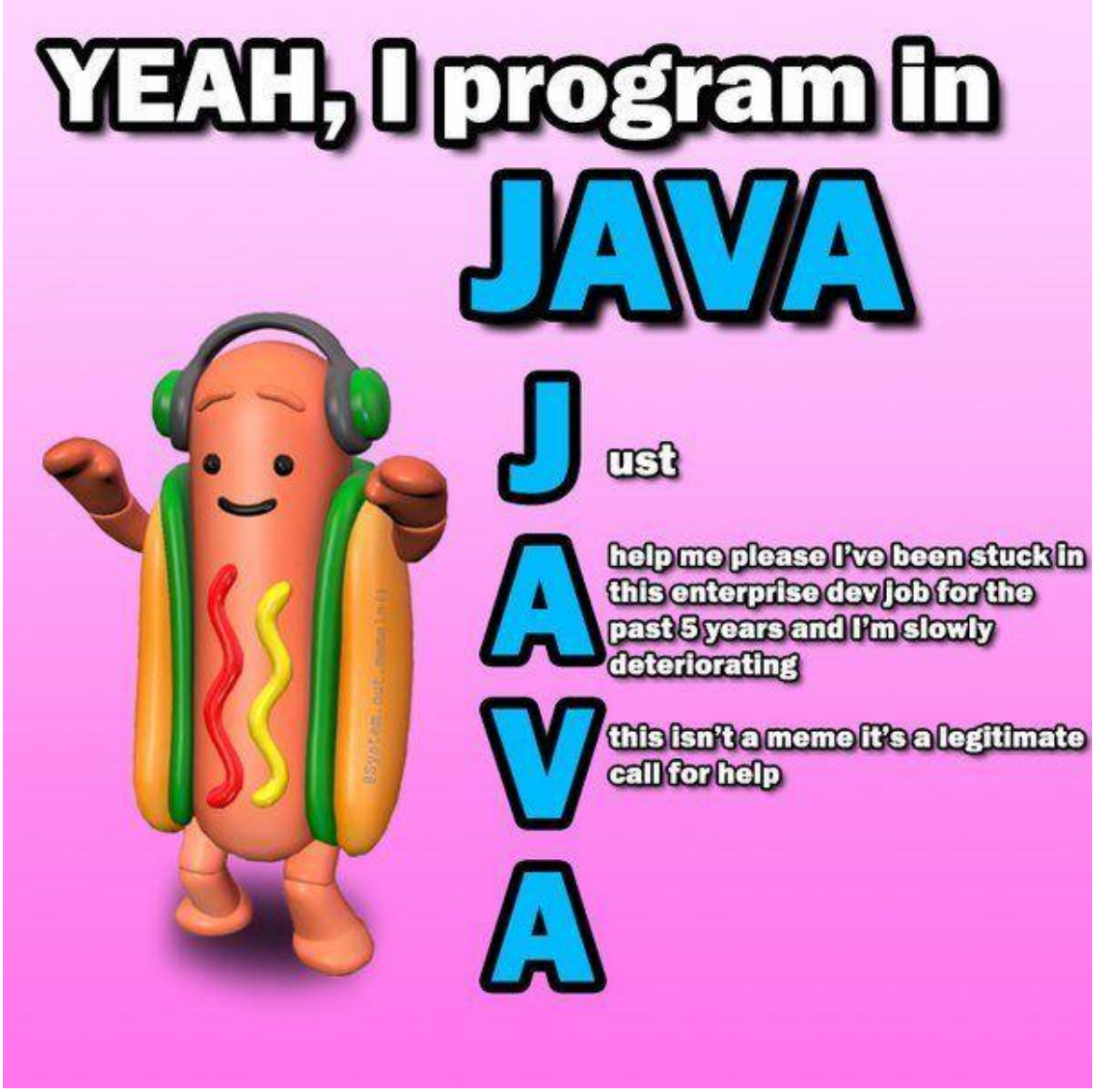


Вопросы теоретические:

Соня

1. Платформа Java: Java Development Kit (JDK): состав, назначение. Кроссплатформенность языка Java.

JDK (Java Development Kit) — это бесплатный комплект инструментов, с помощью которого создают и запускают программы при разработке приложений на языке Java.

В отличие от многих языков программирования, которые компилируются в определенных операционных системах, Java не имеет привязки к ОС(кроссплатформенность). Вместо этого, он может работать в любой операционной системе при наличии Java Runtime Environment.

Java Development Kit состоит из двух частей:

инструменты для разработки программ;
средства для запуска программ.

Инструменты для разработки — это различные утилиты, средства безопасности, документация, примеры и так далее. Весь набор устанавливать не обязательно; разработчик может выбрать и установить только то, что нужно ему для работы:

Библиотека классов. Это готовые решения ко многим задачам, с которыми сталкиваются практически все разработчики. Их использование ускоряет создание приложений.

Компилятор Javac. Он переводит исходный текст в байт-код — набор инструкций, понятный виртуальной Java-машине.

Отладчик. Помогает разработчику при поиске и исправлении ошибок. Программу в режиме отладчика можно остановить и запустить снова, можно узнать значения переменных и вычислить выражения на определённом этапе.

API Java — это программные компоненты, позволяющие интегрировать различные приложения и веб-сайты, то есть оперативно обмениваться данными между ними.

JRE (Java Runtime Environment) — среда выполнения Java. Программы на Java можно запускать на выполнение, только если установлена JRE. Она действует как посредник между программой и операционной системой, позволяет выполнять программу на разных устройствах и ОС. Благодаря ей разработчик может создавать приложение, не задумываясь о том, где оно будет выполняться.

В состав JRE входит JVM — виртуальная машина Java, которая выполняет скомпилированный байт-код. Ещё в неё входят стандартные библиотеки и вспомогательные файлы.

В Java Development Kit *нет среды разработки* — для этой цели используются внешние программы. Гораздо удобнее установить интегрированную среду разработки (IDE): это упростит работу, особенно если вы создаёте большое приложение. Самая популярная IDE для Java — IntelliJ IDEA от компании JetBrains

2. Java virtual machine (JVM), JIT-компилятор – определение, свойства, функции. Принципы работы сборщика мусора.

Виртуальная машина Java («Java Virtual Machine» — JVM) — это основная часть платформы Java Runtime Environment (JRE), которая интерпретирует байт-код Java для запуска программ.

Одним из наиболее значительных преимуществ использования является использование JVM для запуска программы Java в любой операционной среде. В её основе реализуется принцип WORA (Write once, run anywhere — «написал один раз, запускай везде»), который сильно упростил процессы разработки.

JVM выполняет основные функции:

- Выполнение байт-кода: JVM интерпретирует и выполняет Java-байт-код.
- Управление классами: JVM загружает и проверяет классы, обеспечивая их правильную инициализацию.
- Сборка мусора: JVM автоматически удаляет неиспользуемые объекты, освобождая память, даёт доступ к управлению памятью программ и её оптимизации.
- Безопасность и изоляция: проверяет байт-код на безопасность и изолирует приложения в своей среде выполнения.
- позволяет запускать Java-программы на любом устройстве или в любой операционной системе;

JIT (Just-In-Time) компиляция - это технология компиляции, которая позволяет генерировать машинный код на лету, во время выполнения программы. Это отличается от традиционной компиляции, которая генерирует машинный код заранее, на этапе компиляции исходного кода

До появления JIT компиляции, компиляторы генерировали машинный код заранее, на этапе компиляции исходного кода. Это приводило к тому, что компиляторы были ограничены тем, что могли сделать до запуска программы. Компилятор не мог адаптироваться к специфическим условиям выполнения программы, таким как ввод-вывод, динамическое изменение данных, и так далее

Свойства:
Динамическая компиляция: JIT-компилятор анализирует частоту выполнения кода и компилирует наиболее часто используемые части в машинный код.
Оптимизация: JIT-компилятор может применять различные оптимизации, такие как инлайнинг методов или удаление мертвого кода, чтобы еще больше повысить производительность.

Функции:

Ускорение выполнения: JIT-компиляция позволяет ускорить выполнение Java-программ, особенно тех, которые имеют повторяющиеся операции.
Анализ производительности: JIT-компилятор анализирует поведение программы и оптимизирует код на основе полученных данных.

Основные принципы

Определение недостижимых объектов:
Сборщик мусора определяет объекты, которые больше не используются программой, анализируя ссылки на них. Объекты, на которые нет ссылок из корневых объектов (таких как глобальные переменные, стек вызова), считаются недостижимыми и подлежат удалению.

