

Список вопросов к экзамену по дисциплине

Программирование на языке Джава зима 2023-2024 год

Тема 1. Особенности платформы Java. Синтаксис языка Java

1. Парадигма объектно-ориентированного программирование. Основные принципы ООП и их реализация в языке программирования Java и C++

Объектно-ориентированное программирование (ООП) – это такой стиль программирования, который фиксирует поведение реального мира таким способом, при котором детали его реализации скрыты. Если подобное удается, то это позволяет тому, кто занимается решением проблемы, вести рассуждения в терминах предметной области. ООП позволяет разложить проблему на связанные между собой задачи, каждая проблема становится самостоятельным объектом, содержащим свои собственные данные и код.

ООП – это взгляд на программирование, сосредоточенный на данных, в котором данные и поведение (код) жестко связаны. Данные и поведение представлены в виде **классов**, экземпляры которых – **объекты**. Класс – это то, о чем можно мыслить как об отдельном понятии, а объект некоторого класса – это то, о чем можно мыслить как об отдельной сущности.

Инкапсуляция – это возможность скрыть внутренние детали при описании общего интерфейса АТД с целью защиты его от внешнего вмешательства или неправильного использования. Инкапсуляция включает как детали внутренней реализации специфического типа, так и доступные извне операции и функции, которые могут оперировать объектами этого типа. Детали реализации могут делать недоступным для пользователя код, который использует тип. Например, стек – одна из разновидностей динамических структур данных – может быть реализован как массив фиксированной длины, доступ к которому осуществляется посредством специального индекса, называемого *top* (верхушка стека), при этом общедоступные операции будут включать вталкивание данных в стек (*push*) и выталкивание данных из стека (*pop*). Операции *push* и *pop* в терминологии ООП называются **методами**. Изменение внутренней реализации в таком линейном списке не будет влиять на то, как будут извне использоваться операции *push* и *pop*. Реализация стека в данном случае скрыта от его пользователей. АТД, такой как стек, является описанием идеального общего поведения типа. Пользователь этого типа понимает, что операции *push* и *pop* приводят к определенному общему поведению. Конкретная реализация АТД имеет также свои ограничения, например, после большого числа операций *push* область стека переполняется. Эти ограничения действуют на общее поведение.

Наследование – это процесс, посредством которого один объект может приобретать основные свойства другого объекта и добавлять к ним черты, характерные только для него.

Наследование – это средство получения новых классов, называемых *производными классами*, из существующих, называемых *базовыми классами*. При этом повторно используется уже имеющийся код. Производный класс образуется из базового путем добавления или изменения кода. Посредством наследования может быть создана *иерархия* родственных типов (иерархия классов), которые совместно используют код и общий интерфейс. Наследование – это первооснова ООП.

Полиморфизм – это механизм, обеспечивающий многократное использование кода. Полиморфизм может применяться и к функциям, и к операторам – в C++ имя функции или оператор перегружаемы – *статический полиморфизм*. Например, когда одно и то же имя функции используется для множества различных действий, то это называется *перегрузкой функции*. Фактически в С ограниченно применяется полиморфизм, например, при выполнении арифметических операций. Так, например, для операции деления – если аргументы целого типа, то используется целочисленное деление, если один или оба аргумента вещественного типа, то используется деление с плавающей точкой, т.е. поведение операции деления определяется во время компиляции программы типом данных. Этот механизм применим и к другим, определяемых пользователем, типам данных, в C++ это называется *перегрузкой оператора*.

2. Организация программы на Java. Основные структурные единицы. Процесс интерпретации и компиляции. Роль JVM

Я честно в душе не ебу че тут просят

Поскольку Джава объектно-ориентированный язык, то весь код пишется в классах, каждый класс пишется в отдельном файле.

Для того чтобы запускать программу нужно написать метод-функцию main, через который и будет работать программа.

Java — мультиплатформенный язык программирования. Это значит, что программы, написанные на языке Java, можно выполнять на любой платформе, где установлена специальная исполняющая система Java. Такая система называется Java Virtual Machine (JVM). Для того, чтобы перевести программу из исходного кода в код, понятный JVM, нужно её скомпилировать. Код, понятный JVM называется байт-кодом и содержит набор инструкций, которые в дальнейшем будут исполнять виртуальная машина.

Для компиляции исходного кода в байт-код существует компилятор javac, входящий в поставку JDK (Java Development Kit). На вход компилятор принимает файл с расширением .java, содержащий исходный код программы, а на выходе

выдает файл с расширением .class, содержащий байт-код, необходимый для исполнения программы виртуальной машиной.

После того, как программа была скомпилирована в байт-код, она может быть выполнена с помощью виртуальной машины.

3. Структурирование Java приложения, пакеты. Уровни доступа и видимости

Как правило, в Java классы объединяются в пакеты. Пакеты позволяют организовать классы логически в наборы. По умолчанию java уже имеет ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д. Кроме того, пакеты могут иметь вложенные пакеты.

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. Принадлежность к пакету позволяет гарантировать однозначность имен.

Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву package, после которой указывается имя пакета:

Как правило, названия пакетов соответствуют физической структуре проекта, то есть организации каталогов, в которых находятся файлы с исходным кодом. А путь к файлам внутри проекта соответствует названию пакета этих файлов. Например, если классы принадлежат пакету turack, то эти классы помещаются в проекте в папку turack.

Если нам нужен класс из другого пакета, то нам нужно его импортировать(исключением будут классы из java.lang, всякие String и т.д., они по дефолту уже импортированы). На примере импорта Сканнера, который нам нужен, чтобы считывать данные из консоли:

Я хз что за уровни доступа, есть модификаторы доступа, тип там private public и т.д

Данные и методы класса, объявленные с public доступны в любом месте программы.

Данные и методы класса, объявленные с private доступны внутри класса.

Все, что объявлено без модификатора видимости по умолчанию имеет видимость и может быть доступно из любого класса в том же пакете

Protected Поля и методы, обозначенные модификатором доступа protected, будут видны, в пределах всех классов, находящихся в том же пакете, что и наш и в пределах всех классов-наследников нашего класса.

4. Примитивные и ссылочные типы данных. Использование механизмов автоупаковки и автораспаковки. Операция приведения типов. Пониждающее и повышающее приведение.

*Примитивные (простые) типы данных**Целочисленные*

1. char – 16-битовый символ Unicode,
2. byte – 8-битовое целое число со знаком,

3. short – 16-битовое целое число со знаком,
4. int – 32-битовое целое число со знаком,
5. long – 64-битовое целое число со знаком.

Вещественные типы:

6. float – 32-битовое число с плавающей точкой (IEEE 754-1985),
7. double – 64-битовое число с плавающей точкой (IEEE 754-1985).

boolean допускает хранение значений true или false.

*Ссылочные типы данных**Переменные типа класс – ссылки на объекты*

- Классы – переменные типа класс
- Интерфейсы – ссылки интерфейсного типа
- Перечисления – поименованные константы
- Типы оболочки соответствуют каждому **примитивному** типу

Элементы массивов:

- float – 32-битовое число с плавающей точкой (IEEE 754-1985),
- double – 64-битовое число с плавающей точкой (IEEE 754-1985).

Конвенция кода на Java

Примитивные типы можно автоматически преобразовать к объекту обертки (упаковка)

```
Integer obj;
int num = 42;
obj = num;
```

Присваивание создает соответствующий объект Integer

Обратное преобразование (называется автораспаковка) и также происходит автоматически, по мере необходимости.

```
int in = 0;
in = new Integer(9);
```

Приведение типов для интов работает так:

```
public static void main(String[] args) {  
  
    int bigNumber = 10000000;  
  
    short littleNumber = 1000;  
  
    littleNumber = (short) bigNumber;  
    System.out.println(littleNumber);  
}
```

если мы делаем повышающее приведение, то можно еще и так сделать

```
public class Main {  
  
    public static void main(String[] args) {  
  
        int bigNumber = 10000000;  
  
        byte littleNumber = 16;  
  
        bigNumber = littleNumber;  
        System.out.println(bigNumber);  
    }  
}
```

Но нужно учитывать что разрядная сетка уменьшится, можно потерять значение.

Char можно также перевести к другим типам, ведь это просто цифра в юникоде обозначающая символ.

Понижающее приведение - это когда ты приводишь int к byte, можно потерять значение, так как уменьшается разрядная сетка

Повышающее приведение – это когда ты приводишь byte к int, разрядная сетка повышается, большие цифры хранить можно

5. Этапы проектирования, разработки и отладки ООП программ. Понятие конвенции кода языка и стиля программирования.

Этапы проектирования, разработки и отладки ООП программ могут быть описаны следующим образом:

- Проектирование:** на этом этапе определяются классы, объекты, интерфейсы, методы и их взаимодействия. Также происходит проектирование основных структур данных и алгоритмов, которые будут использоваться в программе.
- Разработка:** написание кода, используя уже определенную структуру и объекты. Осуществляется написание классов, методов, интерфейсов, а

также реализация алгоритмов и логики программы с применением объектно-ориентированного подхода.

3. **Отладка:** после написания кода происходит этап отладки, где исправляются ошибки, проверяется корректность работы программы и ее отклик на различные сценарии использования.

Конвенции кода и стили программирования включают в себя следующие основные принципы:

- **Именование переменных, классов, методов и пакетов:** в Java рекомендуется использовать CamelCase нотацию. Например, MyClass, myMethod, myVariable.

- **Отступы:** используются отступы в 4 пробела (табуляция) для обозначения вложенности кода.

- **Комментирование кода:** для улучшения читаемости и понимания кода следует использовать комментарии, описывающие основные функции и блоки кода.

- **Использование классов и объектно-ориентированного подхода:** в Java стиль программирования ориентирован на использование объектов и классов для реализации функциональности.

- **Использование библиотек и фреймворков:** Java имеет обширный набор стандартных библиотек, поэтому рекомендуется использовать их при разработке.

Следование этим конвенциям и стилю программирования помогает создавать более читаемый, структурированный и легко поддерживаемый код.

6. Массивы в Java. Способы создания массивов. Индексы. Размерность массивов. Доступ к элементам массива и примеры использования

Массив – структура данных, в которой хранятся элементы одного типа, получается доступ по индексу

Можно так же как в Си

```
int[] sample = {12, 56, 7, 34, 89, 43, 23, 9};
```

Или вот так:

```
Circle[] array = { new Circle(1, 1, "red"), new Circle(3, 4, "green"), new Circle(1, 3, "")};
```

Размерность массивов – количество индексов, требуемое для однозначной адресации всех элементов.

Применение максимально обширно, можно хранить в них функции, можно хранить данные, упорядочивать данные.

7. Класс Scanner и его использование для чтения стандартного потока ввода, конструктор класса Scanner

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.

```
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 истину если с потока ввода можно считать целое число
            i = sc.nextInt(); // считывает целое число с потока ввода и сохраняет в переменную
            System.out.println(i*2);
        } else {
            System.out.println("Вы ввели не целое число");
        }
    }
}
```

Сам конструктор может принимать и, например строку, с которой он будет работать.

```
public class Main {

    public static void main(String[] args) {

        Scanner scanner = new Scanner("Люблю тебя, Петра творенье,\n" +
            "Люблю твой строгий, стройный вид,\n" +
            "Невы державное теченье,\n" +
            "Береговой ее гранит");
        String s = scanner.nextLine();
        System.out.println(s);
        s = scanner.nextLine();
        System.out.println(s);
        s = scanner.nextLine();
        System.out.println(s);
        s = scanner.nextLine();
        System.out.println(s);
    }
}
```

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

Метод nextLine() обращается к источнику данных (нашему тексту с четверостишием), находит там следующую строку, которую он еще не считывал (в нашем случае — первую) и возвращает ее.

Метод nextInt() обращается к источнику данных и берет целое число из строки и возвращает его. Если там будет НЕ число, то будет ошибка

hasNextInt() проверяет чтобы след. строка была целым числом

hasNextLine() проверяет есть ли вообще след. строка

9. Виды типов данных в Джава. Примитивные типы данных, объявление и присваивание переменных. Константы в Джава: объявление и использование константы

Есть целочисленные, с плавающей точкой, логические и символьные примитивные типы данных.

Константы объявляются вот так

```
public static final String MY_CONSTANT="Мой текст";
```

Применить их можно, например, когда нужно хранить число Pi

10. Виды типов данных в Джава. Объектные типы данных

Есть целочисленные, с плавающей точкой, логические и символьные примитивные типы данных.

Объектные(ссылочные) типы данных: экземпляры классов, массивы

Величины ссылочного типа - это ссылки на объекты. Переменная, имеющая ссылочный тип, содержит ссылку на объект.

11. Объявление и использование бестиповых переменных в Джава

Дженерики (обобщения) — это особые средства языка Java для реализации обобщенного программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания.

Одно из назначений — более сильная проверка типов во время компиляции и устранение необходимости явного приведения.

Допустим, мы определяем класс для представления банковского счета. К примеру, он мог бы выглядеть следующим образом:

```
1 class Account{  
2  
3     private int id;  
4     private int sum;  
5  
6     Account(int id, int sum){  
7         this.id = id;  
8         this.sum = sum;  
9     }  
10  
11     public int getId() { return id; }  
12     public int getSum() { return sum; }  
13     public void setSum(int sum) { this.sum = sum; }  
14 }
```

Класс Account имеет два поля: id - уникальный идентификатор счета и sum - сумма на счете.

В данном случае идентификатор задан как целочисленное значение, например, 1, 2, 3, 4 и так далее. Однако также нередко для идентификатора используются и строковые значения. И числовые, и строковые значения имеют

свои плюсы и минусы. И на момент написания класса мы можем точно не знать, что лучше выбрать для хранения идентификатора - строки или числа. Либо, возможно, этот класс будет использоваться другими разработчиками, которые могут иметь свое мнение по данной проблеме. Например, в качестве типа id они захотят использовать какой-то свой класс.

И на первый взгляд мы можем решить данную проблему следующим образом: задать id как поле типа Object, который является универсальным и базовым суперклассом для всех остальных типов:

```
1 public class Program{
2
3     public static void main(String[] args) {
4
5         Account acc1 = new Account(2334, 5000); // id - число
6         int acc1Id = (int)acc1.getId();
7         System.out.println(acc1Id);
8
9         Account acc2 = new Account("sid5523", 5000);    // id - строка
10        System.out.println(acc2.getId());
11    }
12 }
13 class Account{
14
15     private Object id;
16     private int sum;
17
18     Account(Object id, int sum){
19         this.id = id;
20         this.sum = sum;
21     }
22
23     public Object getId() { return id; }
24     public int getSum() { return sum; }
25     public void setSum(int sum) { this.sum = sum; }
26 }
```

В данном случае все замечательно работает. Однако тогда мы сталкиваемся с проблемой **безопасности типов**. Например, в следующем случае мы получим ошибку:

```
1 Account acc1 = new Account("2345", 5000);
2 int acc1Id = (int)acc1.getId(); // java.lang.ClassCastException
3 System.out.println(acc1Id);
```

Эти проблемы были призваны устранить обобщения или generics. Обобщения позволяют не указывать конкретный тип, который будет использоваться. Поэтому определим класс Account как обобщенный:

```
1 class Account<T>{
2
3     private T id;
4     private int sum;
5
6     Account(T id, int sum){
7         this.id = id;
8         this.sum = sum;
9     }
10
11    public T getId() { return id; }
12    public int getSum() { return sum; }
13    public void setSum(int sum) { this.sum = sum; }
14 }
```

С помощью буквы **T** в определении класса `class Account<T>` мы указываем, что данный тип **T** будет использоваться этим классом. Параметр **T** в угловых

скобках называется **универсальным параметром**, так как вместо него можно подставить любой тип. При этом пока мы не знаем, какой именно это будет тип: String, int или какой-то другой. Причем буква T выбрана условно, это может и любая другая буква или набор символов.

```
1 public class Program{
2
3     public static void main(String[] args) {
4
5         Account<String> acc1 = new Account<String>("2345", 5000);
6         String acc1Id = acc1.getId();
7         System.out.println(acc1Id);
8
9         Account<Integer> acc2 = new Account<Integer>(2345, 5000);
10        Integer acc2Id = acc2.getId();
11        System.out.println(acc2Id);
12    }
13 }
14 class Account<T>{
15
16     private T id;
17     private int sum;
18
19     Account(T id, int sum){
20         this.id = id;
21         this.sum = sum;
22     }
23
24     public T getId() { return id; }
25     public int getSum() { return sum; }
26     public void setSum(int sum) { this.sum = sum; }
27 }
```

!! При определении переменной данного класса и создании объекта после имени класса в угловых скобках нужно указать, какой именно тип будет использоваться вместо универсального параметра.

!! При этом надо учитывать, что они работают только с объектами, но не работают с примитивными типами. То есть мы можем написать Account<Integer>, но не можем использовать тип int или double, например, Account<int>. Вместо примитивных типов надо использовать классы-обертки: Integer вместо int, Double вместо double и т.д.

Например, первый объект будет использовать тип String, то есть вместо T будет подставляться String:

```
1 Account<String> acc1 = new Account<String>("2345", 5000);
```

В этом случае в качестве первого параметра в конструктор передается строка.

А второй объект использует тип int (Integer):

```
1 Account<Integer> acc2 = new Account<Integer>(2345, 5000);
```

12. Объявление переменных типа класс и их инициализация

Допустим у нас есть класс MyClass

Если мы напишем так, то мы объявим переменную ссылочного типа MyClass

```
public class Main {  
    public static void main(String[] args) {  
        MyClass cls;  
    }  
}
```

Если мы напишем так то инициализируем

```
public class Main {  
    public static void main(String[] args) {  
        MyClass cls = new MyClass();  
    }  
}
```

13. Массивы в Джава, как объектные типы данных, контроль доступа за выход за границы массива. Объявление и инициализация массивов, длина массива, получение доступа к элементу массива

Ну, когда мы создаем массив, мы выделяем ячейку в памяти под указанное количество элементов какого-то типа. А сама переменная хранит ссылку на ту часть памяти где хранится этот массив

Пример создания массива строк

```
String [] birthdays = new String[10]; //массив строк Java
```

Контроль доступа за выход за границы массива накладывается на руки программиста, то есть нужно проверять, чтобы индекс был в границах от нуля до размера массива – 1, иначе будет out of bounds Ошибка.

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

Ну, когда мы создаем массив, мы выделяем ячейку в памяти под указанное количество элементов какого-то типа. А сама переменная хранит ссылку на ту часть памяти где хранится этот массив

```
String [] birthdays = new String[10]; //массив строк Java
```

Без инициализации - это когда мы просто указываем первую часть, до знака = и ставим точку с запятой, т.е. мы просто объявляем массив, не вызывая new. Для примера выше это будет выглядеть так: String [] birthdays;.

По сути, неинициализированных значений и массивов в джава нет, когда мы создаём массив он автоматически заполняется нулями (если это массив примитивов) или значениями null, если это массив объектов.

15. Инициализация полей класса и локальных переменных (отличие), инициализатор и статический инициализатор (когда вызывается).

Поле класса - это переменная, объявленная в теле класса. Она представляет собой характеристику или состояние объектов данного класса.

Локальная переменная - это переменная, объявленная внутри метода, блока кода или конструктора и существующая только в пределах этого блока.

Инициализация полей:

- могут быть инициализированы при объявлении или в блоке инициализации (инициализаторе) или в конструкторе.
- при инициализации полей класса при объявлении, их значения будут установлены при создании объекта и будут доступны всем методам объекта.
- поля класса могут быть статическими или нестатическими (инстанс-полями)

Нестатическое поле класса принадлежит конкретному экземпляру класса и каждый объект имеет свою собственную копию этого поля.

