из указанной строки.
boolean	<code>hasNextBigDecimal()</code>	Возвращает <code>true</code> , если следующий токен на входе этого сканера можно интерпретировать как <code>BigDecimal</code> с помощью метода <code>nextBigDecimal()</code> .
boolean	<code>hasNextBigInteger()</code>	Возвращает <code>true</code> , если следующий токен на входе этого сканера может быть интерпретирован как <code>BigInteger</code> , по умолчанию с использованием метода <code>nextBigInteger()</code> .
boolean	<code>hasNextBigInteger(int radix)</code>	аналогично методу выше, но в указанном основании с помощью метода <code>nextBigInteger()</code>
boolean	<code>hasNextBoolean()</code>	проверяет, имеет ли объект логический тип данных в своем буфере.

boolean	hasNextByte()	возвращает значение true, если следующий байт в буфере сканера можно преобразовать в тип данных байта, в противном случае — значение false.
boolean	hasNextByte(int radix)	true, если следующий токен на входе этого сканера может быть интерпретирован как значение байта в указанном основании с помощью метода nextByte().
boolean	hasNextDouble()	true, если следующий токен на входе этого сканера можно интерпретировать как двойное значение с помощью метода nextDouble().
boolean	hasNextFloat()	аналогично методу выше, но как значение с плавающей запятой, используя nextFloat().
boolean	hasNextInt()	true, если следующий токен на входе этого сканера можно интерпретировать как значение int по умолчанию с помощью метода nextInt().
boolean	hasNextInt(int radix)	возвращает логическое значение true, если маркер можно интерпретировать как тип данных int относительно radix, используемого объектом сканера, в противном случае — false.
boolean	hasNextLine()	возвращает логический тип данных, который соответствует новой строке String, которую содержит объект Scanner.
boolean	hasNextLong()	Возвращает true, если следующий токен на входе этого сканера может быть интерпретирован как длинное значение по умолчанию с использованием метода nextLong().
boolean	hasNextLong(int radix)	аналогично методу выше, но в указанном основании с помощью метода nextLong ().
boolean	hasNextShort()	как короткое значение с использованием метода nextShort().
boolean	hasNextShort(int radix)	возвращает логическое значение true, если маркер можно интерпретировать как короткий тип данных относительно radix, используемого объектом сканера, в противном случае — false.
IOException	ioException()	Возвращает IOException, последний раз выданный в основе сканера Readable.

Locale	locale()	возвращает Locale
MatchResult	match()	возвращает объект MatchResult, который соответствует результату последней операции с объектом.
String	next()	Находит и возвращает следующий полный токен.
String	next(Pattern pattern)	Возвращает следующий токен, если он соответствует указанному шаблону.
String	next(String pattern)	Возвращает следующий токен, если он соответствует шаблону, созданному из указанной строки.
BigDecimal	nextBigDecimal()	Сканирует следующий токен ввода как BigDecimal.
BigInteger	nextBigInteger()	как BigInteger.
BigInteger	nextBigInteger(int radix)	как BigInteger.
boolean	nextBoolean()	Сканирует следующий токен ввода как логическое значение и возвращает его.
byte	nextByte()	как byte.
byte	nextByte(int radix)	как byte.
double	nextDouble()	double.
float	nextFloat()	float.
int	nextInt()	int.
int	nextInt(int radix)	int.
String	nextLine()	Перемещает сканер за текущую строку и возвращает пропущенный ввод.
long	nextLong()	long.
long	nextLong(int radix)	long.
short	nextShort()	short.
short	nextShort(int radix)	short.
int	radix()	Возвращает основание сканера по умолчанию.

void	remove()	Операция удаления не поддерживается данной реализацией Iterator.
Scanner	reset()	Сбрасывает
Scanner	skip(Pattern pattern)	Пропускает ввод, соответствующий указанному шаблону, игнорируя разделители.
Scanner	skip(String pattern)	Пропускает ввод, соответствующий шаблону, созданному из указанной строки.
String	toString()	Возвращает строковое представление
Scanner	useDelimiter(Pattern pattern)	Устанавливает шаблон ограничения этого сканера в указанный шаблон.
Scanner	useDelimiter(String pattern)	как метод выше, но созданный из указанной строки.
Scanner	useLocale(Locale locale)	устанавливает local в указанный local.
Scanner	useRadix(int radix)	Устанавливает radix равным указанному.

5. Методы класса Scanner `nextLine()`, `nextInt()`, `hasNextInt()`, `hasNextLine()` и их использование для чтения ввода пользователя с клавиатуры

Метод `nextLine` считывает весь ввод до конца строки

Метод `next` класса `Scanner` читает следующий элемент на входе и возвращает его в виде строки.

Такие методы как `nextInt` и `nextDouble` считывают данные конкретных типов.

Метод `hasNextDouble()`, применённый объекту класса `Scanner`, проверяет, можно ли считать с потока ввода вещественное число типа `double`, а метод `nextDouble()` — считывает его. (Если попытаться считать значение без предварительной проверки, то во время исполнения программы можно получить ошибку (отладчик заранее такую ошибку не обнаружит))

6. Примитивные типы данных, объявление и присваивание переменных

Примитивные типы данных

В Java существует 8 примитивных типов данных:

- `byte` (целые числа, 1 байт)
- `short` (целые числа, 2 байта)

- int (целые числа, 4 байта)
- long (целые числа, 8 байтов)
- float (вещественные числа, 4 байта)
- double (вещественные числа, 8 байтов)
- char (символ Unicode, 2 байта)
- boolean (значение истина/ложь, 1 байт)

Эти 8 типов служат основой для всех остальных типов данных. Примитивные типы обладают явным диапазоном допустимых значений.

byte — диапазон допустимых значений от -128 до 127

```
//объявление переменных типа byte.
```

```
byte getByte, putByte;
```

```
// инициализация переменных
```

```
getByte = 0;
```

```
putByte = 0;
```

Переменные типа byte полезны при работе с потоком данных, который поступает из сети или файла.

short — диапазон допустимых значений от -32768 до 32767

```
//объявление и инициализация переменной типа short.
```

```
short employeeID = 0;
```

int — диапазон допустимых значений от -2147483648 до 2147483647

```
//объявление и инициализация переменной типа int.
```

```
int max = 2147483647;
```

Тип int используется чаще при работе с целочисленными данными, нежели byte и short, даже если их диапазона хватает. Это происходит потому, что при указании значений типа byte и short в выражениях, их тип все равно автоматически повышается до int при вычислении.

long — диапазон допустимых значений от -9223372036854775808 до 9223372036854775807

```
//Использование переменных типа long.
```

```
long days = getDays();

long seconds;

seconds = days * 24 * 60 * 60;
```

Тип удобен для работы с большими целыми числами.

float — диапазон допустимых значений от $\sim 1,4 \cdot 10^{-45}$ до $\sim 3,4 \cdot 10^{38}$

```
//Объявление и инициализация переменных типа float.

float usd = 31.24f;

float eur = 44.03f;
```

Удобен для использования, когда не требуется особой точности в дробной части числа.

double — диапазон допустимых значений от $\sim 4,9 \cdot 10^{-324}$ до $\sim 1,8 \cdot 10^{308}$

```
//Объявление и инициализация переменных типа double.

double pi = 3.14159;
```

Математические функции такие как `sin()`, `cos()`, `sqrt()` возвращают значение `double`

char — символьный тип данных представляет собой один 16-битный Unicode символ. Он имеет минимальное значение `'\ u0000'` (или 0), и максимальное значение `'\ uffff'` (или 65535 включительно). Символы `char` можно задавать также при помощи соответствующих чисел. Например символ `'b'` соответствует числу 1067. Рассмотрим на примере:

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

    char symb1=1067;

    char symb2 ='b';
```

```
        System.out.println("symb1 contains "+ symb1);

        System.out.println("symb2 contains "+ symb2);

    }
```

Вывод этой программы будет:

```
symb1 contains Ъ
```

```
symb2 contains Ъ
```

Небольшой пример того, как узнать, какому числу соответствует символ. Основан на претиповании данных.

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

    char ch = 'J';

    int intCh = (int) ch;

    System.out.println("J corresponds with "+ intCh);

}
```

На выводе программа показывает, что символу 'J' соответствует число 74.

```
J corresponds with 74
```

boolean — предназначен для хранения логических значений. Переменные этого типа могут принимать только одно из 2х возможных значений true или false.

```
//Объявление и инициализация переменной типа boolean.
```

```
boolean b = true;
```

Объявление переменных в java

Пример:

```
int x = 1;
```

```
int y = 2;
```

При объявлении переменной, в следующей последовательности указываются:

- **тип данных** (в данном примере — int — переменная содержит целое число),
- **имя переменной** (в данном примере имена — x и y),
- **начальное значение переменной** или, другими словами, **инициализация переменной**. В данном примере переменным x и y присвоены значения 1 и 2. Однако, это не является обязательным условием при объявлении переменной.

Пример: **объявление переменных без инициализации**:

```
int x;
```

```
int y;
```

После каждой строки при объявлении переменных необходимо ставить точку с запятой «;».

Если нужно **объявить несколько переменных одного типа**, то это также можно сделать одной строкой, указав имена переменных через запятую.

```
int x,y;
```

7. Условные операторы, полное и неполное ветвление в Джава, синтаксис

Условные операторы языка if, if-else, switch. Само по себе ветвление — это алгоритмическая конструкция, в которой в зависимости от истинности некоторого условия, выполняется одна из нескольких последовательностей действий.

Полная форма ветвления: если <условие>, то <действие 1>, иначе <действие 2>. При полной форме ветвления действия выполняются в обоих случаях: и при истинности и при ложности условия.

Неполная форма ветвления: если <условие>, то <действие>, при нем действие отсутствует, если условие не выполняется.

Пример конструкции if:

```
if (a != b && a > b) {  
    // Код будет выполнен, если и первое, и второе условие  
    // окажутся верными  
}  
  
if (a < b || a == b) {  
    // Код будет выполнен, если или первое, или второе условие  
    // окажется верным  
}
```

Пример конструкции if-else:

```
int a = 2, b = 10;  
if (a == b) { // Если a будет равным b, тогда будет выполнен код  
    // Здесь код что будет выполнен  
    // Если все одна строка кода, то фигурные скобки {}  
    // можно не ставить  
} else if (a <= b) { // Если a будет меньше или равным b  
    // Если предыдущее условие не будет выполнено,  
    // а здесь условие окажется верным,  
    // то будет выполнен этот код  
} else {  
    // Этот код сработает, если другие условия не будут выполнены  
}
```

Пример конструкции switch:

```
int x = 23;  
switch (x) { // Проверяем переменную x  
    case 1: // Если переменная будет равна 1, то здесь сработает код  
        // Может быть множество строк, а не только одна  
        System.out.print ("Переменная равна 1");  
        break; // Указываем конец для кода для этой проверки  
    case 56: // Если переменная будет равна 56, то здесь сработает код  
        // Может быть множество строк, а не только одна  
        System.out.print ("Переменная равна 56");  
        break; // Указываем конец для кода для этой проверки  
  
    // По аналогии таких проверок может быть множество  
    // Также можно добавить проверку, которая сработает в случае  
    // если все остальные проверки не сработают  
    default:  
        System.out.print ("Что-то другое");  
        break; // Можно и не ставить, так как это последнее условие  
}
```

```
int x = 23;
switch (x) { // Проверяем переменную x
    case 1: // Если переменная будет равна 1, то здесь сработает код
        // Может быть множество строк, а не только одна
        System.out.print ("Переменная равна 1");
        break; // Указываем конец для кода для этой проверки
    case 56: // Если переменная будет равна 56, то здесь сработает код
        // Может быть множество строк, а не только одна
        System.out.print ("Переменная равна 56");
        break; // Указываем конец для кода для этой проверки

    // По аналогии таких проверок может быть множество
    // Также можно добавить проверку, которая сработает в случае
    // если все остальные проверки не сработают
    default:
        System.out.print ("Что-то другое");
        break; // Можно и не ставить, так как это последнее условие
}
```

8. Оператор множественного выбора в Джава, синтаксис

Инструкция множественного выбора switch позволяет выполнять различные части программы в зависимости от того, какое значение будет иметь некоторая целочисленная переменная (её называют «переменной-переключателем», а «switch» с английского переводится как раз как «переключатель»).

Управляющие конструкции языка: **множественный выбор**

```
switch (i) {  
    case 1:  
        ...  
        break;  
    case 2:  
        ...  
        break;  
    case 3:  
        ...  
        break;  
    default:  
        ...  
        break;  
}
```

```
switch (letter) {  
    case 'A': a = a + 1;  
        break;  
    case 'B': b = b + 1;  
        break;  
    case 'C': c = c + 1;  
        break;  
    default:  
        System.out.println(" Unknown letter");  
}
```

9.Класс System. Работа со стандартами потоками вывода

Класс System является финальным и не предоставляет общедоступных конструкторов. Из-за этого все члены и методы, содержащиеся в этом классе, являются статическими по природе.

Команда **print** выводит текст без пропуска строки, а команда **println** выводит текст и ставит пропуск строки в конце.

10.Перегруженные методы out.println() класса System и их использование для вывода в консоль

System.out.println – вызов метода, который записывает информацию на консоль. Может печатать Boolean, char, int, long, float, double, не нулевой массив из char, Строку с возможностью быть пустой.

11. Константы в Джава: объявление константы

Константы в Java - это по сути такие же контейнеры для хранения данных, как и переменные, разница лишь в том, что константы не могут изменяться в течение всего выполнения работы программы.

Константы в Java принято писать символами верхнего регистра - большими буквами, чтобы отличать их от переменных.

Если попытаться изменить константу, компилятор выдаст ошибку.

Чтобы объявить константу нужно использовать ключевое слово **final**.

При объявлении констант не забываем про то, что нужно заранее решить, данные какого типа будет содержать константа.

Как работает в коде:

Создадим новый класс с именем Const

```
class Const
```

```
{  
  
}
```

Внутри фигурных скобок класса Const объявим главный метод main

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

```
{  
  
}
```

Объявим 2 константы - одну типа float, другую строкового типа и присвоим им значения

```
final float PI = 3.14f;
```

```
final String STOLITSA = "Москва";
```

После объявления констант также не забываем ставить точку с запятой.

13.Объявление и использование бестиповых переменных

(про бестиповые ничего не нашел вообще, но я думаю, что они имели ввиду это)

Давайте представим, что у нас есть объект телевизор с некоторыми характеристиками, такими как номер канала, громкость звука и флаг включенности:

```
public class TV {  
    int numberOfChannel;  
    int soundVolume;
```

```
boolean isOn;  
}
```

Как простой тип, например, `int`, может хранить эти данные? Напомним: одна переменная `int` — это 4 байта. А ведь там внутри есть две переменные (4 байта + 4 байта) этого же типа, да ещё и `boolean` (+1 байт)... Итого — 4 к 9, а ведь как правило, в объекте хранится намного больше информации.