Алгоритм пометок (Mark-and-Sweep):
Этот алгоритм включает два основных этапа:
Этап пометок (Mark): Сборщик мусора начинает с корневых объектов и помечает все достижимые объекты, проходя по ссылкам.
Этап очистки (Sweep): После пометки сборщик мусора проходит по памяти и удаляет все непомеченные объекты, освобождая занимаемую ими память.

Генерационная сборка мусора:
В некоторых системах, таких как .NET и Java, память делится на поколения в зависимости от времени жизни объектов. Объекты делятся на короткоживущие (поколение 0) и долгоживущие (поколения 1 и 2). Это позволяет оптимизировать сборку мусора, выполняя ее чаще для короткоживущих объектов.

Сжатие и перемещение объектов:
Некоторые сборщики мусора, особенно в .NET, используют сжатие памяти после удаления объектов, чтобы переместить выжившие объекты и освободить фрагментированную память.

Фоновая и полная сборка мусора:
Сборка мусора может выполняться в фоновом режиме (для молодых поколений) или как полная сборка мусора (для всех поколений), которая может приостановить работу приложения на время выполнения.

3. Системы сборки проектов. Фреймворк Apache Maven: определение, структура, Maven Coordinates, POM-файл.

Системы сборки проектов в Java — это инструменты, которые автоматизируют процесс компиляции, сборки и развертывания Java-проектов. Наиболее популярными системами сборки для

Java являются Ant, Maven и Gradle.

Основные системы сборки
Apache Ant:

Ant — это одна из первых систем сборки для Java. Она использует XML-файлы для описания процесса сборки и позволяет выполнять различные задачи, такие как компиляция и упаковка кода.
Ant в основном используется для небольших проектов или в сочетании с другими инструментами. Он считается морально устаревшим, но все еще используется в некоторых случаях.

Apache Maven:
Maven — это широко используемая система сборки, которая упрощает управление зависимостями и структурирует проекты согласно определенным стандартам. Она использует файлы pom.xml для описания проекта и его зависимостей.
Maven является стандартом индустрии для сборки Java-проектов. Он особенно полезен для управления зависимостями и автоматизации рутинных задач.

Gradle:
Gradle — это более современная система сборки, которая основана на Ant и Maven. Она использует скрипты на языке Groovy или Kotlin для описания процесса сборки и позволяет гибко настраивать проекты.
Gradle особенно подходит для сложных и многопроектных сборок благодаря своей гибкости и эффективности.

Преимущества систем сборки

Автоматизация: Системы сборки автоматизируют процесс компиляции, упаковки и развертывания проектов, что экономит время и снижает риск ошибок.
Управление зависимостями: Они позволяют легко управлять зависимостями проекта, скачивая и подключая необходимые библиотеки.
Стандартизация: Системы сборки, такие как Maven и Gradle, обеспечивают стандартизацию структуры проектов, что облегчает сотрудничество и поддержку кода.

Maven — это инструмент для автоматической сборки проектов на Java и других языках программирования. Он помогает разработчикам правильно подключить библиотеки и фреймворки, управлять их версиями, выстроить структуру проекта и составить к нему документацию.
Структура Maven-проекта определяется в файле pom.xml, который должен находиться в корневом каталоге проекта.

Структура проекта в Maven обычно включает стандартные каталоги:

- src/main/java – исходный код приложения.

- src/test/java – тесты.
- target – директория с собранными артефактами.

pom.xml. это XML-файл, который описывает структуру проекта, его зависимости и плагины. Он является центральным элементом в Maven для управления сборкой проекта

Содержание:

POM-файл содержит следующие основные элементы:

- Project Coordinates: groupId, artifactId, version.
- Dependencies: список библиотек, необходимых для проекта.
- Plugins: плагины, используемые для различных задач сборки.
- Properties: общие свойства проекта.
- Inheritance details: наследование конфигурации от других POM-файлов.
- Profiles: альтернативные конфигурации для разных сред

Maven Coordinates — это уникальные идентификаторы, которые используются для определения артефактов (библиотек или модулей) в репозитории. Они включают в себя groupId, artifactId и version

Идентификатор группы

`groupId` Это уникальный идентификатор проекта, обычно представляемый в виде обратного доменного имени (похожего на имя пакета Java). Это `groupId` помогает избежать конфликтов в именах между различными проектами.

Пример:

```
<groupId>com.example.myproject</groupId>
```

`artifactId` Это название артефакта (например, библиотеки или веб-приложения), созданного в рамках проекта. Это название должно быть описательным и уникальным в рамках `groupId`.

Пример:

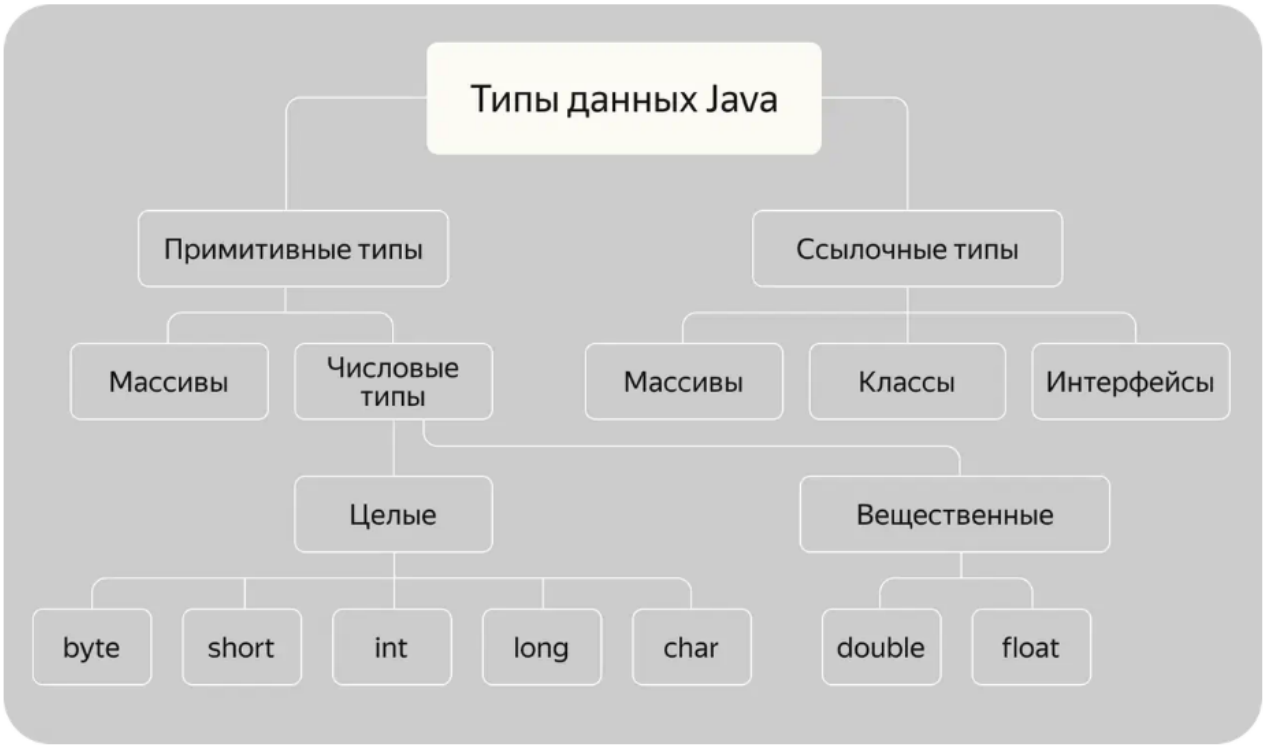
```
<artifactId>моя-библиотека</artifactId>
```

`version` — это текущая версия артефакта. Maven использует номер версии для отслеживания различных выпусков одного и того же артефакта.

4. Типы данных Java: простые и ссылочные. Простые (примитивные) типы данных.

Примитивные типы — это данные, имеющие определённый формат и значение. Они применяются для хранения простой информации, например чисел, символов или логических значений.

Ссылочные типы данных в Java не содержат значения, а ссылаются на место, где они расположены. Проще говоря, это не сами данные, а ссылки. Сюда входят строки, массивы, классы и интерфейсы.



Прежде всего в примитивы входят числовые данные. Они бывают трёх категорий: целые, вещественные и логические.