Статическое поле класса принадлежит самому классу, а не конкретным экземплярам. Оно разделяется между всеми объектами этого класса.

Инициализация локальных переменных:

- должны быть инициализированы перед использованием и это может происходить только внутри метода, блока кода или конструктора.

Инициализатор — это блок кода внутри класса, предназначенный для установки начальных значений для объектов.

- Они выполняются каждый раз при создании объекта класса, до вызова конструктора.

Статический инициализатор используется для инициализации статических полей класса. (static{})

- Он выполняется при загрузке класса и до создания объектов.

16. Циклические конструкции в Java. Использование циклов для работы с массивами. Использование итераторов для обработки массивов. Использование итераторов для работы с коллекциями

В Java существует несколько типов циклических конструкций, которые позволяют выполнять повторяющиеся действия. Кроме того, различные методы работы с массивами и коллекциями, такие как использование циклов и итераторов, помогают эффективно обрабатывать данные. Давайте разберем каждый из этих аспектов более подробно:

for – цикл со счётчиком

```
for (int i = 0; i < 10; i++) {  
    // Действия, выполняемые в цикле  
}
```

while – цикл с предусловием:

```
int i = 0;  
while (i < 10) {  
    // Действия, выполняемые в цикле  
    i++;  
}
```

do-while – цикл с постусловием:

```
int i = 0;  
do {  
    // Действия, выполняемые в цикле  
    i++;  
} while (i < 10);
```

Циклы для работы с массивами:

Использование цикла for:

- initialization — выражение, которое инициализирует выполнение цикла. Исполняется только раз в начале цикла. Чаще всего в данном выражении инициализируют счетчик цикла
- termination — boolean выражение, которое регулирует окончание выполнения цикла. Если результат выражения будет равен false, цикл for прервётся.
- increment — выражение, которое исполняется после каждой итерации цикла. Чаще всего в данном выражении происходит инкрементирование или декрементирование переменной счетчика.
- increment — выражение, которое исполняется после каждой итерации цикла. Чаще всего в данном выражении происходит инкрементирование или декрементирование переменной счетчика.

```
int[] numbers = {1, 2, 3, 4, 5};  
for (int i = 0; i < numbers.length; i++) {  
    System.out.println(numbers[i]);  
}
```

```
int[] numbers = {1, 2, 3, 4, 5};  
for (int num : numbers) {  
    System.out.println(num);  
}
```

Цикл while:

- expression — условие цикла, выражение, которое должно возвращать boolean значение.
- statement(s) — тело цикла (одна или более строк кода).

Перед каждой итерацией будет вычисляться значение выражения expression. Если результатом выражения будет true, выполняется тело цикла — statement(s).

```

int[] numbers = {1, 2, 3, 4, 5};
int i = 0;
while (i < numbers.length) {
    System.out.println(numbers[i]);
    i++;
}

```

Цикл do-while:

- expression — условие цикла, выражение, которое должно возвращать boolean значение.
- statement(s) — тело цикла (одна или более строк кода).

В отличие от while, значение expression будет вычисляться после каждой итерации. Если результатом выражения будет *true*, в очередной раз выполнится тело цикла — statement(s) (как минимум раз).

```

int[] numbers = {1, 2, 3, 4, 5};
int i = 0;
do {
    System.out.println(numbers[i]);
    i++;
} while (i < numbers.length);

```

Коллекции в Java представляют собой структуры данных, предназначенные для хранения и манипулирования группами объектов.

Примеры коллекций: ArrayList, LinkedList, HashSet, Stack

Итератор — это интерфейс, который позволяет перебрать все элементы коллекции, не вникая при этом во внутреннее устройство коллекции (то есть не учитываем особенности работы с ней). Итератор имеет в себе 2 основных метода:

- bool hasNext() — говорит, есть ли следующий элемент
- next() — выдаёт следующий элемент

```

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

        Integer[] array = new Integer[]{64, 49, 37, 399};

        // вывод массива на экран с помощью цикла
        LinkedList<Integer> list = new LinkedList<>();
        Collections.addAll(list, array);
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }

        // вывод массива на экран с помощью итератора
        Iterator <Integer> iterator = list.iterator();
        while (iterator.hasNext()){
            System.out.println(iterator.hasNext());
        }
    }
}

```

Статические поля и методы в Java относятся к классу в целом, а не к его экземплярам. Они могут быть вызваны без создания экземпляра класса и разделяются между всеми экземплярами данного класса. В Java статические методы и поля обозначаются ключевым словом static.

Класс Math в Java содержит множество статических методов, предназначенных для выполнения различных математических операций. Некоторые из основных методов класса Math включают:

1. Math.abs(x) - возвращает абсолютное значение числа x.
2. Math.sqrt(x) - возвращает квадратный корень числа x.
3. Math.pow(x, y) - возведение числа x в степень y.
4. Math.max(x, y) - возвращает наибольшее из двух чисел x и y.
5. Math.min(x, y) - возвращает наименьшее из двух чисел x и y.
6. Math.sin(x), Math.cos(x), Math.tan(x) - тригонометрические функции - синус, косинус и тангенс угла x (значение угла в радианах).
7. Math.random() - возвращает случайное число в диапазоне от 0.0 (включительно) до 1.0 (исключительно).

Пример использования статических методов класса Math:

```
public class MathExample {  
    public static void main(String[] args) {  
        int x = -10;  
        System.out.println("Абсолютное значение числа x: " + Math.abs(x));  
  
        double y = 16;  
        System.out.println("Квадратный корень числа y: " + Math.sqrt(y));  
  
        double a = 5;  
        double b = 3;  
        System.out.println("Максимальное из чисел a и b: " + Math.max(a, b));  
  
        double angleInRadians = Math.toRadians(45); // Преобразование угла из градусов в радианы  
        System.out.println("Синус угла 45 градусов: " + Math.sin(angleInRadians));  
  
        System.out.println("Случайное число: " + Math.random());  
    }  
}
```

Ключевое слово static отсутствует в примере вызова методов класса Math, поскольку все его методы и поля уже статические.

18. Понятие перечисления. Состав и приемы использования в ООП программах на Java

В Java перечисление (enum) - это специальный тип данных, который позволяет определять набор констант. Перечисления представляют собой отдельный класс с фиксированным набором экземпляров, которые представляют различные константы.

Создание перечисления в Java осуществляется с использованием ключевого слова enum. Вот пример простого перечисления с названиями дней недели:

```
public enum Day {  
    MONDAY,  
    TUESDAY,  
    WEDNESDAY,  
    THURSDAY,  
    FRIDAY,  
    SATURDAY,  
    SUNDAY  
}
```

Каждый элемент перечисления (например, MONDAY, TUESDAY и т. д.) является экземпляром самого перечисления.

Перечисления в Java широко используются для определения списков констант, например, цветов, дней недели и т. д. Они могут быть удобны при работе с ограниченным набором значений, когда требуется явное перечисление всех возможных вариантов.

Перечисления также могут содержать методы, поля и конструкторы, что позволяет создавать более сложные структуры данных.

Пример использования перечисления в Java:

```
public class EnumExample {  
    public static void main(String[] args) {  
        Day today = Day.MONDAY;  
        System.out.println("Сегодня " + today);  
    }  
}
```

Этот код выведет "Сегодня MONDAY".

Кроме того, перечисления могут использоваться в switch-конструкциях, что делает код более читаемым и удобным для работы с константами. Вот пример использования перечисления в switch:

```
public class EnumSwitchExample {  
    public static void main(String[] args) {  
        Day today = Day.MONDAY;  
        switch(today) {  
            case MONDAY:  
                System.out.println("Сегодня понедельник");  
                break;  
            case TUESDAY:  
                System.out.println("Сегодня вторник");  
                break;  
            // далее следуют остальные дни недели  
        }  
    }  
}
```

Использование перечислений способствует повышению читаемости и понятности кода, а также решает проблему магических чисел и строк, заменяя их более понятными константами.

29. Понятие класса. Определение, инициализация. Модификаторы доступа. Константы и переменные. Объявление классов.

Класс (простыми словами) — это, по сути, шаблон для объекта. Он определяет, как объект будет выглядеть и какими функциями обладать. Каждый объект является объектом какого-то класса.

Класс (сложнее но красивее) — это шаблонная конструкция, которая позволяет описать в программе объект, его свойства (атрибуты или поля класса) и поведение (методы класса). Каждый класс имеет своё имя, чтобы в будущем к нему можно было обратиться.

Чтобы создать (определить) класс на Java, необходимо написать слово class, дать ему название и поставить фигурные скобки: class *имя класса* {}. Также можно задать поле видимости у любого из классов, а можно и не задавать.

Инициализация - это когда мы впервые задаем переменной какое-либо значение. Например мы объявили переменную int a;. Мы ее не инициализировали - другими словами, не задали начальное, стартовое значение. Если попытаться вывести её значение мы получим ошибку "переменная не инициализирована".

Модификаторы доступа — это чаще всего ключевые слова, которые регулируют уровень доступа к разным частям кода. 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
- protected
- default (package visible)
- public

Переменная — это часть памяти, которая может содержать значение данных, она имеет тип данных. Переменные обычно используются для хранения информации, необходимой вашей программе для выполнения своей работы. Это может быть информация любого типа, от текстов, кодов до цифр, временных результатов многоступенчатых расчетов и т. д.

В Java имеется **несколько типов констант**, соответствующих его встроенным типам, а именно: константа null нулевого типа; логические константы true (истина) и false (ложь) типа boolean; символьные константы типа char, например, 'a' или '\t'; строковые константы класса, например, "Привет всем!"; целые константы типов int и long, например, 111 или -2L; плавающие константы типов float и double, например, 3.1415926.

30. Получение информации о типе. Создание экземпляров классов. Вызов методов класса. Объявление класса на Джава, пример объявления

Рефлексия – это получение информации о типах во время выполнения программы.

Java для получения информации о типе объекта или переменной можно использовать оператор instanceof или метод getClass(). Давайте рассмотрим оба способа.

Использование оператора instanceof:

```
String str = "Hello";
if (str instanceof String) {
    System.out.println("str is an instance of String");
}
```

Использование метода getClass():

```
String str = "Hello";
Class cls = str.getClass();
System.out.println("The type of str is: " + cls.getName());
```

Создание экземпляров класса

Для создания экземпляра класса в Java используется ключевое слово new, после которого идет вызов конструктора класса.

Пример создания экземпляра класса Car:

```
Car myCar = new Car("Toyota", "Camry", 2023);
```

Вызов методов класса:

После создания экземпляра класса вы можете вызвать его методы, используя оператор точки . и имя метода

```
myCar.drive(100); // Передача аргумента 100 в метод drive
```

Объявление класса в Java:

```
public class Person {
    private String name;
    private int age;

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

    public void greet() {
        System.out.println("Hello, my name is " + name + " and I am " + age + " years old.");
    }
}
```

В этом примере:

- Класс называется Person.
- У класса есть два приватных поля: name (типа String) и age (типа int).
- Есть конструктор класса, который инициализирует поля name и age.
- Есть метод greet, который выводит приветствие с использованием имени и возраста.

31. ООП в Java. Понятие объекта. Что представляет собой Java приложение с точки зрения ООП. Основные характеристики объектов в Java.

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.

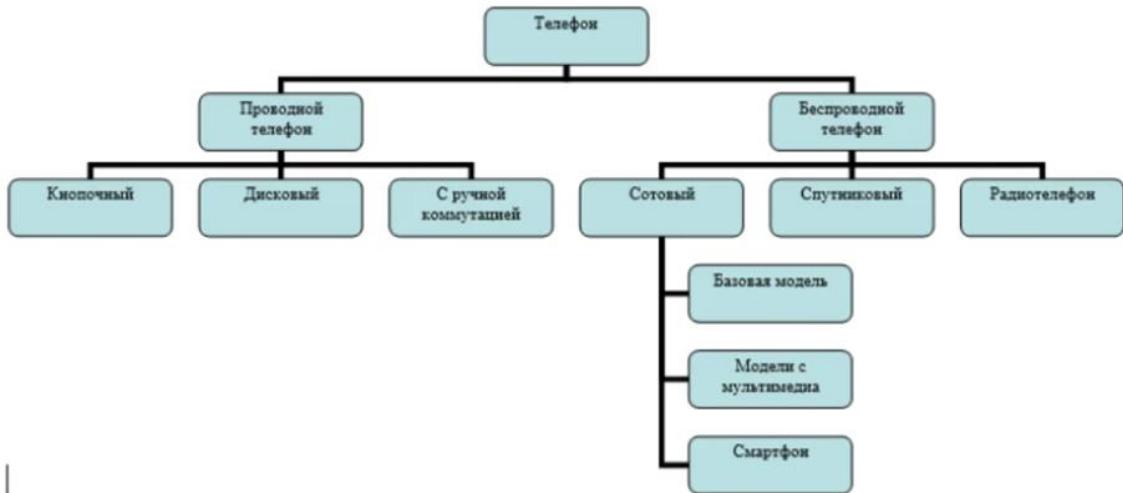
Основные принципы ООП:

- Абстракция

- способ представления элементов задачи из реального мира в виде объектов в программе. Абстракция всегда связана с обобщением некоторой информации о свойствах предметов или объектов.

Пример на классе «Phone»:

Для начала выделим наиболее распространенные типы телефонов от самых первых и до наших дней. Например, их можно представить в виде диаграммы, приведенной на рисунке.



Теперь с помощью абстракции мы можем выделить в этой иерархии объектов общую информацию: общий абстрактный тип объектов — телефон, общую характеристику телефона — год его создания, и общий интерфейс — все телефоны способны принимать и посыпать вызовы.

```

1 public abstract class AbstractPhone {
2     private int year;
3
4     public AbstractPhone(int year) {
5         this.year = year;
6     }
7     public abstract void call(int outputNumber);
8     public abstract void ring (int inputNumber);
9 }
  
```

· Инкапсуляция

- 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.

Задача программиста — определить, какие атрибуты и методы будут доступны для открытого доступа, а какие являются внутренней реализацией объекта и должны быть недоступны для изменений.

· Полиморфизм

- Принцип в ООП, когда программа может использовать объекты с одинаковым интерфейсом без информации о внутреннем устройстве объекта

· Наследование

- использовании уже существующих классов для описания новых

Объектно-ориентированное программирование (ООП) — это методика разработки программ, в основе которой лежит понятие объект.

Объект — это некоторая структура, соответствующая объекту реального мира, его поведению.

Класс — это описание еще не созданного объекта, как бы общий шаблон, состоящий из полей, методов и конструктора, а объект — экземпляр класса, созданный на основе этого описания.

С точки зрения ООП, *Java приложение* - это набор взаимосвязанных объектов, взаимодействующих друг с другом для выполнения задач. Каждый объект выполняет свою уникальную роль и предоставляет методы для манипулирования данными. Приложение организовано через использование классов, которые определяют структуру и поведение объектов, а также через инкапсуляцию, полиморфизм и наследование для создания четкой, модульной и гибкой архитектуры.

Любой объект может обладать *двумя основными характеристиками*: состояние - некоторые данные, которые хранит объект (состояние в виде полей или свойств, которые представляют характеристики объекта. Например, объект класса "Person" может иметь поля "name" и "age"), и поведение - действия, которые может совершать объект (то есть методы).

32. Конструкторы, назначение и использование. Конструктор с параметром, конструктор по умолчанию.

В языке программирования Java конструкторы используются для инициализации объектов при их создании. Конструктор представляет собой метод, который имеет то же имя, что и класс, в котором он определен, и не возвращает никакого значения. Основной целью конструктора является установка начальных значений полей объекта.

В работе с конструкторами в Java следует учитывать следующие особенности:

1. Конструктор по умолчанию: в Java каждый класс по умолчанию имеет конструктор без параметров, который называется конструктором по умолчанию. Если вы не определяете явно какие-либо конструкторы, компилятор Java автоматически добавит конструктор по умолчанию.

2. Конструктор с параметрами: помимо конструктора по умолчанию, класс может также иметь конструкторы с параметрами. Они предоставляют возможность передачи параметров при создании объектов, что позволяет легко инициализировать и настраивать объекты сразу при их создании.

Пример определения конструкторов в классе на Java:

```
public class Car {  
    private String make;  
    private String model;  
  
    // Конструктор по умолчанию  
    public Car() {  
        this.make = "Unknown";  
        this.model = "Unknown";  
    }  
  
    // Конструктор с параметрами  
    public Car(String make, String model)  
    {  
        this.make = make;  
        this.model = model;  
    }  
  
    // Другие методы и поля  
}
```

Пример использования различных конструкторов:

```
public class Main {  
    public static void main(String[] args) {  
        // Использование конструктора по умолчанию  
        Car car1 = new Car();  
  
        // Использование конструктора с параметрами  
        Car car2 = new Car("Toyota", "Camry");  
    }  
}
```

На практике, конструкторы в Java являются важной частью объектно-ориентированного программирования, поскольку позволяют эффективно инициализировать объекты с помощью уже заданных значений и создавать объекты сразу с нужными свойствами.

33. Конструкторы, назначение и использование. Вызов конструктора родительского класса, неявный вызов конструктора родительского класса, порядок инициализации экземпляра Java класса.

Конструктор — это специальный метод, который имеет имя, совпадающее с именем класса, и вызывается при создании экземпляра объекта совместно с оператором *new*. Результатом работы этого метода всегда является экземпляр класса.

Пример использования:

```
public class Cat {  
    private double originWeight;  
    private double weight;  
    private double minWeight;  
    private double maxWeight;  
  
    public Cat() {  
        weight = 1500.0 + 3000.0 * Math.random();  
        originWeight = weight;  
        minWeight = 1000.0;  
        maxWeight = 9000.0;  
    }  
}
```

Видно, что все объявленные в классе переменные в результате работы конструктора получили значение — и объект готов к использованию. Мы можем вызывать различные методы класса, просматривать значение переменных — никаких ошибок не появится.

!! Конструкторы имеются у всех классов, независимо от того, определите вы их или нет, поскольку Java автоматически предоставляет конструктор, используемый по умолчанию (без параметров) и инициализирующий все переменные экземпляра их значениями, заданными по умолчанию.

Но как только вы определите свой собственный конструктор, конструктор по умолчанию предоставляться не будет. Следовательно, если мы удалим конструктор из класса Cat и попытаемся создать объект через new Cat(), то объект будет создан, но все переменные в классе получат значения по умолчанию.

Вызов конструктора родительского класса:

Если у нас есть иерархия классов, то конструкторы подкласса могут вызывать конструкторы родительского класса, чтобы унаследовать инициализацию. Для этого используется ключевое слово `super`. Например, вот как вызывается конструктор родительского класса из конструктора подкласса:

```
public class Parent {  
    public Parent() {  
        // Инициализация родительского класса  
    }  
  
    public class Child extends Parent {  
        public Child() {  
            super(); // Вызов конструктора родительского класса  
            // Инициализация подкласса  
        }  
    }  
}
```

Неявный вызов конструктора родительского класса:

Если в классе отсутствует явное обращение к конструктору родительского класса с помощью `super`, то будет неявно вызван конструктор по умолчанию (без аргументов) родительского класса.

Порядок инициализации экземпляра Java класса:

При создании экземпляра класса в Java инициализация происходит в следующем порядке:

1. **Выделение памяти:** В памяти выделяется место для хранения полей объекта.
2. **Инициализация полей по умолчанию:** Поля объекта инициализируются значениями по умолчанию в зависимости от их типа (например, 0 для чисел, null для ссылочных типов и т.д.).
3. **Вызов конструктора:** Вызывается конструктор, который инициализирует поля объекта и выполняет другие необходимые действия.