Что делать? Нельзя же вложить объект в переменную. На этом моменте в нашей истории появляются ссылочные переменные.

Ссылочные переменные хранят адрес ячейки памяти, в которой расположен определенный объект. То есть это “визитка” с адресом, имея которую мы можем найти наш объект в общей памяти и выполнять с ним некоторые манипуляции. Ссылка на любой объект в Java представляет собой ссылочную переменную.

Как бы это выглядело с нашим объектом телевизора:

```
TV telly = new TV();
```

Переменной типа `TV` с именем `telly` мы задаем ссылку на создаваемый объект типа `TV`. То есть, JVM выделяет память в куче под объект `TV`, создает его и адрес на его местоположение в памяти, кладется в переменную `telly`, которая хранится в стеке.

Переменная типа `TV` и объект типа `TV`, заметили? Это неспроста: объектам определенного типа должны соответствовать переменные того же типа (не считая наследования и реализаций интерфейсов, но сейчас мы это не учитываем). В конце концов, не будем же мы в стаканы наливать суп? Получается, что у нас объект — это телевизор, а ссылочная переменная для него — как бы пульт управления. С помощью этого пульта мы можем взаимодействовать с нашим объектом и его данными. Например, задать характеристики для нашего телевизора:

```
telly.isOn = true;  
telly.numberOfChannel = 53;  
telly.soundVolume = 20;
```

Тут мы использовали оператор точки `.` — чтобы получить доступ и начать использование внутренних элементов объекта, на который ссылается переменная. Например, в первой строке мы сказали переменной `telly`: “Дай нам внутреннюю переменную `isOn` объекта, на который ты ссылаешься, и задай ей значение `true`” (включи нам телевизор).

1. Если простые переменные хранят биты значений, то ссылочные переменные хранят биты, представляющие способ получения объекта.
2. Ссылки на объекты объявляются лишь для одного вида объектов.
3. Любой класс в Java — это ссылочный тип.

4. По умолчанию в Java значение любой переменной ссылки — null.

14. Объявление переменных и инициализация типа класс

Инициализация переменной означает явное (или неявное) установление некоторого значения переменной.

В языке программирования Java переменные, объявленные в методе, обязательно должны быть инициализированы перед их использованием.

Если в теле некоторого метода класса, попробовать использовать объявленную, но не инициализированную переменную, то компилятор выдаст ошибку.

2. Какие существуют способы инициализации членов данных класса?

В Java можно инициализировать переменную, если она является членом класса. Существует четыре способа инициализации членов данных класса:

- инициализация по умолчанию (неявная инициализация);
- явная инициализация начальными значениями (константными значениями);
- явная инициализация методами класса;
- инициализация с помощью конструкторов классов

15. Арифметические операции, операции инкремента и декремента в Джава

Операции в Java бывают унарные (выполняются над одним операндом), бинарные (над двумя операндами).

1. Операция сложения двух чисел

```
int a = 10;
```

```
int b = 7 ;
```

```
int c = a + b;
```

2. Операция вычитания двух чисел

```
int a = 10;
```

```
int b = 7;
```

```
int c = a - b;
```

3. Операция умножения двух чисел

```
int a = 10;
```

```
int b = 7;
```

```
int c = a * b;
```

4. Операция деления двух чисел

```
int a = 10;
```

```
int b = 7;
```

```
int c = a / b;
```

При делении стоит учитывать, что если в операции участвуют два целых числа, то результат деления будет округляться до целого числа, даже если результат присваивается переменной `float` или `double`. Чтобы результат представлял число с плавающей точкой, один из операндов также должен представлять число с плавающей точкой.

5. %

Получение остатка от деления двух чисел:

```
int a = 33;
```

```
int b = 5;
```

```
int c = a % b; // 3
```

6. Инкремент

6.1 Префиксный инкремент

Предполагает увеличение переменной на единицу, например, `z=++y` (вначале значение переменной `y` увеличивается на 1, а затем ее значение присваивается переменной `z`);

```
int a = 8;
```

```
int b = ++a;
```

```
System.out.println(a); // 9
```

```
System.out.println(b); // 9
```

6.2 Постфиксный инкремент

Также представляет увеличение переменной на единицу, например, $z=y++$ (вначале значение переменной y присваивается переменной z , а потом значение переменной y увеличивается на 1)

```
int a = 8;
```

```
int b = a++;
```

```
System.out.println(a); // 9
```

```
System.out.println(b); // 8
```

7. Декремент

7.1 Префиксный декремент

Уменьшение переменной на единицу, например, $z--y$ (вначале значение переменной y уменьшается на 1, а потом ее значение присваивается переменной z)

```
int a = 8;
```

```
int b = --a;
```

```
System.out.println(a); // 7
```

```
System.out.println(b); // 7
```

7.2 Постфиксный декремент

$z=y--$ (сначала значение переменной y присваивается переменной z , а затем значение переменной y уменьшается на 1)

```
int a = 8;
```

```
int b = a--;
```

```
System.out.println(a); // 7
```

```
System.out.println(b); // 8
```

17. Приоритет арифметических операций

Одни операции имеют больший приоритет, чем другие, и поэтому выполняются вначале. Операции в порядке уменьшения приоритета: ++ (инкремент), -- (декремент) * (умножение), / (деление), % (остаток от деления) + (сложение), - (вычитание).

18. Типы данных в языке Джава, классификация, примеры

какие существуют типы данных в java:

- примитивные типы данных,
- ссылочные типы данных,

```
public class TV {  
  
    int numberOfChannel;  
  
    int soundVolume;  
  
    boolean isOn;  
  
}
```

Ссылочные типы данных:

Как простой тип, например, `int`, может хранить эти данные? Напомним: одна переменная `int` — это 4 байта. А ведь там внутри есть две переменные (4 байта + 4 байта) этого же типа, да ещё и `boolean` (+1 байт)... Итого — 4 к 9, а ведь как правило, в объекте хранится намного больше информации. Что делать? Нельзя же вложить объект в переменную. На этом моменте в нашей истории появляются ссылочные переменные. **Ссылочные переменные** хранят адрес ячейки памяти, в которой расположен определенный объект. То есть это “визитка” с адресом, имея которую мы можем найти наш объект в общей памяти и выполнять с ним некоторые манипуляции. Ссылка на любой объект в Java представляет собой ссылочную переменную. Как бы это выглядело с нашим объектом телевизора:

```
TV telly = new TV();
```

Переменной типа `TV` с именем `telly` мы задаем ссылку на создаваемый объект типа `TV`. То есть, JVM выделяет память в куче под объект `TV`, создает его и адрес на его местоположение в памяти, кладется в переменную `telly`, которая хранится в стеке. Подробнее о памяти, а именно — о стеке и еще масса полезного, можно почитать в [этой лекции](#). Переменная типа `TV` и объект типа `TV`, заметили? Это неспроста: объектам определенного типа должны соответствовать переменные того же типа (не считая наследования и реализаций интерфейсов, но сейчас мы это не учитываем). В конце концов, не будем же мы в стаканы наливать суп? Получается, что у нас объект — это телевизор, а ссылочная переменная для него — как бы пульт управления. С помощью

этого пульта мы можем взаимодействовать с нашим объектом и его данными. Например, задать характеристики для нашего телевизора:

```
telly.isOn = true;
```

```
telly.numberOfChannel = 53;
```

```
telly.soundVolume = 20;
```

Тут мы использовали оператор точки . — чтобы получить доступ и начать использование внутренних элементов объекта, на который ссылается переменная. Например, в первой строке мы сказали переменной telly: “Дай нам внутреннюю переменную isOn объекта, на который ты ссылаешься, и задай ей значение true” (включи нам телевизор).

Переопределение ссылочных переменных

Допустим, у нас есть две переменные ссылочного типа и объекты, на которые они ссылаются:

```
TV firstTV = new TV();
```

```
TV secondTV = new TV();
```

Если мы напишем:

```
firstTV = secondTV;
```

это будет означать, что мы первой переменной в качестве значения присвоили копию адреса (значение битов адреса) на второй объект, и теперь обе переменные ссылаются на второй объект (иначе говоря, два пульта от одного и того же телевизора). В тоже время, первый объект остался без переменной, которая на него ссылается. В итоге у нас есть объект, к которому невозможно обратиться, ведь переменная была такой условной ниточкой к нему, без которой он превращается в мусор, просто лежит в памяти и занимает место. Впоследствии этот объект будет уничтожен из памяти [сборщиком](#)

[мусора](#). Прервать связующую ниточку с объектом можно и без другой ссылки:

```
secondTV = null;
```

В итоге ссылка на объект останется одна — firstTV, а secondTV уже ни на кого указывать не будет (что не мешает нам в дальнейшем присвоить ей ссылку на какой-нибудь объект типа TV).

Класс String

Отдельно хотелось бы упомянуть класс [String](#). Это базовый класс, предназначен для хранения и работы с данными, которые хранятся в виде строки. Пример:

```
String text = new String("This TV is very loud");
```

Здесь мы передали строку для хранения в конструкторе объекта. Но никто так не делает. Ведь строки можно создавать:

```
String text = "This TV is very loud";
```

Гораздо удобнее, правда? По популярности использования String не уступает примитивным типам, но всё же это класс, и переменная, которая ссылается на него — не примитивного, а ссылочного типа. У String есть вот такая замечательная возможность конкатенации строк:

```
String text = "This TV" + " is very loud";
```

В итоге мы снова получим текст: This TV is very loud, так как две строки соединятся в одно целое, и переменная будет ссылаться на этот полный текст. Важным нюансом является то, что String — это неизменяемый класс. Что это значит? Возьмем такой пример:

```
String text = "This TV";
```

```
text = text + " is very loud";
```

Вроде, бы всё просто: объявляем переменную, задаем ей значение. На следующей строке изменяем его. Но не совсем-то и изменяем. Так как это неизменяемый класс, на второй строке начальное значение не меняется, а создается новое, которое в свою очередь состоит из первого + " is very loud".

1. Если простые переменные хранят биты значений, то ссылочные переменные хранят биты, представляющие способ получения объекта.
2. Ссылки на объекты объявляются лишь для одного вида объектов.
3. Любой класс в Java — это ссылочный тип.
4. По умолчанию в Java значение любой переменной ссылки — null.
5. String — стандартный пример ссылочного типа. Также этот класс является неизменяемым (immutable).
6. Ссылочные переменные с модификатором final привязаны лишь к одному объекту без возможности переопределения.

19. Массивы в Джава, объявление и инициализация массивов, длина массива, получение доступа к элементу массива

Массив представляет набор однотипных значений. Объявление массива похоже на объявление обычной переменной, которая хранит одиночное значение, причем есть два способа объявления массива:

1. тип_данных название_массива[];

2. тип_данных[] название_массива;

После объявления массива мы можем инициализировать его:

```
int nums[];
```

```
nums = new int[4]; // массив из 4 чисел
```

Длина массива.

Важнейшее свойство, которым обладают массивы, является свойство `length`, возвращающее длину массива, то есть количество его элементов:

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

```
int length = nums.length; // 5
```

Нередко бывает неизвестным последний индекс, и чтобы получить последний элемент массива, мы можем использовать это свойство:

```
int last = nums[nums.length-1];
```

Доступ к конкретной ячейке осуществляется через её номер. Номер элемента в массиве также называют индексом.

20. Массивы в Джава, как объектные типы данных, контроль доступа за выход за границы массива

```
1 public class Cat {
2
3     private String name;
4
5     public Cat(String name) {
6         this.name = name;
7     }
8
9     public static void main(String[] args) {
10
11         Cat[] cats = new Cat[3];
12         cats[0] = new Cat("Томас");
13         cats[1] = new Cat("Бегемот");
14         cats[2] = new Cat("Филипп Маркович");
15     }
16 }
```

Здесь нужно понимать несколько вещей:

В случае с примитивами массивы Java хранят множество конкретных значений (например, чисел `int`). В случае с объектами массив хранит множество ссылок. Массив `cats` состоит из трех ячеек, в каждой из которых есть ссылка на объект `Cat`. Каждая из ссылок указывает на адрес в памяти, где этот объект хранится.

Элементы массива в памяти размещаются в едином блоке. Это сделано для более эффективного и быстрого доступа к ним. Таким образом, ссылка `cats` указывает на блок в памяти, где хранятся все объекты — элементы массива. А `cats[0]` — на конкретный адрес внутри этого блока.

21. Операции над массивами, просмотр элементов массива, поиск по образцу, сортировка массива, сумма элементов массива

Получите первый и последний элемент массива

```
int firstItem = array[0];  
int lastItem = array[array.length - 1];
```

Получить случайное значение из массива

Используя объект `java.util.Random`, мы можем легко получить любое значение из нашего массива:

```
int anyValue = array[new Random().nextInt(array.length)];
```

Добавить новый элемент в массив

Как мы знаем, массивы содержат фиксированный размер значений. Поэтому мы не можем просто добавить элемент и превысить это ограничение.

Нам нужно начать с объявления нового, большего массива и скопировать элементы базового массива во второй.

К счастью, класс `Arrays` предоставляет удобный метод для репликации значений массива в новую структуру разного размера:

```
int[] newArray = Arrays.copyOf(array, array.length + 1);  
newArray[newArray.length - 1] = newItem;
```

При желании, если класс `ArrayUtils` доступен в нашем проекте, мы можем использовать его метод `add` (или его альтернативу `addAll`) для достижения нашей цели в виде однострочного оператора: **

```
int[] newArray = ArrayUtils.add(array, newItem);
```

Как мы можем себе представить, этот метод не изменяет исходный объект `array`; мы должны присвоить его вывод новой переменной.

Вставьте значение между двумя значениями

Из-за его символа индексированных значений вставка элемента в массив между двумя другими не является простой задачей.

