|  |  |  |  |
| --- | --- | --- | --- |
| 1. Понятие платформы JAVA. Виртуальная JAVA – машина. Категории Java программ. JDK.  **Платформа Java** - это среда, которая включает в себя виртуальную машину Java и набор библиотек и инструментов для разработки и выполнения программ на языке Java. Она обеспечивает переносимость и независимость программ, так как код Java компилируется в байт-код, который может выполняться на любой машине, поддерживающей JVM.  **Java** — строго типизированный объектно-ориентированный язык программирования.  **Основные компоненты платформы Java**:  1. **JDK** (Java Development Kit) — это комплект для разработки на Java. Включает в себя следующие компоненты: компилятор (javac) — преобразует исходный код Java в байт-код; интерпретатор (java) — выполняет байт-код на JVM; инструменты для отладки (jdb) — помогают находить и исправлять ошибки в коде; документация (javadoc) — создаёт документацию из комментариев в исходном коде.  2. **JRE** (Java Runtime Environment) — это среда выполнения Java-программ, включающая виртуальную машину Java (JVM) и стандартные библиотеки классов. JRE необходима для выполнения Java-программ.  **JVM** — это ключевой компонент платформы Java, который позволяет выполнять Java-программы. JVM интерпретирует байт-код Java и исполняет его на конкретной аппаратной платформе. Благодаря JVM Java-программы являются кроссплатформенными, т. е. могут выполняться на различных операционных системах без изменения исходного кода.  **Java-программы можно разделить на несколько категорий**:  1. Приложения — самостоятельные программы, которые запускаются на клиентской машине. Они включают в себя консольные и графические приложения.  2. Апплеты — это небольшие программы, которые загружаются и выполняются в контексте веб-браузера.  3. Сервлеты (Servlets) и JSP (JavaServer Pages) — это серверные компоненты, которые используются для создания динамических веб-страниц и веб-приложений.  4. MIDlets — это приложения для мобильных устройств, которые работают на платформе Java ME (Micro Edition).  5. JavaBeans — это повторно используемые компоненты, которые можно внедрять в другие программы. | 2. Тип данных. Примитивные и ссылочные типы. Правила автомат. и явного преобразования. Спец. нечисловые значения: Inf, NaN, Ind. Константы. Соглашения об именовании в JAVA  В Java существуют два основных типа данных: примитивные и ссылочные типы.  **Примитивные типы данных**: **byte** - 8-битное целое число со знаком. **short** - 16-битное целое число со знаком. **int** - 32-битное целое число со знаком. **long** - 64-битное целое число со знаком. **float** - 32-битное число с плавающей запятой. **double** - 64-битное число с плавающей запятой. **char** - 16-битный символ Unicode. **boolean** - логическое значение (true или false).  **Ссылочные типы данных** указ. на объекты и включают в себя:  1. Классы (например, String, Scanner, пользовательские классы).  2. Интерфейсы (например, List, Map).  3. Массивы (например, int[], String[]).  **Автоматическое (неявное) преобразование** происходит, когда преобразование может быть выполнено без потери данных:  int i = 100;  long l = i; // автоматическое преобразование от int к long  float f = l; // автоматическое преобразование от long к float  **Явное преобр-ие** требует явного указания программистом, так как может привести к потере данных:  double d = 100.04;  long l = (long) d; // явное преобразование от double к long  int i = (int) l; // явное преобразование от long к int  **Специальные нечисловые значения**  1. Inf — возн. при делении на 0 или при переполнении полож. числа. Double.POSITIVE\_INFINITY или Float.POSITIVE\_INFINITY.  2. NaN — результат операций, которые не имеют числового значения, например, 0.0 / 0.0. Double.NaN или Float.NaN.  3. В Java нет спец. значения "Ind", но индетерминированное поведение может возникнуть, при работе с непроиниц. Пер-ными (компилятор обычно предотвращает такие случаи).  **Константы** в Java объявляются с помощью ключевого слова final. После присвоения знач. константе его невозможно изменить.  final int DAYS\_IN\_WEEK = 7; final double PI = 3.141592653589793;  **Соглашения об именовании в Java**  1. Классы и интерфейсы: Названия классов и интерфейсов начинаются с заглавной буквы и используют CamelCase.  2. Методы и переменные: Начинаются с маленькой буквы и используют camelCase.  3. Константы: Обычно пишутся в верхнем регистре с разделением слов символом подчеркивания (\_) (MAX\_SIZE, PI).  4. Пакеты: Используют строчные буквы и точки для разделения (например, com.example.project). | 3. Пакеты. Уровни видимости классов. Возможности импорта. Архивация. Комментарии и аннотации. javadoc дескрипторы: @author, @version, @since, @see, @param, @return.  **Пакеты** в Java используются для организации классов и других типов в логические группы. Они помогают избежать конфликтов имен и управлять доступом к классам. Пакеты объявляются с помощью ключевого слова package: package com.example.myapp;  **В Java существуют** **четыре уровня видимости**:  1. public — класс или член класса доступен отовсюду.  2. protected — член доступен в пределах пакета и в подклассах.  3. default (без модификатора) — класс или член класса доступен только в пределах одного пакета.  4. private — член класса доступен только внутри своего класса.  **Ключевое слово import** исп. для импорта классов и пакетов: import java.util.List;  **JAR (Java ARchive**) — это архивный файл, который объединяет множество классов и других ресурсов в один файл. Это упрощает распространение и установку Java-приложений.  **Комментарии** в Java исп. для пояснения кода и бывают 3 типов:  1. Однострочные (//);  2. Многострочные (/\* ... \*/);  3. Документирующие (/\* \* ... \*… \*… \*/): для генерации документации.  **Аннотации** предоставляют метаданные для классов, методов и других элементов.  @Override — метод переопределяет метод суперкласса.  @Deprecated — эл-т устарел и не рекоменд. к использованию.  @SuppressWarnings — подавляет указанные предупреждения компилятора.  **javadoc** — это инструмент для генерации HTML-документации из исходного кода Java. Исп. документирующие комментарии.  /\*  This class represents a simple example.  @author Wandskape  @version 1.0  @since 2024-05-28  \*/  **Основные дескрипторы javadoc**:  - @author — указывает автора класса или метода.  - @version — версия класса или метода.  - @since — указывает, с какой версии доступен класс или метод.  - @see — ссылка на связанный элемент.  - @param — описание параметра метода.  - @return — описание возвращаемого значения метода. | 4. Классы-оболочки. Упаковка и распаковка. Строки. Массивы  **Классы-оболочки (Wrapper classes)** предоставляют объектные представления примитивных типов данных. Они наход. в пакете java.lang и позволяют работать с примитивами как с объектами.  **Основные классы-оболочки**: Byte для byte, Short для short, Integer для int, Long для long, Float для float, Double для double, Character для char, Boolean для boolean.  int primitiveInt = 5;  Integer wrappedInt = Integer.valueOf(primitiveInt); // Упаковка  int unboxedInt = wrappedInt.intValue(); // Распаковка  **Упаковка** — автомат. преобр-ие примитивного типа в объект соответств. класса-оболочки. **Распаковка** — автомат. преобр-ие объекта класса-оболочки в примитивный тип.  Java поддерж. **автоупаковку** и **автораспаковку**, что позволяет более удобно работать с примитивами и их оболочками.  int primitiveInt = 10;  Integer wrappedInt = primitiveInt; // Автоупаковка  int unboxedInt = wrappedInt; // Автораспаковка  **Строки в Java** — объекты класса String и исп. для работы с текст. данными. Строки являются неизменяемыми, что означает, что их содержимое не может быть изменено после создания.  String s1 = "Hello, World!"; String s2 = new String("Hello, World!");  **Основные методы класса String**:  - length() — возвращает длину строки.  - charAt(int index) — возвращает символ на указанной позиции.  - substring(int beginIndex, int endIndex) — возвращает подстроку.  - indexOf(String str) — возвр. индекс перв. вхождения подстроки.  - toUpperCase() и toLowerCase() — в верхний или нижний регистр.  **Массивы** — это структуры данных, которые позволяют хранить несколько значений одного типа в фиксированном размере. Массивы могут быть **одномерными или многомерными**.  int[] intArray = new int[5]; // Одномерный массив  int[] initializedArray = {1, 2, 3, 4, 5};  int[][] twoDimArray = new int[3][4]; // Двумерный массив  int[][] initializedTwoDimArray = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };  **Доступ к элементам массива и их изменение**:  intArray[0] = 10;  int value = intArray[1];  twoDimArray[0][1] = 20;  int twoDimValue = twoDimArray[2][2];  **Основные методы работы с массивами**:  - length — свойство, возвращает длину массива.  - Arrays.copyOf(originalArray, newSize) – копирование массива.  - Arrays.sort(arrayName) – сортировка массива.  - Arrays.binarySearch(arrayName, valueToFind) – поиск элемента. |
| 5. Класс Object. Переопределение методов из Object. Соглашения по переопределению  **Класс Object** — корневой класс для всех других классов в Java. Все классы неявно наследуют от Object. Он предоставляет осн. методы, которые могут быть переопределены для изменения поведения объектов.  **Осн. методы класса Object**: boolean **equals**(Object obj); int **hashCode**(); String **toString**(); Object **clone**(); void **finalize**(); Class<?> **getClass**()  **Метод equals** исп. для проверки равенства 2 объектов. По умолчанию он проверяет равенство ссылок, но его можно переопределить для проверки содержимого объектов.  @Override  public boolean equals(Object obj) {  if (this == obj) return true;  if (obj == null || getClass() != obj.getClass()) return false;  Person p = (Person) obj;  return age == p.age && Objects.equals(name, p.name);  }  **Метод hashCode** возвращает целочисленный хэш-код объекта. Переопределение hashCode должно быть совместимо с equals, то есть если два объекта равны по equals, их хэш-коды должны быть одинаковыми.  @Override  public int hashCode() { return Objects.hash(name, age); }  **Метод toString** возвращает строковое представление объекта. Полезен для отладки и логгирования.  @Override  public String toString() {  return "Person{name='" + name + "', age=" + age + '}';  }  **Соглашения по переопределению**  - equals и hashCode. Если переопределяешь equals, обязательно переопредели hashCode. Два равных объекта должны иметь одинаковый хэш-код.  - toString. Предоставление полезного строк. представления объекта.  - clone должен поддерживать интерфейс Cloneable и возвр. копию объекта.  - finalize исп. для очистки перед уничтожением объекта сборщиком мусора (сейчас редко используется). | 6. Состав класса. Логические блоки. Модификатор native. Абстрактные классы  **Класс в Java состоит из следующих основных компонентов**:  1. Поля: Переменные, которые хранят данные.  2. Методы: Функции, которые выполняют действия.  3. Конструкторы: Спец. методы для инициализации объектов.  4. Блоки иниц-ии: Код, кот. выполн. при создании объекта.  5. Вложенные классы: Классы, объявленные внутри др. класса.  public class MyClass {  private int field1; public MyClass(int field1) { this.field1 = field1; }  public void myMethod() { System.out.println("Hello, World!"); }  public class InnerClass { ... }  }  **Существует два типа блоков инициализации**:  1. Статические: Выполняются один раз при загрузке класса. Используются для инициализации статических полей.  public class MyClass { static { System.out.println("Static block"); } }  2. Нестатические: Выполняются каждый раз при создании нового экземпляра класса. Исп. для инициализации полей объекта.  public class MyClass { { System.out.println("Instance block"); } }  **Модификатор native** используется для обозначения методов, реализация кот. написана на др. языках прогр-ия. Такие методы явл. "родными" для Java и их реализация наход. вне JVM.  **Загрузка библиотеки, содержащей native метод**:  public class MyClass {  public native void nativeMethod();  static { System.loadLibrary("NativeLibrary"); }  }  **Абстрактные классы** не могут быть созданы напрямую и служат для определения общих характеристик для подклассов. Могут содержать как абстрактные, так и не абстрактные методы. **Абстрактные методы** - методы без реализации, кот. должны быть переопределены в подклассах.  public abstract class Animal {  public abstract void makeSound();  public void sleep() { System.out.println("Sleeping..."); }  }  public class Dog extends Animal {  @Override  public void makeSound() { System.out.println("Bark"); }  } | 7. Параметриз. классы (generic): объявления и проблемы реализации. Параметр. методы. Generic-огран-ия. Метасимвол ?.  **Параметриз. классы** позволяют создавать классы и методы, кот. работают с разл. типами данных, сохраняя типобезопасность.  **Объявление** параметриз. класса включает исп-ие угловых скобок (<>) с одним или несколькими параметрами типа:  public class Box<T> {  private T cont; public void setContent(T cont) { this.cont = cont; }  public T getContent() { return content; }  }  Box<String> stringBox = new Box<>(); stringBox.setContent("Hello");  String content = stringBox.getContent();  **Параметризованные методы** позволяют исп. параметры типа в пределах метода. Такие методы могут быть статическими.  