34. Использование языка UML для проектирования и документирования объектно-ориентированных программ. Основные UML диаграммы для отображения отношений между классами в ООП программах

UML — это язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

Использование UML-диаграмм позволит развить навыки продумывания архитектуры приложения и поможет выявить недостатки структуры классов на раннем этапе, а не когда вы уже потратите день на реализацию неправильной модели.

Существует два основных типа диаграмм UML: структурные диаграммы и поведенческие диаграммы (а внутри этих категорий имеется много других). Эти варианты существуют для представления многочисленных типов сценариев и диаграмм, которые используют разные типы людей.

Структурные диаграммы представляют статическую структуру программного обеспечения или системы, они также показывают различные уровни абстракции и реализации. Они используются, чтобы помочь визуализировать различные структуры, составляющие систему, например, базу данных или приложение. Они показывают иерархию компонентов или модулей и то, как они связаны и взаимодействуют между собой. Эти инструменты обеспечивают руководство работы и гарантируют, что все части системы функционируют так, как задумано по отношению ко всем остальным частям.

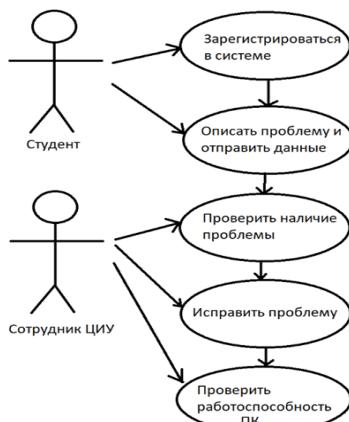
Поведенческие диаграммы Основное внимание здесь уделяется динамическим аспектам системы программного обеспечения или процесса. Эти диаграммы показывают функциональные возможности системы и демонстрируют, что должно происходить в моделируемой системе.

Пример UML отношений

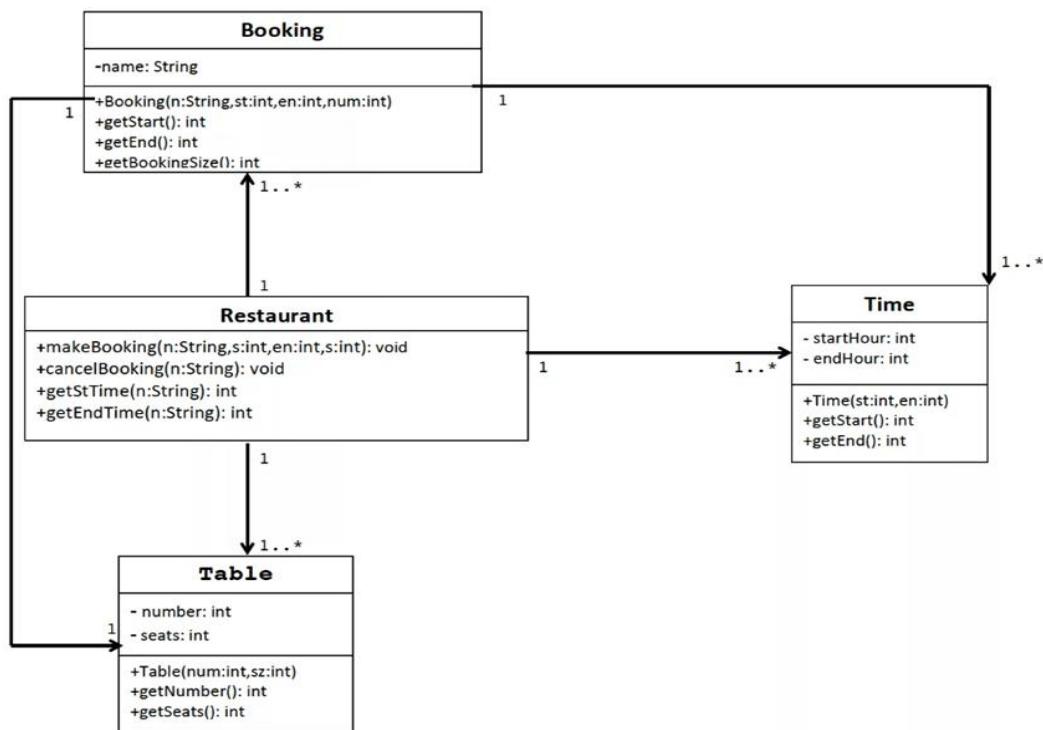
Тип отношения	UML-синтаксис		Краткая семантика
	источник	цель	
Зависимость	----->		Исходный элемент зависит от целевого элемента и изменение последнего может повлиять на первый.
Ассоциация	-----		Описание набора связей между объектами.
Агрегация	◇-----		Целевой элемент является частью исходного элемента.
Композиция	◆-----		Строгая (более ограниченная) форма агрегирования.
Включение	⊕-----		Исходный элемент содержит целевой элемент.
Обобщение	----->		Исходный элемент является специализацией более обобщенного целевого элемента и может замещать его.
Реализация	----->		Исходный элемент гарантированно выполняет контракт, определенный целевым элементом.

MyShared

Пример поведенческой диаграммы:



Пример структурной диаграммы:



35. Управление памятью в Java и C++, процесс освобождения памяти, занимаемой объектом. Метод finalize.

Управление памятью в Java:

В Java память управляется автоматически при помощи механизма сборки мусора (garbage collection). Ключевая особенность состоит в том, что программисту не требуется явно управлять выделением и освобождением памяти. Вместо этого, сборщик мусора отслеживает использование памяти, а затем автоматически освобождает память, занятую объектами, которые больше не доступны.

Процесс освобождения памяти в Java:

1. **Определение ненужных объектов:** Сборщик мусора отслеживает объекты, на которые нет ссылок из активных частей программы.
2. **Маркировка ненужных объектов:** Сборщик мусора маркирует объекты, на которые нет ссылок и которые могут быть освобождены.
3. **Освобождение памяти:** Память, занимаемая прошедшими маркировки объектами, освобождается для повторного использования.

Метод finalize в Java:

Этот метод вызывается Java-машиной у объекта перед тем, как объект будет уничтожен. Фактически этот метод – противоположность конструктору. В нем можно освобождать ресурсы, используемые объектом.

```
1  class Cat
2  {
3      String name;
4
5      Cat(String name)
6      {
7          this.name = name;
8      }
9
10     protected void finalize() throws Throwable
11     {
12         System.out.println(name + " destroyed");
13     }
14 }
```

Управление памятью в C++:

В C++ память управляется явно программистом. Выделение памяти происходит с помощью оператора new, а освобождение — оператора delete. В C++ нет автоматического сбора мусора, поэтому программисту необходимо аккуратно управлять выделением и освобождением памяти.

Процесс освобождения памяти в C++:

1. **Определение ненужных объектов:** Программист должен вручную отслеживать, когда объект больше не нужен.
2. **Освобождение памяти:** После того как объект больше не нужен, программист должен явно освободить память при помощи оператора delete.

```
// Выделение памяти под объект  
ObjectType* ptr = new ObjectType();  
  
// Освобождение памяти  
delete ptr;
```

36. Понятие рекурсии, виды рекурсии и ее использование. Реализация Рекурсивных алгоритмов в ООП программах

Понятие: В контексте языка программирования рекурсия — рекурсия — это некий активный метод (или подпрограмма) вызываемый сам по себе непосредственно, или вызываемой другим методом (или подпрограммой) косвенно.

Виды: Прямая(

Ниже на псевдокоде представлена прямая рекурсия:

```
procedure Alpha;  
  
begin  
  
    Alpha  
  
end;
```

Прямой рекурсией называют вид рекурсии, когда функция вызывает саму себя.

)

Косвенная(

Такой вид рекурсии называют косвенной (или взаимной) рекурсией:

```
procedure Alpha;  
  
begin  
  
    Beta  
  
end;  
  
procedure Beta;  
  
begin  
  
    Alpha  
  
end;
```

Косвенной рекурсией называют вид рекурсии, когда функции вызывают друг друга. В некоторых языках программирования возникают трудности из-за использования прямых ссылок.

)

Использование: Рекурсивные программы часто более емкие, элегантные и их легче понять, чем их итеративные аналоги. Некоторые проблемы легко

решаются с помощью рекурсивных методов, причем гораздо проще, чем на итеративных.

Реализация: Реализация осуществляется в методах, а не основной программе, чтобы была возможность использования рекурсии.

Походы к написанию рекурсивных программ

Как определить когда стоит использовать рекурсию? Если подзадачи похожи на оригинал - тогда мы сможем использовать рекурсию.

Здесь можно выдвинуть два требования:

- (1) подзадачи должны быть проще, чем исходная задача.
- (2) После конечного числа подразделений на подзадачи, должны встретиться такая подзадача, которую можно сразу решить, то есть не нужно больше выполнять декомпозицию.

37. Оператор new. Понятие ссылки и указателя на объект. Реализация в C++ и Java. Время жизни объекта

Обычно мы используем **new** оператор, чтобы создать объект.

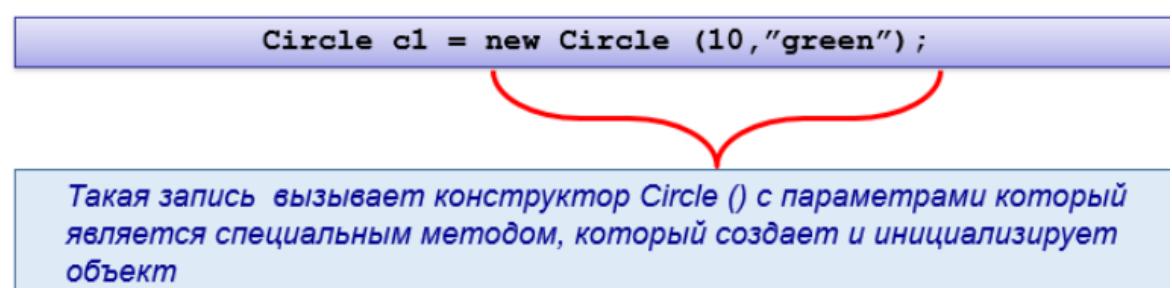


Рисунок 2.4. Синтаксис оператора new

Оператор **new** возвращает ссылку на объект, размещенный в памяти. Как работает **new**? Сначала выделяется память, а потом выполняется присваивание. Фактически это две разных операции. Создание объектов называется инстанцированием. Любой объект - экземпляр типа класс (*instance of class*).

Ссылка - это переменная, содержащая адрес ячейки памяти, в которой хранится объект. Ссылка даёт одно преимущество: можно передать ссылку на объект в какой-нибудь метод, и этот метод будет в состоянии модифицировать (изменять) наш объект используя ссылку на него, вызывая его методы и обращаясь к данным внутри объекта.

Указатели (Pointers) — это объекты, которые хранят адрес памяти и могут экономить память, напрямую указывая на целевой объект, массив или адрес переменной вместо передачи по значению. К сожалению, в Java нет “настоящей” концепции указателей. Но к счастью для нас, существует обходной путь с использованием ссылок на методы, который близок к реальному.

Реализация в Java:

Массив указателей функций

Функциональность массива ссылок на методы можно эмулировать, создав массив интерфейса-оболочки.

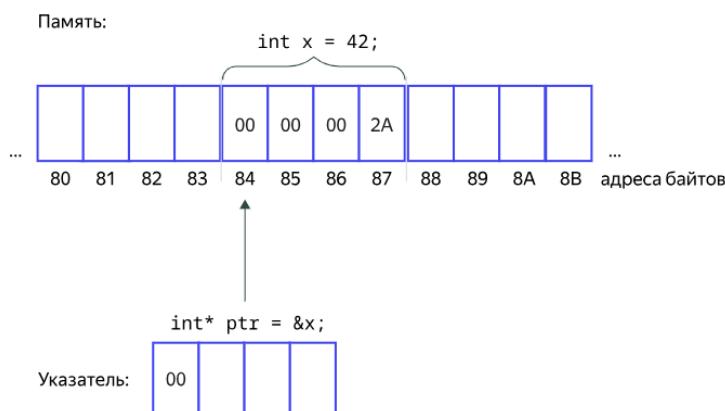
```
1 // Create an array of "pointers"
2 FunctionPointer[] functionPointersArray = new FunctionPointer[3];
3
4 // Assign methods
5 functionPointersArray[0] = this::method1;
6 functionPointersArray[1] = this::method2;
7 functionPointersArray[2] = this::method3;
8
9 // Call methods
10 functionPointersArray[0].methodSignature(3);
11 functionPointersArray[1].methodSignature(4);
12 functionPointersArray[2].methodSignature(5);
```

Код	Описание
<code>String s = null;</code>	<code>s</code> хранит ссылку <code>null</code> .
<code>s = "Привет";</code>	<code>s</code> хранит ссылку на объект-строку
<code>s = null;</code>	<code>s</code> хранит ссылку <code>null</code>

Реализация в C++:

```
1 #include <iostream>
2
3 int main() {
4     int x = 42;
5     int& ref = x; // ссылка на x
6
7     ++x;
8     std::cout << ref << "\n"; // 43
9 }
```

```
1 int main() {
2     int x = 42;
3     int* ptr = &x; // сохраняем адрес в памяти переменной x в указатель ptr
4
5     ++x; // увеличим x на единицу
6     std::cout << *ptr << "\n"; // 43
7 }
```



Если говорить точно, объект является “живым” пока на него есть ссылки.
Как только ссылок не остается — объект “умирает”.

```
1 public class Car {  
2  
3     String model;  
4  
5     public Car(String model) {  
6         this.model = model;  
7     }  
8  
9     public static void main(String[] args) {  
10        Car lamborghini = new Car("Lamborghini Diablo");  
11        lamborghini = null;  
12    }  
13}  
14  
15 }
```

В методе `main()` объект машины Lamborghini Diablo перестает быть живым уже на второй строке. На него была всего одна ссылка, а теперь этой ссылке был присвоен `null`. Поскольку на Lamborghini Diablo не осталось ссылок, он становится «мусором».

38. Переопределение методов в Java, абстрактные методы.

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

Переопределение позволяет взять какой-то метод родительского класса и написать в каждом классе-наследнике свою реализацию этого метода. Новая реализация «заменит» родительскую в дочернем классе.

Чтобы разобраться, возьмем родительский класс `Animal`, обозначающий животных, и создадим в нем метод `voice` — «голос»:

```
1 public class Animal {  
2  
3     public void voice() {  
4  
5         System.out.println("Голос!");  
6     }  
7 }
```

Рассмотрим, как это выглядит на примере. Создадим 4 класса-наследника для нашего класса `Animal`:

```
1 public class Bear extends Animal {
2     @Override
3     public void voice() {
4         System.out.println("Р-р-р!");
5     }
6 }
7 public class Cat extends Animal {
8
9     @Override
10    public void voice() {
11        System.out.println("Мяу!");
12    }
13 }
14
15 public class Dog extends Animal {
16
17     @Override
18     public void voice() {
19         System.out.println("Гав!");
20     }
21 }
22
23
24 public class Snake extends Animal {
25
26     @Override
27     public void voice() {
28         System.out.println("Ш-ш-ш!");
29     }
30 }
```

Чтобы задать нужное нам поведение, мы сделали несколько вещей:

1. Создали в каждом классе-наследнике метод с таким же названием, как и у метода в родительском классе.
2. Сообщили компилятору, что мы не просто так назвали метод так же, как в классе-родителе: хотим переопределить его поведение. Для этого «сообщения» компилятору мы поставили над методом **аннотацию `@Override`** («переопределен»). Проставленная над методом аннотация `@Override` сообщает компилятору (да и читающим твой код программистам тоже): «Все ок, это не ошибка и не моя забывчивость. Я помню, что такой метод уже есть, и хочу переопределить его».
3. Написали нужную нам реализацию для каждого класса-потомка. Змея при вызове `voice()` должна шипеть, медведь — рычать и т.д.

Давай посмотрим, как это будет работать в программе:

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4  
5         Animal animal1 = new Dog();  
6         Animal animal2 = new Cat();  
7         Animal animal3 = new Bear();  
8         Animal animal4 = new Snake();  
9  
10        animal1.voice();  
11        animal2.voice();  
12        animal3.voice();  
13        animal4.voice();  
14    }  
15 }
```

Вывод в консоль:

```
Гав!  
Мяу!  
Р-р-р!  
Ш-ш-ш!
```

Абстрактным называется такой метод, который объявляется, но не предоставляет реализации. Он объявляется при помощи ключевого слова `abstract`, присутствующего в сигнатуре этого метода, и это ключевое слово обязательно должно быть включено в абстрактный класс.