Целые типы данных в Java

Обозначение	Длина	Диапозон
byte	8 бит	от -128 до 127
short	16 бит	от -32 768 до 32 767
int	32 бита	от −2 147 483 648 до +2 147 483 647
long	64 бита	от −9 223 372 036 854 775 808 до +9 223 372 036 854 775 807

Рассмотрим вещественные типы данных в Java. Сюда входят дробные числа, например 0,5 или 13,5743656.

Обозначение	Длина	Диапозон
float	32 бита	от ±1,4E−45 до ±3,4E+38
double	64 бита	от ±4,9E−324 до ±1,8E+308

Логический тип:

boolean: Может иметь значения true или false, занимает 1 бит (в теории, но на практике часто занимает 1 байт).

Символьный тип:

char: Представляет символ Unicode, занимает 2 байта.

5. Переменные: статические и нестатические. Местные переменные, область видимости переменных. Объявление и инициализация переменных. Константы. Спецификаторы доступа.

Статические переменные относятся к классу в целом, а не к отдельным объектам. Они создаются только один раз и существуют до тех пор, пока класс не будет удален из памяти. К ним можно обращаться через имя класса или напрямую внутри класса

Пример использования: Счетчики, которые отслеживают количество созданных объектов класса.

Нестатические переменные (или экземплярные переменные) принадлежат каждому объекту класса отдельно. Каждый объект имеет свою копию нестатической переменной. Они создаются при создании объекта и удаляются при удалении объекта

Местные переменные объявляются внутри методов или блоков кода. Они существуют только в пределах этого блока и удаляются после его завершения. не имеют значений по умолчанию — их необходимо инициализировать перед использованием.

Область видимости переменной определяет, где она доступна для использования. Для статических переменных область видимости ограничена классом, для нестатических — объектом, а для локальных — блоком кода

Объявление переменной включает в себя указание ее типа и имени. Например, int x;. Инициализация переменной — это присвоение ей начального значения. Например, int x = 5;.

Константы — это переменные, которые не могут быть изменены после инициализации. В Java константы обычно объявляются с помощью ключевого слова final. Например, final int MAX_SIZE = 100;.

В Java используются следующие модификаторы доступа:

- public: публичный, общедоступный класс или член класса. Поля и методы, объявленные с модификатором public, видны другим классам из текущего пакета и из внешних пакетов.
- private: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.
- protected: такой класс или член класса доступен из любого места в текущем классе или пакете или в производных классах, даже если они находятся в других пакетах
- Модификатор по умолчанию. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете.

6. Комментарии: виды, особенности применения.

В языке Java есть три способа выделения комментариев в тексте. Чаще всего используются две косые черты //, при этом комментарий начинается сразу за символами // и продолжается до конца строки.

```
// Это однострочный комментарий
```

Многострочные комментарии используются для более длинных пояснений и могут занимать несколько строк. В языках типа C, C++, Java они обозначаются /* */.

```
/* Это многострочный комментарий
   который занимает несколько строк */
```

Эти комментарии объясняют логику кода и помогают другим разработчикам понять, как работает программа. Они должны отвечать на вопросы «что», «как» и «почему»

Документирующие комментарии(Javadoc комментарии (/** */)) используются для автоматической генерации документации. Они обычно оформляются в виде многострочных комментариев и содержат специальные теги для описания функций и классов.

```
/**
 * Описание функции или класса
 * @param param Описание параметра
 * @return Описание возвращаемого значения
 */
```

Документационные комментарии используются для создания официальной документации к коду. Они обычно содержат информацию о функциях, классах и интерфейсах

Особенности применения:

- Комментарии помогают сделать код более понятным и поддерживаемым.
- Javadoc-комментарии используются для автоматического создания документации, что особенно полезно в больших проектах.
- Следует избегать избыточных комментариев, которые могут загромождать код и дублировать очевидную информацию.

7. Операции языка Java – арифметические, отношения и логические, преобразования числовых типов.

Арифметические операции

Бинарные операции:
Сложение (+): Складывает два числа.
Вычитание (-): Вычитает одно число из другого.
Умножение (*): Умножает два числа.
Деление (/): Делит одно число на другое.
Остаток от деления (%): Возвращает остаток от деления двух целых чисел.
Унарные операции:
Инкремент (++): Увеличивает значение переменной на 1.
Декремент (--): Уменьшает значение переменной на 1.

Операции отношений
Сравнение:
Равенство (==): Проверяет, равны ли два значения.
Неравенство (!=): Проверяет, не равны ли два значения.
Больше (>): Проверяет, больше ли одно значение другого.
Меньше (<): Проверяет, меньше ли одно значение другого.
Больше или равно (>=): Проверяет, больше ли или равно одно значение другому.
Меньше или равно (<=): Проверяет, меньше ли или равно одно значение другому.

Логические операции
Логическое И (&&): Возвращает true, если оба условия верны.
Логическое ИЛИ (||): Возвращает true, если хотя бы одно условие верно.
Логическое НЕ (!): Инвертирует логическое значение.

Преобразования числовых типов

Неявное преобразование:
Происходит автоматически при присвоении значения меньшего типа переменной большего типа (например, int в double).
Автоматически без каких-либо проблем производятся расширяющие преобразования (widening) - они расширяют представление объекта в памяти. Происходит от типа с меньшей разрядностью к большей.
byte -> short -> int -> long
int -> double
short -> float -> double
char -> int

Явное преобразование:
Требуется при присвоении значения большего типа переменной меньшего типа (например, double в int). Используется с помощью оператора (тип) перед значением.

```
double x = 10.5;
int y = (int) x; // y будет равно 10
```

При преобразовании чисел с плавающей точкой необходимо проводить округление, так как при преобразовании будет подведен неверный результат.

```
double a = 56.9898;
int b = (int)Math.round(a); // 57
```

При преобразовании операндов во время операций необходимо приводить их к единой системе исчисления double + double ⇒ double

8. Символьные строки, методы работы со строками Java.

При работе со строками важно понимать, что объект String является неизменяемым (immutable). То есть при любых операциях над строкой, которые изменяют эту строку, фактически будет создаваться новая строка.

Строки можно создавать с помощью конструкторов класса String или напрямую присваивая строку в двойных кавычках.

```
String str1 = "Java";
String str2 = new String("Hello");
```

Основные методы класса String
Основные операции со строками раскрывается через методы класса String, среди которых можно выделить следующие:

Соединение строк
Для соединения строк можно использовать операцию сложения ("+"):
concat(): объединяет строки
join(): соединяет строки с учетом разделителя
valueOf(): преобразует объект в строковый вид

Извлечение символов и подстрок
getChars(): возвращает группу символов
charAt(): возвращает символ строки по индексу

Сравнение строк
compareTo(): сравнивает две строки
equals(): сравнивает строки с учетом регистра
equalsIgnoreCase(): сравнивает строки без учета регистра
regionMatches(): сравнивает подстроки в строках

Поиск в строке

indexOf(): находит индекс первого вхождения подстроки в строку

lastIndexOf(): находит индекс последнего вхождения подстроки в строку

startsWith(): определяет, начинается ли строка с подстроки

endsWith(): определяет, заканчивается ли строка на определенную подстроку

Замена в строке

replace(): заменяет в строке одну подстроку на другую

Обрезка строки

trim(): удаляет начальные и конечные пробелы

substring(): возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса

Изменение регистра

toLowerCase(): переводит все символы строки в нижний регистр

toUpperCase(): переводит все символы строки в верхний регистр

9. Классы StringBuffer и StringBuilder.

Объекты String являются неизменяемыми, поэтому все операции, которые изменяют строки, фактически приводят к созданию новой строки, что сказывается на производительности приложения. Для решения этой проблемы, чтобы работа со строками проходила с меньшими издержками в Java были добавлены классы StringBuffer и StringBuilder. По сути они напоминает расширяемую строку, которую можно изменять без ущерба для производительности.

Эти классы похожи, практически двойники, они имеют одинаковые конструкторы, одни и те же методы, которые одинаково используются. Единственное их различие состоит в том, что класс StringBuffer синхронизированный и потокобезопасный. То есть класс StringBuffer удобнее использовать в многопоточных приложениях, где объект данного класса может меняться в различных потоках. Если же речь о многопоточных приложениях не идет, то лучше использовать класс StringBuilder, который не потокобезопасный, но при этом работает быстрее, чем StringBuffer в однопоточных приложениях.