public class Utility {  public static <T> void printArray(T[] arr) {  for (T el : arr) { System.out.println(el); } } }  Integer[] intArray = {1, 2, 3, 4, 5}; String[] strArray = {"A", "B", "C"};  Utility.printArray(intArray); Utility.printArray(strArray);  **Generic-ограничения** позволяют задать верхнюю и нижнюю границы для параметров типа.  1. Ограничение сверху (extends): ограничивает параметр типа определенным типом или его подклассами.  public <T extends Number> void processNumbers(List<T> list) {  (Доступны методы класса Number) }  2. Ограничение снизу (super): ограничивает параметр типа определенным типом или его суперклассами (исп. редко).  public <T super Integer> void processIntegers(List<T> list) { ... }  **Метасимвол ?**:  1. Исп., когда тип неизвестен: List<?> list = new ArrayList<>();  2. Позволяет использовать подтипы указанного типа (extends):  List<? extends Number> numberList = new ArrayList<Integer>();  3. Позволяет использовать суперклассы указанного типа (super):  List<? super Integer> integerList = new ArrayList<Number>();  **Проблемы реализации Generics**  1. Во время компиляции информация о параметрах типа удаляется и заменяется их границами. Это значит, что параметриз. информация недоступна во время выполнения.  2. Нельзя использовать примитивные типы в кач-ве параметров типа. Нужно исп. соответств. классы-оболочки.  3. Нельзя создать экземпляр параметриз. типа напрямую. | 8. Перечисления  **Перечисления** в Java — особый вид класса, кот. представляет собой набор фиксированных констант. Они упрощают работу с набором предопред. значений и делают код более читабельным и безопасным. Переч-ие объявляется с исп-ем ключ. слова **enum**.  public enum Day {  sunday, monday, tuesday, wednesday, thursday, friday, saturday  }  Перечисления могут быть использ. так же, как и любые другие объекты. Например, их можно использовать в операторах switch, а также как параметры методов и возвращаемые значения.  Day day = Day.MONDAY;  **Перечисления в Java имеют несколько встроенных методов**:  - values() — возвращает массив всех значений перечисления.  - valueOf(String name) — возвр. константу переч-ия по ее имени.  for (Day day : Day.values()) { System.out.println(day); }  Day day = Day.valueOf("monday"); System.out.println("Day: " + day);  **Перечисления могут содержать поля, методы и конструкторы**.  public enum Season {  Winter("Cold"), Spring("Mild"), Summer("Hot"), Autumn("Cool");  private String description;  Season(String description) { this.description = description; }  public String getDescription() { return description; }  }  Перечисления могут **реализовывать интерфейсы**:  public interface Describable { String getDescription(); }  public enum Planet implements Describable {  Mercury("Small"), Venus("Hot"), Earth("Our"), Mars("Red");  private String description;  Planet(String description) { this.description = description; }  @Override  public String getDescription() { return description; }  }  Перечисления могут содержать **абстрактные методы**, которые должны быть реализованы каждым элементом перечисления.  **Преимущества использования перечислений**  1. Код становится более понятным и самодокументируемым.  2. Ошибки, связанные с исп-ием неподход. значений, выявляются на этапе компиляции.  3. Предоставляют удобные методы и возможности для работы с предопределенными значениями. |
| 9. Правила наследования. Использование super и this  **Наследование** — это механизм, который позволяет одному классу (подклассу) наследовать свойства и методы др. класса (суперкласса). Реализуется с помощью ключ. слова extends.  **Основные правила наследования**:  1. Единичное наследование: класс может наследоваться только от одного суперкласса: public class Dog extends Animal { ... }  2. Полиморфизм: Подкласс может быть использован везде, где ожидается суперкласс: Animal animal = new Dog();  3. Переопределение методов: Подкласс может переопределить методы суперкласса для изменения или расширения их поведения.  public class Dog extends Animal {  @Override  public void makeSound() { System.out.println("Bark"); }  }  4. Вызов конструктора суперкласса: Конструктор подкласса должен явно или неявно вызывать конструктор суперкласса.  **Ключевое слово super** используется для:  1. Вызова конструктора суперкласса:  public class Animal { public Animal(String name) { ... } }  public class Dog extends Animal {  public Dog(String name) { super(name); }  }  2. Доступа к методам и полям суперкласса:  public class Animal {  public void makeSound() { System.out.println("Some sound"); }  }  public class Dog extends Animal {  @Override public void makeSound() {  super.makeSound(); System.out.println("Bark");  }  }  **Ключевое слово this** используется для:  1. Ссылки на текущий объект:  public class Dog {  private String name; Dog(String name) { this.name = name; }  }  2. Вызова других конструкторов текущего класса:  public class Dog {  private String name; public Dog(String name) { this(name, 0); }  } | 10. Переопределение методов. Методы подставки. «Неглубок.» и «глубокое» клонирование  **Переопределение методов** позволяет подклассу предоставить свою собственную реализацию метода, уже определенного в его суперклассе. **Основные правила переопределения методов**:  1. Метод в подклассе должен иметь ту же сигнатуру (имя и параметры), что и метод в суперклассе.  2. Возвращаемый тип метода в подклассе должен быть таким же или совместимым с возвращаемым типом метода в суперклассе.  3. Модификатор доступа метода в подклассе должен быть таким же или менее строгим, чем в суперклассе.  4. Рекомендуется исп. аннотацию @Override для указания, что метод переопределяется.  public class Dog extends Animal {  @Override  public void makeSound() { System.out.println("Bark"); }  }  В Java все методы, кроме static и final, являются **виртуальными**. Это означает, что при вызове метода через ссылку на суперкласс, в зависимости от фактического типа объекта, будет вызван метод подкласса (если он переопределен).  Animal myDog = new Dog();  myDog.makeSound(); // Вызов метода makeSound из класса Dog  **Клонирование** позволяет создавать копии объектов. Реализуется с помощью интерфейса Cloneable и метода clone().  При **неглубоком клонировании** копируются только основные поля объекта, а ссылки на вложенные объекты остаются прежними. Это означает, что изменения во вложенных объектах будут видны как в оригинале, так и в клоне.  При **глубоком клонировании** создаются новые объекты для всех вложенных объектов, таким образом, изменения в клоне не будут видны в оригинале. | 11. Внутренние классы. Вложенные классы. Анонимные классы  **Внутренний класс** — класс, опред. внутри другого класса. Внутр. классы могут иметь доступ к членам внешнего класса, включая приватные члены.  **Виды внутренних классов**:  1. Не статические внутр. классы: не явл. статическими и имеют доступ к полям и методам внешн. класса, включая приватные.  public class Outer {  private int outerField;  public class Inner { public void innerMethod() { … } }  }  2. Статические внутр. классы: Такие классы явл. статическими и могут быть созданы без создания экземпляра внешнего класса.  public class Outer {  private static int outerStaticField;  public static class StaticInner {  public void staticInnerMethod() { … }  }  }  **Вложенные классы** — это классы, опред. внутри другого класса или метода. Они могут быть статическими или нестатическими.  Пример нестатического вложенного класса:  public class Outer {  private int outerField;  public void outerMethod() {  class LocalInner { public void localInnerMethod() { … } }  LocalInner inner = new LocalInner();  inner.localInnerMethod();  }  }  **Анонимные классы** — это классы без имени, которые создаются на месте и используются для реализации одного интерфейса или наследования одного класса.  Пример анонимного класса для интерфейса:  public class Main {  public static void main(String[] args) {  Runnable runnable = new Runnable() {  @Override public void run() {…}  };  Thread thread = new Thread(runnable); thread.start();  }  } | 12. Интерфейсы. Виды интерфейсов. Параметризация  **Интерфейсы** опред. набор методов, кот. класс должен реализ., не предоставляя их реализаций. Позволяют реализовать множеств. наследование и обеспечивают контракт для классов.  Интерфейсы объявляются с исп-ем ключ. слова **interface**.  public interface Animal { void makeSound(); void eat(); }  Класс, который реализует интерфейс, должен предоставить реализацию всех его методов:  public class Dog implements Animal {  @Override public void makeSound() { System.out.println("Bark"); }  @Override public void eat() { System.out.println("Dog is eating"); }  }  **Виды интерфейсов**  **Функциональные интерфейсы** содержат только один абстракт. метод. Они исп. для лямбда-выражений и объявл. с аннотацией @FunctionalInterface.  @FunctionalInterface  public interface Calculator { int calculate(int a, int b); }  Использование лямбда-выражений с функц. интерфейсами:  Calculator add = (a, b) -> a + b;  System.out.println(add.calculate(5, 3)); // Output: 8  **Маркерные интерфейсы** не содержат методов. Они исп. для обозначения некоторого свойства класса. Примеры: Serializable, Cloneable.  public interface MarkerInterface {…}  **Обобщенные интерфейсы** позволяют параметризовать типы данных, используемые в интерфейсе.  public interface Container<T> { void add(T item); T get(); }  **Параметризация интерфейсов** позволяет исп. обобщенные типы (generics) для определения интерфейсов, которые могут работать с различными типами данных.  public interface Pair<K, V> { K getKey(); V getValue(); }  **Преимущества использования интерфейсов**  1. Множественное наследование: Класс может реализовывать несколько интерфейсов, что позволяет избежать проблем, связанных с множественным наследованием классов.  2. Полиморфизм: Интерфейсы позволяют исп. полиморфизм, создавая общий контракт для различных классов.  3. Гибкость: Интерфейсы делают код более гибким и масштабируемым, так как реализация может быть изменена без изменения интерфейса. |

|  |  |  |  |
| --- | --- | --- | --- |
| 13. Функциональные интерфейсы  **Функциональные интерфейсы** — это интерфейсы, содержащие ровно один абстрактный метод. Они предназначены для использования с лямбда-выражениями и методами ссылок, обеспечивая чистый и удобный способ написания кода.  Можно исп. аннотацию @FunctionalInterface, чтобы явно указать, что интерфейс функц. Это также позволяет проверять, что интерфейс действительно содержит только 1 абстр. метод.  @FunctionalInterface  public interface Calculator { int calculate(int a, int b); }  Java предоставляет мн-во предопределенных функциональных интерфейсов в пакете java.util.function.  1. Predicate: Принимает 1 аргумент и возвращает boolean.  Predicate<Integer> isEven = (n) -> n % 2 == 0;  System.out.println(isEven.test(4)); // Output: true  2. Function: Принимает 1 аргумент и возвращает результат.  Function<String, Integer> lengthFunction = (s) -> s.length();  System.out.println(lengthFunction.apply("hello")); // Output: 5  3. Consumer: Принимает 1 аргумент и ничего не возвращает.  Consumer<String> printConsumer = (s) -> System.out.println(s);  printConsumer.accept("hello world"); // Output: hello world  4. Supplier: Не принимает аргументов, но возвращает результат.  Supplier<Double> randomSupplier = () -> Math.random();  System.out.println(randomSupplier.get());  5. BiFunction: Принимает два аргумента и возвращает результат.  BiFunction<Integer, Integer, Integer> addFunc = (a, b) -> a + b;  System.out.println(addFunc.apply(2, 3)); // Output: 5  **Лямбда-выражения** обеспеч. компактный способ написания анонимных методов (или функций) и исп. для реализации функциональных интерфейсов.  @FunctionalInterface  public interface Calculator {  int calculate(int a, int b);  }  public class Main {  public static void main(String[] args) {  Calculator add = (a, b) -> a + b;  Calculator multiply = (a, b) -> a b;  System.out.println(add.calculate(5, 3)); // Output: 8  System.out.println(multiply.calculate(5, 3)); // Output: 15  }  } | 14. Аннотации. Встроенные аннотации. @Override, @Deprecated, @SuppressWarnings, @Retention, @Documented, @Target, @Inherited и др. Параметры аннотаций  **Аннотации** — это спец. метаданные, кот. могут быть добавлены к элементам кода (классам, методам, полям и т.д.). Они предоставляют информацию о программе компилятору и JVM, но не изменяют логику выполнения программы.  **@Override** используется для указания, что метод переопред. метод суперкласса. Она помогает компилятору поймать ошибки, если метод в не переопред. метод суперкласса.  public class Child extends Parent {  @Override public void display() { System.out.println("Child"); }  }  **@Deprecated** указ., что эл-т (метод, класс или поле) устарел и не рекомендуется к использованию. Также можно добавить описание причины устаревания и альтернативу.  **@SuppressWarnings** подавляет указанные предупреждения компилятора. Она часто исп. для подавления предупреждений, связанных с необработанными типами, устаревшими методами.  public class Example {  @SuppressWarnings("unchecked") public void method() {…}  }  **@Retention** указывает, насколько долго аннотация должна сохраняться: в исходном коде (SOURCE), в байт-коде (CLASS) или в ходе выполнения (RUNTIME).  @Retention(RetentionPolicy.RUNTIME)  public @interface MyAnnotation { String value(); }  **@Documented** указывает, что аннотация должна быть включена в документацию Javadoc.  @Documented public @interface MyAnnotation { String value(); }  **@Target** указывает, к каким элементам кода может быть применена аннотация (классы, методы, поля и т.д.).  @Target(ElementType.METHOD)  public @interface MyAnnotation { String value(); }  **@Inherited** указ., что анн-ия суперкласса будет наследоваться подклассами.  **Параметры аннотаций**  Аннотации могут содерж. параметры, которые можно задавать при их использовании. Параметры могут быть различного типа: примитивы, строки, перечисления, другие аннотации и массивы.  public @interface MyAnnotation { String value(); }  @MyAnnotation(value = "Example") public class Example {…} | 15. Создание пользовательских аннотаций В Java можно создавать **собств. аннотации**, кот. могут быть исп. для предоставления метаданных в вашем коде. Это полезно для создания собств. инструментов, библиотек и фреймворков.  **Основные шаги для создания пользовательской аннотации**  1. Используйте @interface для определения новой аннотации.  2. Укаж. мета-аннотации (@Retention, @Target, @Documented и @Inherited), чтобы настроить поведение вашей аннотации.  3. Определите параметры аннотации, если они необходимы.  **Простая аннотация**  import java.lang.annotation.Retention;  import java.lang.annotation.RetentionPolicy;  @Retention(RetentionPolicy.RUNTIME)  public @interface MySimpleAnnotation { }  Аннотация с параметрами  import java.lang.annotation.Retention;  import java.lang.annotation.RetentionPolicy;  @Retention(RetentionPolicy.RUNTIME)  public @interface MyAnnotation {  String value() default "default value"; int number() default 0;  }  **Использование мета-аннотаций** (где и как может применяться наша аннотация).  import java.lang.annotation.Retention;  import java.lang.annotation.RetentionPolicy;  import java.lang.annotation.Target;  import java.lang.annotation.ElementType;  import java.lang.annotation.Documented;  import java.lang.annotation.Inherited;  @Documented  @Inherited  @Retention(RetentionPolicy.RUNTIME)  @Target({ElementType.METHOD, ElementType.TYPE})  public @interface MyMetaAnnotation {  String description() default "No description";  }  **Пример использования аннотации на классе и методах**  @MyMetaAnnotation(description = "This is a test class")  public class TestClass {  @MyAnnotation(value = "Test method", number = 1)  public void tstMethod() { System.out.println("Testing method"); }  } | 16. Иерархия исключений и ошибок. Способы обр-ки исключений  **Исключения и ошибки** представлены в виде объектов, кот. явл. экземплярами классов, наследуемых от Throwable. Эти классы дел. на 2 категории: ошибки (Errors) и исключения (Exceptions).  **Иерархия Throwable**  1. Throwable: Корневой класс всех исключений и ошибок.  - Error: Классы ошибок, кот. обычно указ. на проблемы, кот. не стоит обрабатывать программно (например, OutOfMemoryError).  - Exception: Классы исключений, кот. могут быть обработаны программно.  - RuntimeException: Исключения времени выполнения, кот. могут возникнуть в любой момент выполнения программы и не проверяются компилятором (например, NullPointerException).  - Проверяемые исключения: остальные исключения, которые нужно обрабатывать (например, IOException, SQLException).  **Обработка исключений** в Java осуществляется с помощью ключевых слов try, catch, finally, и throw. Также можно создавать свои собственные исключения.  1. try-catch: Блок try содержит код, который может выбросить исключение, а блок catch содержит код для его обработки.  try {код, который может вызвать исключение}  catch (ArithmeticException e) {обработка исключения}  2. finally: Блок finally содержит код, который будет выполнен независимо от того, было выброшено исключение или нет. Обычно используется для освобождения ресурсов.  try {код, который может вызвать исключение}  catch (ArithmeticException e) {обработка исключения}  finally {код, который выполнится в любом случае}  3. throw: Используется для явного выбрасывания исключения.  public void checkAge(int age) {  if (age < 18) { throw new IllegalArgumentException("…"); }  }  4. throws: Используется в объявлении метода, чтобы указать, какие исключения могут быть выброшены этим методом.  public void readFile(String fileName) throws IOException {…}  Вы можете создавать свои **собственные исключения**, расширяя класс Exception или RuntimeException.  public class CustomException extends Exception {  public CustomException(String message) {  super(message);  }  } |
| 17. Стандартные коллекции и интерфейсы  Java предоставляет мощную и гибкую библиотеку коллекций в пакете java.util. **Коллекции** облегчают работу с группами объектов, предоставляя удобные и эффективные средства для их хранения, обработки и манипуляции. В этой библиотеке опред. разл. интерфейсы и классы для работы с коллекциями.  **Основные интерфейсы коллекций**  1. Collection<E>: Базовый интерфейс для всех коллекций, за исключением отображений (карт).  2. List<E>: Упоряд. список, кот. допускает дублирование эл-тов.  3. Set<E>: Коллекция, кот. не допускает дублирование эл-тов.  4. Queue<E>: Очередь, кот. упоряд. эл-ты по принципу FIFO.  5. Deque<E>: Двусторонняя очередь, которая поддерживает добавление и удаление элементов с обоих концов.  6. Map<K, V>: Коллекция, которая сопоставляет ключи с значениями (пары ключ-значение).  **Классы, реализующие интерфейс List<E>**  - ArrayList<E>: Динам. массив, кот. позволяет произвольный доступ к элементам.  List<String> arrayList = new ArrayList<>(); arrayList.add("A");  - LinkedList<E>: Двусвязный список, который позволяет быстрые вставки и удаления элементов.  **Классы, реализующие интерфейс Set<E>**  - HashSet<E>: Коллекция, которая не гарантирует порядок элементов и использует хеширование.  - LinkedHashSet<E>: Колл-ия, поддерж. порядок вставки эл-тов.  - TreeSet<E>: Отсортир. коллекция, хранит эл-ты в виде дерева.  **Классы, реализующие интерфейс Queue<E>**  - LinkedList<E>: Также реализует интерфейс Queue<E>.  - PriorityQueue<E>: Очередь с приоритетами, кот. упоряд. эл-ты по их естеств. порядку или по предоставленному компаратору.  **Классы, реализующие интерфейс Deque<E>**  - ArrayDeque<E>: Двусторонняя очередь на базе массива.  - LinkedList<E>: Также реализует интерфейс Deque<E>.  **Классы, реализующие интерфейс Map<K, V>**  - HashMap<K, V>: Коллекция пар ключ-значение, которая не гарантирует порядок элементов.  - LinkedHashMap<K, V>: Коллекция пар ключ-значение, которая поддерживает порядок вставки элементов.  - TreeMap<K, V>: Отсортированная коллекция пар ключ-значение, которая хранит элементы в виде дерева. | 18. Потоки ввода/вывода. Интерфейс Serializable. Классы для архивации  Java предоставляет мощную и гибкую библиотеку для работы с потоками ввода/вывода (I/O) в пакете java.io. Потоки позволяют работать с файлами, сетью и другими источниками данных.  **Основные классы и интерфейсы ввода/вывода**  1. **InputStream и OutputStream**: Базовые классы для работы с байтовыми потоками. 2. **Reader и Writer**: Базовые классы для работы с символьными потоками.  **Байтовые потоки** - FileInputStream и FileOutputStream: Для чтения и записи байтовых данных в файлы.  try (FileInputStream fis = new FileInputStream("input.txt");  FileOutputStream fos = new FileOutputStream("output.txt")) {  int byteData;  while ((byteData = fis.read()) != -1) { fos.write(byteData); }  } catch (IOException e) { e.printStackTrace(); }  **Символьные потоки** - FileReader и FileWriter: Для чтения и записи символьных данных в файлы.  try (FileReader reader = new FileReader("input.txt");  FileWriter writer = new FileWriter("output.txt")) {  int chData;  while ((chData = reader.read()) != -1) { writer.write(chData); }  } catch (IOException e) { e.printStackTrace(); }  **Интерфейс Serializable** используется для указания, что объекты класса могут быть сериализованы, то есть их состояние может быть записано в поток и восстановлено из него. Этот интерфейс не содержит методов и является маркерным.  1. Сериализация: Запись объекта в файл.  2. Десериализация: Чтение объекта из файла.  Java пред. **классы для работы с архивами**, такие как ZIP и GZIP.  **Работа с ZIP файлами**  - ZipOutputStream: Для создания ZIP архива.  - ZipInputStream: Для чтения ZIP архива.  FileOutputStream fos = new FileOutputStream(zipFile);  ZipOutputStream zos = new ZipOutputStream(fos))  **Работа с GZIP файлами**  - GZIPOutputStream: Для создания GZIP архива.  - GZIPInputStream: Для чтения GZIP архива.  FileOutputStream fos = new FileOutputStream(gzipFile);  GZIPOutputStream gzos = new GZIPOutputStream(fos)) | 19. Схема XSD. Простые и сложные типы  **XSD (XML Schema Definition)** – язык для опред-ия структуры и содержания XML доков. С помощью XSD можно описать эл-ты и атрибуты, типы данных, порядок и кол-во их появления в доке.  **Основные компоненты XSD**  - Элементы: Основные строительные блоки XML документа.  - Атрибуты: Дополнительные данные для элементов.  - Типы данных: Описание типа данных для эл-тов и атрибутов.  - Простые типы: Типы данных без вложенной структуры.  - Сложные типы: Типы данных с вложенной структурой.  **Простые типы данных** определяют значения, которые не могут иметь вложенную структуру (например, строки, числа, даты).  1. xs:string: Строковый тип данных.  2. xs:integer: Целочисленный тип данных.  3. xs:date: Дата.  4. xs:boolean: Логический тип данных.  <xs:element name="person"><xs:complexType><xs:sequence>  <xs:element name="name" type="xs:string"/>  <xs:element name="age" type="xs:integer"/>  <xs:element name="birthdate" type="xs:date"/>  </xs:sequence></xs:complexType></xs:element>  **Сложные типы** данных могут содержать др. эл-ты и атрибуты, определяя структуру XML документа. Объявляются с помощью эл-та <xs:complexType> и могут сод. послед-сти (<xs:sequence>), выборы (<xs:choice>) и атрибуты (<xs:attribute>).  <xs:element name="company"><xs:complexType><xs:sequence>  <xs:element name="employee" maxOccurs="unbounded">  <xs:complexType><xs:sequence>  <xs:element name="name" type="xs:string"/>  <xs:element name="position" type="xs:string"/>  <xs:element name="salary" type="xs:decimal"/>  </xs:sequence>  <xs:attribute name="i" type="xs:integer" use="required"/>  </xs:complexType>  </xs:element></xs:sequence></xs:complexType></xs:element>  Простые типы можно объяв. с помощью эл-та **<xs:simpleType>**. Можно исп. встроенные типы или создавать собственные.  <xs:simpleType name="ageType"><xs:restriction base="xs:integer">  <xs:minInclusive value="0"/><xs:maxInclusive value="120"/>  </xs:restriction></xs:simpleType>  <xs:element name="age" type="ageType"/> | 20. JAXB. Маршаллизация и демаршаллизация  JAXB (Java Architecture for XML Binding) – это фреймворк для преобразования Java объектов в XML и обратно. Он позволяет автоматизировать процесс сер-ции (марш-ции) Java объектов в XML и десер-ции (демарш-ции) XML обратно в Java объекты.  Аннотации JAXB: Исп. для описания того, как Java объект должен быть предст. в XML. Осн. аннотации JAXB: @XmlRootElement - корневой эл-т XML; @XmlElement – эл-т XML; @XmlAttribute - атрибут XML; @XmlType - порядок элементов в XML.  **Определение класса Java**  @XmlRootElement public class Person {  private String name;  public Person() {} // Пустой конструктор необходим для JAXB  public Person(String name) { this.name = name; }  @XmlElement public String getName() { return name; }  public void setName(String name) { this.name = name; }  }  **Маршаллизация: Преобразование объекта в XML**  Person person = new Person("John Doe");  try {  JAXBContext context = JAXBContext.newInstance(Person.class);  Marshaller marshaller = context.createMarshaller();  marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, Boolean.TRUE);  StringWriter writer = new StringWriter();  marshaller.marshal(person, writer);  String xmlString = writer.toString();  System.out.println(xmlString);  } catch (JAXBException e) { e.printStackTrace(); }  **Демаршаллизация: Преобразование XML в объект**  String xmlString = "<person><name>John Doe</name></person>";  try {  JAXBContext context = JAXBContext.newInstance(Person.class);  Unmarshaller unmarshaller = context.createUnmarshaller();  StringReader reader = new StringReader(xmlString);  Person person = (Person) unmarshaller.unmarshal(reader);  System.out.println(person.getName());  System.out.println(person.getAge());  } catch (JAXBException e) { e.printStackTrace(); } |