```
public abstract double getArea();
```

Обратите внимание: в абстрактных методах нет тела, но они могут содержать возвращаемый тип, параметры и модификаторы. Ответственность за реализацию абстрактного метода ложится на конкретные субклассы, расширяющие абстрактный класс.

Правила и ограничения при работе с абстрактными методами

Существует несколько правил и ограничений, применимых к абстрактным методам в Java:

- Абстрактный метод не может быть объявлен как `final` или `private`.
- Абстрактный метод обязательно должен объявляться внутри абстрактного класса.
- Конкретный субкласс, расширяющий абстрактный класс, обязательно должен предоставлять реализацию для всех его унаследованных абстрактных методов.

Как использовать абстрактные методы в сочетании с абстрактными классами: примеры
Абстрактные методы часто используются в сочетании с абстрактными классами, так как это позволяет предоставить общий интерфейс для группы взаимосвязанных классов. Продолжая наш пример с фигурами из предыдущего раздела, давайте рассмотрим, как можно использовать абстрактные методы для определения общего интерфейса, который позволял бы вычислять площадь и периметр различных фигур.

[39. Преобразование ссылочных типов в Java, instanceof \(экземпляр класса\).](#)

Преобразование (или приведение) типов в Java может иметь две формы: преобразование примитивных типов и преобразование ссылочных типов.

1. Явное преобразование (Explicit Casting):

- Явное преобразование ссылочных типов используется, когда требуется привести объект к другому типу, если это соответствует иерархии наследования.

Пример:

```
Object obj = "Hello"; // базовый тип Object
String str = (String) obj; // явное преобразование типа
```

2. Проверка типа с использованием оператора instanceof:

- Оператор instanceof используется для проверки, является ли объект экземпляром указанного класса или его подкласса.

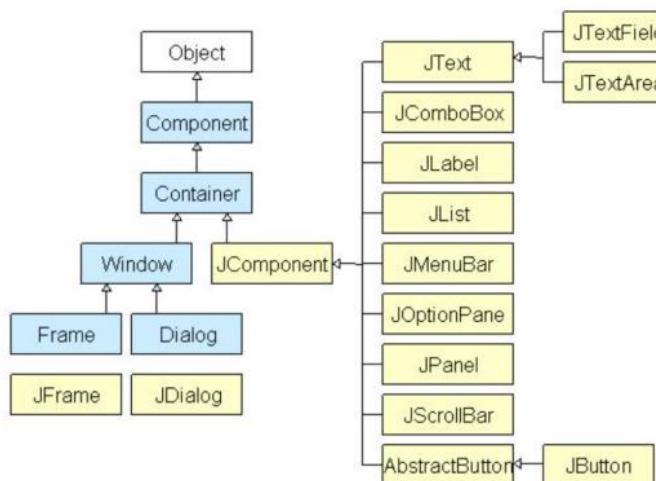
- Возвращает true, если obj является экземпляром класса ClassName, иначе возвращает false.

- Пример использования:

```
Object obj = "Hello"; // базовый тип Object
if (obj instanceof String) {
    String str = (String) obj;
    System.out.println("The object is an instance of String: " + str);
}
```

40. Графическая подсистема. Основы AWT, Swing components. Событийная модель при программировании GUI в ООП программах

Библиотека Java Swing построена на основе Java Abstract Widget Toolkit (AWT), более старого, зависящего от платформы инструментария графического интерфейса пользователя. Вы можете использовать простые программные компоненты Java с графическим интерфейсом пользователя, такие как кнопка, текстовое поле и т. д. Из библиотеки, и вам не нужно создавать компоненты с нуля. Схема иерархии классов графической библиотеки Swing представлена на рисунке



Классы-контейнеры — это классы, на которых могут быть другие компоненты. Итак, для создания графического интерфейса Java Swing нам понадобится хотя бы один объект-контейнер. Существует три типа контейнеров Java Swing:

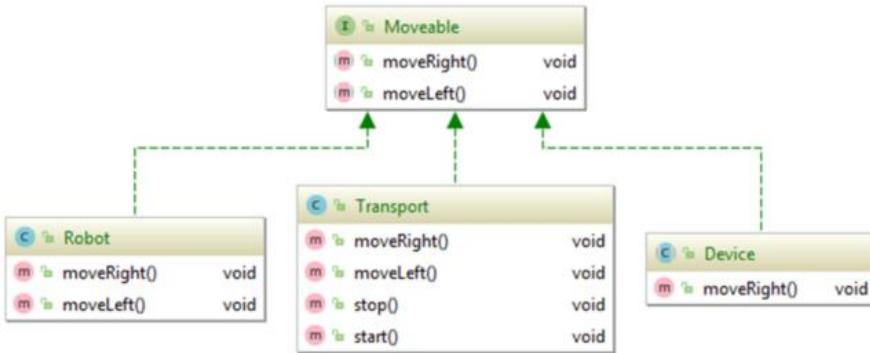
JPanel (Панель): это чистый контейнер, а не окно. Единственная цель Panel - организовать компоненты в окне. • JFrame (Фрейм): это полностью функционирующее окно со своим заголовком и значками. • JDialog (Диалог): это можно представить как всплывающее окно, которое высакивает, когда необходимо отобразить сообщение. Это не полностью функционирующее окно, как Frame. Рассмотрим последовательно все шаги по созданию графического интерфейса на Джава с помощью Swings Листинг 5.1 – Пример 1 программы с GUI

```
import javax.swing.*;
class FirstGui{
    public static void main(String args[]){
        //создаем фрейм окна с помощью конструктора
        //Конструктор берет параметр - название окна - это строка
        JFrame frame = new JFrame("My First GUI");
        // устанавливаем реакцию окна на закрытие по умолчанию
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //задаем свойства окна - его размеры в пикселях
        frame.setSize(300,300);
        //создаем кнопку с помощью конструктора класса JButton
        //конструктор берет параметр строку - название на кнопке
        JButton button = new JButton("Press");
        //добавляем кнопку ук окну
        frame.getContentPane().add(button);
        //делаем окно видимым
        frame.setVisible(true);
    }
}
```

41. Использование языка UML для проектирования и документирования объектно-ориентированных программ. Основные UML диаграммы для отображения отношений между классами в ООП программах

Преимущества интерфейсов Существуют по крайней мере три веские причины использовать интерфейсы: • они используется для достижения абстракции. • Благодаря интерфейсам мы можем поддерживать механизм множественного наследования. • они использовать для достижения слабой связанности кода (low coupling code)

UML диаграмма трех классов и одного интерфейса. Как видно из отношений на схеме, все три класса реализуют общий интерфейс.



42. ООП в Java. Понятие объекта. Что представляет собой Java приложение с точки зрения ООП. Основные характеристики объектов в Java.

43. Модификатор доступа или видимости в Джава, виды и использование. Использования this для доступа к компонентам класса.

Модификаторы – ключевые слова, которые Вы добавляете при инициализации для изменения значений. Язык Java имеет широкий спектр модификаторов, основные из них:

- модификаторы доступа;
- модификаторы класса, метода, переменной и потока, используемые не для доступа.

Чтобы использовать модификатор в Java, нужно включить его ключевое слово в определение класса, метода или переменной. Модификатор должен быть впереди остальной части оператора.

Java предоставляет ряд модификаторов доступа, чтобы задать уровни доступа для классов, переменных, методов и конструкторов. Существует четыре доступа:

- Видимый в пакете (стоит по умолчанию и модификатор не требуется).
- Видимый только для класса (private).
- Видимый для всех (public).
- Видимый для пакета и всех подклассов (protected).

Модификатор доступа по умолчанию – означает, что мы явно не объявляем модификатор доступа в Java для класса, поля, метода и т.д.

Переменная или метод, объявленные без модификатора контроля доступа доступны для любого другого класса в том же пакете. Поля в интерфейсе неявно являются public, static, final, а методы в интерфейсе по умолчанию являются public.

Модификатор private – методы, переменные и конструкторы, которые объявлены как private в Java могут быть доступны только в пределах самого объявленного класса.

Модификатор доступа private является наиболее ограничивающим уровнем доступа. **Класс и интерфейсы не могут быть private.**

Переменные, объявленные как private, могут быть доступны вне класса, если получающие их открытые (public) методы присутствуют в классе (ниже смотрите пример и пояснения).

Использование модификатора private в Java является основным способом, чтобы скрыть данные.

Модификатор public – класс, метод, конструктор, интерфейс и т.д. объявленные как public могут быть доступны из любого другого класса. Поэтому поля, методы, блоки, объявленные внутри public класса могут быть доступны из любого класса, принадлежащего к "вселенной" Java.

Тем не менее, если к public классу в другом пакете мы пытаемся получить доступ, то public класс приходится импортировать.

Благодаря наследованию классов, в Java все публичные (public) методы и переменные класса наследуются его подклассами.

Модификатор protected – переменные, методы и конструкторы, которые объявляются как protected в суперклассе, могут быть доступны только для подклассов в другом пакете или для любого класса в пакете класса protected.

Модификатор доступа protected в Java не может быть применен к классу и интерфейсам. Методы и поля могут быть объявлены как protected, однако методы и поля в интерфейсе не могут быть объявлены как protected.

Доступ protected дает подклассу возможность использовать вспомогательный метод или переменную, предотвращая неродственный класс от попыток использовать их.

Модификатор static – применяется для создания методов и переменных класса.

Ключевое слово static используется для создания переменных, которые будут существовать независимо от каких-либо экземпляров, созданных для класса. Только одна копия переменной static в Java существует вне зависимости от количества экземпляров класса. Статические переменные также известны как переменные класса. В Java локальные переменные не могут быть объявлены статическими (static).

Ключевое слово static используется для создания методов, которые будут существовать независимо от каких-либо экземпляров, созданных для класса.

В Java статические методы или методы static не используют какие-либо переменные экземпляра любого объекта класса, они определены. Методы static принимают все данные из параметров и что-то из этих параметров вычисляется без ссылки на переменные.

Переменные и методы класса могут быть доступны с использованием имени класса, за которым следует точка и имя переменной или метода.

Модификатор final – используется для завершения реализации классов, методов и переменных.

Переменная `final` может быть инициализирована только один раз. Ссылочная переменная, объявленная как `final`, никогда не может быть назначена для обозначения другого объекта. Однако данные внутри объекта могут быть изменены. Таким образом, состояние объекта может быть изменено, но не ссылки. С переменными в Java модификатор `final` часто используется со `static`, чтобы сделать константой переменную класса.

Метод `final` не может быть переопределен любым подклассом. Как упоминалось ранее, в Java модификатор `final` предотвращает метод от изменений в подклассе. Главным намерением сделать метод `final` будет то, что содержание метода не должно быть изменено стороне.

Основная цель в Java использования класса объявленного в качестве `final` заключается в предотвращении класс от быть подклассом. Если класс помечается как `final`, то ни один класс не может наследовать любую функцию из класса `final`.

Модификатор abstract – используется для создания абстрактных классов и методов.

Класс `abstract` не может создать экземпляр. Если класс объявлен как `abstract`, то единственная цель для него быть расширенным.

Класс не может быть одновременно `abstract` и `final`, так как класс `final` не может быть расширенным. Если класс содержит абстрактные методы, то он должен быть объявлен как `abstract`. В противном случае будет сгенерирована ошибка компиляции. Класс `abstract` может содержать как абстрактные методы, а также и обычные.

Метод `abstract` является методом, объявленным с любой реализацией. Тело метода (реализация) обеспечивается подклассом. Методы `abstract` никогда не могут быть `final` или `strict`.

Любой класс, который расширяет абстрактный класс должен реализовать все абстрактные методы суперкласса, если подкласс не является абстрактным классом. Если класс в Java содержит один или несколько абстрактных методов, то класс должен быть объявлен как `abstract`. Абстрактный класс не обязан содержать абстрактные методы. Абстрактный метод заканчивается точкой с запятой. Пример: `public abstract sample();`

Модификатор synchronized – используются в Java для потоков.

Ключевое слово `synchronized` используется для указания того, что метод может быть доступен только одним потоком одновременно. В Java модификатор `synchronized` может быть применен с любым из четырех модификаторов уровня доступа.

Переменная экземпляра отмеченная как **transient** указывает виртуальной машине Java (JVM), чтобы пропустить определенную переменную при сериализации объекта, содержащего её. Этот модификатор включён в оператор, что создает переменную, предшествующую класса или типа данных переменной.

Модификатор volatile – используются в Java для потоков.

В Java модификатор volatile используется, чтобы позволить знать JVM, что поток доступа к переменной всегда должен объединять свою собственную копию переменной с главной копией в памяти. Доступ к переменной volatile синхронизирует все кэшированные скопированные переменные в оперативной памяти. Volatile может быть применен только к переменным экземпляра, которые имеют тип объект или private. Ссылка на объект volatile может быть null.

this Ключевое слово является ссылкой на текущий объект.

Другой способ подумать об этом заключается в том, что this ключевое слово похоже на личное местоимение, которое вы используете для обозначения себя.

Думайте о this как о простом способе для типа сказать "my".

44. Чем отличаются static-метод класса от обычного метода класса. Можно ли вызвать static-метод внутри обычного метода?

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.

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. **Может** обращаться к **обычным** переменным класса.

45. Объявление и использование методов, объявленных с модификатором public static. Как вызвать обычный метод класса внутри static-метода?

В языке программирования Java ключевым словом **static** помечают члены (поля или методы), которые принадлежат классу, а не экземпляру этого класса.

Это означает, что какое бы количество объектов вы не создали, всегда будет создан только один член, доступный для использования всеми экземплярами класса.

Ключевое слово static применимо к переменным, методам, блокам инициализации, импорту.

Подобно статическим полям, статические методы также принадлежат классу, а не объекту, поэтому их можно вызывать без создания экземпляра класса, в котором они находятся. При этом следует помнить, что из статического метода можно получить доступ только к статическим переменным или к другим статическим методам.

Статические методы обычно используются для выполнения операции, не зависящей от создания экземпляра. При этом, они широко используются для создания служебных (утилитных) или вспомогательных классов, поскольку их можно вызывать без создания нового объекта этих классов.

Статические методы не могут напрямую обращаться к переменным экземпляра и методам экземпляра. Для этого им нужна ссылка на объект.

Статические методы могут обращаться ко всем статическим переменным и другим статическим методам.

В Java, чтобы вызвать обычный метод класса внутри static метода, необходимо передать ссылку на экземпляр класса в static метод. В результате этого ссылка на экземпляр будет доступна внутри static метода, и вы сможете вызывать обычные (нестатические) методы через эту ссылку.

46. Синтаксис объявления методов в Джава, тип возвращаемого значения, формальные параметры и аргументы. Методы с пустым списком параметров

Методы могут возвращать или не возвращать значения, могут вызываться с указанием параметров или без. Тип возвращаемых данных указывают при объявлении метода — перед его именем (void, int, string и тд.).

В заголовке метода сначала идут модификаторы, определяющие, на каких условиях он доступен для вызова. Вернёмся к заголовку: после модификаторов — возвращаемый тип, затем идет имя метода, в скобках — параметры.

В Java параметры и аргументы отличаются по своему назначению и использованию, хотя синтаксически они выглядят одинаково. **Формальные параметры** - это переменные, объявленные в объявлении метода или конструктора, которые указывают на тип данных, которые метод ожидает получить. **Фактические параметры, или аргументы**, - это конкретные значения, которые передаются в метод при его вызове.

Методы с пустым списком параметров просто не принимают параметров на вход, совершая все действия, не используя входящих параметров.

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

В Java геттер и сеттер — это два обычных метода, которые используются для получения значения поля класса или его изменения.

Следующий код является примером простого класса с *private*-переменной и реализованных, для доступа к ней извне, геттера и сеттера:

```
public class SimpleGetterSetter {  
    private int number;  
  
    public int getNumber() {  
        return number;  
    }  
  
    public void setNumber(int number) {  
        this.number = number;  
    }  
}
```

Поскольку *number* является *private*, то обратиться к ней напрямую за пределами данного класса не получится:

Чтобы таких проблем не было, внешний код должен вызывать геттер *getNumber()* и сеттер *setNumber()*, чтобы получить или обновить значение переменной:

Итак, **сеттер** — это метод, который изменяет (устанавливает; от слова *set*) значение поля. А **геттер** — это метод, который возвращает (от слова *get*) нам значение какого-то поля.

48. Может ли быть поле данных класса объявлено как с модификатором static и final одновременно и что это означает?

final - значит неизменяемая, если быть точнее, то разрешается только одна операция присвоения

static - означает единая для всех экземпляров класса.

```
class Foo {  
    static final int FOO1=1;  
    final int foo2;  
}
```

В данном примере:

- для всех экземпляров класса *Foo* переменная *FOO1* всегда будет равна 1
- переменная *foo2* - может быть разной для разных экземпляров класса *Foo*

При этом **оба безусловно неизменяемые**.

Короче говоря: может быть и *static* и *final* одновременно, это означает, что для всех объектов она будет всегда одинаковая и неизменяемая, в то время как если мы убираем *static*, она может иметь разные значения для разных объектов,

но при этом она 1 раз получает значение и больше не меняется никогда. А если убрать final, то переменная просто станет статичной для всех объектов.

Тема 3. Реализация наследования в программах на Джаве

49. Наследование, виды наследования и его реализация в Java и C++

Наследование в объектно-ориентированных языках программирования, таких как Java и C++, представляет собой механизм, позволяющий новому классу (подклассу) наследовать свойства и методы от родительского класса (суперкласса).

В Java и C++ есть несколько видов наследования:

1. Наследование в одиночном (или простом) виде:

- Java:

```
public class SubClass extends SuperClass {  
    // тело подкласса  
}
```

- C++:

```
class SubClass : public SuperClass {  
    // тело подкласса  
};
```

2. Множественное наследование (только в C++):

В C++ разрешено множественное наследование, когда один подкласс наследует свойства и методы сразу от нескольких суперклассов.

В обоих языках подкласс может расширить функциональность родительского класса, добавив собственные методы и свойства или переопределив методы суперкласса. Примеры реализации наследования в Java и C++:

Java:

```
class Vehicle {  
    void move() {  
        System.out.println("Moving...");  
    }  
}  
  
class Car extends Vehicle {  
    void beep() {  
        System.out.println("Beep beep!");  
    }  
}
```

C++:

```
class Vehicle {
public:
    void move() {
        std::cout << "Moving..." << std::endl;
    }
};

class Car : public Vehicle {
public:
    void beep() {
        std::cout << "Beep beep!" << std::endl;
    }
};
```

В обоих примерах Car наследует move() от Vehicle, а также имеет свой собственный метод beep().

Наследование позволяет создавать иерархию классов, делая код более гибким и повторно используемым, а также способствует применению принципа DRY (Don't Repeat Yourself - не повторяйся) в программировании.

50. Расширение классов. Порядок создания экземпляра дочернего класса.

Наследование (inheritance) — механизм, который позволяет описать новый класс на основе существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом. По сути, наследование в Java имеет один родительский класс (называемый super), и второй класс как дочерний элемент родителя (подкласс).

1. Для создания наследования нужно создать подкласс (дочерний) из класса Java **super** (родительского), для этого используется ключевое слово **extends** (в переводе - расширить) + имя родительского класса, который вы хотите расширить.

```
package firm;

public class Bonuses {

    private String bonus;
}

package firm;

public class Bonuses extends Sales {

    private String bonus;
}
```

2. Так же, как и для класса **Sales**, мы можем создать конструктор для этого нового класса **Bonuses**. Когда мы создаем объект класса, Java в первую очередь вызывает наш конструктор.

Но! может быть вызван только один конструктор. Если мы вызовем новый конструктор из класса **Bonuses**, все те значения по умолчанию, которые мы устанавливали для экземпляров класса **Sales**, не будут установлены. Чтобы обойти это, в Java есть ключевое слово **super**. Оно делает вызов конструктора из класса **super**. Добавьте следующий конструктор в свой класс с наследованием **Bonuses**:

```
package firm;

public class Bonuses extends Sales {

    private String bonus;

    Bonuses() {
        super();
        bonus = "Бонус не предоставляется";
    }
}
```

51. Наследование в Джава. Вид наследования и синтаксис Ключевое слово extends

В Java наследование реализуется с помощью ключевого слова **extends**. Ключевое слово **extends** используется для создания подкласса (дочернего класса), который наследует свойства и методы родительского класса (суперкласса).

Синтаксис наследования в Java с использованием ключевого слова **extends** выглядит следующим образом:

```
class SuperClass {
    // поля и методы суперкласса
}

class SubClass extends SuperClass {
    // поля и методы подкласса
}
```

Здесь **SubClass** является дочерним классом, который наследует свойства и методы от **SuperClass**.

Пример использования наследования в Java:

```
class Animal {
    void sound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Bark");
    }

    void wagTail() {
        System.out.println("Tail is wagging");
    }
}
```

В данном примере класс Dog наследует метод sound от класса Animal, но также добавляет собственный метод wagTail.

Наследование позволяет создавать иерархию классов, обобщать поведение и свойства, уменьшать повторное использование кода и делать программу более структурированной.

52. Что означает перегрузка метода в Java (overload) и переопределение метода в Java (override)? В чем разница?

Перегрузка методов — это возможность создавать несколько методов с одинаковыми именами, но с различными наборами параметров. Таким образом, при вызове метода можно передать ему разное количество аргументов или аргументы разных типов. Компилятор или интерпретатор определяет, какой именно метод нужно вызвать на основе типов аргументов, переданных при вызове. Это позволяет создавать более удобный интерфейс для работы с классом, предоставляя различные варианты использования метода в зависимости от контекста.

В языках программирования, поддерживающих перегрузку методов, различные параметры, которые могут быть перегружены, включают в себя:

- **Число параметров:** Методы могут быть перегружены по количеству параметров. Например, класс может иметь несколько методов с одним и тем же именем, но с разным числом параметров. Таким образом, можно создать методы для разной логики выполнения в зависимости от передаваемых параметров.
- **Типы параметров:** Методы могут быть перегружены по типу параметров. Например, класс может иметь несколько методов с одним и тем же именем, но с разными типами параметров. Таким образом, можно создать методы, работающие с разными типами данных.
- **Порядок параметров:** Методы могут быть перегружены по порядку параметров. Например, класс может иметь несколько методов с одним и тем же именем, но с разным порядком параметров. Таким образом, можно

создать методы, для которых порядок передачи аргументов имеет значение.

При перегрузке методов важно учесть, что тип возвращаемого значения не может использоваться для разрешения перегрузки методов. Во время вызова перегруженного метода компилятор будет определять, какой конкретно метод должен быть вызван, на основе типа аргументов, переданных в метод. Если типы аргументов соответствуют более чем одному перегруженному методу, то компилятор выбирает более специфичный метод на основе наиболее точного совпадения с типами аргументов.

Также перегрузка методов не зависит от модификаторов доступа (public, private, protected). Два метода с одинаковым именем и параметрами, но разными модификаторами доступа, не будут считаться перегруженными.

Переопределение методов — это изменение реализации метода унаследованного от родительского класса в дочернем классе. Таким образом, при обращении к методу из дочернего класса будет использоваться новая реализация. При этом, сигнатура метода (имя и параметры) остается такой же, что позволяет гарантировать наличие определенного функционала в дочернем классе, а также обеспечивает возможность полиморфного вызова метода.

Условия для переопределения метода:

- Метод в дочернем классе должен иметь такое же имя, как и метод в родительском классе.
- Методы в родительском и дочернем классах должны иметь одинаковую сигнатуру.
- Уровень доступа не может быть более ограниченным, чем уровень доступа переопределенного метода. Например, если метод суперкласса объявлен public, то переопределяемый метод в подклассе не может быть private или protected.
- Метод в дочернем классе должен быть помечен модификатором доступа @Override.
- Методы, которые объявлены как final, не могут быть переопределены.
- Статические методы, которые объявлены как static, не могут быть переопределены, но могут быть повторно объявлены.

Если данные условия не соблюдаются, то метод считается перегруженным, а не переопределенным.

53. Абстрактные классы в Джава и абстрактные методы класса. Вложенные и анонимные классы.

Абстракция - это *принцип ООП*, согласно которому при проектировании классов и создании объектов необходимо выделять только главные свойства сущности, и отбрасывать второстепенные.

Абстрактным называется такой класс, экземпляр которого нельзя создать сам по себе – он служит основой для других классов. При объявлении такого класса в его определении ставится ключевое слово `abstract`.