StringBuffer определяет четыре конструктора:

```
StringBuffer()
StringBuffer(int capacity)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

Аналогичные конструкторы определяет StringBuilder:

```
StringBuilder()
StringBuilder(int capacity)
StringBuilder(String str)
StringBuilder(CharSequence chars)
```

При всех операциях со строками StringBuffer / StringBuilder перераспределяет выделенную память. И чтобы избежать слишком частого перераспределения памяти, StringBuffer/StringBuilder заранее резервирует некоторую область памяти, которая может использоваться. Конструктор без параметров резервирует в памяти место для 16 символов. Если мы хотим, чтобы количество символов было иным, то мы можем применить второй конструктор, который в качестве параметра принимает количество символов.

Третий и четвертый конструкторы обоих классов принимают строку и набор символов, при этом резервируя память для дополнительных 16 символов.

Оба класса имеют одинаковые методы и конструкторы, такие как append(), insert(), delete(), что делает их взаимозаменяемыми в плане функциональности

10. Массивы Java: объявление, инициализация. Основные методы класса Arrays. Доступ к элементам массивов, итерация массивов. Двумерные массивы.

Массив в Java (Java Array) — это структура данных, которая хранит набор пронумерованных значений одного типа (элементы массива). Доступ к конкретной ячейке осуществляется через её номер. Номер элемента в массиве также называют индексом.

Как объявить массив?

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

№	Объявление массива, Java-синтаксис	Примеры	Комментарий

1.	`dataType[] arrayName;`	`int[] myArray;`
		
		`Object[] arrayOfObjects;`
			Желательно объявлять массив именно таким способом, это Java-стиль

2.	`dataType arrayName[];`	`int myArray[];`
		
		`Object arrayOfObjects[];`
			Унаследованный от C/C++ способ объявления массивов, который работает и в Java

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

Инициализация массива — это заполнение его конкретными данными (не по умолчанию).

Инициализация массива происходит с помощью оператора new или с заданием значений сразу:

1. Через new:

```
int[] array = new int[5]; // массив из 5 элементов, все элементы по умолчанию равны 0
```

2. Через инициализацию значений:

```
int[] array = {1, 2, 3, 4, 5}; // массив с заранее заданными значениями
```

Основные методы класса Arrays

Arrays.toString(arr):
Возвращает строковое представление массива, что полезно для вывода содержимого массива на консоль или в лог-файлы

Arrays.copyOf(arr, length):
Создает новый массив, который является копией исходного массива arr, но с указанной длиной length. Если новая длина больше исходной, массив дополняется нулями

Arrays.copyOfRange(arr, from, to):

Создает новый массив, который содержит элементы из исходного массива arr в диапазоне от индекса from до индекса to (не включая to)

Arrays.fill(arr, val):
Заполняет весь массив arr значением val. Также можно указать диапазон индексов для заполнения

Arrays.sort(arr):
Сортирует элементы массива arr в порядке возрастания. Может сортировать как весь массив, так и его часть, указывая начальный и конечный индексы

Arrays.binarySearch(arr, val):
Ищет элемент со значением val в отсортированном массиве arr и возвращает его индекс. Если элемент не найден, возвращает отрицательное значение

Arrays.equals(arr1, arr2):
Сравнивает два массива на равенство. Возвращает true, если массивы одинаковы, и false в противном случае

Arrays.asList(arr):
Возвращает список, содержащий элементы массива arr. Однако этот список является фиксированным и не поддерживает добавление или удаление элементов

Доступ к элементам массивов

Индексация:
Доступ к элементам массива осуществляется по их индексам. Индексация начинается с нуля. Например, первый элемент массива имеет индекс 0, второй — индекс 1 и т.д. 124.

Пример доступа:

```
int[] myArray = {1, 2, 3, 4, 5};
int firstElement = myArray[0]; // получаем значение первого элемента
```

Итерация по массивам

Для обхода массива можно использовать обычный цикл for, улучшенный цикл for-each и другие методы:

1. Цикл for:

```
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]); // вывод каждого элемента
}
```

2. Цикл for-each:

```
for (int element : array) {
```

```
    System.out.println(element); // вывод каждого элемента
}
```

Двумерные массивы

Двумерный массив — это массив массивов. Он представляет собой таблицу с рядами и столбцами.

1. Объявление двумерного массива:
   - Двумерный массив объявляется с помощью двух пар квадратных скобок `[][]`. Например: `int[][] myArray;`.
2. Инициализация двумерного массива:
   - Может быть инициализирован с помощью ключевого слова `new`, указывая размеры каждого измерения. Например: `int[][] myArray = new int[3][4];`
   - Также можно инициализировать сразу при объявлении, перечисляя элементы в фигурных скобках. Например:

```java
int[][] myArray = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

3. Доступ к элементам двумерного массива:
   - Доступ осуществляется по индексам каждого измерения. Например:

```java
int[][] myArray = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
int element = myArray[1][2]; // получаем значение элемента во второй строке и третьем столбце
```

Настя

11. Управляющие конструкции Java: ветвление, циклы. Цикл foreach.

В Java управляющие конструкции делятся на ветвление и циклы, которые обеспечивают управление потоком выполнения программы.

1. **Ветвление (if-else, switch-case)**
   If-else:
   Используется для выполнения частей кода при наличии необходимого выполняющегося условия. Конструкция `if-else` проверяет логическое выражение, которое должно возвращать `true` или `false`, т. е условие в скобках обязательно должно возвращать объект типа boolean.
   Используется, когда заранее известно количество итераций.
   Синтаксис:

```
int number = 10;
if (number > 0) {
    System.out.println("Число положительное");
} else if (number < 0) {
    System.out.println("Число отрицательное");
} else {
    System.out.println("Число равно нулю");
}
```

   Switch-case:
   Используется для проверки значения переменной на совпадение с несколькими случаями
   Поддерживает типы: `int`, `byte`, `short`, `char`, `String`, `enum.`

Синтаксис:

```
int day = 3;
switch (day) {
    case 1:
        System.out.println("Понедельник");
        break;
    case 2:
        System.out.println("Вторник");
        break;
    case 3:
        System.out.println("Среда");
        break;
    default:
        System.out.println("Неизвестный день");
}
```

Особенности ветвления

- Выбор между `if-else` и `switch`:
  - `if-else` – удобен для проверки сложных логических условий.
  - `switch` – лучше для проверки конкретных значений.

2. **Циклы (for, foreach, while, do-while)**

Циклы позволяют выполнять одно и то же действие несколько итераций подряд.

for:

Используется, когда известно количество итераций. Состоит из 4 частей: Инициализация начальной точки: выполняется перед первой итерацией, Условие, при котором цикл заканчивается: проверяется перед каждой итерацией,Счетчик (i++) Тело цикла: выполняется на каждой итерации.

Для досрочного выхода из цикла используется оператор break.

```java
public static void findNumberInLoop(int number){
    for (int i = 0; i < 10; i++) {
        if (i == number) {
            break;
        }
        System.out.println(i);
    }
    System.out.println("cycle was finished");
}
```

Для пропуска необходимой итерации используется оператор continue.

```java
for (int i = 0; i < 10; i++){
    if (i == 5)
        continue;
    System.out.println(i);
}
```

foreach:

For-each — это разновидность цикла for, которая используется, когда нужно обработать все элементы массива или коллекции.
Синтаксис:

```
for (type itVar : array)
{
    Блок операторов;
}
```

Где `type` — тип итерационной переменной (совпадает с типом данных в массиве!), `itVar` — её имя, `array` — массив (тут также может быть другая структура данных, какая-нибудь коллекция, например, `ArrayList`), то есть объект, по которому выполняется цикл. Как вы видите, счётчик в такой конструкции не применяется, итерационная переменная перебирает элементы массива или коллекции, а не значения индекса.

while:

Выполняет блок кода, пока условие остается истинным. Подходит, когда количество итераций заранее неизвестно.

Синтаксис:

```
int count = 0;
while (count < 5) {
    System.out.println("Count: " + count);
    count++;
}
```

do-while:

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

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

Особенность: цикл do гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции while не будет истинно.

12. Перечислить и дать описание основных принципов объектно-ориентированного программирования (ООП). Достоинства и недостатки ООП.

Объектно-ориентированное программирование основывается на четырех ключевых принципах, которые обеспечивают его гибкость, модульность и расширяемость.