| 21. Стратегии обработки XML документов: DOM, SAX, StAX – сравнение записи, чтения и поиска  Существуют три основных стратегии обработки XML документов в Java: DOM, SAX и StAX. Каждая из них имеет свои особенности и подходит для решения определенных задач.  **DOM (Document Object Model)**:  **- Чтение**: строит полное внутреннее представление XML-дока в памяти в виде древовидной структуры. Это позволяет легко перемещаться по документу, выполнять навигацию и поиск. Но для больших доков это требует большого объема памяти.  **- Запись**: Модификация документа также происходит через работу с объектной моделью в памяти, что делает ее удобной для изменения структуры и содержимого XML-документа.  **- Поиск**: Поиск эл-тов и атрибутов очень гибкий и мощный, так как можно использовать методы навигации по дереву.  **SAX (Simple API for XML)**:  **- Чтение**: выполняет последовательный разбор XML-дока, генерируя события для различных его частей (начало/конец эл-та, текстовое содержимое и т.д.). Это позволяет обрабатывать большие документы, не загружая весь документ в память.  **- Запись**: не предоставляет прямой поддержки для записи XML-доков. Однако можно создавать XML-доки, последовательно генерируя SAX-события.  **- Поиск**: Поиск в более сложен, так как нужно самостоятельно отслеживать состояние разбора и искать нужные элементы.  **StAX (Streaming API for XML)**:  **- Чтение**: похож на SAX, но предост. более удобный прогр. интерфейс с исп-ием итераторов для навигации по доку.  **- Запись**: StAX позволяет последовательно создавать XML-документы, генерируя необходимые события.  **- Поиск**: Поиск в StAX проще, чем в SAX, благодаря возможности использовать итераторы для навигации по документу.  В общем случае:  - DOM лучше подходит для небольших документов, где требуется гибкая навигация и модификация.  - SAX эффективнее для больших документов, где важна минимизация потребления памяти.  - StAX является компромиссом между DOM и SAX, обеспечивая более удобный программный интерфейс.  Выбор стратегии зависит от конкретных требований вашего приложения: размер документов, частота чтения/записи, сложность навигации и поиска. | 22. Json парсеры. Tree Model. Data bind. Streaming API  **JSON (JavaScript Object Notation)** - это легкий формат обмена данными, который широко используется в современных веб-приложениях для передачи данных между клиентом и сервером. Для работы с JSON в Java существуют различные библиотеки парсинга, которые предоставляют различные подходы, такие как Tree Model, Data Binding и Streaming API.  **Tree Model.** В модели дерева весь JSON документ загружается в память в виде дерева объектов, что позволяет легко манипул. и обращаться к данным. Примеры таких библиотек в Java: Gson, Jackson ObjectMapper.  public class GsonExample {  public static void main(String[] args) {  String json = "{\"name\":\"John\"}"; Gson gson = new Gson();  Person person = gson.fromJson(json, Person.class);  System.out.println("Name: " + person.getName());  }  }  **Подход Data Bind** позволяет преобразовывать JSON данные непосредственно в объекты Java и наоборот. Он использует аннотации или сопоставление по имени для связывания полей JSON с полями Java объекта. Примеры библиотек: Jackson, Gson.  String json = "{\"name\":\"John\"}";  ObjectMapper mapper = new ObjectMapper();  Person person = mapper.readValue(json, Person.class);  System.out.println("Name: " + person.getName());  **Streaming API** обрабатывает JSON документ поэлементно без загрузки всего документа в память. Он читает и записывает данные по мере их поступления, что позволяет работать с большими файлами или стримами данных без необходимости загрузки их полностью в память. Библиотеки: Jackson, Gson.  JsonFactory factory = new JsonFactory();  JsonParser parser = factory.createParser(new File("data.json"));  while (!parser.isClosed()) {  JsonToken token = parser.nextToken();  if (token == null) { break; }  if (token == JsonToken.FIELD\_NAME && "name".equals(parser.getCurrentName())) {  parser.nextToken();  System.out.println("Name: " + parser.getText());  }  } parser.close(); } } | 23. Потоки выполнения. Thread и интерфейс Runnable  Потоки выполнения в Java позволяют вып. несколько задач одноврем. или асинхронно. 2 осн. способа создания потоков в Java – исп-ие класса Thread и реализация интерфейса Runnable.  **Thread** является основным классом для работы с потоками в Java. Вы можете создать новый поток, наследуясь от класса Thread и переопределяя метод run(), который содержит код задачи, выполняемой в этом потоке.  class MyThread extends Thread {  public void run() { System.out.println("Thread is running"); }  }  public class ThreadExample {  public static void main(String[] args) {  MyThread thread = new MyThread();  thread.start(); // Запускает выполнение потока  }  }  **Интерфейс Runnable** предоставляет еще один способ создания потоков в Java. Он опред. только 1 метод run(), кот. содержит код задачи, выполняемой в потоке. После этого, объект, реализующий Runnable, перед. в конструктор класса Thread.  class MyRunnable implements Runnable {  public void run() { System.out.println("Runnable is running"); }  }  public class RunnableExample {  public static void main(String[] args) {  Thread thread = new Thread(new MyRunnable());  thread.start(); // Запускает выполнение потока  }  }  **Сравнение использования Thread и Runnable**  - Исп-ие Runnable предпочтительнее, поскольку позволяет избежать ограничения множественного наследования в Java, так как классы могут реализовывать мн-во интерфейсов, но наследоваться только от одного класса.  - Кроме того, разделяя код выполнения и код потока, Runnable делает код более модульным и улучшает его читаемость.  - Если вам нужно только выполнить какую-то задачу в фоновом режиме и не требуется управление жизненным циклом потока, Runnable может быть более подходящим выбором. | 24. Жизненный цикл потока. Управление приоритетами и группами потоков  Жизненный цикл потока в Java описывает его состояния от создания до завершения. Управление приоритетами и группами потоков позволяет оптимизировать выполнение задач и эффективно использовать ресурсы процессора.  Жизненный цикл потока:  1. Создание (New): Поток создается, но еще не запущен.  2. Готовность (Runnable): Поток готов к выполнению, ожидает выделения процессорного времени.  3. Выполнение (Running): Поток выполняется.  4. Приостановка (Blocked/Waiting): Поток приостанавливается, например, из-за ожидания ввода-вывода или монитора.  5. Ожидание (Timed Waiting): Поток ждет определенного времени перед возобновлением выполнения.  6. Завершение (Terminated): Поток завершается и уничтожается.  **Приоритет потока** определяет, как часто ему будет выделяться процессорное время. Java поддерживает диапазон приоритетов от 1 до 10, где 1 - наименьший приоритет, а 10 - наивысший.  Thread thread = new Thread();  thread.setPriority(Thread.MAX\_PRIORITY);  **Группа потоков** - это объект, который объединяет несколько потоков вместе. Группы потоков позволяют управлять одновременно несколькими потоками и обрабатывать их как единое целое.  ThreadGroup group = new ThreadGroup("MyThreadGroup");  Thread thread1 = new Thread(group, new MyRunnable());  Thread thread2 = new Thread(group, new MyRunnable()); |

|  |  |  |  |
| --- | --- | --- | --- |
| 25. Атомарные типы и модификатор volatile. Методы wait(), notify() и их использование  Атомарные типы данных и модификатор volatile - это механизмы в Java, предназнач. для обеспечения безопасности потоков и корректной работы с общими ресурсами.  **Атомарные типы данных** обеспечивают атомарные операции чтения/записи для примитивных типов данных. Атомарными являются операции чтения/записи для типов int и long, а также операции для ссылочных типов данных.  public class AtomicExample {  private AtomicInteger count = new AtomicInteger(0);  public void increment() { count.incrementAndGet(); }  public int getCount() { return count.get(); }  }  **Мод-р volatile** применяется к переменным и гарантирует, что изменения этой переменной будут видны всем потокам. Он предотвращает кэширование пер-ной в локал. памяти потока.  public class VolatileExample {  private volatile boolean flag = false;  public void setFlag(boolean newValue) { flag = newValue; }  public boolean getFlag() { return flag; }  }  **Методы wait() и notify()** используются для синхронизации выполнения потоков и управления их состоянием.  - wait(): Помещает поток в режим ожидания, пока другой поток не вызовет метод notify() или notifyAll() для этого же объекта.  - notify(): Будит один из потоков, ожидающих на этом объекте, чтобы продолжить выполнение. Если есть несколько потоков в режиме ожидания, будет выбран случайный из них.  public class WaitNotifyExample {  public void doWork() {  synchronized (this) {  try {  System.out.println("Thread is waiting"); wait();  System.out.println("Thread is awake");  } catch (InterruptedException e) { e.printStackTrace(); }  }  }  public void notifyThread() { synchronized (this) { notify(); } }  } | 26. Методы synchronized. Инструкция synchronized  Методы synchronized и инструкция synchronized - это ключевые механизмы в Java для обеспечения безопасности потоков путем синхронизации доступа к общим ресурсам.  **Методы synchronized**:  1. Синхронизация метода:  public synchronized void syncMethod() { Критическая секция }  Этот подход синхронизирует весь метод, что означает, что только один поток может выполнять этот метод одновременно для данного экземпляра объекта.  2. Синхронизация блока кода:  public void someMethod() {  synchronized(this) { Критическая секция }  }  В этом случае только код внутри блока synchronized будет синхронизирован, что позволяет гибко управлять крит. секцией.  **Инструкция synchronized**:  1. Синхронизация метода:  В байт-коде Java инструкция synchronized применяется к методам для обеспечения блокировки на уровне объекта.  public synchronized void synchMethod() { Код метода }  2. Синхронизация блока кода:  Инструкция synchronized также может использоваться для синхронизации блоков кода.  synchronized(obj) { Критическая секция } | 27. Многопоточность. Блокирующие очереди. Semaphore и CyclicBarrier  **Многопоточность** – спос-ть программы или языка прогр-ия обрабатывать и выполнять несколько потоков одновременно. Блокирующие очереди, Semaphore и CyclicBarrier - некот. из инструментов, предоставляемых Java для реализации синхронизации и координации многопоточных задач.  **Блокирующие очереди** предост. механизм для передачи данных между потоками. Они блокируются, когда пытаются добавить эл-т в заполненную очередь или извлечь элемент из пустой очереди, что обесп. безопасность в многопоточ. среде.  BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(10);  new Thread(() -> {  try {  queue.put(1);  System.out.println("Added element to queue");  } catch (InterruptedException e) { e.printStackTrace(); }  }).start();  new Thread(() -> {  try {  int element = queue.take();  System.out.println("Removed element: " + element);  } catch (InterruptedException e) { e.printStackTrace(); }  }).start();  **Semaphore** - счетчик, который управляет доступом к ресурсам. Он используется для ограничения количества потоков, которые могут получить доступ к опред. ресурсу одновременно.  **CyclicBarrier** - точка синхронизации, где некот. кол-во потоков должно остановиться и дождаться, пока не все потоки достигнут этой точки, после чего выполняется опред. действие. | 28. Многопоточность. CountDownLatch. Phaser  CountDownLatch и Phaser - 2 мощных инструмента, предоставл. Java для координации многопоточных задач. Они позв. потокам синхронизироваться и совместно выполнять операции.  **CountDownLatch** - счетчик, кот. позволяет одному или неск. потокам ждать, пока другие потоки завершат выполнение определенного числа операций.  public class CountDownLatchExample {  public static void main(String[] args) throws InterruptedException {  CountDownLatch latch = new CountDownLatch(3);  for (int i = 0; i < 3; i++) {  new Thread(() -> {  try {  Thread.sleep(1000); // Имитация работы потока  System.out.println("Thread completed its task");  latch.countDown(); // Уменьшение счетчика на 1  } catch (InterruptedException e) { e.printStackTrace(); }  }).start();  }  latch.await(); // Ожидание, пока счетчик не станет равен нулю  System.out.println("All threads have completed their tasks");  }  }  **Phaser** - механизм, кот. позволяет синхронизировать выполнение потоков в несколько фаз. Каждая фаза - это точка синхронизации, где потоки ожидают, пока не все потоки достигнут этой точки.  public class PhaserExample {  public static void main(String[] args) {  Phaser phaser = new Phaser(3);  for (int i = 0; i < 3; i++) {  new Thread(() -> {  System.out.println("Waits at phase " + phaser.getPhase());  phaser.arriveAndAwaitAdvance();  System.out.println("Cont after phase " + phaser.getPhase());  }).start();  }  phaser.arriveAndDeregister();  System.out.println("All threads have completed their tasks");  }  } |