Apache счел это типичным сценарием и реализовал метод в своем классе `ArrayUtils`, чтобы упростить решение:

```
int[] largerArray = ArrayUtils.insert(2, array, 77);
```

Просмотр элементов массива

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

```
String intArrayString = Arrays.toString(intArray);
```

```
// print directly will print reference value
System.out.println(intArray);
// [I@7150bd4d
System.out.println(intArrayString);
// [1, 2, 3, 4, 5]
```

Сумма элементов массива

```
package ua.com.prologistic;
2
3 public class SumOfArray{
4     public static void main(String args[]){
5         int[] array = {10, 30, 20, 50, 40, 10};
6         int sum = 0;
7         // цикл для обхода каждого элемента массива
8         for( int num : array) {
9             // суммирование каждого элемента массива
10            sum = sum + num;
11        }
12        System.out.println("Сумма элементов массива равна: " + sum);
13    }
14 }
```

Сортировка пузырьком

```
public static void bubbleSort(int[] arr){
    /*Внешний цикл каждый раз сокращает фрагмент массива,
    так как внутренний цикл каждый раз ставит в конец
    фрагмента максимальный элемент*/
    for(int i = arr.length-1 ; i > 0 ; i--){
        for(int j = 0 ; j < i ; j++){
            /*Сравниваем элементы попарно,
            если они имеют неправильный порядок,
            то меняем местами
            if( arr[j] > arr[j+1] ){
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

23.Операция конкатенации строк в Джава, ее обозначение и использование и ее использование

При работе со строками, а именно конкатенацией, не желательно использовать оператор «+». Для этих целей следует использовать StringBuffer или StringBuilder.

С помощью “+”

```
package ua.com.prologistic.stringutils;

public class PlusOperator {

    public static void main(String args[]){
        String simpleString = new String ("Prologistic");
        // объединение строк с помощью оператора "+"
        simpleString += ".com.ua";
    }

}
```

Конкатенация строк с помощью StringBuffer и StringBuilder будут аналогичны друг другу:

```
package ua.com.prologistic.stringutils;

public class StringBufferExample {

    public static void main(String args[]){
        StringBuffer stringBuffer = new StringBuffer("Prologistic");
        // объединение строк с помощью StringBuffer
        stringBuffer.append(".com.ua");
    }

}
```

Когда мы объединяем строки с помощью оператора «+», происходит следующее:

1. Создается новый объект StringBuilder.
2. Строка «Prologistic» копируется в только что созданный объект StringBuilder.
3. Вызывается метод append() для добавления строки «.com.ua» к объекту StringBuilder.
4. Вызывается метод toString() для получения объекта типа String с объекта StringBuilder.
5. Ссылка на только что созданный объект типа String присваивается simpleString, а старая строка «Prologistic» становится доступной для сборщика мусора.

А что же происходит, если мы используем StringBuffer или StringBuilder:

1. Создается новый объект StringBuffer со значением «Prologistic».
2. Вызывается метод append() для добавления строки «.com.ua» к объекту.
3. Вызывается метод toString() для получения объекта типа String с объекта StringBuffer.

Судя по количеству действий, необходимых для конкатенации строк, способ с использованием StringBuffer или StringBuilder является менее трудоемким, использует меньше ресурсов и производит меньше мусора для уборщика мусора.

24. Циклы в Джава, цикл с предусловием, цикл с постусловием, пример записи и использование. Условие окончания цикла

Еще одним видом управляющих конструкций являются циклы. Циклы позволяют в зависимости от определенных условий выполнять определенное действие множество раз.

В языке Java есть следующие виды циклов:

- for
- while
- do...while

1) Цикл **for**

Цикл for имеет следующее формальное определение:

```
for ([инициализация счетчика]; [условие]; [изменение счетчика])
{
    // действия
}
for (int i = 1; i < 9; i++)
{ System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
}
```

Первая часть объявления цикла - `int i = 1` создает и инициализирует счетчик `i`. Счетчик необязательно должен представлять тип `int`. Это может быть и любой другой числовой тип, например, `float`. Перед выполнением цикла значение счетчика будет равно 1. В данном случае это то же самое, что и объявление переменной. Вторая часть - условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока `i` не достигнет 9. И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: `i--`. В итоге блок цикла сработает 8 раз, пока значение `i` не станет равным 9. И каждый раз это значение будет увеличиваться на 1.

2) **do**

Цикл `do` сначала выполняет код цикла, а потом проверяет условие в инструкции `while`. И пока это условие истинно, цикл повторяется.

```
int j = 7;
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```

3) Цикл **while**

Цикл `while` сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

```
int j = 6;
while (j > 0){

    System.out.println(j);
    j--;
```

```
}
```

Так как перед выполнением тела цикла мы всегда предварительно вычисляем логическое выражение (условие входа в цикл), то этот вид while часто называют циклом с предусловием.

```
public static void main(String[] args) {  
    int number = 3; // Возводимое в степень число  
    int result = 1; // Результат возведения в степень  
    int power = 1; // Начальный показатель степени  
    while(power <= 10) { // условие входа в цикл  
        result = result * number;  
        System.out.println(number + " в степени " + power + " = " + result);  
        power++;  
    }  
}
```

В постусловном цикле выражение идет после цикла.

```
do {  
    // Тело цикла - периодически выполняемые оператор(ы)  
}while (Логическое выражение);
```

25. Циклы в Джава, итерационный цикл for(), синтаксис, счетчик цикла, условие окончания цикла, модификация счетчика, пример использования

Виды циклов:

- do...while()
- while()
- for()

Цикл for:

```
1 for ([инициализация счетчика]; [условие]; [изменение счетчика])  
2 {  
3     // действия  
4 }
```

Синтаксис цикла for

Возможные виды изменения счетчика: i++, i--, ++i, --i, i+/- const (где i – переменная-счетчик)

Пример использования циклов:

while

Используя цикл while определить сумму $2 + 4 + 6 + \dots + 2n$

Фрагмент кода, который решает данную задачу (ввод данных опущен)

```
// цикл while - вычисление суммы
```

```
int i;
```

```

int n;

int sum;

i=1;

sum=0;

while (i<=n)
{
    sum += 2*i;

    i++;
}

for

```

Используя цикл for написать фрагмент кода, который находит сумму для заданного n:

$5 + 10 + 15 + \dots + 5 \cdot n$

Фрагмент кода, который решает данную задачу

```

int i, n;

int sum; // результат - сумма

sum = 0;

// цикл for

for (i=1; i<=n; i=i+1 )

    sum += 5*i;

```

26.Способы объявления массивов в Джава, использование операции new для выделения памяти для элементов массива. Объявление с инициализацией, объявление массива определенного размера без инициализации.

Способы объявления массивов в Джава

Как и любую переменную, массив в Java нужно объявить. Сделать это можно одним из двух способов. Они равноправны, но первый из них лучше соответствует стилю Java. Второй же — наследие языка Си (многие Си-программисты переходили на Java, и для их удобства был оставлен и альтернативный способ). В таблице приведены оба способа объявления массива в Java:

№	Объявление массива, Java-синтаксис	Примеры	Комментарий
1.	<code>dataType[] arrayName;</code>	<code>int[] myArray;</code> <code>Object[] arrayOfObjects;</code>	Желательно объявлять массив именно таким способом, это Java-стиль
2.	<code>dataType arrayName[];</code>	<code>int myArray[];</code> <code>Object arrayOfObjects[];</code>	Унаследованный от C/C++ способ объявления массивов, который работает и в Java

В обоих случаях **dataType** — тип переменных в массиве. В примерах мы объявили два массива. В одном будут храниться целые числа типа `int`, в другом — объекты типа `Object`. Таким образом при объявлении массива у него появляется имя и тип (тип переменных массива). **arrayName** — это имя массива.

Использование операции `new` для выделения памяти для элементов массива.

Объявление с инициализацией, объявление массива определенного размера без инициализации.

Как и любой другой объект, создать массив Java, то есть зарезервировать под него место в памяти, можно с помощью оператора **new**. Делается это так:

```
new typeOfArray [length];
```

Где **typeOfArray** — это тип массива, а **length** — его длина (то есть, количество ячеек), выраженная в целых числах (`int`). Однако здесь мы только выделили память под массив, но не связали созданный массив ни с какой объявленной ранее переменной. Обычно массив сначала объявляют, а потом создают, например:

```
int[] myArray; // объявление массива
```

```
myArray = new int[10]; // создание, то есть, выделение памяти для массива на 10  
элементов типа      //int
```

Здесь мы объявили массив целых чисел по имени `myArray`, а затем сообщили, что он состоит из 10 ячеек (в каждой из которых будет храниться какое-то целое число). Однако гораздо чаще массив создают сразу после объявления с помощью такого сокращённого синтаксиса:

```
// ниже объявление массива определенного размера без инициализации.
```

```
int[] myArray = new int[10]; // объявление и выделение памяти “в одном флаконе”
```

Обратите внимание: После создания массива с помощью **new**, в его ячейках записаны значения по умолчанию. Для численных типов (как в нашем примере) это будет 0, для `boolean` — `false`, для ссылочных типов — `null`. Таким образом после операции

```
// ниже объявление массива определенного размера без инициализации.
```

```
int[] myArray = new int[10];
```


мы получаем массив из десяти целых чисел, и, пока это не изменится в ходе программы, в каждой ячейке записан 0.

Инициализация массива

Как создать массив в Java уже понятно. После этой процедуры мы получаем не пустой массив, а массив, заполненный значениями по умолчанию. Получаем доступ к элементу массива (то есть записываем в него значение или выводим его на экран или проделываем с ним какую-либо операцию) мы по его индексу. Инициализация массива — это заполнение его конкретными данными (не по умолчанию).

Пример: давайте создадим массив из 4 сезонов и заполним его строковыми значениями — названиями этих сезонов.

```
String[] seasons = new String[4]; /* объявили и создали массив. Java выделила память под массив из 4 строк, и сейчас в каждой ячейке записано значение null (поскольку строка — ссылочный тип)*/
```

```
seasons[0] = "Winter"; /* в первую ячейку, то есть, в ячейку с нулевым номером мы записали строку Winter. Тут мы получаем доступ к нулевому элементу массива и записываем туда конкретное значение */
```

```
seasons[1] = "Spring"; // проделываем ту же процедуру с ячейкой номер 1 (второй)
```

```
seasons[2] = "Summer"; // ...номер 2
```

```
seasons[3] = "Autumn"; // и с последней, номер 3
```

Теперь во всех четырёх ячейках нашего массива записаны названия сезонов.

Инициализацию также можно провести по-другому, совместив **с инициализацией и объявлением:**

```
String[] seasons = new String[] {"Winter", "Spring", "Summer", "Autumn"};
```

Более того, оператор new можно опустить:

```
String[] seasons = {"Winter", "Spring", "Summer", "Autumn"};
```

27.Объявление класса на Джава, пример объявления

Пример создания класса в Java, приводится ниже:

```
public class Dog{
    String breed;
    int age;
    String color;

    void barking(){
    }

    void hungry(){
    }

    void sleeping(){
    }
}
```

Breed, age, color – поля класса
barking(), hungry(), sleeping() – методы класса

28.Использования this для доступа к компонентам класса

```
class Book
{
    public String name;
    public String author;
    public int year;

    Book(){
        this.name = "неизвестно";
        this.author = "неизвестно";
        this.year = 0;
    }

    Book(String name, String author, int year){
        this.name = name;
        this.author = author;
        this.year = year;
    }

    public void Info(){
        System.out.printf("Книга '%s' (автор %s) была издана в %d году \n", name, author, year);
    }
}
```

Класс Book имеет два конструктора. Первый конструктор без параметров присваивает "неопределенные" начальные значения полям. Второй конструктор присваивает полям класса значения, которые передаются через его параметры.

Так как имена параметров и имена полей класса в данном случае у нас совпадают - name, author, year, то мы используем ключевое слово this. Это ключевое слово представляет ссылку на текущий объект. Поэтому в выражении this.name = name; первая часть this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно.

29. Создание или инстанцирование объектов типа класс

Пример:

```
ClassName ObjectName = new classConstructor()
```

30. Что такое класс в Java?

Класс – логическое описание чего-либо, шаблон, с помощью которого можно создавать реальные экземпляры этого самого чего-либо. Другими словами, это просто описание того, какими должны быть созданные сущности: какими свойствами и методами должны обладать.

Свойства – характеристики сущности, **методы** – действия, которые она может выполнять. Хорошим примером класса из реальной жизни, дающим понимание, что же такое класс,

можно считать чертежи: чертежи используются для описания конструкций (катапульта, отвертка), но чертеж – это не конструкция. Инженеры используют чертежи, чтобы создавать конструкции, так и в программировании классы используются для того, чтобы создавать объекты, обладающие описанными свойствами и методами.

```
1 public class Student {
2     private String name, group, specialty;
3
4     public Student(String name, String group, String specialty) {
5         this.name = name;
6         this.group = group;
7         this.specialty = specialty;
8     }
9
10
11 // getters/setters
12 }
```

Классы — общий шаблон того, как должен вести себя объект.

В Java есть 4 вида классов внутри другого класса:

- Вложенные внутренние классы – нестатические классы внутри внешнего класса.
- Вложенные статические классы – статические классы внутри внешнего класса.
- Локальные классы Java – классы внутри методов.
- Анонимные Java классы – классы, которые создаются на ходу.

31. Модификатор доступа или видимости в Джава, виды и использование

Модификаторы доступа — это чаще всего ключевые слова, которые регулируют уровень доступа к разным частям твоего кода. Почему «чаще всего»? Потому что один из них установлен по умолчанию и не обозначается ключевым словом :) Всего в Java есть четыре модификатора доступа. Перечислим их в порядке от самых строгих до самых «мягких»:

- **private;**
- **default (package visible);**
- **protected;**
- **public.**

Private — наиболее строгий модификатор доступа. Он ограничивает видимость данных и методов пределами одного класса.

Собственно, **ограничение доступа к полям и реализация геттеров-сеттеров — самый распространенный пример использования private в реальной работе.** То есть

реализация инкапсуляции в программе — главное предназначение этого модификатора.

Дальше у нас по списку идет модификатор **default** или, как его еще называют, **package visible**. Он не обозначается ключевым словом, поскольку установлен в Java по умолчанию для всех полей и методов.

Случаи его применения ограничены, как и у модификатора `protected`. Чаще всего `default`-доступ используется в пакете, где есть какие-то классы-утилиты, не реализующие функциональность всех остальных классов в этом пакете.

Следующий по строгости модификатор доступа — **protected**.

Поля и методы, обозначенные модификатором доступа `protected`, будут видны:

- в пределах всех классов, находящихся в том же пакете, что и наш;
- в пределах всех классов-наследников нашего класса.

public создан для того, чтобы отдавать что-то пользователям. Например, интерфейс твоей программы. Части кода, помеченные модификатором `public`, предназначены для конечного пользователя.

32. Чем отличаются `static`-метод класса от обычного метода класса:

Статические методы можно вызывать не используя ссылку на объект. В этом их ключевое отличие от обычных методов класса. Для объявления таких методов используется ключевое слово `static`. На методы, объявленные как `static`, накладывается следующие ограничения:

- Они могут непосредственно вызывать только другие статические методы.
- Им непосредственно доступны только статические переменные.
- Они не могут делать ссылки типа `this` или `super`.

Пример использования статических методов:

```
public class StaticMethodClass {
    static int staticVar = 3;
    int nonStaticVar;

    public void nonStaticMethod() {
        System.out.println("Нестатический метод");
    }

    static void staticMethod(int localVar) {
        System.out.println("localVar = " + localVar);
        System.out.println("staticVar = " + staticVar);
        //Нельзя обратиться к нестатической переменной из статического метода
        //System.out.println("nonStaticVar = " + nonStaticVar);
    }
}
```

```

public static void main(String[] args) {
    staticMethod(42);
    //Нельзя обратиться к нестатическому методу без указания объекта
    //nonStaticMethod();
    StaticMethodClass useStatic = new StaticMethodClass();
    useStatic.nonStaticMethod();
    useStatic.staticMethod(67);
}
}

```

```

public class StaticMethodDemo {
    public static void main(String[] args) {
        StaticMethodClass.staticMethod(42);
    }
}

```

33.Для чего используется оператор new?