1. Абстракция. Это процесс выделения значимой информации и придание объекту характеристик, которые отличают его от всех объектов, чётко определяя его концептуальные границы. **Суть абстракции** заключается в том, что каждый верхний слой над объектом (классом) является более абстрактным, чем его «младшая версия». **На основе абстрактных классов** формируются все более конкретные классы и вытекающие из них объекты. При этом реализацию функции не нужно заранее переписывать, программист оставляет базовый шаблон для последующих надстроек. Абстрактный класс отличается от дочерних тем, что он является публичным и включает в себя реализацию методов.

2. Инкапсуляция. Это принцип, согласно которому **вся информация, необходимая для работы конкретного объекта, должна храниться внутри этого объекта**. Если нужно вносить изменения, методы для этого тоже должны лежать в самом объекте — посторонние объекты и классы этого делать не могут.

Для внешних объектов доступны только публичные атрибуты и методы. **Задача программиста** — определить, какие атрибуты и методы будут доступны для открытого доступа, а какие являются внутренней реализацией объекта и должны быть недоступны для изменений. **Инкапсуляция обеспечивает безопасность** и не даёт повредить данные внутри какого-то класса со стороны. Ещё она помогает избежать случайных зависимостей, когда из-за изменения одного объекта что-то ломается в другом.

3. Наследование. Это **концепция, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа**. В общем виде наследование выглядит так: **потомок при создании получает все свойства и методы родителя**. Родитель — это класс, на основе которого создаётся что-то новое. Потомок (или дочерний элемент) — это то, что получилось при создании на основе класса или объекта. **Наследование полезно**, так как оно даёт возможность **структурирования и повторного использования кода**. Это способствует более быстрой и качественной разработке.

4. Полиморфизм. Это **понятие, которое позволяет разным сущностям выполнять одни и те же действия**. При этом неважно, как эти сущности устроены внутри и чем они различаются. Полиморфизм позволяет использовать один интерфейс для работы с разными типами объектов. **Пример**: есть две разных сущности — картинка и видео. И тем, и другим можно поделиться: отправить в личное сообщение другому человеку. Программист может сделать два разных метода — один для картинки, другой для видео. А может воспользоваться полиморфизмом и создать один метод «Отправить» для обеих сущностей. Или такой пример:

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

public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}
```

Преимущества:

1. **Модульность:** ООП позволяет разбивать код на модули, что делает его более управляемым и понятным. Каждый класс представляет собой отдельный модуль, который можно разрабатывать и тестировать независимо от других. Это особенно полезно в больших проектах, где разные команды могут работать над разными частями системы. Модульность также упрощает процесс отладки и тестирования, так как можно изолированно проверять каждый модуль. Модульность способствует лучшей организации кода и облегчает его поддержку.
2. **Повторное использование кода.** Классы и объекты можно использовать повторно в разных частях программы или даже в разных проектах. Это снижает количество дублирующегося кода и упрощает процесс разработки. Повторное использование кода позволяет экономить время и ресурсы, так как разработчикам не нужно писать один и тот же код заново.

Недостатки:

1. **Сложность разработки**. Требует больше времени на проектирование.

2. **Производительность**. ООП может быть менее производительным по сравнению с процедурным программированием из-за дополнительных накладных расходов на управление объектами и методами. Производительность ООП может быть ниже, что может привести к увеличению времени выполнения программ и снижению эффективности.
3. **Избыточность.** Избыточность может возникнуть из-за необходимости создания множества классов и объектов. Это может усложнить структуру программы и сделать её менее понятной.
4. **Зависимость от конкретного языка.**

13. Определение класса. Объявление класса. Спецификаторы доступа. Отношения между классами Java (наследование, зависимость, агрегирование). Статические члены класса. Переменные класса.

Класс в Java — это шаблон (или структура), который описывает свойства (переменные) и поведение (методы) объектов, создаваемых на его основе.
Объявление класса включает в себя спецификатор доступа, ключевое слово class, название класса с заглавной буквы, и тело класса в {}.

```
public class Example {
    int number;  // Поле
    public void showNumber() {  // Метод
        System.out.println(number);
    }
}
```

Спецификаторы доступа. Это ключевые слова в объектно-ориентированных языках, которые задают параметры доступа для классов, методов и прочих элементов.
Существуют четыре типа спецификаторов доступа:

Public. Все члены класса (переменные), объявленные в спецификаторе public, будут доступны всем.
Private. Все члены класса, объявленные в спецификаторе private, будут доступны только этому классу.
Default (пакетный доступ). Все члены класса, объявленные в спецификаторе default, будут доступны только для одного и того же пакета.
Protected. Все члены класса, объявленные в спецификаторе protected, будут доступны только для классов из того же пакета и всех подклассов.

Спецификаторы доступа помогают контролировать, какая часть программы может получить доступ к членам класса, чтобы предотвратить неправильное использование данных.

Отношения между классами.
1) **Наследование.** Класс может наследовать свойства и методы другого класса с помощью ключевого слова `extends`. Родительский класс (superclass) передает свои поля и методы дочернему классу (subclass).
2) **Зависимости.** Зависимость возникает, если один класс использует другой в качестве параметра или внутри методов. Это временная связь.

```
class Engine {
    void start() {
        System.out.println("Engine started.");
    }
}

class Car {
    void startCar(Engine engine) {  // Зависимость от класса Engine
        engine.start();
    }
```

3) **Агрегирование**. Агрегирование — это "has-a" отношение. Один класс содержит ссылку на объект другого класса. Оба объекта могут существовать независимо друг от друга.

```java
class Engine {
    void start() {
        System.out.println("Engine started.");
    }
}

class Car {
    Engine engine = new Engine();  // Агрегирование
    void startCar() {
        engine.start();
    }
}
```

Статические члены (поля и методы) класса (спецификатор static).
При создании объектов класса для каждого объекта создается своя копия нестатических обычных полей. А статические поля являются общими для всего класса. Поэтому они могут использоваться без создания объектов класса. Интересный и наглядный пример: Есть класс Person, который содержит переменную id и статическую переменную counter. Она увеличивается в конструкторе и ее значение присваивается переменной id. То есть при создании каждого нового объекта Person эта переменная будет увеличиваться, поэтому у каждого нового объекта Person значение поля id будет на 1 больше чем у предыдущего. К статическим переменным можно обращаться через класс, а не экземпляр.

Переменные класса (Локальные, переменные экземпляра и переменные класса(статические)).
1) Локальные.
Локальные переменные объявляются в методах, конструкторах или блоках. Создаются, когда метод, конструктор или блок запускается и уничтожаются после того, как завершиться метод, конструктор или блок. Модификаторы доступа нельзя использовать для локальных переменных. Они являются видимыми только в пределах объявленного метода, конструктора или блока. В Java не существует для локальных переменных значения по умолчанию, так что они должны быть объявлены и начальное значение должны быть присвоено перед первым

использованием.

2) Переменные экземпляра.

Переменные экземпляра объявляются в классе, но за пределами метода, конструктора или какого-либо блока. Когда для объекта в стеке выделяется пространство, создается слот для каждого значения переменной экземпляра. В Java переменные экземпляра создаются тогда, когда объект создан с помощью ключевого слова «new» и разрушаются тогда, когда объект уничтожается. Переменные экземпляра могут быть объявлен на уровне класса, до или после использования. Имеют разные модификаторы (спецификаторы) доступа. Видны для всех методов класса. Переменные содержат значения, которые должны ссылаться более чем на один метод, конструктор или блок, или на основные части состояния объекта, которые должны присутствовать на протяжении всего класса, можно провести аналогию с атрибутами (так надеюсь понятнее).

3) Статические переменные.

Переменные класса, также известные в Java как статические переменные, которые объявляются со статическим ключевым слово в классе, но за пределами метода, конструктора или блока. Существует только одна копия этих переменных, независимо от количества экземпляров. В Java статические переменные создаются при запуске программы и уничтожаются, когда выполнение программы остановится. Видимость похожа на переменную экземпляра. Однако большинство статических переменных объявляются как public, поскольку они должны быть доступны для пользователей класса. Статические переменные могут быть доступны посредством вызова с именем класса *ClassName.VariableName*.

```
class Example {
    int instanceVariable;       // Переменная экземпляра
    static int classVariable;   // Статическая переменная

    void display() {
        int localVariable = 10; // Локальная переменная
        System.out.println(localVariable);
    }
}
```

14. Объект класса. Создание объекта. Конструктор класса, конструктор по умолчанию. Ключевое слово this. Перегрузка конструкторов. Доступ к переменным экземпляра.

Объект — это экземпляр класса. Он содержит конкретное состояние (значения полей) и поведение (методы), определенные в классе. Объект создается с помощью ключевого слова new, которое вызывает конструктор класса. Синтаксис: ClassName objectName = new ClassName(type valName);

Конструктор класса в Java — это специальный метод, который имеет имя, совпадающее с именем класса, и вызывается при создании экземпляра объекта совместно с оператором new, и не имеет возвращаемого типа. Конструктор инициализирует объект, его поля и методы.