| 29. JDBC Типы драйверов. Установка соединения, выполнение запросов  **JDBC (Java Database Connectivity)** - API, предоставляющее доступ к разл. реляц. базам данных из Java-приложений. При работе с JDBC важно понимать типы драйверов, установку соединения с базой данных и выполнение запросов.  **Типы драйверов JDBC:**  1. JDBC-ODBC мост (JDBC-ODBC Bridge Driver):  - Использует ODBC (Open Database Connectivity) для обеспечения связи между Java-приложениями и БД.  - Требует наличия ODBC-драйвера для каждой БД.  - Не рекомендуется для новых разработок, т. к. поддержка ODBC снижает производительность и надежность.  2. Нативный драйвер JDBC (Native-API Driver):  - Использует библиотеки, предоставляемые поставщиками баз данных, для взаимодействия с БД.  - Напрямую связан с конкретной БД и обеспечивает более высокую производительность, чем мост JDBC-ODBC.  - Каждый поставщик БД предост. собств. нативный драйвер.  3. Сетевой протокольный драйвер (Network Protocol Driver):  - Взаимод. с БД через сетевой протокол, такой как TCP/IP.  - Предоставляет независимость от конкретной БД и обеспечивает поддержку для большинства баз данных.  4. JDBC-техн-ия встроенного сервера (JDBC-technology Server):  - Взаимодействует с сервером БД, который предоставляет JDBC-сервер.  - Позволяет Java-приложениям работать с БД, находящимися на удаленных серверах.  Для **установки соединения с БД** и выполнения запросов в JDBC необходимо выполнить следующие шаги:  1. Загрузка драйвера: Class.forName("com.mysql.jdbc.Driver");  2. Установка соединения:  Connection connect = DriverManager.getConnection ("jdbc:mysql://localhost:3306/db\_name", "username", "password");  3. Созд. запроса: Statement stat = connect.createStatement();  4. Выполнение запроса:  ResultSet resultSet = stat.executeQuery("SELECT FROM table");  5. Обработка результатов: while (resultSet.next()) { … }  6. Закрытие ресурсов:  resultSet.close(); statement.close(); connection.close(); | 30. JDBC. Обработка результатов запроса  **JDBC (Java Database Connectivity)** - API, предоставляющее доступ к разл. реляц. базам данных из Java-приложений. При работе с JDBC важно понимать типы драйверов, установку соединения с базой данных и выполнение запросов.  **Обработка результатов запроса** в JDBC вкл. в себя извлечение данных из объекта ResultSet, кот. представляет набор рез-тов запроса к БД. Для обработки результатов запроса в JDBC обычно используются циклы и методы доступа к данным ResultSet.  1. Извлечение данных из ResultSet: обычно используются методы доступа, такие как getInt(), getString(), getDouble() и т. д., которые позволяют получить значения столбцов по их индексу или имени.  ResultSet resultSet = statement.executeQuery("SELECT id, name, age FROM users");  while (resultSet.next()) {  int id = resultSet.getInt("id");  String name = resultSet.getString("name");  int age = resultSet.getInt("age");  // Обработка данных  }  2. Проверка наличия данных: исп. метод next(). Когда метод next() возвр. false, это означает, что больше нет строк данных.  while (resultSet.next()) { Обработка данных }  3. Закрытие ResultSet, Statement и Connection: Важно закрывать ресурсы ResultSet, Statement и Connection после завершения работы с ними, чтобы избежать утечки ресурсов.  resultSet.close(); statement.close(); connection.close();  При обработке результатов запроса в JDBC важно учитывать возможные **исключительные ситуации**, такие как ошибки при выполнении запроса, отсутствие соединения с БД и другие.  try {  Connection connection = DriverManager.getConnection ("jdbc:mysql://localhost:3306/my\_db", "username", "password");  Statement statement = connection.createStatement();  ResultSet resultSet = statement.executeQuery("SELECT FROM my\_table");  // Обработка результатов запроса  resultSet.close(); statement.close(); connection.close();  } catch (SQLException e) { e.printStackTrace(); } | 31. Модульное тестирование на основе JUnit. Принципы написания тестов  **Модульное тестирование** на основе JUnit - процесс проверки отдельных модулей или компонентов ПО для обеспечения их корректности и соответствия ожидаемому поведению. **Принципы написания тестов** включают в себя набор правил и рекомендаций, кот. помогают создать эффект. и надежн. тесты.  1. Каждый тест должен быть независимым от других тестов. Это означает, что результат выполнения одного теста не должен влиять на результаты других тестов.  2. Тесты должны быть автоматизированы, чтобы их можно было запускать без необходимости ручного вмешательства. Это обесп. быстрое выполнение тестов и повышает их надежность.  3. Тесты должны быть понятными и легко читаемыми, чтобы другие разработчики могли легко понять их назначение и ожидаемое поведение.  4. Тесты должны охватывать все возможные случаи использования и все ветви кода. Это помогает обнаружить потенциальные проблемы и ошибки в ПО.  5. Тесты должны быть изолированы от внешних зависимостей, таких как базы данных, сетевые ресурсы или внешние службы. Для этого исп. заглушки (mock objects) или фиктивные объекты.  public class MyMathTest {  @Test  public void testAddition() {  MyMath math = new MyMath();  int result = math.add(3, 5);  assertEquals(8, result); // Проверка ожидаемого результата  }  @Test  public void testSubtraction() {  MyMath math = new MyMath();  int result = math.subtract(10, 3);  assertEquals(7, result); // Проверка ожидаемого результата  }  } | 32. JUnit. Основные аннотации. проверки  **JUnit** предоставляет набор аннотаций для определения и управления тестами, а также для проверки ожидаемых результатов. **Вот осн. аннотации и проверки**:  1. @Test: Обозначает метод как тестовый.  2. @Before: Метод будет выполняться перед каждым тестовым методом. Исп. для иниц-ии ресурсов перед каждым тестом.  3. @After: Метод будет выполняться после каждого тестового метода. Исп. для освобождения ресурсов после каждого теста.  4. @BeforeClass: Метод будет выполняться 1 раз перед запуском всех тестов. Исп. для иниц-ии общих ресурсов перед запуском тестов.  5. @AfterClass: Метод будет выполняться один раз после выполнения всех тестов. Используется для освобождения общих ресурсов после выполнения всех тестов.  **Проверки (Assertions) в JUnit:**  1. assertEquals(expected, actual):  - Проверяет, что ожидаемое значение равно фактическому значению.  - Пример: assertEquals(10, result);  2. assertTrue(boolean condition):  - Проверяет, что условие истинно.  - Пример: assertTrue(result > 0);  3. assertFalse(boolean condition):  - Проверяет, что условие ложно.  - Пример: assertFalse(result < 0);  4. assertNotNull(object):  - Проверяет, что объект не является нулевым.  - Пример: assertNotNull(result);  5. assertNull(object):  - Проверяет, что объект является нулевым.  - Пример: assertNull(result);  6. assertSame(expected, actual):  - Проверяет, что две ссылки указывают на один и тот же объект.  - Пример: assertSame(expected, result);  7. assertNotSame(expected, actual):  - Проверяет, что две ссылки указывают на разные объекты.  - Пример: assertNotSame(expected, result); |
| 33. Модульное тестирование TestNG  **TestNG (Test Next Generation)** - фреймворк для модульного тестирования Java-приложений, кот. предост. более широкий набор функц-ти по сравнению с JUnit. Он поддерж. аннотации, параметризованные тесты, группировку тестов, зависимости между тестами и многое другое.  **Вот некоторые особенности TestNG:**  1. Аннотации для определения тестов, настроек и действий перед и после тестов. Это позволяет управлять тестами более гибко и читаемо.  2. Позволяет создавать параметризованные тесты, где один и тот же тест может быть запущен с разн. наборами входных данных.  3. Можно опред. группы тестов и запускать только выбранные группы. Это полезно для организации и запуска различных наборов тестов.  4. Позволяет определять зависимости между тестами, где один тест зависит от успешного выполнения другого теста.  5. Предоставляет подробные отчеты о выполнении тестов, которые позволяют легко отслеживать результаты тестирования.  public class MyMathTest {  @Test public void testAddition() {  MyMath math = new MyMath();  int result = math.add(3, 5); assertEquals(result, 8);  }  @Test public void testSubtraction() {  MyMath math = new MyMath();  int result = math.subtract(10, 3); assertEquals(result, 7);  }  }  **Запуск тестов TestNG:**  1. Создать конфигурационный файл XML для указания классов тестов, групп тестов и других параметров.  2. Многие среды разработки, такие как IntelliJ IDEA и Eclipse, имеют интеграцию с TestNG и позволяют запускать тесты непосредственно из IDE.  3. TestNG также поддерживает запуск тестов из командной строки с помощью утилиты TestNG. | 34. Сетевые программы java.net Понятие клиент- сервер  Java предоставляет набор классов в пакете java.net, которые позволяют разрабатывать сетевые приложения, включая клиент-серверные приложения. Понятие клиент-сервер в сетевом программировании описывает архитектурную модель, где клиент и сервер взаимодействуют друг с другом по сети.  **Клиент:**  - Клиент - это приложение или устройство, которое отправляет запросы к серверу и получает ответы от него.  - В клиент-серверной модели клиент обычно инициирует соединение с сервером, отправляет запрос и ожидает ответа.  - Клиент может быть представлен в виде веб-браузера, мобильного приложения, десктопного приложения и т. д.  **Сервер:**  - Сервер - это приложение или устройство, которое ожидает запросы от клиентов, обрабатывает их и отправляет ответы.  - Сервер обычно работает на удаленной машине и ждет запросы от клиентов по определенному сетевому порту.  - Сервер может быть веб-сервером, почтовым сервером, файловым сервером и т. д.  **Пример клиент-серверной модели:**  1. Клиентское приложение:  - Клиентское приложение может отправлять запросы к серверу, например, запрос на получение веб-страницы или запрос на загрузку файла.  - Пример: веб-браузер, который отправляет запросы к веб-серверу для получения веб-страниц.  2. Серверное приложение:  - Серверное приложение принимает запросы от клиентов, обрабатывает их и отправляет ответы.  - Пример: веб-сервер, который принимает HTTP-запросы от веб-браузеров и отправляет HTML-страницы в ответ. | 35. Сокеты. Установка соединения по протоколу TCP\IP и обмен. Серверный сокет ServerSocket. Клиентский сокет Socket  **Сокеты** - конечные точки двустороннего соединения между двумя программами на сети. Они позв. программам обмен. данными через сеть, независимо от того, где они находятся.  **Установка соединения по протоколу TCP/IP:**  1. Серверный сокет (ServerSocket):  - ServerSocket - серверный сокет, кот. прослушивает опред. порт на сервере и ожидает подключения от клиентов.  ServerSocket serverSocket = new ServerSocket(8080);  Socket clientSocket = serverSocket.accept(); // Ожидание клиента  2. Клиентский сокет (Socket):  - Socket - клиентский сокет, который инициирует соединение с сервером через определенный порт.  Socket socket = new Socket("localhost", 8080);  **Обмен данными между клиентом и сервером**:  1. Чтение и запись данных:  - После установки соединения клиент и сервер могут читать и записывать данные через соответств. потоки ввода-вывода.  // Получение потоков ввода-вывода для обмена данными  BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));  PrintWriter out = new PrintWriter(socket.getOutputStream(), true);  // Отправка данных от клиента серверу  out.println("Hello, server!");  // Чтение данных от сервера клиентом  String response = in.readLine();  2. Закрытие сокетов и потоков:  - После завершения обмена данными сокеты и потоки ввода-вывода должны быть закрыты для освобождения ресурсов.  // Закрытие потоков ввода-вывода и сокетов  in.close();  out.close();  socket.close(); | 36. Типы и принципы работы web-приложений.  **Web-приложения** - программные приложения, кот. работают через интернет и доступны через веб-браузер. Они позволяют пользователям взаимод. с данными и функциональностью, предоставляемой сервером, с помощью веб-интерфейса.  **Типы web-приложений:**  1. Статические web-приложения: набор веб-страниц, которые отображаются пользователю без изменений и обновлений. Они обычно создаются с использованием HTML, CSS и JavaScript.  2. Динамические web-приложения: генерируют содержимое веб-страницы на сервере в зависимости от запросов пользователя. Исп. серверные технологии, такие как PHP, Python, Ruby, Java и т. д., для обработки запросов и взаимодействия с базами данных.  3. Серверные web-приложения: работают на сервере и отвечают на запросы клиентов, обрабатывая данные и отправляя ответы. Могут быть реализованы как динам. приложения, обрабатывая запросы с исп-ем серверных языков программирования.  4. Клиентские web-приложения выполняются на стороне клиента, обычно в веб-браузере, и взаимодействуют с сервером для получения данных и обновлений. Могут быть написаны с исп-ем JS и работать с сервером через AJAX-запросы или WebSocket.  **Принципы работы web-приложений:**  1. Веб-приложения следуют принципу клиент-сервер, где клиент отпр. запросы серверу, а сервер отвечает на них.  2. Веб-приложения используют протокол HTTP для обмена данными между клиентом и сервером. Клиент отправляет HTTP-запросы, а сервер отвеч. на них HTTP-ответами.  3. Для улучшения масштабируемости и обслуживания веб-приложения обычно разделяют frontend и backend.  4. Обработка запросов на сервере: Серверные web-приложения обрабатывают запросы клиентов, выполняя необходимые операции, взаимодействуя с базами данных и генерируя ответы.  5. Многие web-приложения требуют доступа к БД для хранения и получ. данных (с исп. SQL или ORM (Object-Relational Mapping)).  6. Важным аспектом веб-приложений является обеспечение безопасности, включая защиту от атак, аутентификацию пользователей, контроль доступа и шифрование данных. |

|  |  |  |  |
| --- | --- | --- | --- |