Оператор (операторная функция) **new** создает экземпляр объекта, встроенного или определенного пользователем, имеющего конструктор.

Синтаксис

```
new constructor([[arguments]])
```

Параметры

constructor

Функция, задающая тип объекта.

arguments

Список параметров, с которыми будет вызван конструктор.

Описание

Создание объекта, определенного пользователем, требует два шага:

1. Написать функцию, которая задаст тип объекта.
2. Создать экземпляр объекта, используя **new**.

Чтобы определить новый тип объекта, создайте функцию, которая задаст его и имя и свойства. Свойство объекта также может быть объектом. Примеры приведены ниже.

Когда выполняется `new Foo(...)`, происходит следующее:

1. Создается новый объект, наследующий `Foo.prototype`.
 2. Вызывается конструктор — функция `Foo` с указанными аргументами и `this`, привязанным к только что созданному объекту. `new Foo` эквивалентно `new Foo()`, то есть если аргументы не указаны, `Foo` вызывается без аргументов.
 3. Результатом выражения `new` становится объект, возвращенный конструктором. Если конструктор не возвращает объект явно, используется объект из п. 1. (Обычно конструкторы не возвращают значение, но они могут делать это, если нужно переопределить обычный процесс создания объектов.)
- Всегда можно добавить свойство к уже созданному объекту. Например, `car1.color = "black"` добавляет свойство `color` к объекту `car1`, и присваивает ему значение `"black"`. Это не затрагивает другие объекты. Чтобы добавить свойство ко всем объектам типа, нужно добавлять его в определение типа `Car`.

Добавить свойство к ранее определенному типу можно используя свойство `Function.prototype`. Это определит свойство для всех объектов, созданных этой функцией, а не только у какого-либо экземпляра. Следующий пример добавляет свойство `color` со значением `null` всем объектам типа `car`, а потом меняет его на `"black"` только у экземпляра `car1`.

34. Можно ли вызвать static-метод внутри обычного метода?

Статические методы можно вызывать откуда угодно — из любого места программы. А значит, их можно вызывать и из статических методов, и из обычных. Никаких ограничений тут нет.

35. Как вызвать обычный метод класса внутри static-метода?

В обычном методе класса всегда есть скрытый параметр — `this` — ссылка на объект класса, у которого был вызван метод. Каждый раз, когда вы вызываете обычный метод внутри другого обычного метода, для этого вызова используется скрытый параметр `this`. Именно поэтому нельзя вызвать обычный метод из статического. Внутри статического метода просто нет скрытой переменной с именем `this`.

36. Для чего используется в Джава ключевое слово `this`?

В Java `'this'` это ссылка на сам объект (в C++ это указатель). Служебное слово `'this'` также используется, чтобы вызвать другой конструктор в том же классе (методичка)

Ключевое слово Java «`this`» используется для ссылки на текущий экземпляр метода, в котором он используется.

Ниже приведены способы его использования:

- 1) Чтобы конкретно обозначить, что переменная экземпляра используется вместо статической или локальной переменной.
- 2) This используется для ссылки на конструкторы
- 3) This используется для передачи текущего экземпляра java в качестве параметра
- 4) Может быть использовано для возврата текущего экземпляра
- 5) This можно использовать для получения дескриптора текущего класса
- 6) this можно также использовать для ссылки на внешний объект
(из интернета)

37.Объявление и использование методов, объявленных с модификатором public static

Метод, объявленный с модификатором public static доступен везде (в самом классе, внутри дочерних классов и т.д.).

Пример использования статических методов:

```
public class StaticMethodClass {
    static int staticVar = 3;
    int nonStaticVar;

    public void nonStaticMethod() {
        System.out.println("Нестатический метод");
    }

    static void staticMethod(int localVar) {
        System.out.println("localVar = " + localVar);
        System.out.println("staticVar = " + staticVar);
        //Нельзя обратиться к нестатической переменной из статического метода
        //System.out.println("nonStaticVar = " + nonStaticVar);
    }

    public static void main(String[] args) {
        staticMethod(42);
        //Нельзя обратиться к нестатическому методу без указания объекта
        //nonStaticMethod();
        StaticMethodClass useStatic = new StaticMethodClass();
        useStatic.nonStaticMethod();
        useStatic.staticMethod(67);
    }
}

public class StaticMethodDemo {
    public static void main(String[] args) {
        StaticMethodClass.staticMethod(42);
    }
}
```

38. Синтаксис объявления методов, тип возвращаемого значения, формальные параметры и аргументы

Метод — это именованный блок кода, объявляемый внутри класса. Он содержит некоторую законченную последовательность действий (инструкций), направленных на решение отдельной задачи, который можно многократно использовать. Иными словами, метод — это некоторая функция: что-то, что умеет делать ваш класс. В других языках тоже присутствуют функции. Только в Java они являются членами классов и, согласно терминологии ООП, называются методами. Но прежде чем продолжить, давайте рассмотрим небольшой пример:

```
public String constructHelloSentence(String name) {  
    String resultSentence = "Hello world! My name is " + name;  
    System.out.println(resultSentence);  
    return resultSentence;  
}
```

Тут ничего сложного: метод Java, задача которого сформировать строку приветствия, с именем, которое мы ему передаем. Как например — Hello world! My name is Bobby. Давайте как следует разберемся с построением метода, рассмотрев каждое ключевое слово в объявлении метода (слева направо). Наше первое ключевое слово — **public**, и оно обозначает модификатор доступа.

В языке Java применяют такие модификаторы доступа:

- **public:** публичный. Методы или поля с этим модификатором общедоступны, видимы другим классам (а точнее, их методам и полям) из текущего пакета и из внешних пакетов. Это самый широкий уровень доступа из известных;
- **protected:** к методам или переменным с этим модификатором есть доступ из любого места в текущем классе или пакете, или в классах, наследующих данный, а заодно — и методы или поля, даже если они находятся в других пакетах.

```
protected String constructHelloSentence(String name) {...}
```

- **Модификатор по умолчанию.** Если у поля или метода класса нет модификатора, применяется модификатор по умолчанию. В таком случае поля или методы видны всем классам в текущем пакете (как **protected**, только с отсутствием видимости при наследовании).

```
String constructHelloSentence(String name) {...}
```

- **private:** антипод модификатора **public**. Метод или переменная с таким модификатором доступны исключительно в классе, в котором они объявлены.


```
private String constructHelloSentence(String name) {...}
```

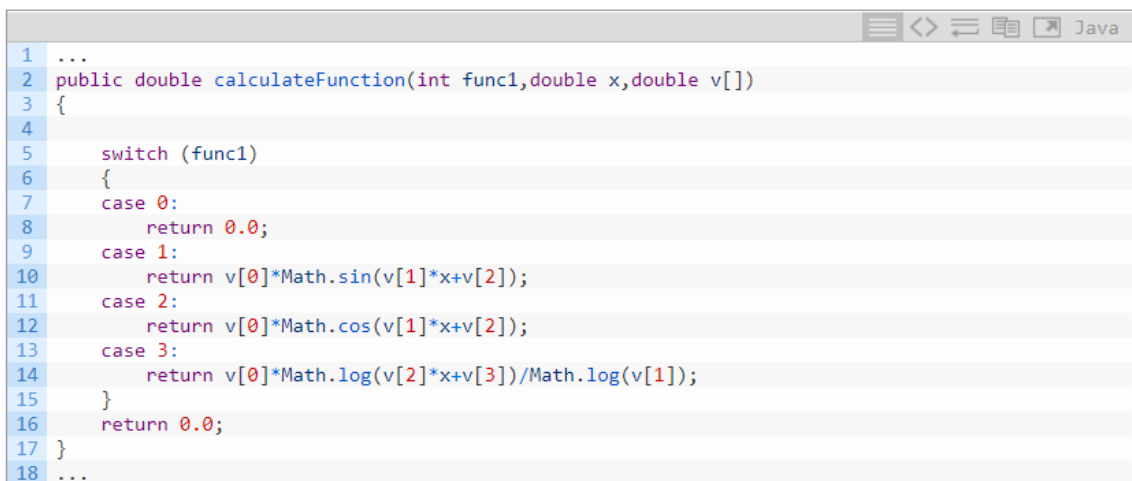
Далее мы имеем String в сигнатуре метода (первая строка метода, описывающая его свойства).

Возвращаемое значение

Возвращаемое значение — это данные (некий результат выполнения метода), которые приходят на его место после вызова.

Формальные параметры или просто **параметры** — это параметры, перечисленные в описании функции, процедуры или метода. Не следует путать их с фактическими параметрами.

Формальные параметры в Java:



```
1 ...
2 public double calculateFunction(int func1,double x,double v[])
3 {
4
5     switch (func1)
6     {
7         case 0:
8             return 0.0;
9         case 1:
10            return v[0]*Math.sin(v[1]*x+v[2]);
11        case 2:
12            return v[0]*Math.cos(v[1]*x+v[2]);
13        case 3:
14            return v[0]*Math.log(v[2]*x+v[3])/Math.log(v[1]);
15    }
16    return 0.0;
17 }
18 ...
```

В этом примере func1, x, v — это формальные параметры метода calculateFunction.

39. Методы пустым списком параметров

Если метод не имеет параметров, то Вы можете внутри метода объявить локальные переменные. Они могут использоваться точно так же, как любая другая переменная, но доступна только внутри области действия метода. Вот пример:

```
public void writeText() {
int localVariable1 = 1;
int localVariable2 = 2;
System.out.println( localVariable1 + localVariable2 );
}
```

Локальные переменные также могут быть объявлены как окончательные. Если вы объявите их как final, значение не может быть изменено. Если переменная является

ссылкой на объект, то ее нельзя изменить, но значения внутри ссылочного объекта все еще можно изменить.

40. Стандартные методы класса сеттеры и геттеры, синтаксис и их назначение?

```
1 class Date
2 {
3     private:
4         int m_day;
5         int m_month;
6         int m_year;
7
8     public:
9         int getDay() { return m_day; } // геттер для day
10        void setDay(int day) { m_day = day; } // сеттер для day
11
12        int getMonth() { return m_month; } // геттер для month
13        void setMonth(int month) { m_month = month; } // сеттер для month
14
15        int getYear() { return m_year; } // геттер для year
16        void setYear(int year) { m_year = year; } // сеттер для year
17};
```

геттеры — это функции, которые возвращают значения закрытых переменных-членов класса;

сеттеры — это функции, которые позволяют присваивать значения закрытым переменным-членам класса.

41. Может ли быть поле данных класса определено как с модификатором `static` и `final` одновременно, и что это означает?

«Constant (константа)» — слово в английском языке, относящееся в основном к «ситуации, которая не меняется». Это одна из фундаментальных концепций программирования в Java, и у нее нет каких-либо специальных предпосылок или концепций, которые необходимо знать перед изучением, кроме базовых навыков программирования.

Константы в Java используются, когда необходимо реализовать «статическое» или постоянное значение для переменной. Язык программирования напрямую не поддерживает константы. Чтобы сделать любую переменную ею, мы должны использовать модификаторы `static` и `final`.

- Модификатор `static` делает переменную доступной без загрузки экземпляра ее определяющего класса.
- Последний модификатор делает переменную неизменной.

Причина, по которой мы должны использовать как статические, так и конечные модификаторы, заключается в том, что:

- Когда мы объявим переменную «var» только как статическую, все объекты одного класса смогут получить доступ к этому 'var' и изменить его значения.
- Когда мы объявляем переменную только как final, для каждого отдельного объекта будет создано несколько экземпляров одного и того же значения константы, и это неэффективно / нежелательно.
- Когда мы используем как static, так и final, тогда «var» остается статичным и может быть инициализирован только один раз, что делает его надлежащей константой, которая имеет общую ячейку памяти для всех объектов своего содержащего класса.

42. Методы класса, конструкторы, синтаксис и назначение

Допустим, создадим класс кот, с некоторыми полями данных:

```
1. public class Cat {  
2.  
3.     String color;  
4.     int weight;  
5.     String sex;  
6.  
7.  
8.     void myaukat(){  
9.  
10.    }  
11.    void walk(){  
12.  
13.    }  
14.  
15. }
```

Название объекта — название класса; свойства — поля класса и действия — это будут методы нашего класса.

В классе мы описали сам объект, его свойства и методы, но мы не создали конкретного кота. Для того, чтобы создавать конкретные объекты в джаве предусмотренное ключевое слово new и конструктор. По сути дела, конструктор — это тоже метод. Только он создан чтобы создавать объект. Правильная форма записи конструктора такая: ИмяКласса(). Это конструктор по умолчанию. Такой конструктор создаст нашего котенка, но животное будет без набора параметров. Для того, чтобы создать объект с параметрами можно применять конструктор с параметрами: ИмяКласса(параметр1, параметр2){ поле класса = параметр1; поле класса = параметр2;}. Конструкторов в классе можно создавать неограниченное количество и с разным набором параметров. Если Вы не создали конструктора, то будет использован конструктор по умолчанию: без параметров.

43. Может ли класс иметь в своем составе несколько конструкторов?

Да, может. Конструкторы одного класса могут иметь одинаковое имя и различную сигнатуру. Такое свойство называется совмещением или перегрузкой (overloading). Если класс имеет несколько конструкторов, то присутствует перегрузка конструкторов.

44. Может ли конструктор класса возвращать значение?

Сигнатура конструктора – это количество и типы параметров, а также последовательность их типов в списке параметров конструктора. Тип возвращаемого результата не учитывается.

Конструктор не возвращает никаких параметров. Это положение объясняет в некотором смысле, как Java различает перегруженные конструкторы или методы. Java различает перегруженные методы не по возвращаемому типу, а по числу, типам и последовательности типов входных параметров. Конструктор не может возвращать даже тип **void**, иначе он превратится в обычный метод, даже не смотря на сходство с именем класса.

В конструкторе разрешается записывать оператор **return**, но только пустой, без всякого возвращаемого значения.

45. Наследование в Джава. Вид наследования и синтаксис Ключевое слово **extends**

Наследование в программировании на Java может быть определено как процесс, в котором один класс приобретает свойства (методы и поля) другого. С использованием наследования информация становится управляемой в иерархическом порядке

Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово **extends**, после которого идет имя базового класса

Пример наследования:

```
public class Sedan extends Car {  
  
    public Sedan(String model, int maxSpeed, int yearOfManufacture {  
  
        super(model, maxSpeed, yearOfManufacture);  
  
    }  
  
}
```

46. Что означает перегрузка метода в Java (overload)? Пестерников

В Java разрешается в одном и том же классе определять два или более метода с одним именем, если только объявления их параметров отличаются. Это называется *перегрузкой методов*.

Перегрузка методов является одним из способов поддержки полиморфизма в Java.

47. Что означает переопределение метода в Java (override)? Пестерников

Переопределение метода (англ. *Method overriding*) в объектно-ориентированном программировании — одна из возможностей языка программирования, позволяющая подклассу или дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в одном из суперклассов или родительских классов.