```
class Person {
    String name;
    int age;
    // Конструктор
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

В Java существует одна особенность - если в классе не было создано ни одного конструктора, компилятор автоматически добавляет **конструктор по умолчанию.** Он не имеет параметров и инициализирует все переменные со значением по умолчанию. Для числовых - 0, для булевых - false, для ссылочных - null.

Ключевое слово this.

Ключевое слово this в Java используется для ссылки на текущий объект. Другими словами, this ссылается на экземпляр класса, внутри которого оно используется. С помощью ключевого слова можно обращаться не только к параметрам, но и к методам класса. Использование ключевого слова ‘this’ для ссылки на текущие переменные экземпляра класса.

```
class Test {
    int a;
    int b;

     // Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
```

```
        this.b = b;
    }
```

Использование this() для вызова текущего конструктора класса.

```
class Test {
    int a;
    int b;

     // Default constructor
    Test()
    {
        this(10, 20);
        System.out.println(
            "Inside  default constructor \n");
    }

     // Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
        System.out.println(
            "Inside parameterized constructor");
    }
```

Использование ключевого слова ‘this’ для возврата текущего экземпляра класса.

```
class Test {
    int a;
    int b;

     // Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

     // Method that returns current class instance
    Test get() { return this; }
```

Использование ключевого слова ‘this’ в качестве параметра метода.

```
class Test {
    int a;
```

```
    int b;

     // Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

     // Method that receives 'this' keyword as parameter
    void display(Test obj)
    {
        System.out.println("a = " + obj.a
                           + "  b = " + obj.b);
    }

     // Method that returns current class instance
    void get() { display(this); }
```

Использование ключевого слова ‘this’ для вызова текущего метода класса.

```
class Test {

     void display()
    {
        // calling function show()
        this.show();

          System.out.println("Inside display function");
    }

     void show()
    {
        System.out.println("Inside show function");
    }
```

Перегрузка конструкторов.

Перегрузка (overloading) — это определение нескольких конструкторов с разными наборами параметров. Она позволяет создавать объекты с разной степенью детализации.

```java
class Car {
    String brand;
    int speed;

    // Конструктор без параметров
    Car() {
        this.brand = "Unknown";
        this.speed = 0;
    }

    // Конструктор с одним параметром
    Car(String brand) {
        this.brand = brand;
        this.speed = 0;
    }

    // Конструктор с двумя параметрами
    Car(String brand, int speed) {
        this.brand = brand;
        this.speed = speed;
    }
```

Доступ к переменным осуществляется через созданный объект. Путь строится следующим образом: Создается объект класса ClassName objectName = new ClassName();
Вызывается переменная: objectName.valName;

15. Методы, объявление, имя. Статические методы. Доступ к методам. Спецификаторы доступа.

Методы - это именованный блок кода, который объявляется внутри класса и может быть использован многократно.
Метод состоит из шести частей:

> **Спецификатор доступа**. Позволяет указать, из какого кода можно получить доступ к методу. Существует четыре возможных модификатора доступа: public, protected, private и default (также называемый package-private).
> **Модификатор: static** говорит о том, что класс, объект или поле статичны, т е

могут быть вызваны без созданного экземпляра класса. **Final** служит для переменной указанием на то, что она является константой. Для методов — что они не могут быть переопределены при наследовании, а для классов — это указание на то, что наследоваться от него нельзя. **Abstract** применим только к методам и классам. Если класс помечается как абстрактный, он либо содержит абстрактные методы, либо это делается для того чтобы запретить создание экземпляров этого класса.

Возвращаемый тип. Тип значения, возвращаемого методом, если таковое имеется. Метод может возвращать примитивное значение или ссылку на объект, или он может ничего не возвращать, если используется ключевое слово void в качестве возвращаемого типа.

Идентификатор метода. Имя, которое даётся методу. должно быть уникальным в пределах класса и соответствовать camelCase/

Список параметров. Необязательный список входных данных для метода, разделённый запятыми. Метод может иметь от 0 до 255 параметров, разделённых запятыми.

Список исключений. Необязательный список исключений, которые может выдавать метод.

Тело метода. Определение логики (может быть пустым)

Статические методы принадлежат классу, а не конкретному объекту, и вызываются без создания объекта. Объявляются с ключевым словом `static`. Используются для операций, не зависящих от состояния объекта (например, математические вычисления, вспомогательные функции). Статические методы не могут напрямую обращаться к нестатическим полям или методам.

Доступ к методам. Для вызова нестатического метода необходимо создать объект класса. Статические методы вызываются через имя класса.

Спецификаторы доступов.

Public. Все члены класса, объявленные в этом спецификаторе, будут доступны всем. Доступ к общедоступным членам класса можно получить из любой точки программы с помощью оператора прямого доступа (.) с объектом этого класса.

Private. Все члены класса, объявленные в этом спецификаторе, будут доступны только этому классу.

Protected. Метод доступен из классов в том же пакете и из подклассов (даже если они в другом пакете).

Default. Метод доступен только в пределах одного пакета**.**

16. Пакеты Java. Импорт пакетов и классов. Статический импорт.

Пакет (package) — это способ организации классов и интерфейсов в структурированную иерархию. Пакеты помогают:

- Избежать конфликта имен (например, два класса с именем `Employee` могут находиться в разных пакетах).
- Упростить управление проектами, особенно большими.
- Контролировать доступ к классам и методам (через модификаторы доступа).

Импорт пакетов и классов.
1) Импортирование конкретного класса: import mypackage.MyClass;
2) Импортирование всех классов: Импортирует все классы пакета, но не включает вложенные пакеты. Это удобно, если используется много классов из одного пакета. import mypackage.*;
Статический импорт: позволяет напрямую использовать статические члены (поля и методы) класса, без указания имени класса. import static имя_класса.имя_статического_члена; или import static имя_класса.*;
Особенности статического импорта:

- Уменьшает количество кода, но может снижать читаемость, если неочевидно, из какого класса вызываются методы.
- Использовать с осторожностью, чтобы избежать конфликтов имен.

17. Вложенные и внутренние классы Java. Статические и нестатические внутренние классы.

Вложенный класс (Nested Class) — это класс, который определен внутри другого класса. Область действия вложенного класса ограничена областью действия внешнего класса. Если класс B определен в классе A, то класс B не может существовать независимо от класса A. Вложенный класс имеет доступ к членам (в том числе закрытым) того класса, в который он объявлен. Вложенный класс создается для того, чтобы обслуживать окружающий его класс. Если вложенный класс оказывается полезен в каком-либо ином контексте, он должен стать классом верхнего уровня.
Внутренний класс (Inner Class) — это вложенный класс, который не является статическим, не может содержать статические переменные или методы, если только они не являются константами. Экземпляр **внутреннего класса** может существовать только тогда, когда существует конкретный экземпляр внешнего класса.
!По определению внутренний класс не может быть статическим, но вопрос содержит термин статический внутренний класс???????? В интете нет такого тоже!

Типы вложенных классов

1. **Статические вложенные классы** (Static Nested Class).объявляется с ключевым словом `static`. Он не связан с объектом внешнего класса,

поэтому не имеет доступа к нестатическим членам внешнего класса.
Особенности:Может содержать как статические, так и нестатические методы. Доступ к статическим членам внешнего класса.

2. **Нестатические внутренние классы** (Non-Static Inner Class). Не статические вложенные классы называются внутренними классами, если они связанны с внешним классом. Имеет доступ к **всем членам внешнего класса**, включая `private`. Для создания объекта внутреннего класса требуется объект внешнего класса.
3. **Локальные классы** (Local Classes) — объявляются внутри метода. Объявляется внутри метода и доступен только в пределах этого метода.
4. **Анонимные классы** (Anonymous Classes) — используются для создания объектов "на месте". Используется для создания объекта класса без явного объявления нового класса. Обычно применяется для реализации интерфейсов или наследования классов на месте.