| 37. Структура и параметры HTTP протокола. Понятие Request/ Response  **HTTP** - протокол передачи гипертекста, который используется для обмена данными между клиентом и сервером в сети Интернет. Он опред. формат запросов и ответов, кот. клиент и сервер отпр. друг другу.  **Структура HTTP протокола:**  1. Запрос (Request): Каждый запрос начинается с строки запроса, кот. содержит метод запроса, URL и версию протокола. Затем следуют заголовки запроса, которые содержат дополнительную информацию о запросе, такую как тип содержимого, параметры аутентификации и т. д. После заголовков может следовать тело запроса, которое содержит данные, отправляемые с запросом (например, данные формы в POST запросе).  2. Ответ (Response): Каждый ответ начин. с строки состояния, кот. содержит версию протокола, код состояния и текстовое описание состояния. Затем следуют заголовки ответа, которые содержат дополнительную информацию о ответе, такую как тип содержимого, дата и время создания и т. д. После заголовков может следовать тело ответа, кот. сод. данные, отправленные с ответом (например, HTML-страница или данные JSON).  **Параметры HTTP протокола:**  1. Методы запроса (Request Methods): HTTP опред. различные методы запроса, такие как GET, POST, PUT, DELETE и т. д., кот. определяют тип операции, выполняемой клиентом на сервере.  2. Коды состояния (Status Codes): HTTP определяет различные коды состояния, такие как 200 OK, 404 Not Found, 500 Internal Server Error и т. д., которые определяют результат выполнения операции на сервере.  3. Заголовки (Headers): Заголовки содержат дополнительную информацию о запросе или ответе, такую как тип содержимого, дата и время, параметры аутентификации и т. д.  4. Тело (Body): Тело сод. данные, отправляемые с запросом или ответом. Напр-р, в POST запросе тело может сод. данные формы. | 38. HTTP методы запроса и коды состояний  **HTTP методы запроса** определяют тип операции, которую клиент хочет выполнить с ресурсом на сервере. Каждый метод имеет свою специфическую функцию и предназначен для определенных действий. **Вот основные HTTP методы запроса**:  1. GET: Исп. для запроса ресурса с сервера. Пар-ры запроса могут быть переданы в URL. Не должен изменять состояние сервера.  2. POST: Исп. для отправки данных на сервер для обработки. Пар-ры запроса передаются в теле запроса. Может изменять состояние сервера (например, создание нового ресурса).  3. PUT: Используется для обновления или создания ресурса на сервере. Параметры запроса передаются в теле запроса. Часто используется для обновления существующего ресурса.  4. DELETE: Исп. для удаления ресурса на сервере. Пар-ры запроса м. б. переданы в URL. Удаляет указ. ресурс с сервера.  5. PATCH: Исп. для частичного обновления ресурса на сервере. Позволяет отправить только измененные данные. Может быть полезен, когда требуется обновить только опред. поля ресурса.  6. HEAD: Аналогичен методу GET, но сервер отправляет только заголовки ответа без тела. Часто используется для получения метаданных о ресурсе без его фактического получения.  7. OPTIONS: Используется для запроса информации о доступных методах или параметрах ресурса на сервере. Позволяет клиенту определить, какие операции можно выполнить с ресурсом.  **Коды состояний HTTP:**  HTTP статус-коды используются для передачи информации о состоянии выполнения запроса клиента к серверу. Они разделяются на пять групп:  1xx - Информационные статусы - обозначают, что сервер получил запрос и продолжает процесс обработки.  2xx - Успешные статусы - обозначают успешное выполнение запроса клиента к серверу.  3xx - Перенаправления - обозначают, что клиент должен выполнить дополнительные действия, чтобы завершить запрос.  4xx - Ошибки клиента - обозначают ошибки, произошедшие из-за некорректного запроса клиента.  5xx - Ошибки сервера - обозначают ошибки, произошедшие на стороне сервера в процессе обработки запроса. | 39. Арх-ра Java EE. Станд. контейнеры Java EE. Java EE Сервера  Java Platform, Enterprise Edition предост. набор спецификаций и стандартов для разработки и выполнения масштабируемых и надежных приложений. **Архитектура Java EE:**  1. Приложение: Java EE приложение состоит из компонентов, которые разрабатываются и настраиваются для выполнения определенных функций в приложении.  2. Контейнеры: Java EE предоставляет контейнеры для выполнения различных типов компонентов приложения, таких как сервлеты, EJB (Enterprise JavaBeans), JSP (JavaServer Pages), JMS (Java Message Service) и т. д.  3. API (Application Programming Interface): Java EE определяет API для разработки компонентов приложения и их взаимодействия с контейнерами и другими компонентами.  4. Сервисы: Java EE предоставляет различные службы и сервисы, такие как сервисы транзакций, безопасности, навигации и т. д., которые могут исп. в приложениях.  **Стандартные контейнеры Java EE:**  1. Web-контейнер: Отвечает за выполнение веб-компонентов (сервлеты и JSP). Предост. среду выполнения для обраб-ки HTTP запросов и генерации HTTP ответов.  2. EJB-контейнер: Отвечает за выполнение компонентов EJB (Enterprise JavaBeans). Предоставляет механизмы управления транзакциями, безопасностью и жизненным циклом для EJB.  3. Приложение-клиентский контейнер: Предоставляет среду выполнения для клиентских приложений Java EE, которые запускаются независимо от веб-браузера.  4. Контейнеры JMS (Java Message Service): Отвечают за выполнение компонентов, использующих сервис JMS для обмена сообщениями между разл. частями приложения.  **Java EE серверы** предоставляют реализацию спецификаций Java EE и служат платформой для выполнения Java EE приложений.  1. Apache Tomcat: Легковесный веб-контейнер, который поддерживает выполнение сервлетов и JSP. Часто используется для разработки и тестирования веб-приложений.  2. GlassFish является надежным и производительным Java EE-сервером с широким набором функций, поддерживающим развертывание и управление корпоративных веб-приложений. | 40. Понятие и назначение сервлета. Servlet в архитектуре Web-приложения. Жизненный цикл сервлета - интерфейс Servlet.  **Сервлеты** - Java классы, которые используются для создания динам. веб-страниц и обработки HTTP запросов в рамках веб-приложений на платформе Java EE. Они явл. ключ. компонентами веб-приложений и предост. мощные ср-ва для взаимодействия с клиентами через протокол HTTP.  **Назначение сервлета:**  1. Сервлеты могут генерировать содержимое веб-страницы на основе данных из разл. источников, таких как БД, файлы и т. д.  2. Сервлеты могут обраб. HTTP запросы от клиентов, выполнять операции на сервере и генерир. соответствующие HTTP ответы.  3. Сервлеты могут управлять состоянием клиентов, используя разл. механизмы, такие как cookies, сессии и параметры запроса.  **В архитектуре веб-приложения** сервлеты обычно используются вместе с другими компонентами, такими как JSP (JavaServer Pages), фильтры, слушатели событий и EJB (Enterprise JavaBeans). Они работают внутри контейнера сервлетов, кот. обеспечивает их выполнение и управление жизненным циклом.  **Жизненный цикл сервлета** описывает процесс его создания, инициализации, обработки запросов и уничтожения. Он управляется контейнером сервлетов и основан на интерфейсе javax.servlet.Servlet, который определяет методы для работы с сервлетом. Вот основные методы интерфейса Servlet:  1. init(ServletConfig config): Вызывается при инициализации сервлета. Исп. для выполнения начальной настройки сервлета.  2. service(ServletRequest request, ServletResponse response): Вызывается для обработки каждого HTTP запроса к сервлету. Метод абстрактный и должен быть реализован в классе сервлета.  3. destroy(): Вызывается при уничтожении сервлета. Исп. для освобождения ресурсов и выполнения завершающих операций.  При выполнении запроса от клиента контейнер сервлетов созд. новый поток выполнения (или исп. пул потоков), вызывает методы init() и service(), а затем при завершении запроса вызывает метод destroy() (если необходимо). Это позволяет сервлетам эффективно обраб. мн-во одновременных запросов от клиентов и поддерживать состояние между запросами. |
| 41. Взаимодействие сервлета и JSP.  Взаимодействие сервлетов и JSP (JavaServer Pages) часто используется в веб-приложениях для разделения логики приложения и представления. Обычно сервлеты выполняют обработку бизнес-логики и генерацию данных, а JSP используются для отображения этих данных и формирования веб-страниц для клиентов. Вот основные способы взаимодействия между сервлетами и JSP:  1. Передача данных из сервлета в JSP:  - Сервлеты могут устанавливать атрибуты запроса, сессии или контекста, которые могут быть использованы в JSP для отображения данных.  - Например, сервлет может установить атрибут запроса с помощью request.setAttribute("name", value), а затем JSP может получить доступ к этому атрибуту с помощью выражения EL (Expression Language) ${requestScope.name}.  2. Передача данных из JSP в сервлет:  - JSP может отправлять данные на обработку сервлету с помощью HTTP параметров или форм. Например, форма HTML в JSP может быть отправлена на сервлет с использованием метода POST, а данные могут быть извлечены из запроса в сервлете с помощью request.getParameter("parameterName").  3. Использование JSP для отображения данных:  - Сервлет может использовать методы для генерации содержимого JSP и отправки его клиенту. Например, сервлет может использовать объект RequestDispatcher, чтобы передать управление JSP для отображения страницы.  4. Использование JSP в качестве шаблонов:  - JSP страницы могут быть использованы в качестве шаблонов для создания повторяющихся частей веб-страниц. Сервлет может динамически подставлять данные в шаблоны JSP и отправлять клиенту результирующую страницу.  Эффективное взаимодействие между сервлетами и JSP позволяет разработчикам создавать динамические и интерактивные веб-приложения, которые могут эффективно обрабатывать запросы от клиентов и предоставлять им актуальную информацию. | 42. Интерфейс ServletContext. Интерфейс ServletConfig.  Интерфейсы ServletContext и ServletConfig являются частями спецификации Java Servlet API и предоставляют доступ к различным параметрам и ресурсам, связанным с конкретным веб-приложением и конкретным сервлетом соответственно.  **Интерфейс ServletContext** предст. контекст веб-приложения и позволяет взаимод. с параметрами контекста, атрибутами, ресурсами и др. инфой о приложении. Методы ServletContext:  1. getInitParameter(String name): Возвращает значение инициализационного параметра контекста по его имени.  2. getInitParameterNames(): Возвращает имена всех инициализационных параметров контекста.  3. getAttribute(String name): Возвращает значение атрибута контекста по его имени.  4. setAttribute(String name, Object value): Устанавливает значение атрибута контекста.  5. getRequestDispatcher(String path): Возвращает объект RequestDispatcher для заданного пути, который может быть использован для передачи управления другому ресурсу (например, сервлету или JSP странице).  Интерфейс ServletConfig представляет конфигурацию конкретного сервлета и предоставляет доступ к его инициализационным параметрам и контексту приложения. Некоторые методы ServletConfig включают:  1. getInitParameter(String name): Возвращает значение инициализационного параметра сервлета по его имени.  2. getInitParameterNames(): Возвращает имена всех инициализационных параметров сервлета.  3. getServletContext(): Возвращает объект ServletContext, представляющий контекст веб-приложения, в кот. работает сервлет.  4. getServletName(): Возвращает имя сервлета, как определено в развертывании приложения.  Использование ServletContext и ServletConfig позволяет сервлетам взаимод. с окружением своего веб-приложения, получать доступ к его ресурсам и параметрам, а также выполнять разл. операции, связанные с его конфигурацией. | 43. Интерфейс HttpServletRequest. Интер-с HttpServletResponse.  Интерфейсы HttpServletRequest и HttpServletResponse являются ключевыми частями Java Servlet API и предоставляют доступ к информации о HTTP запросах от клиентов и формированию HTTP ответов от сервера соответственно.  **Интерфейс HttpServletRequest** представляет HTTP запрос от клиента к серверу и содержит информацию о запросе, такую как параметры, заголовки, метод запроса и тело запроса. Некоторые методы HttpServletRequest включают:  1. getMethod(): Возвр. HTTP метод запроса (напр-р, GET, POST).  2. getParameter(String name): Возвращает значение параметра запроса по его имени.  3. getHeader(String name): Возвращает значение заголовка запроса по его имени.  4. getSession(boolean create): Возвращает сессию клиента, связанную с этим запросом. Может создать новую сессию, если параметр create установлен в true.  5. getRequestDispatcher(String path): Возвращает объект RequestDispatcher для передачи управления другому ресурсу (например, сервлету или JSP странице).  **Интерфейс HttpServletResponse** представляет HTTP ответ, который сервер отправляет клиенту после обработки запроса. Он содержит методы для управления заголовками ответа, отправки содержимого и установки статуса ответа. Некоторые методы HttpServletResponse включают:  1. setStatus(int sc): Устанавливает статус ответа HTTP.  2. setContentType(String type): Устанавливает тип контента ответа (например, "text/html", "application/json").  3. getWriter(): Возвращает объект PrintWriter, который может быть исп. для отправки текстового содержимого ответа.  4. sendRedirect(String location): Отправляет перенаправление клиенту на другой ресурс по заданному URL.  5. addCookie(Cookie cookie): Добавляет cookie в ответ.  Использование HttpServletRequest и HttpServletResponse позволяет сервлетам обрабатывать HTTP запросы от клиентов и генерировать соответствующие HTTP ответы, управляя содержимым, заголовками и другими аспектами ответа. | 44. Многопоточность в сервлете. Переадресация запросов и перенаправление откликов  **Многопоточность** в сервлете может стать проблемой из-за того, что сервлеты обычно исп. для обработки одновременных запросов от разных клиентов. В контексте многопоточности следует учитывать следующее:  1. Состояние сервлета: Сервлеты могут использовать разделяемые ресурсы, которые могут привести к состоянию гонки при доступе из нескольких потоков. Поэтому важно обеспечить синхронизацию доступа к общим данным.  2. Использование потоков: В сервлете можно создать новый поток для обработки долгого или асинхронного запроса. Однако это может повысить сложность управления потоками и безопасность приложения.  3. ThreadLocal: Для избежания конфликтов между потоками можно использовать ThreadLocal для хранения потоко-локальных данных, которые не будут разделяться между потоками.  4. Пул потоков: Сервлет контейнер может использовать пул потоков для обработки запросов, что может снизить нагрузку на систему и обеспечить более эффективное исп-ие ресурсов.  **Переадресация запросов и перенаправление откликов** - это механизмы, позволяющие сервлету передавать управление другому ресурсу для обработки запроса или генерации ответа.  1. Переадресация запросов: При переадресации запроса сервлет отправляет клиенту HTTP ответ со статусом "302 Found" и заголовком "Location", указывающим на новый URL ресурса. Клиент затем отправляет новый запрос на указанный URL.  2. Перенаправление откликов: При перенаправлении отклика сервлет сохраняет текущий HTTP запрос и его атрибуты и передает управление другому ресурсу (например, сервлету или JSP странице) для генерации содержимого ответа. Обычно используется для динам. генерации содержимого ответа.  Оба механизма могут быть полезными в различных сценариях веб-разработки и могут помочь в управлении навигацией и обработкой запросов в веб-приложении. Они обеспечивают гибкость и возможность создания динамических и интерактивных веб-приложений. |