48. В чем разница между перегрузкой и переопределением методов, поясните

Перегрузка метода имеет дело с понятием наличия двух или более методов в одном классе с одинаковым именем, но разными аргументами.

Переопределение метода означает наличие двух методов с одинаковыми аргументами, но разными реализациями. Один из них будет существовать в родительском классе, а другой — в производном или дочернем классе. Аннотация `@Override`, хотя и не является обязательной, может быть полезна для обеспечения надлежащего переопределения метода во время компиляции.

49. Абстрактные классы в Джава и абстрактные методы класса

Кроме обычных классов в Java есть абстрактные классы. Абстрактный класс похож на обычный класс. В абстрактном классе также можно определить поля и методы, но в то же время нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы призваны предоставлять базовый функционал для классов-наследников. А производные классы уже реализуют этот функционал.

При определении абстрактных классов используется ключевое слово `abstract`:

```
1 public abstract class Human{
2
3     private String name;
4
5     public String getName() { return name; }
6 }
```

Но главное отличие состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта. Например, следующим образом:

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

Кроме обычных методов абстрактный класс может содержать абстрактные методы. Такие методы определяются с помощью ключевого слова `abstract` и не имеют никакой реализации:

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

Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

50. Виды наследования в Джава, использование интерфейсов для реализации наследования

Интерфейсы, как и классы, могут наследоваться:

При применении этого интерфейса класс BaseAction должен будет реализовать как методы и свойства интерфейса IRunAction, так и методы и свойства базового интерфейса IAction, если эти методы и свойства не имеют реализации по умолчанию.

Однако в отличие от классов мы не можем применять к интерфейсам модификатор **sealed**, чтобы запретить наследование интерфейсов.

Также мы не можем применять к интерфейсам модификатор **abstract**, поскольку интерфейс фактически итак, как правило, предоставляет абстрактный функционал, который должен быть реализован в классе или структуре (за исключением методов и свойств с реализацией по умолчанию).

Однако методы интерфейсов могут использовать ключевое слово **new** для сокрытия методов из базового интерфейса

При наследовании интерфейсов следует учитывать, что, как и при наследовании классов, производный интерфейс должен иметь тот же уровень доступа или более строгий, чем базовый интерфейс

Но не наоборот. Например, в следующем случае мы получим ошибку, и программа не скомпилируется, так как производный интерфейс имеет менее строгий уровень доступа, нежели базовый

Виды наследования:

Одиночное наследование

Многоуровневое наследование

Иерархическое наследование

Множественное наследование (Очень важно запомнить, что Java не поддерживает множественное наследование. Это значит, что класс не может продлить более одного класса. Значит, следующее утверждение НЕВЕРНО: тем не менее, класс может реализовать один или несколько интерфейсов, что и помогло Java избавиться от невозможности множественного наследования.)

51. Что наследуется при реализации наследования в Джава (какие компоненты класса), а что нет?

Модификаторы	То же класс	То же пакет	Подкласс	Другие пакеты
public	ДА	ДА	ДА	ДА
protected	ДА	ДА	ДА	НЕТ
default	ДА	ДА	НЕТ	НЕТ
private	ДА	НЕТ	НЕТ	НЕТ

52. К каким методам и полям базового класса производный класс имеет доступ (даже если базовый класс находится в другом пакете), а каким нет? Область видимости полей и данных из производного класса

Если вы создаёте класс, который в дальнейшем планируете использовать, как базовый, то объявляйте в нём поле **protected** вместо **private**. Иначе объекты производного класса не смогут обращаться к элементам базового.

Важной особенностью производного класса, является то, что хоть он и может использовать все методы и элементы полей **protected** и **public** базового класса, но он не может обратиться к конструктору с параметрами. Если конструкторы в производном классе не определены, при создании объекта сработает конструктор без аргументов базового класса. А если нам надо сразу при создании объекта производного класса внести данные, то для него необходимо определить свои конструкторы. В нашем примере показано, как же мы всё-таки можем использовать уже готовые конструкторы базового класса, чтобы не набирать код конструкторов снова. Для этого при определении конструктора производного класса после его имени следует поставить оператор **:** и имя конструктора базового класса, который необходимо вызвать, при создании объекта производного класса — **SecondClass(): FirstClass() {}**. Тело конструктора оставляем пустым т.к. всю работу проделает конструктор базового класса. В случае конструктора с параметром, этот параметр мы передаем в конструктор с параметром базового класса **SecondClass(int inputS) : FirstClass (inputS){}**

В main-функции создаем объекты базового и производного классов — **FirstClass F_object(3);** и **SecondClass S_object(4);** и отображаем их значения **value** на экран. Объект производного класса без проблем обращается к методу **show_value()** базового класса. Так, будто это его собственный метод. Ниже вызываем метод, который возводит значения **value** производного класса в квадрат. И выводим это изменённое значение на экран. А вот если мы захотим вызвать этот метод — **F_object.ValueSqr();** — для объекта базового класса,

компилятор нам этого не позволит сделать и выдаст ошибку. Это еще одна важная особенность — производный класс имеет доступ к базовому классу, а базовый класс, даже «не знает» о существовании производного и не может пользоваться его кодом.

53.Объявление и инициализация переменных типа String

Тип String не является примитивным типом данных, однако это один из наиболее используемых типов в Java.

Тип String — означает, что переменная хранит строковый тип данных (в нее мы записываем текст).

```
String someText = "hi!"; //объявили и инициализировали переменную "someText" со значением "hi!"
```

Обратите внимание: когда мы инициализируем переменную числового типа - в качестве значения переменной мы указываем просто число. Когда мы инициализируем переменную строкового типа - мы пишем текст в двойных кавычках: "text". В Java строковые (String) данные всегда выделяются двойными кавычками, и только отдельные символы (char) - одинарными 'O'.

Еще один пример:

```
String s = "One more time"; //строковые данные в двойных кавычках
```

```
char sign = 'x'; //переменная "sign" может хранить исключительно символы; сейчас она хранит символ "x"
```

54.Операция конкатенации строк и ее использование

Конкатенация — присоединение второй строки к концу первой. Используется для слияния двух строк. В Джаве обозначается символом +

55. Что означает утверждение, что объект класса String является неизменяемым

Безопасность и String pool основные причины неизменяемости String в Java.

Безопасность объекта неизменяемого класса String обусловлена такими фактами:

- вы можете передавать строку между потоками и не беспокоиться что она будет изменена
- нет проблем с синхронизацией (не нужно синхронизировать операции со String)
- отсутствие утечек памяти

- в Java строки используются для передачи параметров для авторизации, открытия файлов и т.д. - неизменяемость позволяет избежать проблем с доступом
 - возможность кэшировать hash code
- String pool позволяет экономить память и не создавать новые объекты для каждой повторяющейся строки. В случае с изменяемыми строками - изменение одной приводило бы к изменению всех строк одинакового содержания.

57.Объявление и инициализация массива строк. Организация просмотра элементов массива



1. Понятие массива строк. Общая форма объявления одномерного массива строк

Любая строка в Java имеет тип `String`. Одномерный массив строк имеет тип `String[]`.

Двумерный массив строк имеет тип `String[][]`.

Общая форма объявления и выделение памяти для одномерного массива строк

`String[] arrayName = new String[size];` //где

- `String` – встроенный в Java класс, который реализует строку символов. Объект типа `String` поддерживает большой набор операций;
- `arrayName` – имя объекта (экземпляра) типа `String`. Фактически, `arrayName` есть ссылкой на объект типа `String`;
- `size` – размер массива (количество строк, количество элементов типа `String`).

Объявление одномерного массива строк и выделение памяти для него можно реализовать и по другому

`String[] arrayName;`

`arrayName = new String[size];`

2. Каким образом объявляется одномерный массив строк? Пример

Ниже приведен пример объявления и использования одномерного массива строк.

// объявление одномерного массива строк

`String[] array = new String[5];`

// заполнение начальными значениями

`array[0] = "abcd";`

`array[1] = "Hello";`

`array[2] = "";` // пустая строка

`array[3] = "bestprog";`

`array[4] = ";\+=";` // комбинация `"\"` заменяется на `"\"`

// использование в выражениях

```
arrayS[4] = arrayS[1] + " " + arrayS[3]; // arrayS[4] = "Hello bestprog"  
arrayS[4] += ".net"; // arrayS[4] = "Hello bestprog.net"
```

3. Двумерный массив строк. Общая форма

Общая форма объявления двумерного массива строк следующая:

```
String[][] matrName = new String[n][m];
```

где

- **matrName** – имя объекта (ссылка на объект), который есть двумерным массивом типа **String**;
- **n** – количество строк в массиве **matrName**;
- **m** – количество столбцов в массиве **matrName**.

Возможен также другой способ объявления и выделения памяти для двумерного массива строк:

```
String[][] matrName; // объявление ссылки на двумерный массив строк  
matrName = new String[n][m];
```

4. Пример объявления и использования двумерного массива строк

```
// объявление двумерного массива строк
```

```
String[][] matr = new String[2][3];
```

```
// заполнение массива значениями
```

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

```
    for (int j=0; j<matrS[i].length; j++)
```

```
        matrS[i][j] = "matrS[" + i + "][" + j + "];
```

```
// проверка
```

```
String s;
```

```
s = matrS[0][0]; // s = "matrS[0][0]"
```

```
s = matrS[1][1]; // s = "matrS[1][1]"
```

5. Как определяется длина массива строк? Свойство **length. Пример**

Чтобы определить количество строк в массиве используется свойство **length**.

Для одномерных массивов количество строк **n** определяется следующим образом:

```
String[] arrayS = new String[25];
```

```
int n;
```

```
n = array.length;
```

Для двумерных массивов количество строк и столбцов определяется следующим образом

```
// matr - двумерный массив строк
```

```
String[][] matrS = new String[2][3];
```

```
int n, m;
```

```
n = matr.length; // n = 2 - количество строк
```

```
m = matr[0].length; // m = 3 - количество столбцов
```

```
m = matr[1].length; // m = 3
```

6. Как осуществляется инициализация одномерного массива? Пример

Инициализация одномерного массива строк точно такая же как инициализация одномерного массива любого другого типа.

```
// инициализация одномерного массива строк
```

```
String[] M = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};
```

```
String s;
```

```
s = M[2]; // s = "Tuesday"
```

```
s = M[4]; // s = "Thursday"
```

7. Поиск заданной строки в одномерном массиве строк. Пример

```
// поиск заданной строки в массиве строк
```

```
// объявление массива строк
```

```
String M[] = new String[5];
```

```
String s = "May"; // строка, которую нужно найти
```

```
boolean f_is;
```

```
// заполнение массива значениями
```

```
M[0] = "January";
```

```
M[1] = "February";
```

```
M[2] = "May";
```

```
M[3] = "October";
```

```
M[4] = "December";
```

```
// поиск строки
```

```
f_is = false;
```

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

```
    if (M[i]==s) {
```

```
        f_is = true;
```

```
        break;
```

```
    }
```

```
// вывод результата
```

```
if (f_is)
```

```
    System.out.println("Искомая строка есть в массиве.");
```

```
else
```

```
    System.out.println("Искомой строки нет в массиве.");
```

8. Сортировка одномерного массива строк по алфавиту методом вставки. Пример

Для сравнения двух строк в лексикографическом порядке в классе String разработан метод `compareTo()`. Общая форма метода следующая:

```
int compareTo(вторая_строка)
```

Метод возвращает

- **<0**, если вторая строка следует после первой строки в лексикографическом порядке;
- **=0**, если строки одинаковы;
- **>0**, если вторая строка следует перед первой в лексикографическом порядке.

Фрагмент, который демонстрирует сортировку массива строк методом вставки:

```
// сортировка массива строк методом вставки
```

```
String[] M = {"abc", "bde", "fgh", "abcd", "bcdef", "cdef", "fghij", "aaa"};
```

```
String s;
```

```
// сортировка
```

```
for (int i=0; i<M.length-1; i++)
```

```

for (int j=i; j>=0; j--)
    if (M[j].compareTo(M[j+1])>0) {
        // обменять M[j] и M[j+1] местами
        s = M[j];
        M[j] = M[j+1];
        M[j+1] = s;
    }

```

// вывод результата

```

for (int i=0; i<M.length; i++)
    System.out.println(M[i]);

```

В результате выполнения вышеприведенного кода, на экран будет выведено следующее

```

aaa
abc
abcd
bcdef
bde
cdef
fgh
fghij

```

9. Как осуществляется инициализация двумерного массива строк? Пример

Инициализация двумерного массива строк ничем не отличается от инициализации двумерного массива любого примитивного типа. Элементами массива есть обычные строки.

Ниже приведен пример инициализации двумерного массива строк с именем M

// объявление массива M с начальной инициализацией

```

String M[][] = new {
    { "a1", "a2", "a3" },
    { "b1", "b2", "b3" },
    { "a1", "c2", "a1" }
};

```

// проверка

```

String s;
s = M[0][1]; // s = "a2"
s = M[1][0]; // s = "b1"

```

10. Пример подсчета количества вхождений заданной строки в двумерном массиве строк

// вычисление количества вхождений заданной строки в двумерном массиве

// объявление массива M с начальной инициализацией

```

String M[][] = {
    { "abcd", "abc", "bcd" },
    { "acd", "bcd", "abcd" },
    { "abc", "bc", "cde" }
}

```

```

};
String s = "abc"; // строка, количество вхождений которой нужно вычислить
int k = 0; // количество вхождений, результат
for (int i=0; i<M.length; i++)
    for (int j=0; j<M[i].length; j++)
        if (M[i][j]==s)
            k++; // k = 2

```

11. Пример замены строки в двумерном массиве строк

Заданы:

- двумерный массив строк с именем `matr`;
- строка `s1`, которая ищется для замены;
- строка `s2`, которая заменяет строку `s1`.

Разработать программу, которая заменяет строку `s1` в матрице `matr` новой строкой `s2`.

Фрагмент кода, который решает данную задачу:

```

// объявление двумерного массива строк
String[][] matr = new String[2][3];
// заполнение матрицы matr произвольными значениями
matrS[0][0] = "abc";
matrS[0][1] = "cba";
matrS[0][2] = "def";
matrS[1][0] = "abc";
matrS[1][1] = "fff";
matrS[1][2] = "qqq";
// заполнение значениями строк s1 и s2
String s1 = "abc"; // заменяемая строка
String s2 = "mmm"; // заменяющая строка
// цикл вычисления
for (int i=0; i<matrS.length; i++)
    for (int j=0; j<matrS[i].length; j++)
        if (matrS[i][j]==s1)
            matr[i][j] = s2;

// вывод результата
for (int i=0; i<matrS.length; i++) {
    for (int j=0; j<matrS[i].length; j++)
        System.out.print(matrS[i][j] + " ");
    System.out.println();
}

```

В результате выполнения вышеприведенного кода, на экран будет выведен следующий результат:

```

mmm cba def
mmm fff qq

```

58. Понятие и объявление интерфейсов в Джава

Интерфейс — это ссылочный тип в Java. Он схож с классом. Это совокупность абстрактных методов. Класс реализует интерфейс, таким образом наследуя абстрактные методы интерфейса.