```
public abstract class Shape {  
    // class body  
}
```

!! Абстрактные классы могут содержать как абстрактные, так и неабстрактные методы, а также переменные экземпляров, конструкторы и другие члены. Правда, напрямую инстанцировать абстрактные классы нельзя, а это означает, что их не получится напрямую использовать для создания объектов.

!! У абстрактных классов могут быть конструкторы, но их нельзя вызывать прямо из субклассов. Напротив, конструктор конкретного субкласса обязан явно вызывать конструктор соответствующего суперкласса. Для этого вызывается либо `super()`, либо конкретный конструктор суперкласса.

Абстрактные классы часто применяются для определения общего интерфейса или такого поведения, которое может совместно использоваться сразу множеством субклассов. *Например*, представьте, что вы проектируете игру, в которой используются различные фигуры – скажем, круги, квадраты и треугольники. У всех этих фигур есть определённые общие свойства, например, периметр и площадь. Но у них могут быть и уникальные характеристики, такие как количество сторон и радиус.

Чтобы представить эти фигуры в вашей игре, можно создать абстрактный класс `Shape`, определяющий общее поведение для всех фигур:

```
public abstract class Shape {  
    protected int x, y;  
  
    public Shape(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}
```

В этом примере класс `Shape` содержит переменные экземпляра для координат `x` и `y` некоторой фигуры, а также абстрактные методы для вычисления площади и периметра. Определяя эти методы как абстрактные, мы позволяем каждому конкретному субклассу реализовать их по-своему.

Например, можно было бы создать субкласс `Circle`, расширяющий класс `Shape` и реализующий

собственные версии методов `getArea()` и `getPerimeter()`:

```
public class Circle extends Shape {  
    private double radius;  
  
    public Circle(int x, int y, double radius) {  
        super(x, y);  
        this.radius = radius;  
    }  
  
    public double getArea() {  
        return Math.PI * radius * radius;  
    }  
  
    public double getPerimeter() {  
        return 2 * Math.PI * radius;  
    }  
}
```

Используя абстрактные классы таким образом, мы создаём гибкую и расширяемую систему, позволяющую представлять в нашей игре различные фигуры, в то же время выдерживая общий интерфейс для всех субклассов, обеспечивающий общность их поведения.

Вложенные классы – это классы, которые используются для более подробного описания каких-то деталей нашего внешнего класса

```
1  public class Airplane {  
2      private String name, id, flight;  
3      private Wing leftWing = new Wing("Red", "X3"), rightWing = new Wing("Blue", "X3");  
4  
5      public Airplane(String name, String id, String flight) {  
6          this.name = name;  
7          this.id = id;  
8          this.flight = flight;  
9      }  
10  
11     private class Wing {  
12         private String color, model;  
13  
14         private Wing(String color, String model) {  
15             this.color = color;  
16             this.model = model;  
17         }  
18  
19         // getters/setters  
20     }  
21  
22     // getters/setters
```

Так мы создали нестатический вложенный класс `Wing` (крыло) внутри класса `Airplane` (самолет), и добавили две переменные – левое крыло и правое крыло. И у каждого крыла есть свои свойства (цвет, модель), которые мы можем изменять. Так можно укомплектовывать структуры столько, сколько нужно.

Особенности нестатических вложенных классов Java:

1. Они существуют только у объектов, потому для их создания нужен объект. Другими словами: мы укомплектовали наше крыло так,

чтобы оно было частью самолета, потому, чтобы создать крыло, нам нужен самолет, иначе оно нам не нужно.

2. Внутри Java класса не может быть статических переменных. Если вам нужны какие-то константы или что-либо еще статическое, выносить их нужно во внешний класс. Это связано с тесной связью нестатического вложенного класса с внешним классом.
3. У класса полный доступ ко всем приватным полям внешнего класса. Данная особенность работает в две стороны.
4. Можно получить ссылку на экземпляр внешнего класса. Пример: Airplane.this – ссылка на самолет, this – ссылка на крыло.

Анонимные классы - представляют собой интересную особенность Java, позволяющую создавать классы без явного имени, которые могут быть использованы для реализации интерфейсов или создания подклассов существующих классов непосредственно внутри кода.

Пример кода без анонимных классов (используем интерфейс и классы, наследующие его):

```
1 public interface MonitoringSystem {  
2  
3     public void startMonitoring();  
4 }  
  
1 public class GeneralIndicatorsMonitoringModule implements MonitoringSystem {  
2  
3     @Override  
4     public void startMonitoring() {  
5         System.out.println("Мониторинг общих показателей стартовал!");  
6     }  
7 }  
8  
9  
10 public class ErrorMonitoringModule implements MonitoringSystem {  
11  
12     @Override  
13     public void startMonitoring() {  
14         System.out.println("Мониторинг отслеживания ошибок стартовал!");  
15     }  
16 }  
17  
18  
19 public class SecurityModule implements MonitoringSystem {  
20  
21     @Override  
22     public void startMonitoring() {  
23         System.out.println("Мониторинг безопасности стартовал!");  
24     }  
25 }
```

Пример кода с анонимными классами (используем интерфейс и 3 класса без имени):

```

1 public class Main {
2
3     public static void main(String[] args) {
4
5         MonitoringSystem generalModule = new MonitoringSystem() {
6             @Override
7             public void startMonitoring() {
8                 System.out.println("Мониторинг общих показателей стартовал!");
9             }
10        };
11
12
13        MonitoringSystem errorModule = new MonitoringSystem() {
14            @Override
15            public void startMonitoring() {
16                System.out.println("Мониторинг отслеживания ошибок стартовал!");
17            }
18        };
19
20        MonitoringSystem securityModule = new MonitoringSystem() {
21            @Override
22            public void startMonitoring() {
23                System.out.println("Мониторинг безопасности стартовал!");
24            }
25        };
26
27        generalModule.startMonitoring();
28        errorModule.startMonitoring();
29        securityModule.startMonitoring();
30    }
31 }
32 }
```

Ключевые особенности анонимных классов:

1. Отсутствие имени: Анонимные классы не имеют названия. Они создаются и использованы на месте, где они нужны, что делает их удобными для обработки небольших кусков кода без необходимости создания отдельного класса.
2. Использование для интерфейсов и подклассов: Анонимные классы обычно используются для реализации интерфейсов или создания подклассов существующих классов без явного создания нового класса.

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

В Java существуют два вида наследования: наследование классов (class inheritance) и наследование интерфейсов (interface inheritance).

1. Наследование классов (class inheritance):

- Один класс может наследовать свойства (поля и методы) другого класса.
- Наследующий класс называется подклассом (subclass) или производным классом (derived class), а класс, от которого наследуется, называется суперклассом (superclass) или базовым классом (base class).
- Подкласс наследует все доступные открытые (public) и защищенные (protected) свойства и методы суперкласса, а также может добавлять свои собственные свойства и методы.

- Наследование классов в Java реализуется с помощью ключевого слова "extends".

Пример наследования классов:

```
class Vehicle {  
    void start() {  
        System.out.println("The vehicle starts");  
    }  
}  
  
class Car extends Vehicle {  
    void accelerate() {  
        System.out.println("The car accelerates");  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Car car = new Car();  
        car.start(); // Метод start() унаследован от класса Vehicle  
        car.accelerate();  
    }  
}
```

2. Наследование интерфейсов (interface inheritance):

- Интерфейс определяет набор методов, которые должны быть реализованы классами, которые реализуют этот интерфейс.
- Класс может реализовать несколько интерфейсов и унаследовать их методы.
- Наследование интерфейсов позволяет создавать иерархию интерфейсов и обеспечивает гибкость при определении общего поведения для различных классов.
- Наследование интерфейсов в Java реализуется с помощью ключевого слова "implements".

Пример наследования интерфейсов:

```
interface Animal {  
    void eat();  
}  
  
interface Pet {  
    void play();  
}  
  
class Cat implements Animal, Pet {  
    public void eat() {  
        System.out.println("The cat eats");  
    }  
}
```

```

    }

    public void play() {
        System.out.println("The cat plays");
    }
}

public class Main {
    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.eat(); // Метод eat() реализован из интерфейса Animal
        cat.play(); // Метод play() реализован из интерфейса Pet
    }
}

```

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

Компоненты, которые наследуются при наследовании классов в Java:

1. Поля (переменные): Подкласс наследует все поля (переменные) суперкласса, если они не являются приватными (private). Поля суперкласса, объявленные как private, не наследуются и недоступны в подклассе.
2. Методы: Подкласс наследует все открытые (public) и защищенные (protected) методы суперкласса. Открытые методы могут быть вызваны из подкласса, а защищенные методы могут быть вызваны из подкласса и классов из того же пакета.
3. Внутренние классы: Если суперкласс содержит внутренние классы, они также наследуются подклассом.

Компоненты, которые не наследуются при наследовании классов в Java:

1. Конструкторы: Подкласс не наследует конструкторы суперкласса. Однако, при создании объекта подкласса, конструктор суперкласса будет вызван автоматически для инициализации унаследованных полей.
2. Приватные (private) поля и методы: Приватные поля и методы суперкласса не наследуются подклассом и недоступны в нем.
3. Статические (static) поля и методы: Статические поля и методы суперкласса не наследуются подклассом, а доступ к ним осуществляется через имя суперкласса.
4. Финальные (final) методы: Финальные методы суперкласса не могут быть переопределены в подклассе.

56. 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

57. Класс Object, его методы, их назначение. Иерархия классов в Java.

58. Наследование. Использование ключевых слов this и super. Пример использования в языках C++ и Java

Наследование - это механизм в объектно-ориентированных языках программирования, который позволяет создавать новые классы на основе уже существующих классов. При использовании наследования, новый класс, называемый подклассом или производным классом, наследует свойства и методы от другого класса, называемого суперклассом или базовым классом.

Ключевое слово "this" используется для обращения к текущему объекту внутри класса. Оно позволяет ссылаться на текущий объект и использовать его свойства и методы. Ключевое слово "super" используется для обращения к свойствам и методам суперкласса из подкласса. Оно позволяет вызывать конструкторы суперкласса, обращаться к его полям и вызывать его методы.

Пример использования ключевых слов this и super в языке C++:

```
```cpp
#include <iostream>
using namespace std;

class Animal {
public:
 Animal() {
 cout << "Animal constructor" << endl;
 }
};

class Dog : public Animal {
public:
 Dog() : Animal() {
 cout << "Dog constructor" << endl;
 }
}
```

```
};

int main() {
 Dog dog;
 return 0;
}
```

В этом примере класс Dog наследуется от класса Animal с помощью ключевого слова "public". В конструкторе класса Dog используется ключевое слово "super" для вызова конструктора суперкласса Animal. При выполнении программы будет выведено:

```
Animal constructor
Dog constructor
```

Пример использования ключевых слов this и super в языке Java:

```
class Animal {
 Animal() {
 System.out.println("Animal constructor");
 }
}

class Dog extends Animal {
 Dog() {
 super();
 System.out.println("Dog constructor");
 }
}

public class Main {
 public static void main(String[] args) {
 Dog dog = new Dog();
 }
}
```

В этом примере класс Dog наследуется от класса Animal с помощью ключевого слова "extends". В конструкторе класса Dog используется ключевое слово "super" для вызова конструктора суперкласса Animal. При выполнении программы будет выведено:

```
Animal constructor
```

Dog constructor

59. Паттерны проектирования программ. Паттерн Фабрика.

60. Паттерны проектирования программ. Паттерн Фабричный метод.

61. Расширение классов в Джава. Переопределение методов. Сокрытие полей данных.

62. Паттерны проектирования программ. Паттерн Observer и модель MVC

Тема 4. Полиморфизм в Джава. Работа со строками. Интерфейсы.

63. Интерфейсы. Общий синтаксис и расширение. Пустые интерфейсы. Реализация и применение. Сравнение с абстрактными классами.

64. Обработка строк в Java. Класс StringBuffer. Класс StringBuilder

В Java классы **StringBuffer** и **StringBuilder** предназначены для работы с изменяемыми строками, в отличие от класса **String**, который является неизменяемым. Оба класса предоставляют методы для изменения содержимого строки без создания новых объектов. Однако есть некоторые различия между ними.

**StringBuffer:**

- **StringBuffer** является потокобезопасным (thread-safe) классом, что означает, что его методы синхронизированы и могут быть безопасно использованы в многопоточных приложениях.
- Из-за синхронизации **StringBuffer** может быть менее эффективным в однопоточных приложениях по сравнению с **StringBuilder**.
- Пример использования **StringBuffer**:

```
StringBuffer stringBuffer = new StringBuffer("Hello");
stringBuffer.append(" World");
System.out.println(stringBuffer.toString());
```

### StringBuilder:

- **StringBuilder** представляет собой несинхронизированный аналог **StringBuffer**.
- По сравнению с **StringBuffer**, **StringBuilder** может обеспечивать более высокую производительность в однопоточных приложениях.
- Пример использования **StringBuilder**:

```
StringBuilder stringBuilder = new StringBuilder("Hello");
stringBuilder.append(" World");
System.out.println(stringBuilder.toString());
```

(Как видите, классы почти идентичны, просто буффер для многопоточных приложений, а билдер для однопоточных)

Оба класса имеют методы вставки **insert(offset, N\*)**, добавления **append(N\*)**, замены **replace(start, end, string)**, а также другие, менее используемые методы.

\* - N - любой примитивный тип данных, строка или **StringBuffer/StringBuilder**.  
Методы **insert** и **append** перегружены.

## 65. Работа со строками в Java, строковый кэш. Операция конкатенации строк

## 66. Интерфейс Comparable и Comparator. Использование интерфейсных ссылок для написания обобщенных алгоритмов

**Comparable** и **Comparator** в Java предоставляют способы сравнения объектов для упорядочивания. Эти интерфейсы особенно полезны при сортировке коллекций объектов.

### 1. Comparable интерфейс:

- Реализуя интерфейс **Comparable**, объекты становятся сравнимыми с помощью их собственного метода **compareTo**.

```
public class Person implements Comparable<Person> {
 private String name;
 private int age;

 // constructors, getters, setters...

 @Override
 public int compareTo(Person other) {
 return Integer.compare(this.age, other.age);
 }
}
```

## 2. Comparator интерфейс:

- **Comparator** предоставляет более гибкий подход, позволяя определить сравнение объектов вне их класса.

```
public class PersonComparator implements Comparator<Person> {
 @Override
 public int compare(Person p1, Person p2) {
 return p1.getName().compareTo(p2.getName());
 }
}
```

## 3. Использование интерфейсных ссылок для написания обобщенных алгоритмов:

С использованием интерфейсных ссылок и лямбда-выражений можно создавать обобщенные алгоритмы для сортировки и сравнения.

```
List<Person> people = Arrays.asList(/*...*/);
people.sort(Comparator.comparing(Person::getName));
```

*Здесь мы создаём список людей (List<Person> people), а затем сортируем его методом sort. В метод мы передаем компаратор, который сортирует людей по методу getName класса Person.*

67. Понятие сортировки массивов. Сортировка пузырьком. Сортировка вставками. Использование полиморфизма (ООП) для программирования алгоритмов сортировок в массивах и коллекциях

68. Понятие поиска в массивах. Последовательный поиск. Сортировка методом прямого выбора. Использование полиморфизма (ООП) для программирования алгоритмов поиска в массивах и коллекциях

69. Объявление и инициализация переменных типа String. Операция конкатенации строк и ее использование

70. При создании объектов строк с помощью класса StringBuffer, например StringBuffer strBuffer = new StringBuffer(str) можно ли использовать операцию конкатенации строк или необходимо использовать методы класса StringBuffer

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

72. Понятие и объявление интерфейсов в Джава. Может ли один класс реализовывать несколько интерфейсов?

73. Что входит в состав интерфейса (какие компоненты может содержать интерфейс)? Может ли интерфейс наследоваться от другого интерфейса?

74. Интерфейсные ссылки и их использование в Джава

Полиморфизм является одним из принципов ООП, и позволяет нам создавать универсальные конструкции программного обеспечения. Полиморфизм в Джава реализуется следующими способами:

- использование наследования для создания полиморфных ссылки
- использование интерфейсов для создания полиморфных ссылок На основе использования полиморфизма мы можем реализовывать универсальные алгоритмы, например, алгоритмы сортировки и поиска.

Использование **интерфейсных ссылок**, которые являются потенциально полиморфными дает нам большое преимущество. Используя этот механизм

языка Джава, мы можем писать универсальные алгоритмы для обработки различных типов данных.

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

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

**Тема 5. Основные принципы и типы исключительных ситуаций.**

77. Понятие исключительной ситуации, причины возникновения, механизм обработки. Классификация исключений. Исключения, классификация и использование исключений. Генерация (порождение исключений).