| 45. Сохранение данных приложения между запросами. Атрибуты и параметры  **Сохранение данных между запросами в веб-приложении** - важный аспект, который позволяет передавать информацию от одного запроса к другому и поддерживать состояние приложения. Для этого можно использовать атрибуты контекста, атрибуты сессии и параметры запроса.  **Атрибуты контекста** доступны для всех сервлетов и др. компонентов приложения. Они устанавливаются через объект ServletContext и могут использоваться для хранения информации о приложении, доступной на всем его протяжении.  servletContext.setAttribute("attrName", attributeValue);  Object attributeValue = servletContext.getAttribute("attrName");  **Атрибуты сессии** связаны с конкретной сессией пользователя и сохр. между запросами от одного и того же клиента. Они устанавливаются через объект HttpSession и могут исп. для хранения инфы о состоянии сеанса пользователя.  HttpSession session = request.getSession();  session.setAttribute("attributeName", attributeValue);  Object attributeValue = session.getAttribute("attributeName");  **Параметры запроса** передаются клиентом в URL или в теле HTTP запроса. Они доступны внутри сервлета через объект HttpServletRequest и могут использоваться для передачи данных от клиента к серверу.  String paramValue = request.getParameter("parameterName");  Выбор между атрибутами контекста, атрибутами сессии и параметрами запроса зависит от конкретных потребностей приложения. Атрибуты контекста подходят для глобальных данных приложения, атрибуты сессии - для данных, привязанных к пользовательской сессии, а параметры запроса - для передачи данных от клиента к серверу с каждым запросом. | 46. Сессии, события, файлы Cookie  Сессии, события и файлы Cookie являются ключевыми инструментами для управления состоянием и взаимодействия с клиентами в веб-приложениях.  **Сессия** представляет собой период активности между клиентом и сервером. Она начинается, когда клиент отправляет первый запрос к серверу, и заканчивается после определенного периода неактивности или явного завершения. Сервлеты могут использовать сессии для хранения состояния между запросами от одного и того же клиента. Для этого используется объект HttpSession, который обеспечивает доступ к атрибутам сессии.  HttpSession session = request.getSession();  session.setAttribute("attributeName", attributeValue);  Object attributeValue = session.getAttribute("attributeName");  **События:** Сервлет контейнер может генерировать события, связанные с жизн. циклом сессии или контекста приложения (например, создание, уничтожение, активация, деактивация). Сервлеты могут реагировать на эти события, реализуя интерфейсы HttpSessionListener или ServletContextListener, и выполнять соответствующие действия.  public class MySessionListener implements HttpSessionListener {  public void sessionCreated(HttpSessionEvent se) {…}  public void sessionDestroyed(HttpSessionEvent se) {…}  }  **Файлы Cookie** представляют собой небольшие текстовые файлы, которые сервер отправляет клиенту, а клиент сохраняет их и отправляет обратно с каждым запросом. Могут использоваться для хранения инфы о пользовательской сессии, предпочтениях пользователя, отслеживания состояния сеанса и т. д.  Cookie cookie = new Cookie("name", "value");  response.addCookie(cookie);  Cookie[] cookies = request.getCookies(); | 47. Интерфейс Filter  **Интерфейс Filter** в Java Servlet API предоставляет возможность сервлетам выполнять общие операции фильтрации для HTTP запросов и ответов, прежде чем они будут обработаны или отправлены на сервер. Фильтры могут быть использованы для обработки запросов и ответов, изменения их содержимого, добавления дополнительной логики и т. д. Осн. цель фильтров - обеспечить переисп-ие кода и модульность в веб-приложениях.  **Интерфейс Filter содержит три метода:**  1. init(FilterConfig config): вызывается 1 раз при инициализации фильтра и исп. для выполнения любых инициализационных действий, таких как чтение параметров конфигурации. Пар-р FilterConfig предост. доступ к конфигурационной инфе фильтра.  2. doFilter(ServletRequest request, ServletResponse response, FilterChain chain): Метод doFilter вызывается каждый раз, когда сервлет обрабатывает запрос или отправляет ответ. Этот метод содержит основную логику фильтрации. Он получает объекты ServletRequest и ServletResponse, которые представляют текущий запрос и ответ, а также объект FilterChain, который исп. для передачи запроса и ответа следующему фильтру в цепочке или целевому ресурсу (например, сервлету).  3. destroy(): Метод destroy вызывается один раз при завершении работы фильтра и используется для освобождения ресурсов или завершения каких-либо активных процессов, созданных фильтром во время его жизненного цикла.  public class MyFilter implements Filter {  public void init(FilterConfig config) throws ServletException {…}  public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  // Передача запроса и ответа след. фильтру в цепочке  chain.doFilter(request, response);  }  public void destroy() {…}  }  После создания фильтра его необходимо зарегистрировать в файле конфигурации веб-приложения (например, в web.xml) или с помощью аннотаций. | 48. Java Server Pages (JSP). Жизненный цикл. Неявн. объекты в JSP.  **JSP** - технология Java для создания динамических веб-страниц, которая позволяет разработчикам встраивать Java код прямо в HTML страницы. JSP страницы компилируются в сервлеты при первом запросе и выполняются на сервере, что обеспечивает гибкость и удобство веб-разработки.  **Жизненный цикл JSP:**  1. Компиляция: При первом запросе JSP страницы сервер компилирует ее в сервлет, который затем исполняется. Если JSP файл изменился, сервер повторно компилирует его.  2. Инициализация: После компиляции и перед отправкой ответа клиенту сервер создает экземпляр сервлета, представляющего JSP страницу, и вызывает метод init() для инициализации.  3. Обработка запроса: При каждом запросе сервер вызывает метод service(), который генерирует HTML содержимое страницы и отправляет его клиенту.  4. Уничтожение: При завершении работы приложения сервер вызывает метод destroy() для освобождения ресурсов.  **Неявные объекты в JSP:**  1. request: HTTP запрос от клиента, содержит параметры запроса, заголовки и другую информацию.  2. response: HTTP ответ сервера, исп. для отправки содержимого клиенту.  3. out: поток вывода, который используется для записи HTML содержимого, возвращаемого клиенту.  4. session: сессия клиента, используется для хранения состояния между запросами от одного и того же клиента.  5. application: контекст приложения, исп. для хран-ия глобальной инфы о приложении, доступной на всем его протяжении.  6. config: конфигурация сервлета, исп. для доступа к параметрам инициализации.  7. pageContext: контекст JSP страницы, предоставляет доступ к другим неявным объектам и атрибутам страницы.  Эти неявные объекты предоставляют доступ к различным аспектам HTTP запроса, ответа и среды выполнения JSP страницы, что делает JSP более мощным и гибким инструментом для разработки динамических веб-приложений. |

|  |  |  |  |
| --- | --- | --- | --- |
| 49. Синтаксис страницы JSP. Директивы (directives); объявления (declarations); скриптлеты (scriptlets); выражения (expressions); комментарии (comments)  **Синтаксис страницы JSP** состоит из нескольких основных элементов, которые позволяют встраивать Java код и другие элементы в HTML страницу.  1. Директивы исп. для указания настройки и конфигурации JSP страницы. Синтаксис: <%@ directiveName attribute="value" %>.  <%@ page language="java" contentType="text/html; charset=UTF-8" %>  <%@ include file="header.jsp" %>  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  2. Объявления используются для объявления переменных и методов в JSP странице. Синтаксис: <%! declaration %>.  <%! int count = 0; %>  3. Скриптлеты исп. для вставки Java кода непосредственно в HTML страницу. Синтаксис: <% javaCode %>.  <% for (int i = 0; i < 5; i++) { %>  <p>Iteration <%= i %></p>  <% } %>  4. Выражения используются для вставки значений переменных или вычисленных выражений в HTML страницу. Синтаксис:  <%= expression %>.  <p>Current time: <%= new java.util.Date() %></p>  5. Комментарии исп. для добавления комментов в код JSP.  - Синтаксис: <%-- comment --%>.  Эти элементы позволяют создавать динамические веб-страницы, в которых Java код и другие элементы интегрируются с HTML кодом для генерации контента, который отправляется клиенту. | 50. Стандартные action-теги JSP. jsp:useBean, jsp:setProperty, jsp:getProperty, jsp:include, jsp:forward, jsp:param. Expression Language (EL)  **Стандартные action-теги JSP** представляют собой специальные элементы, которые используются для выполнения различных операций в JSP страницах. Вот некоторые из них:  1. <jsp:useBean>: исп. для создания экземпляра Java объекта (обычно JavaBean) или получения ссылки на сущ-щий объект.  <jsp:useBean id="user" class="com.example.User" />  2. <jsp:setProperty>: исп. для установки свойств Java объекта, созданного с помощью <jsp:useBean>.  <jsp:setProperty name="user" property="fName" value="John" />  3. <jsp:getProperty>: исп. для получения значений свойств Java объекта и вывода их на страницу.  <jsp:getProperty name="user" property="firstName" />  4. <jsp:include>: исп. для включения содержимого другой JSP страницы или ресурса в текущую страницу.  <jsp:include page="header.jsp" />  5. <jsp:forward>: исп. для перенаправления запроса на другую JSP страницу или ресурс.  <jsp:forward page="error.jsp" />  6. <jsp:param>: исп. внутри <jsp:forward> или <jsp:include> для передачи параметров другой JSP странице или ресурсу.  <jsp:param name="error" value="404" />  EL- это специальный язык, который позволяет получать доступ к данным и выполнить вычисления в JSP страницах. Он предоставляет удобный и компактный синтаксис для работы с объектами и свойствами.  - Получение значения свойства объекта: ${user.firstName}  - Выполнение арифметической операции: ${2 + 2}  - Вызов метода объекта: ${user.getName()}  EL позв. упростить доступ к данным и выполнение операций в JSP страницах, делая код более читаемым и поддерживаемым. | 51. Библиотека тэгов JSTL. Стандартные теги: core, formatting, sql, xml, functions  **Библиотека JSTL (JavaServer Pages Standard Tag Library)** предоставляет набор стандартных тегов, которые упрощают разработку динамических веб-приложений в JSP.  1. **Core Tags** (<c:core>): базовые операции и структуры управления потоком выполнения: <c:if>, <c:choose>, <c:set>, <c:forEach>, <c:remove>, <c:catch> и т. д.  2. **Formatting Tags** (<fmt:formatting>): исп. для форматирования данных, таких как числа и даты: <fmt:formatNumber>, <fmt:formatDate>, <fmt:parseNumber>, <fmt:parseDate> и т. д.  3. **SQL Tags** (<sql:sql>): возможность выполнения SQL запросов к базе данных прямо из JSP страницы: <sql:setDataSource>, <sql:update>, <sql:query>, <sql:param> и т. д.  4**. XML Tags** (<x:xml>): работа с XML данными в JSP страницах: <x:parse>, <x:out>, <x:forEach> и т. д.  5. **Functions Tags** (<fn:functions>): набор функций для выполнения разл. операций в JSP страницах: <fn:substring>, <fn:toLowerCase>, <fn:toUpperCase>, <fn:split>, <fn:length> и т. д.  Эти теги позволяют разработчикам выполнять различные операции, такие как управление потоком выполнения, форматирование данных, выполнение SQL запросов, работа с XML и выполнение функций, прямо в JSP страницах, что делает процесс разработки более простым и эффективным. | 52. Библиотека JSTL. Пользовательские теги  В дополнение к стандартным тегам, **библиотека JSTL** также позволяет создавать **пользоват. теги**, кот. могут быть исп. для расширения функциональности JSP страниц и упрощения повторяющихся задач. Пользоват. теги позволяют разработчикам создавать собств. теги с пользоват. логикой, которые могут быть использованы на JSP страницах. **Основные шаги их создания:**  1. Создайте класс Java, который расширяет один из классов тегов JSTL, таких как TagSupport или BodyTagSupport, и реализуйте методы для обработки тега.  2. Опред. новый пользоват. тег в файле описания библиотеки тегов (TLD - Tag Library Descriptor), указав имя, атрибуты и обработчик тега.  3. Используйте новый пользоват. тег на JSP странице, указав его URI и имя, и задав значения атрибутов при необходимости.  Пример TLD файла:  <?xml version="1.0" encoding="UTF-8"?>  <taglib xmlns="http://java.sun.com/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-jsptaglibrary\_2\_1.xsd"  version="2.1">  <tlib-version>1.0</tlib-version>  <short-name>custom</short-name>  <uri>http://example.com/tags</uri>  <tag>  <name>hello</name>  <tag-class>com.example.tags.HelloTag</tag-class>  <body-content>empty</body-content>  </tag>  </taglib> |