Вместе с абстрактными методами интерфейс в Java может содержать константы, обычные методы, статические методы и вложенные типы. Тела методов существуют только для обычных методов и статических методов.

Далее разберём зачем нужны интерфейсы в Java и для чего используются, разницу абстрактного класса и интерфейса.

Реализация Интерфейса

Ключевое слово `interface` используется для объявления интерфейса. Вот пример того, как можно создать интерфейс:

Пример 1

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Любое количество запросов импорта

public interface NameOfInterface { //создание интерфейса
    // Любое количество полей final и static
    // Любое количество объявлений абстрактных методов
}
```

Интерфейсы имеют следующие свойства:

- Интерфейс абстрактный косвенно. Вам не нужно использовать ключевое слово `abstract` во время объявления интерфейса.
- Каждый метод в интерфейсе косвенно абстрактным, поэтому ключевое слово `abstract` не нужно.
- Методы в интерфейсе косвенно публичны.

Пример 2

```
/* File name : Animal.java */
interface Animal {
    public void eat();
    public void travel();
}
```

59. Может ли один класс реализовывать несколько интерфейсов?

Да.

Класс Java может распространять только один родительский класс. Множественное наследование не допускается. Однако интерфейсы не являются классами, а интерфейс может расширять несколько родительских интерфейсов.

60. Что входит в состав интерфейса. (какие компоненты может содержать интерфейс)?

Хз, что сюда писать, ибо по этой формулировке ничего нормального не нашёл, максимум – такое:

Интерфейсы состоят из абстрактных методов и конечных переменных.

61. Может ли интерфейс наследоваться от другого интерфейса?

Да.

Интерфейс может расширяться от одного или нескольких интерфейсов. Класс может расширять только один класс, но реализовывать любое количество интерфейсов

Интерфейс не может реализовать другой интерфейс. Он должен расширить другой интерфейс, если это необходимо.

62. Интерфейс Comparable, назначение, его методы и использование в Джава

Интерфейс Comparable

В интерфейсе **Comparable** объявлен только один метод **compareTo** (Object obj), предназначенный для упорядочивания объектов класса. Данный метод удобно использовать для сортировки списков или массивов объектов.

Метод **compareTo** (Object obj) сравнивает вызываемый объект с obj. В отличие от метода **equals**, который возвращает true или false, **compareTo** возвращает:

- 0, если значения равны;
- Отрицательное значение (обычно -1), если вызываемый объект меньше obj;
- Положительное значение (обычно +1), если вызываемый объект больше obj.

Если типы объектов не совместимы при сравнении, то **compareTo** (Object obj) может вызвать исключение **ClassCastException**. Необходимо помнить, что аргумент метода **compareTo** имеет тип сравниваемого объекта класса.

Обычные классы Byte, Short, Integer, Long, Double, Float, Character, String уже реализуют интерфейс **Comparable**.

Пример реализации интерфейса Comparable

```
package test;

import java.util.TreeSet;

class Compare implements Comparable<Object>
{
    String str;
    int num;
    String TEMPLATE = "num = %d, str = '%s'";

    Compare(String str, int num)
    {
        this.str = str;
        this.num = num;
    }

    @Override
    public int compareTo(Object obj)
    {
        Compare entry = (Compare) obj;
        int result = str.compareTo(entry.str);
        if(result != 0)
            return result;

        result = num - entry.num;
        if(result != 0)
            return (int) result / Math.abs( result );

        return 0;
    }

    @Override
    public String toString()
    {
        return String.format(TEMPLATE, num, str);
    }
}

public class Example
{
    public static void main(String[] args)
    {
        TreeSet<Compare> data = new TreeSet<Compare>();
        data.add(new Compare("Начальная школа" , 234));
        data.add(new Compare("Начальная школа" , 132));
        data.add(new Compare("Средняя школа" , 357));
        data.add(new Compare("Высшая школа" , 246));
        data.add(new Compare("Музыкальная школа", 789));
        for (Compare e : data)
            System.out.println(e.toString());
    }
}
```

Результат выполнения программы:


```
num = 246, str = 'Высшая школа'
num = 789, str = 'Музыкальная школа'
num = 132, str = 'Начальная школа'
num = 234, str = 'Начальная школа'
num = 357, str = 'Средняя школа'
```

В примере значения сортируются сначала по полю `str` (по алфавиту), а затем по `num` в методе `compareTo`. Это хорошо видно по двум строкам с одинаковыми значениями `str` и различными `num`. Чтобы изменить порядок сортировки значения `str` (в обратном порядке), необходимо внести небольшие изменения в метод `compareTo`.

```
@Override
public int compareTo(Object obj)
{
    int result = ((Comp)obj).str.compareTo(str);
    if(result != 0)
        return result;

    result = entry.number - number;

    if(result != 0) {
        return (int) result / Math.abs(result);
    }
    return 0;
}
```

63. Какое значение возвращает вызов метода `object1.compareTo(object2)`, который сравнивает 2 объекта `obj1` и `obj2` в зависимости от объектов?

Булево

64. Интерфейсные ссылки и их использование в Джава

Ранее неоднократно отмечалось, что в Java запрещено множественное наследование. Причина отказа от множественного наследования связана с теми потенциальными проблемами, которые могут при этом возникать. Однако множественное наследование открывает широкие перспективы для составления эффективных программных кодов и значительно повышает гибкость программ. Выход был найден в использовании интерфейсов.

Интерфейсы во многом напоминают классы. Принципиально от класса интерфейс отличается тем, что содержит только сигнатуры методов без описания, а также поля-константы (поля, значения которых постоянны и не могут изменяться).

Описание интерфейса аналогично описанию класса, только ключевое слово `class` необходимо заменить ключевым словом `interface`. Как отмечалось, для методов интерфейса указываются только сигнатуры. Описываемые в интерфейсе поля по умолчанию считаются неизменяемыми (как если бы они были описаны с ключевым словом `final`) и статическими (то есть `static`). Таким образом, поля интерфейса играют роль глобальных констант. Практическое использование интерфейса подразумевает его

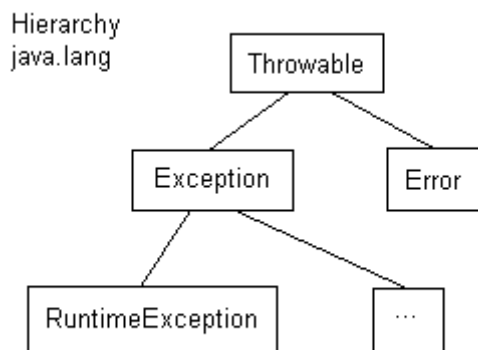
реализацию. Эта процедура напоминает наследование абстрактных классов. Реализуется интерфейс в классе. Класс, который реализует интерфейс, должен содержать описание всех методов интерфейса. Методы интерфейса при реализации описываются как открытые. Один и тот же класс может реализовать одновременно несколько интерфейсов, равно как один и тот же интерфейс может реализовываться несколькими классами.

При создании объектов класса в качестве типа объектной переменной может указываться имя реализованного в классе интерфейса. Другими словами, если класс реализует интерфейс, то ссылку на объект этого класса можно присвоить интерфейсной переменной — переменной, в качестве типа которой указано имя соответствующего интерфейса. Ситуация очень напоминает ту, что рассматривалась в предыдущей главе при наследовании, когда объектная переменная суперкласса ссылалась на объект подкласса. Как и в случае с объектными ссылками суперкласса, через интерфейсную ссылку можно сослаться не на все члены объекта реализующего интерфейс класса. Доступны только те методы, которые объявлены в соответствующем интерфейсе. С учетом того, что класс может реализовать несколько интерфейсов, а один и тот же интерфейс может быть реализован в разных классах, ситуация представляется достаточно пикантной.

65. Понятие исключительной ситуации и ее обработка

В языке Java исключения (Exceptions) и ошибки (Errors) являются объектами. Когда метод вызывает, еще говорят "бросает" от слова "throws", исключительную ситуацию, он на самом деле работает с объектом. Но такое происходит не с любыми объектами, а только с теми, которые наследуются от Throwable.

Упрощенную диаграмму классов ошибок и исключительный вы можете увидеть на следующем рисунке:



RuntimeException, Error и их наследников еще называют **unchecked exception**, а всех остальных наследников класса Exception - **checked exception**.

Checked Exception обязывает пользователя обработать ее (используя конструкцию try-catch) или же отдать на откуп обрамляющим методам, в таком случае к декларации метода, который бросает проверяемое (checked) исключение, дописывают конструкцию **throws**, например:

```
public Date parse(String source) throws ParseException { ... }
```

К unchecked исключениям относятся, например, `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ClassCastException` и так далее. Это те ошибки, которые могут возникнуть практически в любом методе. Несомненно, описывать каждый метод как тот, который бросает все эти исключения, было бы глупо.

Обрабатывать ошибку лучше там, где она возникла. Если в данном фрагменте кода нет возможности принять решение, что делать с исключением, его нужно бросать дальше, пока не найдется нужный обработчик, либо поток выполнения программы не вылетит совсем.

Вы наверняка знаете, что обработка исключений происходит с помощью блока `try-catch-finally`. Сразу скажу вам такую вещь: **никогда не используйте пустой catch блок!**. Выглядит этот ужас так:

```
try { ... }  
catch(Exception e) { }
```

Если Вы уверены, что исключения в блоке `try` не возникнет никогда, напишите комментарий, как например в этом фрагменте кода:

```
StringReader reader = new StringReader("qwerty");  
try { reader.read(); }  
catch (IOException e) { /* cannot happen */ }
```

Если же исключение в принципе может возникнуть, но только действительно в "исключительной ситуации" когда с ним ничего уже сделать будет нельзя, лучше оберните его в `RuntimeException` и пробросьте наверх, например:

```
String siteUrl = ...;  
...  
URL url;  
try { url = new URL(siteUrl); }  
catch (MalformedURLException e) {  
    throw new RuntimeException(e);  
}
```

Скорее всего ошибка здесь может возникнуть только при неправильной конфигурации приложения, например, `siteUrl` читается из конфигурационного файла и кто-то допустил опечатку при указании адреса сайта. Без исправления конфига и перезапуска приложения тут ничего поделать нельзя, так что `RuntimeException` вполне оправдан.

66. В каком случае программа должна использовать оператор `throw`?

В языке C++ оператор `throw` используется для сигнализирования о возникновении исключения или ошибки (аналогия тому, когда свистит арбитр). Сигнализирование о том, что произошло исключение, называется генерацией исключения (или «выбрасыванием

исключения»). Для использования оператора throw применяется ключевое слово throw, а за ним указывается значение любого типа данных, которое вы хотите задействовать, чтобы сигнализировать об ошибке. Как правило, этим значением является код ошибки, описание проблемы или настраиваемый класс-исключение.

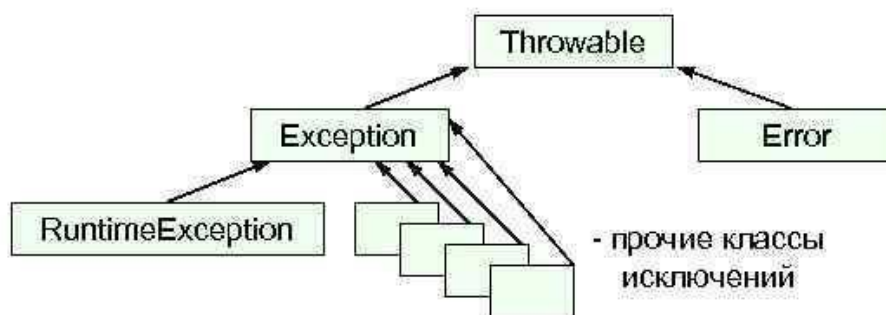
67. В Java все исключения делятся на два основных типа. Что это за типы и какие виды ошибок ни обрабатывают?



4

Классы исключений

- Исключения в Java являются объектами с собственной иерархией классов:



- Класс **Error** включает серьезные ошибки выполнения, которые должны приводить к завершению программы и, как правило, не должны перехватываться.
- Класс **RuntimeException** – менее серьезные ошибки (например, деление на ноль), которые можно перехватить и обработать.
- Эти два класса (и их потомки) называются *непроверяемыми* исключениями (невозможно предсказать, в каких методах они могут возникнуть). Остальные классы являются *проверяемыми*, и программист обязан указать в объявлении метода, какие прерывания в нем могут возникнуть.



САМЫЕ РАСПРОСТРАНЕННЫЕ НЕКОНТРОЛИРУЕМЫЕ ИСКЛЮЧЕНИЯ JAVA

Тип исключения	С чем связана ошибка
ArithmeticException	Выполнение арифметических операций (например, деление на ноль)
ArrayIndexOutOfBoundsException	Индекс массива выходит за допустимые границы
ArrayStoreException	Присвоение элементу массива значения несовместимого типа
ClassCastException	Попытка выполнить приведение несовместимых типов
IllegalArgumentException	Методу указан неправильный аргумент
IllegalMonitorStateException	Работа с монитором (относится к многопоточному программированию)
IllegalStateException	Ресурс находится в некорректном состоянии
IllegalThreadStateException	Предпринята попытка выполнить некорректную операцию на потоке
IndexOutOfBoundsException	Индекс выходит за допустимый диапазон значений
NegativeArraySizeException	Попытка создать массив отрицательного размера
NullPointerException	Некорректное использование ссылок (обычно когда объектная переменная содержит пустую ссылку)
NumberFormatException	Преобразование строки к числовому значению (когда в число преобразуется строка, содержащая некорректное текстовое представление числа)
SecurityException	Попытка нарушить режим защиты
StringIndexOutOfBoundsException	Неверное индексирование при работе с текстовой строкой
UnsupportedOperationException	Попытка выполнить некорректную операцию

КОНТРОЛИРУЕМЫЕ ИСКЛЮЧЕНИЯ JAVA

Тип исключения	С чем связана ошибка
ClassNotFoundException	Невозможно найти нужный класс
CloneNotSupportedException	Некорректная попытка клонировать объект
IllegalAccessException	Ошибка доступа к ресурсу
InstantiationException	Попытка создать объект абстрактного класса или интерфейса
InterruptedException	Прерывание одного потока другим
NoSuchFieldException	Отсутствует нужное поле
NoSuchMethodException	Отсутствует нужный метод

68.Код ниже вызовет ошибку: **Exception <...>**

java.lang.ArrayIndexOutOfBoundsException: 4: Что она означает?

Это исключение, появляющееся во время выполнения. Оно возникает тогда, когда мы пытаемся обратиться к элементу массива по отрицательному или превышающему размер массива индексу.

69.Контролируемые исключения (checked) и 70.Неконтролируемые исключения (unchecked) и ошибки, которые они обрабатывают

Все исключительные ситуации делятся на «проверяемые» (checked) и «непроверяемые» (unchecked) (смотрите картинку в начале статьи). Это свойство присуще «корневищу» (Throwable, Error, Exception, RuntimeException) и передается по наследству. Никак не видимо в исходном коде класса исключения. В дальнейших примерах просто учтите, что — Throwable и Exception и все их наследники (за исключением наследников Error-а и RuntimeException-а) — checked — Error и RuntimeException и все их наследники — unchecked checked exception = проверяемое исключение, проверяемое компилятором.

1. Checked исключения, это те, которые должны обрабатываться блоком catch или описываться в сигнатуре метода. Unchecked могут не обрабатываться и не быть описанными.

2. Unchecked исключения в Java – наследованные от RuntimeException, checked – от Exception (не включая unchecked).

Checked исключения отличаются от Unchecked исключения в Java, тем что:

1)Наличие\обработка Checked исключения проверяются на этапе компиляции.
Наличие\обработка Unchecked исключения происходит на этапе выполнения.

71.Как реализуется принципы ООП в Java при создании исключений?

Java является объектно-ориентированным языком. Это означает, что писать программы на Java нужно с применением объектно-ориентированного стиля. И стиль этот основан на использовании в программе объектов и классов.

Основные принципы ООП:

- [Абстракция](#)
- [Инкапсуляция](#)
- [Наследование](#)
- [Полиморфизм](#)

Абстракцию в ООП можно также определить, как способ представления элементов задачи из реального мира в виде объектов в программе. Абстракция всегда связана с обобщением некоторой информации о свойствах предметов или объектов, поэтому главное — это отделить значимую информацию от незначимой в контексте решаемой задачи. При этом уровней абстракции может быть несколько.