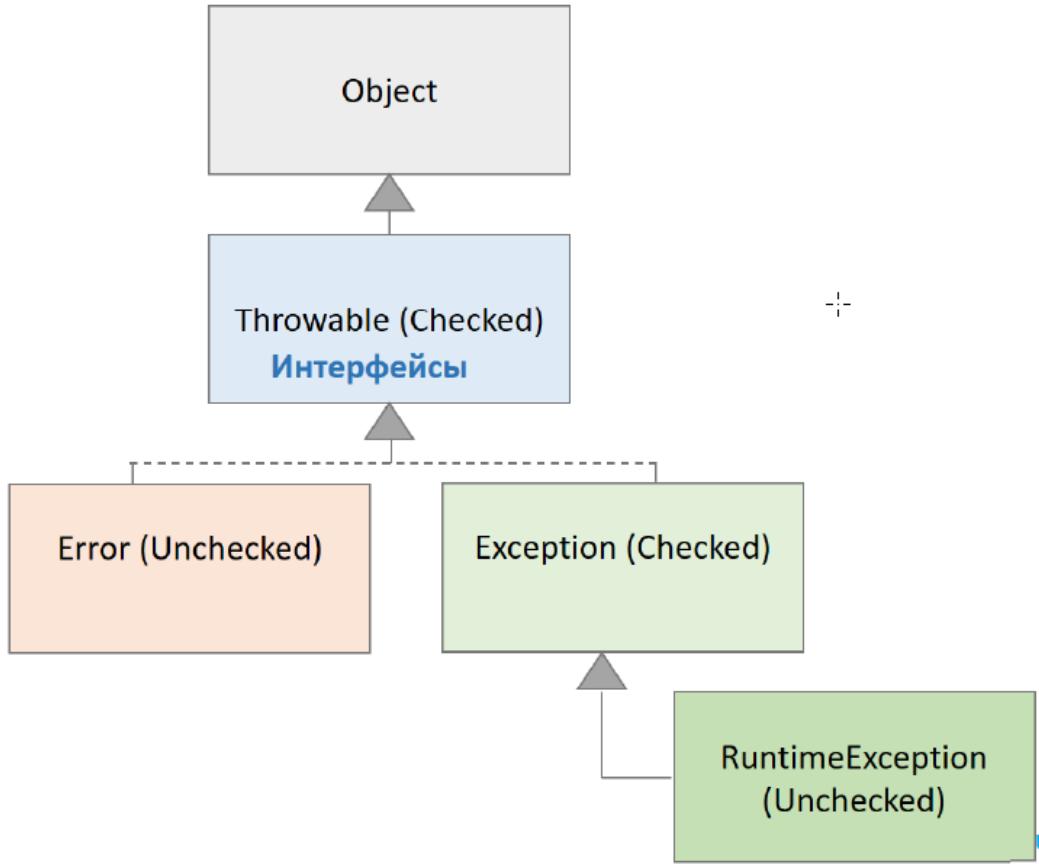
**Исключительная ситуация** - момент, когда выполняемая программа генерирует ошибку, не позволяющую продолжение работы программы в штатном режиме. Например деление на ноль, невозможность считать данные с носителя, ошибка работы с API.

Если исключение игнорируется программой, то программа будет завершена аварийно и производится вывод соответствующего сообщения.

Сообщение включает в себя трассировку стека вызовов:

- указывает на строку кода, где произошло исключение
- показывает след вызова метода, который привел к попытки выполнения ошибочной строчки кода

Также программист может сам обрабатывать исключения применяя блоки try catch finally



Ошибки в джаве можно разделить на два вида, checked(Exceptions) и unchecked(Error)

Под checked ошибками подразумеваются те виды ошибок, который предусмотрены и являются ошибкой программиста, их нужно обрабатывать. Например деление на ноль.

Под unchecked ошибками понимается серьезная ошибка, которую согласно спецификации Java, не следует пытаться обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM.

Например, исключения такого рода возникают, если закончилась память, доступная виртуальной машине. Программа дополнительную память всё равно не сможет обеспечить для JVM. Все исключения делятся на 4 вида, которые на самом деле являются классами, унаследованными друг от друга.

## Класс Throwable

Самым базовым классом для всех исключений является класс Throwable. В классе Throwable содержится код, который записывает текущий стек-трейс вызовов функций в массив.

## Класс Error

Следующим классом исключений является класс Error — прямой наследник класса Throwable. Объекты типа Error (и его классов-наследников)

создает Java-машина в случае каких-то серьезных проблем. Например, сбой в работе, нехватка памяти, и т.д.

Обычно вы как программист ничего не можете сделать в ситуации, когда в программе возникла ошибка типа Error: слишком серьезна такая ошибка. Все, что вы можете сделать — уведомить пользователя, что программа аварийно завершается или записать всю известную информацию об ошибке в лог программы.

## Класс Exception

Исключения типа Exception (и RuntimeException) — это обычные ошибки, которые возникают во время работы многих методов. Цель каждого выброшенного исключения — быть захваченным тем блоком catch, который знает, что нужно сделать в этой ситуации.

Когда какой-то метод не может выполнить свою работу по какой-то причине, он сразу должен уведомить об этом вызывающий метод, выбрасывая исключение соответствующего типа.

## Класс RuntimeException

RuntimeException — это разновидность (подмножество) исключений Exception. Можно даже сказать, что RuntimeException — это облегченная версия обычных исключений (Exception): на такие исключения налагается меньше требований и ограничений

Можно создавать и собственные исключение, просто унаследовав от любого из видов исключений, указанных выше

```
class ИмяКласса extends RuntimeException
{
}
```

Можно также генерировать исключения самостоятельно, использовав ключевое слово throw

Допустим у нас есть программа которая считывает файл, который должен обязательно быть длины 1044 символа, но вдруг оказалось что файл оказался длиной 733 символа, в такой ситуации нам нужно выбросить ошибку.

Используем такой вид исключения EOFException. Тогда в программе в случае если у нас окажется длина меньше 1044, то мы должны сделать следующее:

```
throw new EOFException();
```

## 78. Служебное слово *throw* и его использование при определении методов. В каком случае программа должна использовать оператор *throw*?

Можно также генерировать исключения самостоятельно, использовав ключевое слово **throw**

Допустим у нас есть программа которая считывает файл, который должен обязательно быть длины 1044 символа, но вдруг оказалось что файл оказался длиной 733 символа. С точки зрения языка ничего страшного не произошло же, файл считался, он дошел до конца, но для работы нашей программы это не катит, нам обязательно нужно, чтобы файл был длиной 1044 символа. В такой ситуации нам нужно выбросить ошибку.

Используем такой вид исключения EOFException

Тогда в программе в случае если у нас окажется длина меньше 1044, то мы должны сделать следующее:

```
throw new EOFException();
```

## 79. Создание собственных классов исключений

Можно создавать и собственные исключения, просто создав класс, который наследуется от классов исключений.

```
class ИмяКласса extends RuntimeException
{
}
}
```

## 80. Блок *try/catch/finally*, его предназначение и особенности

При написании программ всегда будут случаи, когда будут происходить непредсказуемые исключения, результирующие в прекращении работы программы с ошибкой.

Но чаще всего предусматривается, что программе следует сделать что-то при встрече ошибки, например записать стек вызовов в какой-нибудь лог, чтобы потом было проще разобраться в чем ошибка. Или же это может быть предусмотренная проверка введенных пользователем данных.

Для таких случаев были созданы блоки try... catch... finally

Принцип прост

```
import java.util.Scanner;

public class Main {
 public static void main(String[] args) {
 Scanner sc = new Scanner(System.in);
 int divide = sc.nextInt();
 int divisible = 5;
 //У нас есть код, который возможно допустит ошибку
 try{
 System.out.println(5 / divide);
 }
 //В случае возникновения этой ошибки мы ее обрабатываем
 catch (ArithmetricException e){
 System.out.println("Logging the exception");
 }
 //Этот блок кода произойдет в любом случае
 finally {
 System.out.println("Программа закончила работать");
 }
 }
}
```

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

При возникновении ошибки в процессе выполнения программы исполняющая среда JVM создает объект нужного типа из иерархии исключений Java – множества возможных исключительных ситуаций, унаследованных от общего «предка» – класса Throwable.

Исключительные ситуации, возникающие в программе, можно разделить на две группы:

Ситуации, при которых восстановление дальнейшей нормальной работы программы невозможно

Восстановление возможно.

К первой группе относят ситуации, когда возникают исключения, унаследованные из класса Error. Это ошибки, возникающие при выполнении программы в результате сбоя работы JVM, переполнения памяти или сбоя системы. Обычно они свидетельствуют о серьезных проблемах, устраниить которые программными средствами невозможно. Такой вид исключений в Java относится к неконтролируемым (unchecked) на стадии компиляции.

К этой группе также относят RuntimeException – исключения, наследники класса Exception, генерируемые JVM во время выполнения программы. Часто причиной возникновения их являются ошибки программирования. Эти

исключения также являются неконтролируемыми (unchecked) на стадии компиляции, поэтому написание кода по их обработке не является обязательным.

Ко второй группе относят исключительные ситуации, предвидимые еще на стадии написания программы, и для которых должен быть написан код обработки. Такие исключения являются контролируемыми (checked). Основная часть работы разработчика на Java при работе с исключениями – обработка таких ситуаций.

82. Код ниже вызовет ошибку: Exception <...> java.lang.ArrayIndexOutOfBoundsException: 4: Что она означает?

```
import java.util.Scanner;

public class Main {
 public static void main(String[] args) {
 String[] arr = {"Раз", "Два"};
 System.out.println(arr[2]);
 }
}
```

C:\Users\mrval\jdks\openjdk-20.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2023.2.1\lib\idea\_rt.jar" -Dfile.encoding=UTF-8 Main
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2 out of bounds for length 2
at Main.main(Main.java:6)

Ну, мы попытались обратиться к массиву по индексу, которого в массиве нет, все.

83. Контролируемые исключения (checked) и неконтролируемые исключения (unchecked) и ошибки, которые они обрабатывают

К первой группе относят ситуации, когда возникают исключения, унаследованные из класса Error. Это ошибки, возникающие при выполнении программы в результате сбоя работы JVM, переполнения памяти или сбоя системы. Обычно они свидетельствуют о серьезных проблемах, устранить которые программными средствами невозможно. Такой вид исключений в Java относится к неконтролируемым (unchecked) на стадии компиляции.

К этой группе также относят RuntimeException – исключения, наследники класса Exception, генерируемые JVM во время выполнения программы. Часто причиной возникновения их являются ошибки программирования. Эти исключения также являются неконтролируемыми (unchecked) на стадии компиляции, поэтому написание кода по их обработке не является обязательным.

Ко второй группе относят исключительные ситуации, предвидимые еще на стадии написания программы, и для которых должен быть написан код обработки. Такие исключения являются контролируемыми (checked). Основная часть работы разработчика на Java при работе с исключениями – обработка таких ситуаций.

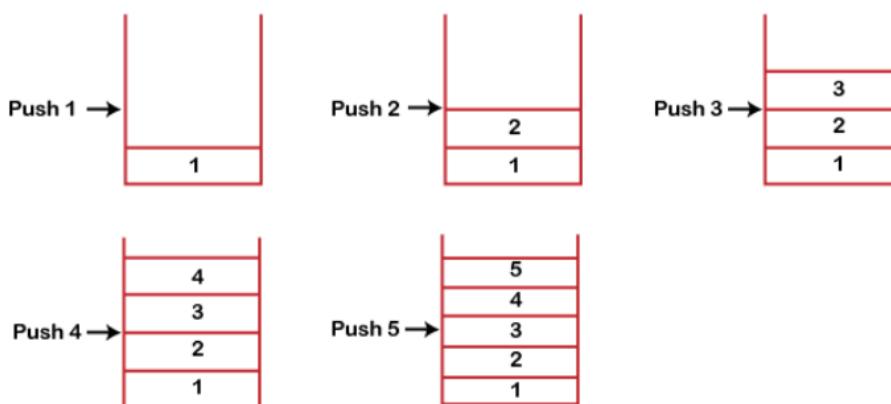
## 84. Как реализуются принципы ООП в Java при создании исключений? Порядок выполнения операторов при обработке блока блока try...catch

## Тема 6. Абстрактные типы данных Джениерики и использование контейнерных классов в Джава

### 85. Абстрактный тип данных Stack (стек) в Джава

**Стек** — это линейная структура данных, которая следует принципу LIFO (Last In First Out). Стек имеет один конец, куда мы можем добавлять элементы и извлекать их оттуда, в отличие от очереди, которая имеет два конца (спереди и сзади). Стек содержит только один указатель top (верхушка стека), указывающий на самый верхний элемент стека. Всякий раз, когда элемент добавляется в стек, он добавляется на вершину стека, и этот элемент может быть удален только из стека только сверху. Другими словами, стек можно определить как контейнер, в котором вставка и удаление элементов могут выполняться с одного конца, известного как вершина стека. Примеры стеков – стопка тарелок или книг.

**Стек** — это структура данных, в которой строго определен порядок вставки и удаления элементов, и этот порядок может быть LIFO или FILO.

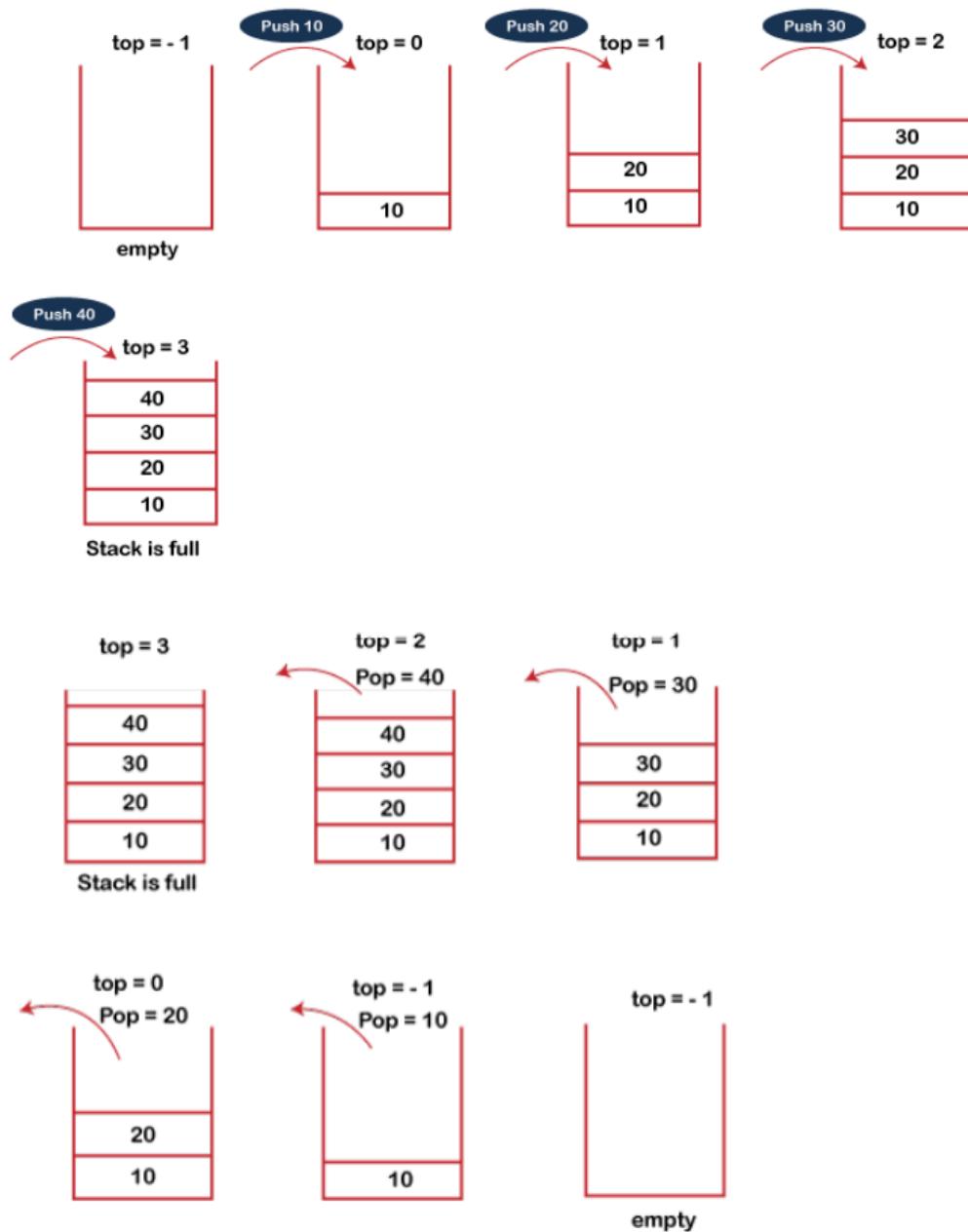


#### **Общие операции, реализованные в стеке:**

- **push()**: добавляем элемент в стек. Если стек заполнен, возникает состояние переполнения.
- **pop()**: удаляем элемент из стека. Если стек пуст, это означает, что в стеке нет элементов, это состояние известно как состояние потери значимости.
- **isEmpty()**: определяет, пуст стек.
- **isFull()**: определяет, заполнен ли стек
- **peek()**: возвращает элемент в заданной позиции.
- **count()**: возвращает общее количество элементов.
- **change()**: изменяет элемент в заданной позиции.

- `display()`: печатает все элементы, доступные в стеке.

## Операции `pop` и `push`



## Реализация стека с помощью Stack

```

import java.util.Stack;

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

 // create an object of Stack class
 Stack<String> animals= new Stack<>();

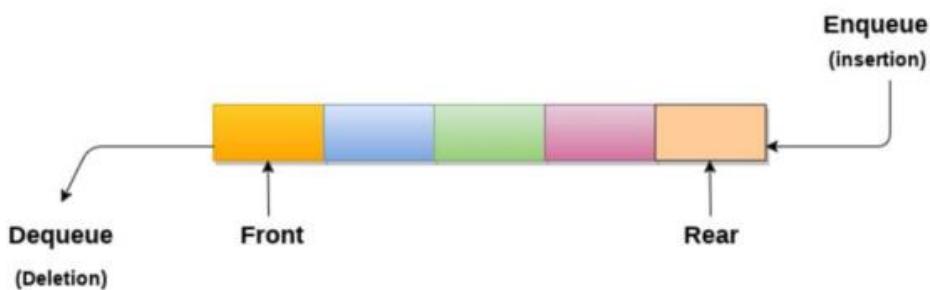
 // push elements to top of stack
 animals.push("Dog");
 animals.push("Horse");
 animals.push("Cat");
 System.out.println("Stack: " + animals);

 // pop element from top of stack
 animals.pop();
 System.out.println("Stack after pop: " +
 animals);
 }
}

```

## 86. Абстрактный тип данных Queue (очередь) в Джава

**Очередь** - упорядоченный список, который позволяет выполнять операции вставки на одном конце, называемом REAR , и операции удаления, которые выполняются на другом конце, называемом FRONT. Очередь работает по принципу «первый пришел — первый обслужен» ( FCFS, first come first served). Например: люди, стоящие в кассу магазина. Операция dequeue означает удаление элемента из начала очереди, а операция enqueue добавление элемента в конец очереди.



### **Использование очередей - Очереди широко используются:**

- в качестве списков ожидания для одного общего ресурса, такого как принтер, диск, ЦП;
- при асинхронной передаче данных (когда данные не передаются с одинаковой скоростью между двумя процессами), например. трубы, файловый ввод-вывод, сокеты;
- в качестве буферов в большинстве приложений, таких как медиаплеер MP3, проигрыватель компакт-дисков и т. д.;

- для ведения списка воспроизведения в медиаплеерах, чтобы добавлять и удалять песни из списка воспроизведения;
- в операционных системах для обработки прерываний и при реализации работы алгоритмов планирования и диспетчеризации.

## Виды очередей:

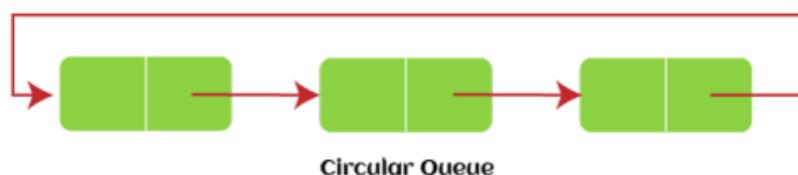
### 1. Простая очередь или линейная очередь

- Вставка элемента происходит с одного конца, а удаление — с другого.
- Основным недостатком использования линейной очереди является то, что вставка выполняется только с заднего конца. Если первые три элемента будут удалены из очереди, мы не сможем вставить больше элементов, даже если в линейной очереди есть свободное место. В этом случае линейная очередь показывает состояние переполнения, поскольку задняя часть указывает на последний элемент очереди.