| 53. Maven. Назначение. Состав объект. модели проекта pom.xml  **Apache Maven** - инструмент управления проектами, кот. исп. в Java-проектах для автоматизации сборки, управления зав-тями, тестирования и развертывания ПО. Его осн. цель - упростить процесс разработки, повысить производ-сть и обеспечить единообразие в проектах.  **Основные задачи Maven включают в себя:**  1. Автоматически собирает проект из исх. кода в исполняемый артефакт (например, JAR, WAR или EAR файл).  2. Обесп. удобный способ управления зависимостями проекта, включая их загрузку из центр. репоз-ия и включение в проект.  3. Возможность запуска тестов и ген-ции отчетов о тестировании.  4. Позволяет автоматизировать процесс развертывания проекта в целевой среде.  **pom.xml (Project Object Model)** явл. осн. конфигурационным файлом Maven проекта. Он содержит инфу о проекте и его зависимостях, а также настройки сборки и тестирования. **Вот основные элементы pom.xml**:  1. project: Основной эл-т, содержащий всю инфу о проекте.  - groupId: идентификатор группы, к кот. принадлежит проект.  - artifactId: Идентификатор артефакта (название проекта).  - version: Версия проекта.  - packaging: Тип артефакта (например, jar, war, pom).  - dependencies: Список зависимостей проекта.  2. dependencies: Эл-т, содержащий инфу о зависимостях проекта.  - dependency: Определение одной зависимости.  - groupId: groupId зависимости.  - artifactId: artifactId зависимости.  - version: Версия зависимости.  - scope: Область видимости зависимости (compile, runtime, test, provided).  3. build: Настройки сборки проекта.  - plugins: Плагины Maven, используемые для различных этапов сборки (например, компиляции, тестирования, упаковки).  - plugin: Определение плагина.  - groupId: groupId плагина.  - artifactId: artifactId плагина.  - version: Версия плагина.  - configuration: Конфигурация плагина. | 54. Основные фазы сборки проекта maven. Жизнен. цикл сборки  В Maven существует концепция **"жизненного цикла сборки"**, который представляет собой последовательность этапов, через которые проходит проект при сборке. Каждая фаза представляет собой опред. этап в процессе сборки проекта. **Вот основные фазы сборки проекта Maven:**  1. validate: Проверка проекта на наличие всех необходимых информационных элементов, их корректность и целостность.  2. compile: Компиляция исходных кодов проекта.  3. test: Запуск тестов для проверки корректности работы приложения.  4. package: Упаковка скомпилированных исходных кодов в артефакт (например, JAR, WAR или EAR файл).  5. verify: Проверка качества кода и выполнение доп. проверок.  6. install: Установка артефакта в локальный репозиторий Maven.  7. deploy: Размещение артефакта в удаленном репозитории для общего доступа.  Эти фазы образуют **"жизненный цикл сборки" проекта Maven**, и проект проходит через них последовательно при выполнении цели mvn clean install или других целей сборки. Каждая фаза может содержать набор задач (goal), которые выполняются в соответствии с определенными настройками проекта.  Жизненный цикл сборки Maven позв. автоматизировать процесс сборки, тестирования и развертывания проекта, обеспечивая единообразие и предсказуемость процесса разработки. | 55. Понятие профайла сборки. Управление зависимостями. Maven plugin.  **Профайл сборки в Maven** - механизм, кот. позв. настраивать сборку проекта в зависимости от определенных условий или требований. Профайлы позволяют опред. наборы настроек, которые могут быть активированы или деактивированы в завис-ти от различных факторов, таких как окружение выполнения, целевая платформа или требования к конфигурации.  **Профайлы могут быть использованы для следующих целей:**  1. Управление зависимостями: В зависимости от профиля можно задать различные зависимости, кот. будут включены в проект при сборке.  2. Настройка среды выполнения: Можно настроить различные параметры среды выполнения, такие как переменные среды, параметры JVM и т. д.  3. Активация конфигураций: Профили могут быть активированы при выполнении опред. задач или в завис-ти от пер-ных среды.  <profiles>  <profile>  <id>dev</id><dependencies>…</dependencies>  <properties><environment>…</environment></properties>  </profile>  </profiles>  Управление зависимостями в Maven осуществляется с помощью раздела <dependencies> в файле pom.xml. Зависимости указываются с помощью элемента <dependency>, где указываются groupId, artifactId и version зависимости.  Пример зависимости в файле pom.xml:  <dependencies><dependency><groupId>com.example</groupId>  <artifactId>my-library</artifactId><version>1.0.0</version>  </dependency></dependencies>  **Maven плагин** - инструмент, кот. предоставляет доп. функциональность и возможности в Maven. Плагины могут исп. для выполнения различных задач в процессе сборки, тестирования, развертывания и т. д. В Maven существует множество стандартных плагинов, таких как maven-compiler-plugin, maven-surefire-plugin, maven-jar-plugin и другие. Плагины также могут быть созданы и исп. пользователем для выполнения специфических задач в проекте. | 56. Архитектуры построения web-приложений. MVC. Многоур. архитектура.  Архитектура веб-прил-й опред. способ организации компонентов приложения для достижения целей, таких как удобство сопровождения, масштабируемость, гибкость и безопасность. Две распр. архитектуры для построения веб-приложений - MVC (Model-View-Controller) и многоуровневая архитектура.  **1. MVC (Model-View-Controller):**  - Модель (Model): компонент, кот. представляет собой данные и бизнес-логику приложения. Модель обрабатывает запросы от контроллера, выполняет операции с данными и передает результаты представлению.  - Представление (View): компонент, кот. отвечает за отобр-ие данных пользователю. Представление получает данные от модели и отображает их в удобной для пользователя форме (например, HTML страницы).  - Контроллер (Controller): компонент, который обрабатывает запросы от пользователей, взаимодействует с моделью для получения данных и обновления состояния, а затем передает эти данные представлению. Контроллер также отвечает за управление навигацией и потоком управления приложения.  **2. Многоуровневая архитектура:**  - Представление (View): Отвечает за отображение данных пользователю и взаимодействие с ним. В этой архитектуре представление может быть представлено в виде HTML страниц, пользовательских интерфейсов и т. д.  - Бизнес-логика (Business Logic): Этот уровень содержит бизнес-логику приложения, которая определяет правила и процессы, которые управляют данными и операциями в приложении.  - Доступ к данным (Data Access): Этот уровень обеспечивает доступ к данным, необходимым для работы приложения. Он может вкл. в себя БД, внешние API и другие источники данных.  MVC часто используется в веб-приложениях, так как он обеспечивает разделение ответственности между компонентами и упрощает сопровождение и тестирование приложений. Многоуровневая архитектура также широко применяется и обеспечивает логическое разделение приложения на независимые слои, что упрощает масштабирование и поддержку. |
| 57. JMS. Понятие MOM. Основные понятие и режимы работы.  **JMS (Java Message Service)** - API для отправки сообщений между клиентами в распределенных приложениях. Оно обеспечивает стандартизированный способ создания, отправки, получения и обработки сообщений в асинхр. режиме. JMS построено на основе понятий MOM (Message-Oriented Middleware) - ПО, кот. обраб. отправку, получение и маршр-цию сообщений между клиентами.  **Основные понятия в JMS:**  1. Сообщение (Message): осн. Эл-т данных в JMS. Оно содержит информацию, которую клиенты обмениваются друг с другом.  2. Очередь (Queue): место, где отправители помещают сообщения для получателей. Сообщения в очереди обрабатываются в порядке их поступления (FIFO).  3. Тема (Topic): место, где отправители посылают сообщения, и одновременно все получатели, подписанные на эту тему, получают сообщения. Темы обесп. публикацию/подписку модель взаимодействия.  **Основные режимы работы JMS:**  1. Point-to-Point (P2P): Этот режим работы используется с очередями. Каждое сообщение отправляется в конкретную очередь и принимается только одним получателем (consumer). Это обеспечивает однократную обработку сообщения и гарантирует, что каждое сообщение будет обработано.  2. Publish/Subscribe (Pub/Sub): Этот режим работы используется с темами. Отправитель (publisher) посылает сообщение в тему, и все подписанные на тему получатели (subscribers) получают это сообщение. Это позволяет широковещательную передачу сообщений, где каждый получатель может получить копию сообщения.  JMS позволяет создавать асинхр. и надежные системы обмена сообщениями, что делает его популярным инструментом в распределенных приложениях, таких как корпоративные системы, финанс. приложения, системы мониторинга и другие. | 58. JMS. Особенности работы P2P, pub/sub. Программные интерфейсы.  В JMS (Java Message Service) сущ. 2 основных режима работы: Point-to-Point (P2P) и Publish/Subscribe (Pub/Sub).  **1. Point-to-Point (P2P):**  - Особенности работы: В режиме P2P каждое сообщение отправляется в определенную очередь (Queue), и только один получатель (consumer) может получить и обработать это сообщение. После того как сообщение было получено и обработано, оно удаляется из очереди.  - Программные интерфейсы: Для работы с P2P моделью в JMS используются следующие интерфейсы:  - javax.jms.Queue: очередь, в кот. отправляются сообщения.  - javax.jms.QueueSender: отправка сообщений в очередь.  - javax.jms.QueueReceiver: получение сообщений из очереди.  **2. Publish/Subscribe (Pub/Sub):**  - Особенности работы: В режиме Pub/Sub отправитель посылает сообщения в тему (Topic), а все подписанные на эту тему получатели (subscribers) получают копии этого сообщения. Каждый подписчик получает свою собств. копию сообщения.  - Программные интерфейсы: Для работы с Pub/Sub моделью в JMS используются следующие интерфейсы:  - javax.jms.Topic: тема, в которую отправляются сообщения.  - javax.jms.TopicPublisher: отправка сообщений в тему.  - javax.jms.TopicSubscriber: подписка на тему и получение сообщений из нее.  **Программные интерфейсы JMS** обеспечивают удобные методы для отправки и получения сообщений в соответствии с выбранным режимом работы. Разработчики могут использовать эти интерфейсы для создания асинхронных и надежных систем обмена сообщениями в своих приложениях. | 59. Алгоритм настройки работы клиентов JMS P2P. Алгоритм настройки работы клиентов JMS pub/sub  **Алгоритм настройки работы клиентов JMS P2P:**  1. Клиенту необходимо установить соединение с брокером сообщений (например, Apache ActiveMQ, RabbitMQ), который обеспечивает отправку и получение сообщений. Это делается с исп-ем фабрики соединений JMS (javax.jms.ConnectionFactory).  2. После установки соединения клиент создает сессию обмена сообщениями с брокером. В JMS сессия представлена интерфейсом javax.jms.Session.  3. Клиент создает очередь, в которую будет отправляться и из которой будет получаться сообщения. Это делается с исп-ем интерфейса javax.jms.Queue.  4. Клиент создает отправителя (javax.jms.QueueSender), чтобы отправлять сообщения в очередь, и получателя (javax.jms.QueueReceiver), чтобы получ. сообщения из очереди.  5. Клиент исп. отправителя для отправки сообщений в очередь и получателя для получ. и обработки сообщений из очереди.  **Алгоритм настройки работы клиентов JMS Pub/Sub:**  1. Как и в случае с P2P, клиент должен установить соединение с брокером сообщений.  2. Создание сессии обмена сообщениями с брокером, исп. интерфейс javax.jms.Session.  3. Клиент создает тему, в которую будут отправляться сообщения. Это делается с исп-ем интерфейса javax.jms.Topic.  4. Клиент создает издателя (javax.jms.TopicPublisher), чтобы отправлять сообщения в тему, и подписчика (javax.jms.TopicSubscriber), чтобы подписаться на тему и получать сообщения из нее.  5. Клиент использует издателя для отправки сообщений в тему, а подписчика для получения и обработки сообщений из темы. | 60. JMS. Способы обеспечения надежности доставки сообщений  В JMS (Java Message Service) сущ. несколько способов обеспечения надежности доставки сообщений:  1. Транзакции (Transactions): JMS поддерживает транзакционную модель, которая позволяет отправителю атомарно отправлять несколько сообщений и гарантировать, что либо все сообщения будут успешно доставлены и обработаны, либо ни одно из них. Это обеспечивает согласованность сообщений в системе и предотвращает потерю данных в случае сбоев.  2. Подтверждения (Acknowledgements): В режиме P2P отправ-ль может запросить от получателя подтверждение доставки сообщения. Получатель должен отправить подтверждение, когда сообщение было успешно получено и обработано. Это обеспечивает надежность доставки сообщений в сети.  3. Долговременные (постоянные) сообщения (Durable Messages): В режиме Pub/Sub JMS позволяет создавать постоянные подписки, которые сохраняют сообщения для подписчиков, даже если они временно отключены. Это гарантирует, что подписчик получит все сообщения, отправленные в тему, даже если он был неактивен во время их отправки.  4. Персистентность (Persistence): JMS позволяет отправителю указывать, должны ли сообщения быть сохранены на диске или в памяти брокера сообщений. Сообщения, сохраненные на диске, будут сохранены даже в случае сбоев системы и перезапуска брокера, что обеспечивает более высокую надежность доставки.  5. Обработка исключений (Exception Handling): JMS предоставляет механизмы обработки исключений для обработки ошибок во время отправки или получения сообщений. Клиенты могут использовать обработку исключений для повторной отправки сообщений в случае неудачной доставки или для выполнения других действий в зависимости от ситуации.  Эти механизмы обеспечивают надежность и целостность сообщений в JMS, что делает его широко используемым в распределенных системах для обмена данными. |