```
public abstract class AbstractPhone {  
    private int year;  
  
    public AbstractPhone(int year) {
```

```

        this.year = year;
    }
    public abstract void call(int outputNumber);
    public abstract void ring (int inputNumber);
}

```

Для исключения подобного вмешательства в конструкцию и работу объекта в ООП используют принцип инкапсуляции – еще один базовый принцип ООП, при котором атрибуты и поведение объекта объединяются в одном классе, внутренняя реализация объекта скрывается от пользователя, а для работы с объектом предоставляется открытый интерфейс.

```

public class SomePhone {

    private int year;
    private String company;
    public SomePhone(int year, String company) {
        this.year = year;
        this.company = company;
    }
    private void openConnection(){
        //findComutator
        //openNewConnection...
    }
    public void call() {
        openConnection();
        System.out.println("Вызываю номер");
    }

    public void ring() {
        System.out.println("Дзынь-дзынь");
    }

}

```

В программировании наследование заключается в использовании уже существующих классов для описания новых.

```

public class Smartphone extends CellPhone {

    private String operationSystem;

    public Smartphone(int year, int hour, String operationSystem) {
        super(year, hour);
        this.operationSystem = operationSystem;
    }
    public void install(String program){
        System.out.println("Устанавливаю " + program + " для " + operationSystem);
    }

}

```


Принцип в ООП, когда программа может использовать объекты с одинаковым интерфейсом без информации о внутреннем устройстве объекта, называется **полиморфизмом**.

```
AbstractPhone firstPhone = new ThomasEdisonPhone(1879);
AbstractPhone phone = new Phone(1984);
AbstractPhone videoPhone=new VideoPhone(2018);
User user = new User("Андрей");
user.callAnotherUser(224466,firstPhone);
// Вращайте ручку
//Сообщите номер абонента, сэр
user.callAnotherUser(224466,phone);
//Вызываю номер 224466
user.callAnotherUser(224466,videoPhone);
//Подключаю видеоканал для абонента 224466
```

72.Какой оператор позволяет принудительно выбросить исключение?

Оператор throw используется для возбуждения исключения «вручную». Для того, чтобы сделать это, нужно иметь объект подкласса класса Throwable, который можно либо получить как параметр оператора catch, либо создать с помощью оператора new. Ниже приведена общая форма оператора throw: throw [ОбъектТипаThrowable];

При достижении этого оператора нормальное выполнение кода немедленно прекращается, так что следующий за ним оператор не выполняется. Ближайший окружающий блок try проверяется на наличие соответствующего возбужденному исключению обработчика catch. Если такой отыщется, управление передается ему. Если нет, проверяется следующий из вложенных операторов try, и так до тех пор пока либо не будет найден подходящий раздел catch, либо обработчик исключений исполняющий системы Java не остановит программу, выведя при этом состояние стека вызовов.

Только подклассы класса Throwable могут быть возбуждены или перехвачены. Простые типы (int, char и т.п.), а также классы, не являющиеся подклассами Throwable (например, String и Object) использоваться в качестве исключений

не могут. Наиболее общий путь для использования исключений — создание своих собственных подклассов класса Exception.

При создании собственных классов исключений следует принимать во внимание следующие аспекты: - Все исключения должны быть дочерними элементами Throwable. - Если планируете создать контролируемое исключение, следует расширить класс Exception. - Если хотите создать исключение на этапе выполнения, следует расширить класс RuntimeException.

73.Порядок выполнения операторов при обработке блока try...catch?

Конструкция try..catch состоит из двух основных блоков: try, и затем catch:

Работает она так:

1. Выполняется код внутри блока try.
2. Если в нём ошибок нет, то блок catch(err) игнорируется, то есть выполнение доходит до конца try и потом прыгает через catch.
3. Если в нём возникнет ошибка, то выполнение try на ней прерывается, и управление прыгает в начало блока catch(err).

При этом переменная err (можно выбрать и другое название) будет содержать объект ошибки с подробной информацией о произошедшем.

Таким образом, при ошибке в try скрипт не «падает», и мы получаем возможность обработать ошибку внутри catch.

74. Абстрактный тип данных Stack (стек) в Java?

Класс Stack в Java

Класс стека в Java является частью платформы Collection, которая упрощает различные операции, такие как push, pop и т. д.

Что такое класс Stack в Java?

Класс Stack в Java — это структура данных, которая следует за LIFO (Last In First Out). Java Stack Class подпадает под базовую платформу Collection Hierarchy Framework, в которой вы можете выполнять базовые операции, такие как push, pop и т. д.

75. Универсальные типы или обобщенные типы данных, для чего создаются?

Обобщённые методы или дженерики позволяют работать с различными типами данных без изменения их описания, одно из назначений этих методов — более сильная проверка типов во время компиляции и устранение необходимости явного приведения. Создаётся более безопасный и легко читаемый код, который не перегружен переменными типа Object и приведением классов.

76. Объявление обобщённого класса коллекции с параметризованным методом для обработки массива элементов коллекции на основе цикла for each (определение общего метода для отображения элементов массива)

Пример использования цикла for-each с коллекциями

Создадим коллекцию из имён и выведем все имена на экран.

```
1 List<String> names = new ArrayList<>();
2     names.add("Snoopy");
3     names.add("Charlie");
4     names.add("Linus");
5     names.add("Shroeder");
6     names.add("Woodstock");
7
8     for(String name : names){
9         System.out.println(name);
10    }
```

77. Что представляет из себя класс ArrayList и в каком случае используется?

При разработке часто бывает сложно предсказать, какого размера понадобятся массивы. Поэтому функция динамического выделения памяти во время работы программы необходима каждому языку программирования. Динамическим называется массив, размер которого может измениться во время исполнения программы. В Java для такой цели существует класс ArrayList.

Что такое класс ArrayList?

ArrayList — реализация изменяемого массива интерфейса List, часть Collection Framework, который отвечает за список (или динамический массив), расположенный в пакете java.util. Этот класс реализует все необязательные операции со списком и предоставляет методы управления размером массива, который используется для хранения списка. В основе ArrayList лежит идея динамического массива. А именно, возможность добавлять и удалять элементы, при этом будет увеличиваться или уменьшаться по мере необходимости.

Что хранит ArrayList?

Только ссылочные типы, любые объекты, включая сторонние классы. Строки, потоки вывода, другие коллекции. Для хранения примитивных типов данных используются классы-обертки.

78. Класс Pattern и его использование

Экземпляр класса `Pattern` представляет собой скомпилированное регулярное выражение, известное также как шаблон.

Класс `Java Pattern` можно использовать двумя способами.

1. через метод `Pattern.matches()`, чтобы быстро проверить, соответствует ли текст (`String`) заданному выражению;
2. скомпилировать экземпляр `Pattern`, используя `Pattern.compile()`, который можно использовать несколько раз, чтобы сопоставить выражение с несколькими текстами.

79. Класс `Math` и его использование

Для выполнения различных математических операций в `Java` в пакете `java.lang` определен класс `Math`. Рассмотрим его основные методы:

Методы реализованы как `static`, поэтому можно сразу вызывать через `Math.methodName()` без создания экземпляра класса.

В классе определены две константы типа `double`: `E` и `PI`.

Популярные методы для тригонометрических функций принимают параметр типа `double`, выражающий угол в радианах.

- `sin(double d)`
- `cos(double d)`
- `tan(double d)`
- `asin(double d)`
- `acos(double d)`
- `atan(double d)`
- `atan2(double y, double x)`

Существуют также гиперболические функции: `sinh()`, `cosh()`, `tanh()`.

Экспоненциальные функции: `cbrt()`, `exp()`, `expm1()`, `log()`, `log10()`, `log1p()`, `pow()`, `scalb()`, `sqrt()`.

Из них хорошо знакомы возведение в степень - `pow(2.0, 3.0)` вернёт `8.0`.

Также популярен метод для извлечения квадратного корня - `sqrt(4.0)`. Если аргумент меньше нуля, то возвращается `NaN`. Похожий метод `cbrt()` извлекает кубический корень.

Если аргумент отрицательный, то и возвращаемое значение будет отрицательным: -27.0->-3.0.

Функции округления:

- `abs()` - возвращает абсолютное значение аргумента
- `ceil()` - возвращает наименьшее целое число, которое больше аргумента
- `floor()` - возвращает наибольшее целое число, которое меньше или равно аргументу
- `max()` - возвращает большее из двух чисел
- `min()` - возвращает меньшее из двух чисел
- `nextAfter()` - возвращает следующее значение после аргумента в заданном направлении
- `nextUp()` - возвращает следующее значение в положительном направлении
- `rint()` - возвращает ближайшее целое к аргументу
- `round()` - возвращает аргумент, округлённый вверх до ближайшего числа
- `ulp()` - возвращает дистанцию между значением и ближайшим большим значением

Другие методы

- `copySign()` - возвращает аргумент с тем же знаком, что у второго аргумента
- `getExponent()` - возвращает экспоненту
- `IEEEremainder()` - возвращает остаток от деления
- `hypot()` - возвращает длину гипотенузы
- `random()` - возвращает случайное число между 0 и 1 (единица в диапазон не входит)
- `signum()` - возвращает знак значения
- `toDegrees()` - преобразует радианы в градусы
- `toRadians()` - преобразует градусы в радианы

80.Как вызываются методы класса `Math` и что при этом происходит?

Вызов одного из методов `Math` на примере `Math.abs()`