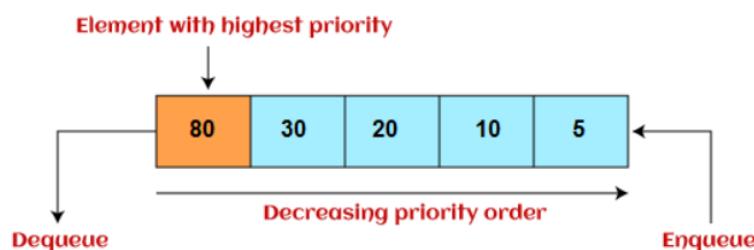
### 2. Циклическая очередь

- После последнего элемента очереди сразу идет первый или начальный элемент.
- Недостаток линейной очереди преодолевается при использовании круговой очереди. Если в циклической очереди есть пустое место, новый элемент можно добавить в пустое место, просто увеличив значение Rear.



### 3. Очередь с приоритетами

- Это особый тип очереди, в которой элементы располагаются в зависимости от их приоритета. Это особый тип структуры данных очереди, в которой каждый элемент такой очереди имеет связанный с ним приоритет. Допустим, какие-то элементы встречаются с одинаковым приоритетом, тогда они будут располагаться по принципу FIFO. Вставка в такую очередь происходит на основе поступления элемента в соответствии с его приоритетом, а удаление в данной очереди происходит на основе приоритета.
- Существует два типа: очередь с возрастающим приоритетом и очередь с убывающим приоритетом.



### 4. Двусторонняя очередь или Дек (англ. Deque)

- В Deque или Double Ended Queue вставка и удаление могут выполняться с обоих концов очереди либо спереди, либо сзади. Это означает, что мы можем вставлять и удалять элементы как с переднего, так и с заднего конца очереди. Deque можно использовать и как стек, и как очередь, поскольку он позволяет выполнять операции вставки и удаления на обоих концах.
- Существует два типа дека: очередь ограниченным вводом и очередь с ограниченным выводом.



## Операции, выполняемые над очередью

- Enqueue(): вставка элемента в конец очереди. Возвращает пустоту.
- Dequeue(): удаление из внешнего интерфейса очереди. Он также возвращает элемент, который был удален из внешнего интерфейса. Он возвращает целочисленное значение.
- Peek() Просмотр очереди: возвращает элемент, на который указывает передний указатель в очереди, но не удаляет его.
- isFull() (Queue overflow):
- isEmpty() (Queue underflow):

## Способы реализации очереди

- Реализация с использованием массива.
- Реализация с использованием связанного списка.

## Реализация очереди с помощью интерфейса Queue

```
import java.util.Queue;
import java.util.LinkedList;

class Main {

 public static void main(String[] args) {
 // Creating Queue using the LinkedList class
 Queue<Integer> numbers = new LinkedList<>();

 // enqueue
 // insert element at the rear of the queue
 numbers.offer(1);
 numbers.offer(2);
 numbers.offer(3);
 System.out.println("Queue: " + numbers);

 // dequeue
 // delete element from the front of the queue
 int removedNumber = numbers.poll();
 System.out.println("Removed Element: " +
removedNumber);

 System.out.println("Queue after deletion: " +
numbers);
 }
}
```

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

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

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

90. Обобщенное программирование. Понятие и использование дженериков в Java.

91. Параметризованные классы и методы. Их определение и использование

92. Стирание типов.

93. Понятие структуры данных список. Линейный список. Виды списков и их реализация на Java. Доступ к элементу структуры данных список. Использование списков. Трудоемкость операций со списками.

94. Односвязный и двусвязный список. Способы реализации на языке Джава

Тема 7. Java Core. Дженерики (продолжение) и использование контейнерных классов Java Framework Collection

95. Возможности Java Framework Collection. Контейнер ArrayList и его основные методы.

Java Framework Collection предоставляет различные структуры данных, такие как списки, множества, карты(мап) и т.д., для удобного хранения и обработки объектов. Одним из таких контейнеров является ArrayList.

**ArrayList** - это динамический массив, который позволяет добавлять, удалять, изменять и получать элементы. Его основные методы включают:

1. add(E element) - добавляет элемент в конец списка.
2. add(int index, E element) - добавляет элемент по указанному индексу, сдвигая последующие элементы, если необходимо.
3. remove(int index) - удаляет элемент по указанному индексу.
4. set(int index, E element) - заменяет элемент по указанному индексу на новый элемент.
5. get(int index) - возвращает элемент по указанному индексу.
6. size() - возвращает размер списка.

ArrayList также автоматически расширяет свой размер по мере необходимости, что делает его удобной структурой данных для хранения и обработки коллекций объектов.

#### 96. Возможности Java Framework Collection. Контейнер LinkedList и его основные методы.

Наряду с ArrayList, в Java Framework Collection имеется также контейнер **LinkedList**. LinkedList является двусвязным списком, предоставляющим эффективные операции вставки и удаления элементов в начале, середине и конце списка.

**Основные методы LinkedList включают:**

1. add(E element) - добавляет элемент в конец списка.
2. add(int index, E element) - добавляет элемент по указанному индексу.
3. remove(int index) - удаляет элемент по указанному индексу.
4. get(int index) - возвращает элемент по указанному индексу.
5. set(int index, E element) - заменяет элемент по указанному индексу на новый элемент.
6. addFirst(E element) - добавляет элемент в начало списка.
7. addLast(E element) - добавляет элемент в конец списка.
8. removeFirst() - удаляет первый элемент списка.
9. removeLast() - удаляет последний элемент списка.
10. getFirst() - возвращает первый элемент списка.
11. getLast() - возвращает последний элемент списка.

LinkedList также предоставляет реализацию интерфейса Queue, что

позволяет использовать его в качестве структуры данных для FIFO (первым вошел, первым вышел) очереди.

## 97. Возможности Java Framework Collection. Интерфейс Map и его основные методы.

Интерфейс **Map** в Java представляет собой структуру данных, которая хранит пары "ключ-значение". Он является частью Java Framework Collection и предоставляет функционал для доступа и изменения данных.

### **Основные методы интерфейса Map java.util.Map:**

1. `put(K key, V value)`: Добавляет в карту новую пару "ключ-значение". Если карта уже содержит элемент с таким ключом, то его значение обновляется.
2. `get(Object key)`: Возвращает значение по ключу. Если такого ключа нет, возвращает `null`.
3. `remove(Object key)`: Удаляет из карты пару "ключ-значение" по ключу.
4. `clear()`: Очищает карту, удаляя все пары "ключ-значение".
5. `containsKey(Object key)`: Проверяет есть ли в карте указанный ключ.
6. `containsValue(Object value)`: Проверяет есть ли в карте указанное значение.
7. `keySet()`: Возвращает набор всех ключей карты.
8. `values()`: Возвращает набор всех значений карты.
9. `entrySet()`: Возвращает набор всех элементов в виде пар "ключ-значение".
10. `size()` : Возвращает количество пар "ключ-значение" в карте.

Эти методы являются основой для работы с объектами, реализующими интерфейс Map, такими как `HashMap`, `LinkedHashMap`, `TreeMap` и др.

## 98. Возможности Java Framework Collection. Контейнер HashMap и его основные методы.

**HashMap в Java** - это часть коллекций Framework, она реализует интерфейс Map и использует хеш-таблицу для хранения информации, что обеспечивает быстрый доступ к данным.

### **Основные методы класса HashMap включают:**

1. `put(K key, V value)` - добавляет пару "ключ-значение" в карту. Если карта уже содержит значение для ключа, старое значение заменяется.
2. `get(Object key)` - возвращает значение по указанному ключу, или `null`, если такого ключа не существует.
3. `containsKey(Object key)` - возвращает `true`, если этот ключ существует в карте.
4. `containsValue(Object value)` - возвращает `true`, если это значение существует в карте.
5. `remove(Object key)` - удаляет специфический ключ и его значение из

карты.

6. clear() - удаляет все пары "ключ-значение" из карты.
7. isEmpty() - проверяет, является ли карта пустой.
8. size() - возвращает количество пар "ключ-значение" в карте.
9. values() - возвращает коллекцию всех значений карты.
10. keySet() - возвращает множество всех ключей.
11. entrySet() - возвращает множество всех элементов карты в формате "ключ-значение".

Этот класс позволяет проводить операции, такие как вставка, удаление и извлечение значений с большой скоростью, и используется во многих приложениях Java.

## 99. Коллекция HashMap, принципы создания и методы работы с ней

**HashMap в Java** - это реализация интерфейса Map, которая создаёт соответствие между ключами и значениями типа Key-Value pair. HashMap обеспечивает быстрый доступ к содержимому по ключу, поскольку внутренне используется хеш-таблица для хранения данных.

**Создать HashMap можно следующим способом:**

```
java
```

Копировать

```
Map<K, V> map = new HashMap<K, V>();
```

Где K это тип ключей, а V это тип значений.

**Основные методы работы с HashMap:**

1. put(Key K, Value V): Добавляет элемент в карту.
2. get(Object key): Возвращает значение для ключа в карте.
3. remove(Object key): Удаляет и возвращает значение, соответствующее указанному ключу.
4. containsKey(Object key): Проверяет, есть ли в карте указанный ключ.
5. containsValue(Object value): Проверяет, есть ли в карте указанное значение.
6. size(): Возвращает количество элементов в карте.
7. isEmpty(): Проверяет, пустая ли карта.
8. clear(): Удаляет все элементы из карты.
9. keySet(): Возвращает Set из всех ключей, содержащихся в карте.
10. values(): Возвращает Collection всех значений, которые содержатся в карте.

Работу с HashMap можно продемонстрировать на следующем примере:

java

Копировать

```
Map<String, Integer> map = new HashMap<>();
map.put("Вася", 25);
map.put("Петя", 30);
map.put("Маша", 23);

System.out.println(map.get("Вася")); // выведет 25
System.out.println(map.containsKey("Петя")); // выведе
т true
System.out.println(map.size()); // выведет 3
```

Важно помнить, что **HashMap** позволяет хранить null значения и null ключи. Кроме того, **HashMap** не гарантирует порядок вставки элементов, элементы могут возвращаться не в том порядке, в котором они были вставлены.

#### 100. Использование обобщенного класса **HashMap**, которая реализует интерфейс **Map** для хранения пар ключ-значение в разработке программ.

**HashMap** является обобщенным классом в Java, который реализует интерфейс **Map**. Это означает, что вы можете указать типы данных для ключей и значений при создании объекта **HashMap**.

Объект **HashMap** хранит данные в формате пары "ключ-значение". Это значит, что каждое значение в карте связано с уникальным ключом. Доступ к данным в карте осуществляется по ключу, что делает **HashMap** особенно полезным при работе с большими объемами данных.

**Пример использования** обобщенного класса **HashMap**:

```
// Создание объекта HashMap, который хранит ключи типа
String и значения типа Integer
HashMap<String, Integer> map = new HashMap<String, Integer>();

// Добавление элементов в карту
map.put("Apple", 10);
map.put("Banana", 20);
map.put("Cherry", 30);

// Получение значения по ключу
int value = map.get("Banana"); // value будет равно 20

// Удаление элемента из карты
map.remove("Apple");

// Проверка наличия ключа в карте
boolean exists = map.containsKey("Banana"); // exists
будет равно true
```

**HashMap** также предоставляет методы такие как `keySet()`, который возвращает все ключи в карте, и `values()`, который возвращает все значения в карте.

Использование **параметризованного HashMap** (или любой другой коллекции Java) помогает обеспечить типобезопасность. Это означает, что вы можете уведомить компилятор о том, какие типы данных вы намереваетесь использовать, и получить ошибки во время компиляции, если вы случайно попытаетесь использовать неправильный тип данных.

### 101. Возможности Java Framework Collection. Конейнер HashSet и его основные методы.

**HashSet в Java** представляет собой класс, который реализует интерфейс `Set` и включён в Java Collection Framework. Он создаёт набор, который не содержит дублирующихся элементов. Как и `HashMap`, `HashSet` использует хеш-таблицу для хранения информации, но он хранит только объекты, а не пары ключ/значение.

**Основные методы класса HashSet** включают:

1. `add(E e)` - добавляет указанный элемент в этот набор, если он еще не присутствует.

2. remove(Object o) - удаляет указанный элемент из этого набора, если он присутствует.

3. clear() - удаляет все элементы из этого набора.

4. contains(Object o) - возвращает true, если этот набор содержит указанный элемент.

5. isEmpty() - проверяет, является ли этот набор пустым.

6. size() - возвращает количество элементов в этом наборе (его мощность).

7. iterator() - возвращает итератор по элементам в этом наборе.

Вот пример использования HashSet:

```
java Копировать

HashSet<String> set = new HashSet<String>();

// добавление элементов в HashSet
set.add("Apple");
set.add("Banana");
set.add("Cherry");

// добавление дублирующего элемента
set.add("Apple");

// вывод размера набора – должно вывести 3, так как дубликаты не
// учитываются
System.out.println(set.size());

// проверка, содержит ли набор определенный элемент
System.out.println(set.contains("Banana")); // выводит true

// удаление элемента
set.remove("Apple");

// очистка набора
set.clear();
```

Этот класс достаточно полезен, когда вам нужно избежать дублирования элементов в вашем наборе данных.

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

**HashSet** - это обобщенный класс в библиотеке Java, который наследует функционал от класса **AbstractSet** и реализует интерфейс **Set**. Внутренне **HashSet** использует экземпляр **HashMap** для выполнения всех операций.

Представляет собой коллекцию, которая не допускает дублирования элементов. Если подробнее, каждый такой элемент представляет собой нечто уникальное, и операция добавления дубликатов будет проигнорирована. **HashSet** не обеспечивает порядок своих элементов, и элементы могут возвращаться в произвольном порядке.

Так как класс HashSet является обобщенным (HashSet<E>), то при создании объекта этого класса необходимо указать конкретный тип элементов, которые будут в нём храниться, где E - тип элементов коллекции.

### Основные методы HashSet:

- add(E e): Добавляет указанный элемент в набор, если его там еще нет.
- remove(Object o): Удаляет указанный элемент из набора (если он там есть).
- contains(Object o): Возвращает true, если этот набор содержит указанный элемент.
- size(): Возвращает количество элементов в этом наборе.
- isEmpty(): Возвращает true, если этот набор не содержит элементов.
- clear(): Удаляет все элементы из этого набора.

### Пример использования:

```
java Копировать

// Создаем коллекцию
HashSet<String> set = new HashSet<String>();

// Добавляем в коллекцию ряд элементов
set.add("Apple");
set.add("Banana");
set.add("Cherry");

// Добавляем в нее же дубликат
set.add("Apple"); // этот элемент будет проигнорирован

// Проверяем размер коллекции
System.out.println(set.size()); // выведет "3"

// Проверяем наличие элемента в коллекции
System.out.println(set.contains("Banana")); // выведет "true"

// Удаляем элемент "Apple"
set.remove("Apple");

// Очищаем коллекцию
set.clear();
```

## 103. Регулярные выражения и организация работы с ними в Java. Примеры

Регулярные выражения в Java используются для обработки строк по определенным шаблонам. Они могут быть применены для поиска, редактирования или манипулирования текстом.

В Java регулярные выражения поддерживаются в классе java.util.regex, который включает два основных класса: Pattern и Matcher.

- Pattern: Это скомпилированное представление регулярного выражения. Статический метод Pattern.compile(String regex) используется для компиляции регулярного выражения переданного в виде строки.

- Matcher: Используется для выполнения операций сопоставления с шаблоном на входной строке. Объекты типа Matcher создаются вызовом метода matcher() в объекте класса Pattern.

Самые популярные операции с регулярными выражениями:

- Поиск в строке: Matcher.find(), Matcher.start(), Matcher.end()
- Замена в строке: String.replaceAll(String regex, String replacement)
- Разбиение строки на части: String.split(String regex)

Вот простой пример использования регулярных выражений в Java:

```
java □ Копировать

import java.util.regex.*;

public class Main {
 public static void main(String[] args) {
 // Создаем шаблон регулярного выражения
 Pattern pattern = Pattern.compile("[a-z]+");

 // Создаем объект Matcher
 Matcher matcher = pattern.matcher("hello world");

 // Проверяем на совпадение
 while (matcher.find()) {
 System.out.println("Найденное совпадение: " + matcher.group());
 System.out.println("Начало совпадения: " + matcher.start());
 System.out.println("Конец совпадения: " + (matcher.end() - 1));
 }
 }
}
```

В этом примере мы используем регулярное выражение "[a-z]+", которое соответствует одному или нескольким символам в нижнем регистре. Мы ищем все такие совпадения во входной строке и печатаем их.

#### 104. Структура коллекций в Java Collection Framework. Иерархия интерфейсов

В Java Collection Framework существуют четыре основных интерфейса: Collection, List, Set и Map.

##### **1. Интерфейс Collection**

Collection является корневым интерфейсом в иерархии коллекций. Ему подчиняются интерфейсы List, Set и Queue.

## 2. Интерфейс List

List - это упорядоченная коллекция (последовательность) содержащая элементы, в которых допускаются дубликаты. ArrayList, LinkedList, Vector, Stack - это реализации интерфейса 'List'.

## 3. Интерфейс Set

Set - это неупорядоченная коллекция, которая не содержит дубликатов. HashSet, LinkedHashSet, TreeSet - это реализации интерфейса Set.

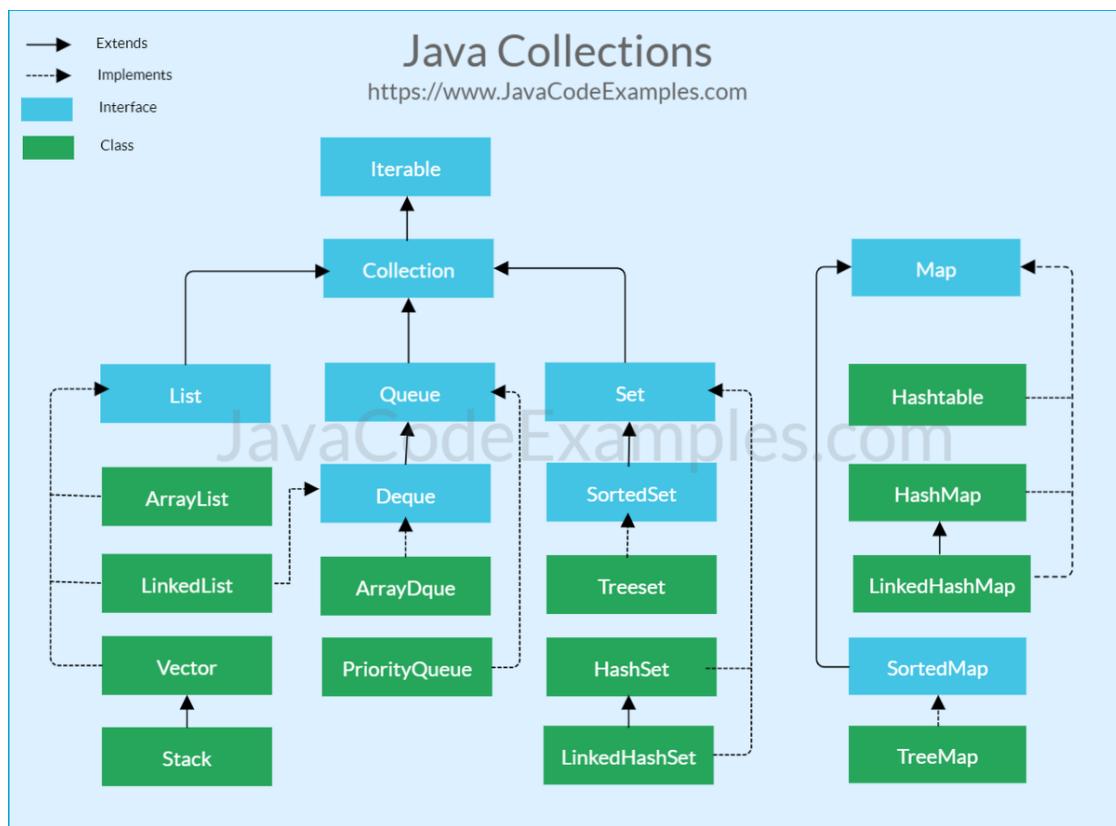
## 4. Интерфейс Queue

Queue - это интерфейс для коллекций, предназначенных для хранения упорядоченных элементов перед обработкой. PriorityQueue, Deque, ArrayDeque - это реализации интерфейса Queue.