Гпт такое сгенерировал:

Характеристика	Статический вложенный класс	Нестатически
Доступ к членам внешнего класса	Только к статическим	К любым, вкл
Требуется объект внешнего класса	Нет	Да
Объявление `static`	Да	Нет
Типы членов	Может содержать статические члены	Не может соде члены

18. Наследование. Подклассы и суперклассы. Доступ к членам класса. Конструкторы при наследовании, ключевое слово super.

Наследование — это механизм, который позволяет классу (подклассу) наследовать свойства (поля) и методы другого класса (суперкласса). При этом подкласс может добавлять собственные поля и методы, а также переопределять унаследованные методы.
Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово **extends**, после которого идёт

имя базового класса.
В Java существует три типа наследования:

> **Одиночное наследование** (Single inheritance) — когда один подкласс наследует свойства и методы только у одного суперкласса.
> **Многоуровневое -** когда существует цепочка наследования
> **Иерархическое -** когда два или более классов наследуют один класс

Все классы в Java наследуются от класса `Object`, который предоставляет базовые методы:

- `toString()` — преобразует объект в строку.
- `equals()` — сравнивает объекты.
- `hashCode()` — возвращает хэш-код объекта.

Суперкласс:

- Это класс, который предоставляет свои свойства и методы для подклассов.
- Может быть любым классом, кроме `final` (класс, объявленный как `final`, нельзя наследовать).

Подкласс:

- Класс, который расширяет суперкласс, используя ключевое слово `extends`.
- Может переопределять методы суперкласса.

Доступ к элементам класса. (package-private - это default)

Модификатор	Видимость в подклассе (в том же пакете)	Видимость в подклассе (в
`public`	Доступен	Доступен
`protected`	Доступен	Доступен
(package-private)	Доступен	Недоступен
`private`	Недоступен	Недоступен

Конструкторы при наследовании.
Конструктор суперкласса не наследуется, но всегда вызывается при создании объекта подкласса. Вызов конструктора суперкласса происходит автоматически

или явно через ключевое слово `super`.

Неявный вызов:

```
class Superclass {
    Superclass() {
        System.out.println("Constructor of Superclass");
    }
}

class Subclass extends Superclass {
    Subclass() {
        System.out.println("Constructor of Subclass");
    }
}

public class Main {
    public static void main(String[] args) {
        Subclass obj = new Subclass();
        // Вывод:
        // Constructor of Superclass
        // Constructor of Subclass
    }
}
```

Явный вызов (super):

```
class Superclass {
    Superclass(String message) {
        System.out.println("Superclass constructor: " + message);
    }
}

class Subclass extends Superclass {
    Subclass(String message) {
        super(message); // Явный вызов конструктора суперкласса
        System.out.println("Subclass constructor: " + message);
    }
}

public class Main {
    public static void main(String[] args) {
        Subclass obj = new Subclass("Hello");
```

```
        // Вывод:
        // Superclass constructor: Hello
        // Subclass constructor: Hello
    }
}
```

`super` — это ссылка на суперкласс, которая используется для:

1. Вызова конструктора суперкласса.
2. Доступа к членам суперкласса, если они скрыты в подклассе.

Переопределение методов (Override)
Подкласс может переопределить метод суперкласса для изменения его поведения.
Правила:

1. **Сигнатура метода должна совпадать.**
2. **Уровень доступа должен быть таким же или более открытым.**
3. **Нельзя переопределить метод, объявленный как `final`.**

19. Иерархия наследования Java. Преобразование типов при наследовании. Ключевое слово instanceof.

Иерархия наследования - это набор классов, связанных отношением наследования. В Java можно строить иерархии, состоящие из любого количества уровней наследования. Например, если имеются три класса A, B и C, то класс C может наследовать от класса B, а тот, в свою очередь, от класса A. В таком случае каждый подкласс наследует характерные особенности всех своих суперклассов. **В иерархии классов конструкторы выполняются в порядке наследования**, начиная с суперкласса и кончая подклассом.
Все классы в Java неявно наследуются от класса `Object`.
Этот класс предоставляет общие методы:

- `toString()` — возвращает строковое представление объекта.
- `equals()` — сравнивает объекты.
- `hashCode()` — возвращает хэш-код объекта.
- `getClass()` — возвращает класс объекта.

- `clone()` — создает копию объекта (если класс реализует интерфейс `Cloneable`).

Преобразование типов при наследовании
Т.к. подкласс содержит все методы суперкласса, от которого он был унаследован, то объект этого класса можно сохранить в переменную любого из его типов родителей.
Преобразование типов при наследовании нужно для того, чтобы использовать все функциональные возможности объекта после того, как его фактический тип был на время забыт.
Java поддерживает два типа преобразования при наследовании:

1. Расширение (Upcasting, восходящее преобразование) — преобразование объекта подкласса в объект суперкласса. Движение по цепочке наследования вверх. Но при этом теряется возможность вызвать методы, которые были добавлены в класс при наследовании.
2. Сужение (Downcasting) — преобразование объекта суперкласса в объект подкласса.

Расширение. Это безопасное преобразование. Не требует явного указания. При upcasting доступны только методы и поля суперкласса.
Upcasting не изменяет объект, а лишь ограничивает доступ к методам и полям суперкласса.
Пример: есть суперкласс Animal и подкласс Dog

```
class Animal {
    void eat() {
        System.out.println("Animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        animal.eat();            // Доступен метод суперкласса
        // animal.bark();        // Ошибка: метод bark() недоступен
```

```
    }
}
```

Сужение. Требует явного указания (Если объект суперкласса не был изначально объектом подкласса, возникнет ошибка `ClassCastException`.) Downcasting потенциально небезопасен. Всегда используйте `instanceof` перед приведением типов.

```
class Animal {
    void eat() {
        System.out.println("Animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog();
        Dog dog = (Dog) animal;     // Downcasting, animal приводится к типу Dog
        dog.bark();                 // Теперь доступен метод bark(), функционал объекта Dog,  доступен через dog
    }
}
```

Ключевое слово instanceof.

Используется для проверки, является ли объект экземпляром определенного класса или его подкласса. Возвращает true/false.

Гриша

20. Полиморфизм в Java. Перегрузка и переопределение методов.

Полиморфизм в Java – это способность объекта принимать множество форм. Реализуется двумя способами: через переопределение методов (runtime полиморфизм) и перегрузку методов (compile-time полиморфизм).

Перегрузка методов (Overloading):

Один класс, методы с одинаковым именем, но разной сигнатурой (количество/тип аргументов).
Происходит на этапе компиляции.

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b;
    }
}
```

Переопределение методов (Overriding):
Подкласс изменяет поведение метода, унаследованного от суперкласса.
Требуется одинаковая сигнатура метода и аннотация @Override.
Работает во время выполнения.

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

21. Абстрактные методы и классы Java.

Абстрактные классы в Java — это классы, которые не могут быть созданы как объекты. Они служат для определения общих характеристик и поведения для подклассов.

Основные свойства:
Абстрактный метод — метод без реализации, который подклассы обязаны переопределить.

```
abstract class Shape {
    abstract void draw(); // Абстрактный метод
}
```

Абстрактный класс может содержать:

Абстрактные и обычные методы.
Поля и конструкторы.

java
Копировать
Редактировать
```
abstract class Animal {
    String name;
    Animal(String name) {
        this.name = name;
    }
    abstract void sound(); // Абстрактный метод
    void sleep() { // Обычный метод
        System.out.println(name + " is sleeping");
    }
}
```
Подклассы обязаны реализовать все абстрактные методы родительского класса.

```
class Dog extends Animal {
    Dog(String name) {
        super(name);
    }
    @Override
    void sound() {
        System.out.println(name + " barks");
    }
}
```
Отличия от интерфейсов:
Абстрактный класс может содержать поля и методы с реализацией.
Класс может наследовать только один абстрактный класс, но реализовать множество интерфейсов.

22. Интерфейсы Java: определение интерфейса, реализация интерфейса. Преимущества применения интерфейсов. Переменные интерфейсов. Наследование интерфейсов. Методы по умолчанию. Статические методы интерфейсов.

Интерфейс — это коллекция абстрактных методов (без реализации), которые класс должен реализовать. Интерфейсы используются для задания контракта, который должны соблюдать классы.
Определение интерфейса:
```
interface Animal {
    void sound();
    void eat();
}
```

Реализация интерфейса:

Класс реализует интерфейс с помощью ключевого слова implements. Класс должен переопределить все методы интерфейса.