```

1 public static void main(String[] args) {
2     System.out.println(Math.abs(-1));
3     // 1
4
5     System.out.println(Math.abs(-21.8d));
6     // 21.8
7
8     System.out.println(Math.abs(4532L));
9     // 4532
10
11    System.out.println(Math.abs(5.341f));
12    // 5.341
13
14    }

```

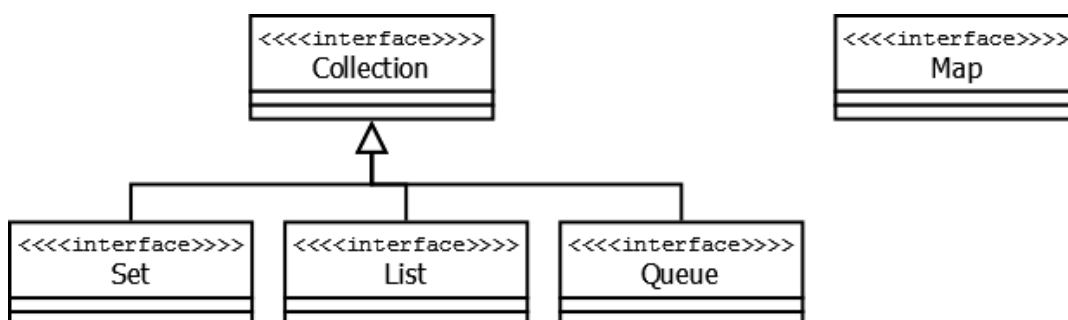
<https://metanit.com/java/tutorial/12.1.php> - все методы класса Math.

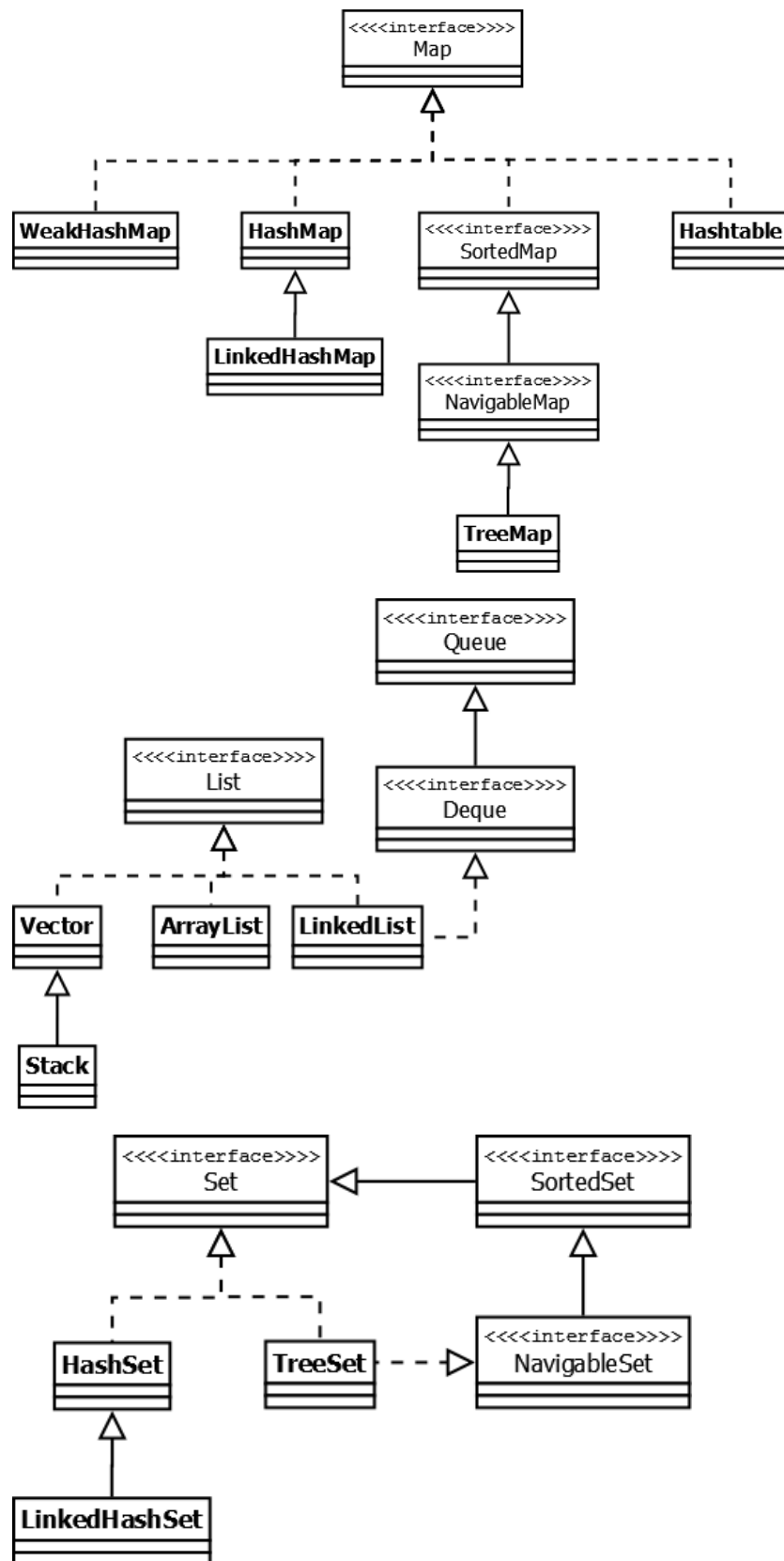
81. Структура коллекций в Java Collection Framework. Иерархия интерфейсов

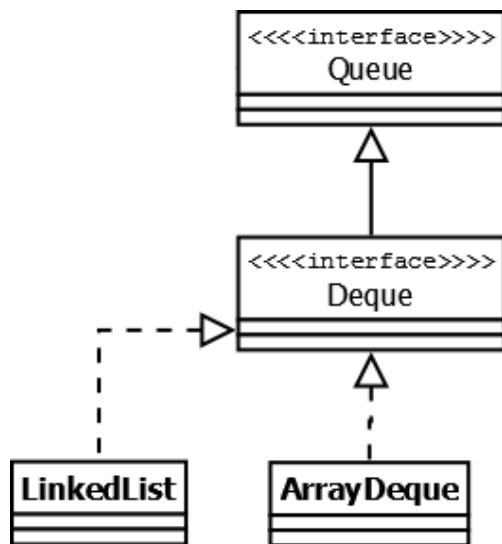
Java Collection Framework — иерархия интерфейсов и их реализаций, которая является частью JDK и позволяет разработчику пользоваться большим количеством структур данных из «коробки».

Базовые понятия

На вершине иерархии в Java Collection Framework располагаются 2 интерфейса: Collection и Map. Эти интерфейсы разделяют все коллекции, входящие во фреймворк на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ — значение» (словари).







<https://habr.com/ru/post/237043/>

82. Коллекция HashMap, создание и методы работы с ней

Хеш-таблицей называется структура данных, реализующая интерфейс ассоциативного массива (абстрактная модель «ключ – значение» или entry), которая обеспечивает очень быструю вставку и поиск: независимо от количества элементов вставка и поиск (а иногда и удаление) выполняются за время, близкое к константе – $O(1)$.

Конструкторы класса:

1. `public HashMap()` — создает хеш-отображение по умолчанию: объемом (capacity) = 16 и с коэффициентом загрузки (load factor) = 0.75;
2. `public HashMap(Map< ? extends K, ? extends V> m)` — создает хеш-отображение, инициализируемое элементами другого заданного отображения с той начальной емкостью, которой хватит вместить в себя элементы другого отображения;
3. `public HashMap(int initialCapacity)` — создает хеш-отображение с заданной начальной емкостью. Для корректной и правильной работы HashMap размер внутреннего массива обязательно должен быть степенью двойки (т.е. 16, 64, 128 и т.д.);
4. `public HashMap(int initialCapacity, float loadFactor)` — создает хеш-отображение с заданными параметрами: начальной емкостью и коэффициентом загрузки.

Методы:

- `put(key, value)` – добавление новой пары
- `containsKey(key)` – проверка на наличие введенного ключа

- `containsValue(value)` – проверка на наличие введенного значения
- `keySet()` – получение списка всех ключей
- `values()` – получение списка всех значений
- `size()` – получение количества пар ключ-значение
- `clear()` – удаление всех элементов
- `isEmpty()` – проверка на наличие хотя бы одной пары ключ-значение
- `putAll(secondMap)` – объединение двух объектов класса `HashMap` (второй объект дописывается в конец первого)
- `entrySet()` – получение всех пар ключ-значение

Интерфейс `Map.Entry` обозначает пару “ключ-значение” внутри словаря.

83. Чем является класс `LinkedList`

Представляет структуру данных в виде связанного списка. Он наследуется от класса `AbstractSequentialList` и реализует интерфейсы `List`, `Deque` и `Queue`. То есть он соединяет функциональность работы со списком и функциональность очереди.

84. Одним из ключевых методов интерфейса `Collection` является метод `Iterator iterator()`. Что возвращает этот метод?

Одним из ключевых методов интерфейса `Collection` является метод `Iterator<E> iterator()`. Он возвращает итератор - то есть объект, реализующий интерфейс `Iterator`.

85. Что возвращает метод `next()`

Метод `next()` возвращает объект с двумя свойствами `done` и `value`. Также вы можете задать параметр для метода `next`, чтобы отправить значение в генератор.

Синтаксис

```
gen.next(value)
```

Параметры

`value`

Значение, отправляемое в генератор. Значение будет установлено в виде результата выражения `yield`, т. е. в `[переменная] = yield [выражение]` значение, которое было передано в функцию `.next` будет присвоено `[переменной]`.

Возвращаемое значение

Object с двумя свойствами:

- `done` (boolean)
 - Имеет значение `true`, если итератор находится за окончанием итерируемой последовательности. В этом случае `value` может указывать *возвращаемое значение* итератора.
 - Имеет значение `false`, если итератор может создать следующее значение в последовательности. Это эквивалентно вообще не указанному свойству `done`.
- `value` - любое JavaScript значение, возвращаемое итератором. Может быть опущено, когда `done` имеет значение `true`.

86. Что возвращает метод `hasNext()`

`hasNext()` проверяет есть ли еще объекты, которые вы можете получить от итератора, и если есть, то он возвращает `true`. Если больше нет объектов, он вернет `false`.

87.Обобщенный класс `HashSet` класс коллекция, наследует свой функционал от класса `AbstractSet`, а также реализует интерфейс `Set`. Что он себя представляет?

Интерфейс `Set` расширяет интерфейс `Collection` и представляет набор уникальных элементов. `Set` не добавляет новых методов, только вносит изменения унаследованные. В частности, метод `add()` добавляет элемент в коллекцию и возвращает `true`, если в коллекции еще нет такого элемента.

Обобщенный класс `HashSet` представляет хеш-таблицу. Он наследует свой функционал от класса `AbstractSet`, а также реализует интерфейс `Set`.

Хеш-таблица представляет такую структуру данных, в которой все объекты имеют уникальный ключ или хеш-код. Данный ключ позволяет уникально идентифицировать объект в таблице.

Для создания объекта `HashSet` можно воспользоваться одним из следующих конструкторов:

- `HashSet()`: создает пустой список
- `HashSet(Collection<? extends E> col)`: создает хеш-таблицу, в которую добавляет все элементы коллекции `col`
- `HashSet(int capacity)`: параметр `capacity` указывает начальную емкость таблицы, которая по умолчанию равна 16

- `HashSet(int capacity, float koef)`: параметр `koef` или коэффициент заполнения, значение которого должно быть в пределах от 0.0 до 1.0, указывает, насколько должна быть заполнена емкость объектами прежде чем произойдет ее расширение. Например, коэффициент 0.75 указывает, что при заполнении емкости на 3/4 произойдет ее расширение.

Класс `HashSet` не добавляет новых методов, реализуя лишь те, что объявлены в родительских классах и применяемых интерфейсах:

88.Обобщенный класс `HashMap` класс коллекция, которая реализует интерфейс `Map` для хранения пар ключ-значение. Что он себя представляет?

Интерфейс `Map<K, V>` представляет отображение или иначе говоря словарь, где каждый элемент представляет пару "ключ-значение". При этом все ключи уникальные в рамках объекта `Map`. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.

Следует отметить, что в отличие от других интерфейсов, которые представляют коллекции, интерфейс `Map` НЕ расширяет интерфейс `Collection`.

Среди методов интерфейса `Map` можно выделить следующие:

- `void clear()`: очищает коллекцию
- `boolean containsKey(Object k)`: возвращает `true`, если коллекция содержит ключ `k`
- `boolean containsValue(Object v)`: возвращает `true`, если коллекция содержит значение `v`
- `Set<Map.Entry<K, V>> entrySet()`: возвращает набор элементов коллекции. Все элементы представляют объект `Map.Entry`
- `boolean equals(Object obj)`: возвращает `true`, если коллекция идентична коллекции, передаваемой через параметр `obj`
- `boolean isEmpty`: возвращает `true`, если коллекция пуста
- `V get(Object k)`: возвращает значение объекта, ключ которого равен `k`. Если такого элемента не окажется, то возвращается значение `null`
- `V getOrDefault(Object k, V defaultValue)`: возвращает значение объекта, ключ которого равен `k`. Если такого элемента не окажется, то возвращается значение `defaultValue`
- `V put(K k, V v)`: помещает в коллекцию новый объект с ключом `k` и значением `v`. Если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После

добавления возвращает предыдущее значение для ключа `k`, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается значение `null`

- `V putIfAbsent(K k, V v)`: помещает в коллекцию новый объект с ключом `k` и значением `v`, если в коллекции еще нет элемента с подобным ключом.
- `Set<K> keySet()`: возвращает набор всех ключей отображения
- `Collection<V> values()`: возвращает набор всех значений отображения
- `void putAll(Map<? extends K, ? extends V> map)`: добавляет в коллекцию все объекты из отображения `map`
- `V remove(Object k)`: удаляет объект с ключом `k`
- `int size()`: возвращает количество элементов коллекции

Чтобы положить объект в коллекцию, используется метод `put`, а чтобы получить по ключу - метод `get`. Реализация интерфейса `Map` также позволяет получить наборы как ключей, так и значений. А метод `entrySet()` возвращает набор всех элементов в виде объектов `Map.Entry<K, V>`.

Обобщенный интерфейс `Map.Entry<K, V>` представляет объект с ключом типа `K` и значением типа `V` и определяет следующие методы:

- `boolean equals(Object obj)`: возвращает `true`, если объект `obj`, представляющий интерфейс `Map.Entry`, идентичен текущему
- `K getKey()`: возвращает ключ объекта отображения
- `V getValue()`: возвращает значение объекта отображения
- `V setValue(V v)`: устанавливает для текущего объекта значение `v`
- `int hashCode()`: возвращает хеш-код данного объекта

При переборе объектов отображения мы будем оперировать этими методами для работы с ключами и значениями объектов.

Классы отображений. `HashMap`

Базовым классом для всех отображений является абстрактный класс `AbstractMap`, который реализует большую часть методов интерфейса `Map`. Наиболее распространенным классом отображений является `HashMap`, который реализует интерфейс `Map` и наследуется от класса `AbstractMap`.

Чтобы добавить или заменить элемент, используется метод `put`, либо `replace`, а чтобы получить его значение по ключу - метод `get`. С помощью других методов интерфейса `Map` также производятся другие манипуляции над элементами: перебор, получение ключей, значений, удаление.

89.Стандартные поток ввода-вывода, предоставляемые Java

- **InPutStream** – поток ввода используется для считывания данных с источника.
- **OutPutStream** – поток вывода используется для записи данных по месту назначения.

90.Понятие сериализации, интерфейс Serializable?

Сериализация представляет процесс записи состояния объекта в поток, соответственно процесс извлечения или восстановления состояния объекта из потока называется десериализацией. Сериализация очень удобна, когда идет работа со сложными объектами.

Интерфейс Serializable

Сразу надо сказать, что сериализовать можно только те объекты, которые реализуют интерфейс `Serializable`. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект, реализующий его, может быть сериализован.

91. Какие объекты можно сериализовать?

Сериализация — это процесс сохранения состояния объекта в последовательность байт.

Любой Java-объект преобразуется в последовательность байт.

92.Какие методы определяет интерфейс Serializable?

Сериализация (Serialization) — это процесс, который переводит объект в последовательность байтов, по которой затем его можно полностью восстановить. Зачем это нужно? Дело в том, при обычном выполнении программы максимальный срок жизни любого объекта известен — от запуска программы до ее окончания. Сериализация позволяет расширить эти рамки и «дать жизнь» объекту так же между запусками программы.

Сериализация. Класс ObjectOutputStream

Для сериализации объектов в поток используется класс **ObjectOutputStream**. Он записывает данные в поток.

Для создания объекта `ObjectOutputStream` в конструктор передается поток, в который производится запись:

1 ObjectOutputStream(OutputStream out)

Для записи данных ObjectOutputStream использует ряд методов, среди которых можно выделить следующие:

- **void close():** закрывает поток
- **void flush():** очищает буфер и сбрасывает его содержимое в выходной поток
- **void write(byte[] buf):** записывает в поток массив байтов
- **void write(int val):** записывает в поток один младший байт из val
- **void writeBoolean(boolean val):** записывает в поток значение boolean
- **void writeByte(int val):** записывает в поток один младший байт из val
- **void writeChar(int val):** записывает в поток значение типа char, представленное целочисленным значением
- **void writeDouble(double val):** записывает в поток значение типа double
- **void writeFloat(float val):** записывает в поток значение типа float
- **void writeInt(int val):** записывает целочисленное значение int
- **void writeLong(long val):** записывает значение типа long
- **void writeShort(int val):** записывает значение типа short
- **void writeUTF(String str):** записывает в поток строку в кодировке UTF-8
- **void writeObject(Object obj):** записывает в поток отдельный объект

93. Что означает понятие десериализация?

Десериализация — это обратный процесс: восстановление структур и объектов из сериализованной строки или последовательности байтов.

94. Класс File, определенный в пакете java.io, не работает напрямую с потоками. В чем состоит его задача?

Класс File, определенный в пакете java.io, не работает напрямую с потоками. Его задачей является управление информацией о файлах и каталогах. Хотя на уровне операционной системы файлы и каталоги отличаются, но в Java они описываются одним классом File.

95. При работе с объектом класса FileOutputStream происходит вызов метода FileOutputStream.write(), что в результате этого происходит?

Главное назначение класса FileOutputStream — запись байтов в файл.

В зависимости от значения параметра boolean append который был передан при вызове конструктора (True/False, если не указано то по умолчанию передается False) либо в файл допишется содержимое передаваемой в метод переменной, либо содержимое файла будет заменено на содержимое переменной.