## 5. Интерфейс Map

Map - это объект, который хранит соответствия ключей к значениям. Ключи уникальны, значения могут дублироваться. HashMap, TreeMap, LinkedHashMap - это реализации интерфейса Map.

Таким образом, иерархия интерфейсов в Java Collection Framework выглядит следующим образом:



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

Метод `iterator()` интерфейса `Collection` возвращает объект `Iterator<E>`, который используется для итерации по элементам коллекции.

Тип E здесь - это тип элементов коллекции. Iterator<E> предоставляет методы для работы с элементами коллекции:

- boolean hasNext(): Возвращает true, если в коллекции ещё остались элементы для итерации.
- E next(): Возвращает следующий элемент из коллекции. Если такого элемента нет, выбрасывается исключение NoSuchElementException.
- void remove(): Удаляет из коллекции последний элемент, возвращенный этим итератором. Если метод next() не был вызван, или метод remove() уже был вызван после последнего вызова next(), выбрасывается исключение IllegalStateException.

Здесь пример использования итератора:

```
java Копировать

import java.util.*;

public class Main {
 public static void main(String[] args) {
 Collection<Integer> collection = new ArrayList<>(Arrays.a
sList(1, 2, 3, 4, 5));

 Iterator<Integer> iterator = collection.iterator();

 while(iterator.hasNext()) {
 Integer element = iterator.next();
 System.out.println(element);
 }
 }
}
```

В этом примере мы создали коллекцию из пяти элементов и используем итератор для перебора этих элементов. Каждый из элементов выводится в консоль.

## 106. Класс Pattern и его использование

Класс Pattern в языке программирования Java является частью пакета java.util.regex и используется для представления скомпилированного регулярного выражения. Регулярные выражения широко используются в программировании для работы со строками, включая поиск, замену и извлечение данных.

Основные методы класса Pattern включают:

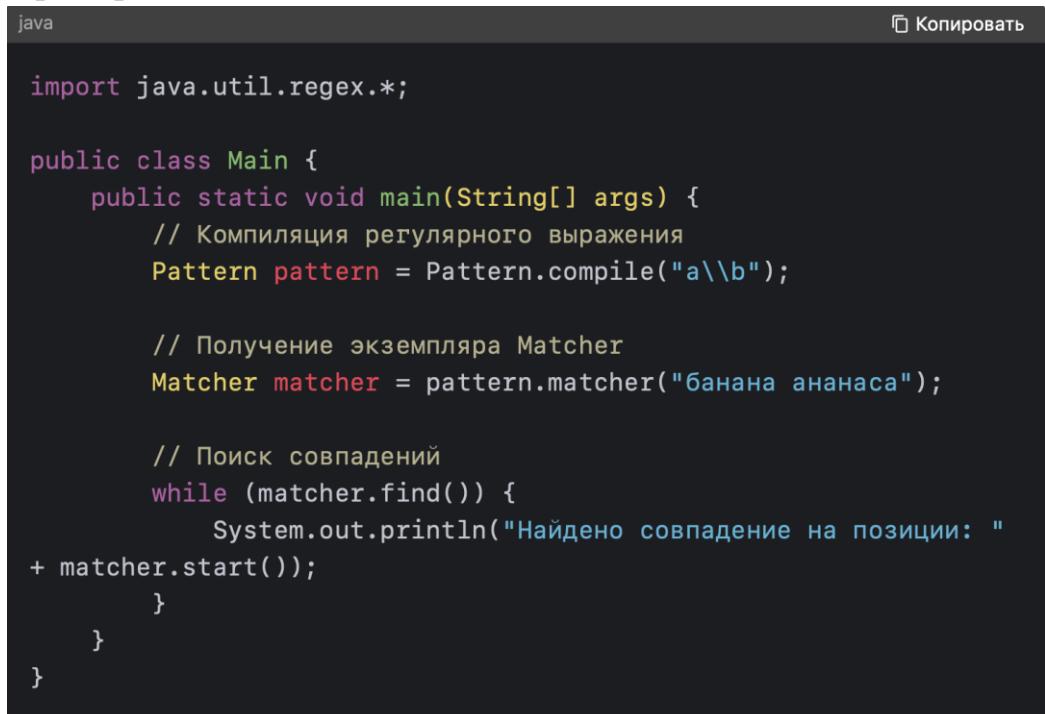
- public static Pattern compile(String regex): этот метод используется для компиляции регулярного выражения в экземпляр класса Pattern.

- public Matcher matcher(CharSequence input): этот метод создает Matcher для данного входного символьного потока, который можно затем использовать для выполнения операций совпадения, поиска и замены.

- public static boolean matches(String regex, CharSequence input): этот метод является упрощенной формой использования регулярных выражений, который проверяет, соответствует ли вся входная последовательность данному регулярному выражению.

- public String pattern(): этот метод возвращает регулярное выражение, из которого был скомпилирован данный экземпляр Pattern.

Пример использования класса Pattern:



The screenshot shows a code editor window with a dark theme. The title bar says "java". The code is as follows:

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

public class Main {
 public static void main(String[] args) {
 // Компиляция регулярного выражения
 Pattern pattern = Pattern.compile("a\\b");

 // Получение экземпляра Matcher
 Matcher matcher = pattern.matcher("банана ананаса");

 // Поиск совпадений
 while (matcher.find()) {
 System.out.println("Найдено совпадение на позиции: "
+ matcher.start());
 }
 }
}
```

В этом примере регулярное выражение "a\b" представляет собой паттерн поиска буквы "a" в конце слова. После компиляции регулярного выражения при помощи метода compile создается экземпляр Matcher для строки "банана ананаса". При последующем вызове метода find происходит поиск всех совпадений, и, если они найдены, выводится их позиция в исходной строке.

## 107. Класс Math и его использование

Первый набор методов - это **базовые математические функции**, такие как степень, квадратный корень, максимум или минимум между двумя значениями, а также случайное значение.

В дополнение к основным математическим функциям, класс Math содержит методы для решения **экспоненциальных и логарифмических, а также тригонометрических функций**.

Также данный класс содержит различные **методы округления** (в большую сторону, в меньшую и тд.)

Класс Math используется для самых различных математических вычислений внутри программы.

Класс Math, представленный в пакете java.lang, является ключевым элементом Java, предоставляющим готовые методы для выполнения различных математических операций. В силу того, что все методы класса Math являются статическими, нет необходимости создавать экземпляр класса для их использования, и они могут быть применены напрямую.

Подробнее, вот некоторые важные аспекты класса Math в Java:

## 1. Основные методы

Класс Math предоставляет широкие возможности для работы с числовыми значениями. Большинство его методов подразделяются на следующие категории:

- Абсолютное значение: abs().
- Округление: ceil(), floor(), rint(), round().
- Минимум/максимум: min(), max().
- Случайное число: random().
- Тригонометрические: sin(), cos(), tan(), asin(), acos(), atan(), toDegrees(), toRadians().
- Логарифмические и экспоненциальные: exp(), log(), log10(), pow(), sqrt() и т.д.

## 2. Примеры использования

Знакомство с базовым использованием класса Math:

```
java Копировать

double x = Math.abs(-10.5); // x становится 10.5
double y = Math.max(5, 10); // y становится 10
double z = Math.sqrt(64); // z становится 8.0
double t = Math.random(); // t становится случайным числом между
0.0 и 1.0
```

## 3. Поле PI и E

В классе Math также есть две важные константы: PI и E. Они предоставляют значения числа Пи и основания натурального логарифма соответственно.

## 4. Семантика NaN и Infinity

Класс Math также обеспечивает корректную обработку специальных значений, таких как NaN (Not a Number) и Infinity.

В общем, класс Math - это мощный инструмент в руках программиста Java. Он предоставляет практически все необходимое для выполнения математических вычислений, как простых, так и сложных.

## 108. Возможности Java Framework Collection. Интерфейс Map и его основные методы

Интерфейс Map в Java представляет собой структуру данных, которая хранит пары "ключ-значение". Он является частью Java Framework Collection и предоставляет функционал для доступа и изменения данных.

Основные методы интерфейса `Map` java.util.Map:

1. put(K key, V value): Добавляет в карту новую пару "ключ-значение". Если карта уже содержит элемент с таким ключом, то его значение обновляется.

2. get(Object key): Возвращает значение по ключу. Если такого ключа нет, возвращает null.

3. remove(Object key): Удаляет из карты пару "ключ-значение" по ключу.

4. clear(): Очищает карту, удаляя все пары "ключ-значение".

5. containsKey(Object key): Проверяет есть ли в карте указанный ключ.

6. containsValue(Object value): Проверяет есть ли в карте указанное значение.

7. keySet(): Возвращает набор всех ключей карты.

8. values(): Возвращает набор всех значений карты.

9. entrySet(): Возвращает набор всех элементов в виде пар "ключ-значение".

10. size(): Возвращает количество пар "ключ-значение" в карте.

Эти методы являются основой для работы с объектами, реализующими интерфейс Map, такими как HashMap, LinkedHashMap, TreeMap и др.

(Вопрос дословно аналогичен 97!!!)

## 109. Возможности Java Framework Collection. Контейнер Hashtable его основные методы.

## 110. Возможности Java Framework Collection. Интерфейс Iterator и Iterable.

## 111. Работа с Датой и временем в Java. Примеры использования

Тема 8. Стандартные потоки ввода-вывода. СерIALIZАЦИЯ.

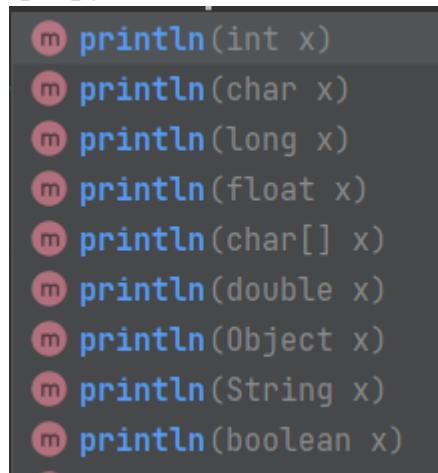
## 112. Класс System. Стандартные потоки ввода-вывода, предоставляемые Java. Работа со стандартными потоками вывода

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

В Java метод **System.out.println()** используется для вывода данных в консоль. Он перегружен для различных типов данных, что позволяет удобно выводить разнообразную информацию. Вот несколько примеров:

- Вывод строки: `System.out.println("Hello");`
- Вывод числа: `System.out.println(123);`

Короче, выводить может все примитивные типы, а также массив `char`, строку и объект. Вот все перегрузки:



### 114. Класс Scanner. Ввод и вывод данных. Стандартные потоки ввода и вывода.

**Scanner** - это класс в пакете `java.util`, используемый для получения ввода примитивных типов, таких как `int`, `double` и т. д. и строки.

Использование класса `Scanner` в Java - самый простой способ чтения ввода в Java-программе, хотя и не очень эффективен, если вам нужен метод ввода для сценариев, где время является ограничением, как в конкурентном программировании.

Для создания **потока вывода** в классе `System` определен объект `out`. В этом объекте определен метод `print` и другие его виды, которые позволяют вывести на консоль некоторое значение.

Java Streams использует два основных потока для чтения и записи данных:

1. **InputStream** – входной поток используется для чтения данных из источника в приложении Java. Данными может быть что угодно: файл, массив, периферийное устройство или сокет. В Java класс `java.io.InputStream` является базовым классом для всех входных потоков ввода-вывода Java.

2. **OutputStream** – выходной поток используется для записи данных (файла, массива, периферийного устройства или сокета) в пункт назначения. В

Java класс `java.io.OutputStream` является базовым классом для всех выходных потоков Java IO.

Класс `Scanner` в Java является частью пакета `java.util` и используется для получения ввода примитивных типов данных, таких как `int`, `double`, и т.д. и строк. Для работы со сканером требуется импортировать пакет `java.util.Scanner`.

Создание объекта класса `Scanner` происходит следующим образом:

```
java
Scanner scanner = new Scanner(System.in);
```

Здесь `System.in` является стандартным потоком ввода, который обычно связан с клавиатурой.

Класс `Scanner` предлагает различные методы для чтения данных различных типов. Вот некоторые из них:

- `next()`: Чтение ввода пользователя до тех пор, пока не встретится пробел.
- `nextLine()`: Чтение всей строки ввода.
- `nextInt()`, `nextDouble()`, `nextLong()` и т.п.: Чтение числовых данных в указанном формате.

Пример использования класса `Scanner` для чтения данных:

```
java
Scanner scanner = new Scanner(System.in);
System.out.println("Введите имя:");
String name = scanner.nextLine(); // считывание строки
System.out.println("Введите возраст:");
int age = scanner.nextInt(); // считывание числа
```

Стандартные потоки ввода и вывода в Java обозначены в классе `System`. `System.in` является стандартным потоком ввода (клавиатура), `System.out` - стандартным потоком вывода (консоль), и `System.err` - стандартным потоком вывода ошибок.

Для вывода информации на консоль используется метод `System.out.println()`, который автоматически переводит каретку на новую строку после вывода, и метод `System.out.print()`, который выводит информацию без перевода каретки. Стандартный поток вывода ошибок `System.err` работает аналогично `System.out` и обычно используется для вывода информации об ошибках разработчика.

Пример использования стандартных потоков:

```
java
System.out.println("Это сообщение выведено в стандартный поток вывода");
System.err.println("Это сообщение выводится в стандартный поток ошибок");
```

115. Использование форматированного ввода-вывода в Java. Классы для форматирования вывода.

При выводе или отображении данных часто их нужно представить в удобочитаемом виде. Чтобы упростить процесс форматирования строковых данных в Java существует специальный класс.

**Formatter**, который располагается в пакете **java.util**. Если до этого вы программировали на языке С, то возможности данного класса покажутся вам знакомыми по сравнению с функцией **printf()** (которую также можно использовать в программах на java), предназначеннной как раз для форматированного вывода. Внутри библиотеки классов Java возможности Formatter'a используются в методах **print()** и **printf()**, а также в классе String — метод **format()**.

Основной метод **format()** преобразует переданную строку в ее форматированное представление в соответствие с заданным шаблоном. Чтобы понять работу Formatter'a лучше всего рассмотреть практический пример.

Объект класса Formatter можно получить, вызвав один из нескольких предопределенных конструкторов. Ниже перечислены только некоторые из них.

```
Formatter()
Formatter(Appendable a)
Formatter(File file)
Formatter(Locale l)
Formatter(OutputStream ous)
```

**String.format()** — не единственный метод для форматирования строки. Его аналогами могут служить **System.out.printf()** и **System.out.format()**.

**DecimalFormat** — класс для форматирования любого числа в Java, будь то целое число или число с плавающей запятой.

Когда происходит создание объекта DecimalFormat, прямо в конструкторе можно задать шаблон форматирования приходящих чисел.

Как будет выглядеть наш пример с использованием DecimalFormat:

```
1 DecimalFormat df = new DecimalFormat("#.###");
2 double value = 72.224463;
3 System.out.print(df.format(value));
```

Вывод в консоли:

```
72,224
```

## 116. Понятие сериализации и ее использование в ООП программах. Использование интерфейса Serializable в программах на Джава

**Сериализация** - это процесс сохранения состояния объекта в последовательность байт.

Любой Java-объект преобразуется в последовательность байт. Для чего это нужно?

Мы уже не раз говорили, что программы не существуют сами по себе. Чаще всего они взаимодействуют друг с другом, обмениваются данными и т.д. И байтовый формат для этого удобен и эффективен. Мы можем, например, превратить наш объект класса SavedGame (сохраненная игра) в последовательность байт, передать эти байты по сети на другой компьютер, и на втором компьютере превратить эти байты снова в Java-объект!

В Java за процессы сериализации отвечает интерфейс Serializable. Этот интерфейс крайне прост: чтобы им пользоваться, не нужно реализовывать ни одного метода. Его достаточно implementировать в класс, чтобы представить информацию о сериализации джава-машине.

### 117. Какие объекты можно сериализовать?

**Сериализация** - это процесс сохранения состояния объекта в последовательность байт. **Десериализация** - это процесс восстановления объекта, из этих байт. Java Serialization API предоставляет стандартный механизм для создания сериализуемых объектов.

Для начала следует убедиться, что класс сериализуемого объекта реализует интерфейс java.io.Serializable. Интерфейс Serializable это интерфейс-маркер; в нём не задекларировано ни одного метода. Но говорит сериализирующему механизму, что класс может быть сериализован.

### 118. Какие методы определяет интерфейс Serializable?

Интерфейс Serializable это **интерфейс-маркер**; в нём не задекларировано ни одного метода. Но говорит сериализирующему механизму, что класс может быть сериализован. При попытке сериализовать объект, не реализующий этот интерфейс, будет брошено java.io.NotSerializableException.

### 119. Что означает понятие десериализация?

**Десериализация** - это процесс восстановления объекта, из последовательности байт. Java Serialization API предоставляет стандартный механизм для создания сериализуемых объектов.

### 120. Организация работы с файлами в Джава. Класс File, определенный в пакете java.io, не работает напрямую с потоками. В чем состоит его задача?

Класс File в Java предназначен для работы с файлами и директориями на уровне файловой системы. Его основная задача - предоставить удобные методы для управления файлами, такие как создание, удаление, переименование, проверка существования и другие операции.

Однако `File` не предназначен для работы напрямую с потоками данных. Вместо этого, для чтения и записи данных из файла в Java обычно используются классы из пакета `java.io` или `java.nio`, такие как `FileInputStream`, `FileOutputStream`, `BufferedReader`, `BufferedWriter` и другие. Эти классы предоставляют функциональность для работы с потоками данных, в то время как `File` сосредоточен на манипуляциях с метаданными файловой системы.

`File` предоставляет интерфейс для управления файлами и директориями, а классы из пакетов `java.io` и `java.nio` обеспечивают работу с потоками данных для чтения и записи содержимого файлов.

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

Метод `FileOutputStream.write()` используется для записи байтовых данных в файл, на который направлен `FileOutputStream`. Когда этот метод вызывается, переданные байты добавляются в конец файла.

*122. Иерархия классов ввода вывода. Работа с файлами в Java. Работа с файлами. СерIALIZАЦИЯ ОБЪЕКТОВ*