```
class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
    @Override
    public void eat() {
        System.out.println("Dog eats");
    }
}
```

Преимущества интерфейсов:
Поддержка множественного наследования (класс может реализовать несколько интерфейсов).
Обеспечение гибкости и расширяемости системы.
Улучшение модульности (легче изменять или заменять компоненты).
Интерфейсы используются для работы с полиморфизмом.

Переменные интерфейсов:
Все переменные в интерфейсах:
Неявно public, static, и final.

Должны быть инициализированы при объявлении.

```
interface Constants {
    int MAX_SIZE = 100; // public static final автоматически
}
```

Наследование интерфейсов:
Интерфейс может наследовать другой интерфейс с помощью extends.

```
interface Flyable {
    void fly();
}
interface Bird extends Flyable {
    void sing();
}
```

Методы по умолчанию (default methods):
Позволяют добавлять реализацию в интерфейс без нарушения существующего кода.

```
interface Animal {
    default void sleep() {
        System.out.println("Animal is sleeping");
    }
}
```

Статические методы интерфейсов:
Статические методы принадлежат самому интерфейсу, вызываются через имя интерфейса.

```
interface Utils {
    static void print(String message) {
        System.out.println(message);
    }
}
Utils.print("Hello, static method!");
```

23. Исключения (exception) Java. Синтаксис объявления исключений. Классификация исключений. Основные классы для работы с исключениями. Исключения при наследовании.

Исключения — это события, возникающие во время выполнения программы, которые нарушают её нормальный поток. Исключения помогают обрабатывать ошибки без прерывания работы программы.

Синтаксис объявления исключений:
Обработка исключений (try-catch):

```
try {
    // Код, который может вызвать исключение
} catch (ExceptionType e) {
    // Обработка исключения
} finally {
    // Код, который выполнится всегда (необязательно)
}
```

Генерация исключений (throw):

```
throw new ExceptionType("Сообщение об ошибке");
```

Указание исключений в методе (throws):

```
void method() throws ExceptionType {
    // Код, который может вызвать исключение
}
```

Классификация исключений:
Проверяемые (Checked Exceptions):
Проверяются во время компиляции.
Пример: IOException, SQLException.

```
import java.io.*;
void readFile() throws IOException {
    FileReader file = new FileReader("file.txt");
```

}

Непроверяемые (Unchecked Exceptions):
Не проверяются во время компиляции.
Пример: NullPointerException, ArithmeticException.
int result = 10 / 0; // ArithmeticException

Ошибки (Errors):
Указывают на серьёзные проблемы в работе JVM.
Пример: OutOfMemoryError, StackOverflowError.

Основные классы для работы с исключениями:
Throwable: Базовый класс для всех исключений и ошибок.
Exception: Базовый класс для проверяемых исключений.
RuntimeException: Базовый класс для непроверяемых исключений.
Error: Представляет ошибки JVM.

Исключения при наследовании:
Если метод родительского класса **объявляет исключение:**
Подкласс может:
Не выбрасывать исключение.
Выбросить то же самое исключение.
Выбросить исключение, являющееся подклассом объявленного.

Если метод родительского класса **не объявляет исключений:**

Подкласс не может выбрасывать проверяемые исключения.

```
class Parent {
    void method() throws IOException {
        // Может выбросить IOException
    }
}
class Child extends Parent {
    @Override
    void method() throws FileNotFoundException { // ОК, FileNotFoundException — подкласс
IOException
        // Реализация
    }
}
```

24. Коллекции: Java collections framework. Классификация интерфейсов коллекций. Интерфейс Collection.

Java Collections Framework предоставляет множество классов и интерфейсов для работы с группами объектов.

Классификация интерфейсов коллекций:
List – упорядоченные коллекции с возможностью дублирования элементов.
Примеры: ArrayList, LinkedList, Vector.

Set – коллекции, содержащие уникальные элементы (без дублирования).
Примеры: HashSet, LinkedHashSet, TreeSet.

Map – коллекции, где каждый элемент связан с уникальным ключом.
Примеры: HashMap, HashTable, TreeMap.

Queue – интерфейсы для работы с очередями.
Примеры: PriorityQueue, LinkedList (реализует Queue).

Интерфейс Collection

Collection – родительский интерфейс для всех коллекций из Java Collections Framework.
Основные операции:
add(): добавляет элемент.
remove(): удаляет элемент.
contains(): проверяет наличие элемента.
size(): возвращает количество элементов.
isEmpty(): проверяет пустоту коллекции.

Методы из интерфейса Collection:
boolean add(E e)
boolean remove(Object o)
boolean contains(Object o)
int size()
boolean isEmpty()

25. Списки. Интерфейс List. Основные классы, реализующие интерфейс List. ArrayList, особенности, методы. Comparator.

List — это интерфейс из Java Collections Framework, представляющий упорядоченные коллекции с возможностью дублирования элементов.

Интерфейс List
Основные операции интерфейса List:

Упорядоченность элементов.
Вставка и удаление элементов по индексу.
Поддержка дублированных элементов.

Методы из интерфейса List:
void add(int index, E element)
E get(int index)
E remove(int index)
int indexOf(Object o)
int size()

Основные классы, реализующие интерфейс List:
ArrayList
LinkedList
Vector
Stack
Сегодня наиболее популярны ArrayList и LinkedList.

ArrayList: Особенности и методы
ArrayList — это динамический массив, который увеличивается по мере добавления новых элементов.

Основные методы ArrayList:
add(E e): добавляет элемент в конец списка.
add(int index, E element): вставляет элемент по указанному индексу.
get(int index): возвращает элемент по индексу.
remove(int index): удаляет элемент по индексу.
size(): возвращает количество элементов.
contains(Object o): проверяет наличие элемента.
clear(): очищает список.
set(int index, E element): заменяет элемент по указанному индексу.

Comparator используется для сравнения объектов в коллекциях, таких как List, для изменения порядка сортировки.

26. Интерфейс Set. Основные реализации. HashSet. TreeSet.

Set — это интерфейс из Java Collections Framework, представляющий коллекцию элементов без дублирования. То есть в Set нельзя добавлять повторяющиеся элементы.

Основные реализации интерфейса Set:
HashSet
TreeSet

HashSet

HashSet — это неупорядоченная коллекция, которая не гарантирует сохранение порядка элементов. Внутри используется хэширование для обеспечения уникальности элементов.

Особенности HashSet:
Не поддерживает дублирование элементов.
Работает быстро за счет хэширования.
Не гарантирует порядок элементов.

Методы HashSet:
add(E e): добавляет элемент.
remove(Object o): удаляет элемент.
contains(Object o): проверяет наличие элемента.
size(): возвращает количество элементов.
clear(): очищает множество.

TreeSet — это упорядоченная коллекция, основанная на структуре данных дерево (Tree). Элементы сохраняются в отсортированном порядке.

Особенности TreeSet:
Элементы автоматически упорядочиваются.
Использует компаратор (Comparator) для сортировки по умолчанию.
Поддерживает уникальные элементы.

Методы TreeSet:
add(E e): добавляет элемент.
remove(Object o): удаляет элемент.
first(): возвращает первый элемент.
last(): возвращает последний элемент.
headSet(E toElement): возвращает элементы, меньшие заданного.
tailSet(E fromElement): возвращает элементы, большие или равные заданному.

27. Очереди в Java, интерфейс Queue, PriorityQueue. Структура, основные методы.

Интерфейс Queue в Java
Queue — это интерфейс, представляющий очередь с возможностью добавления и извлечения элементов. Очередь поддерживает элементы в порядке их добавления (FIFO — First In, First Out).

Основные методы интерфейса Queue:
add(E e): добавляет элемент в очередь.
offer(E e): также добавляет элемент, но возвращает false в случае переполнения.
remove(): удаляет и возвращает первый элемент (генерирует NoSuchElementException при пустой очереди).

poll(): удаляет и возвращает первый элемент или null, если очередь пуста.
peek(): возвращает первый элемент, но не удаляет его.
size(): возвращает количество элементов в очереди.

PriorityQueue
PriorityQueue — это очередь с приоритетом. Элементы отсортированы на основе их приоритета (естественный порядок или заданный компаратором).

Особенности PriorityQueue:
Упорядоченная очередь, элементы извлекаются в порядке их приоритета.
Приоритет определяется на основе компаратора или естественного порядка.

Методы PriorityQueue:
add(E e): добавляет элемент в очередь.
remove(): удаляет элемент с наименьшим приоритетом.
poll(): возвращает элемент с наименьшим приоритетом, или null при пустой очереди.
peek(): возвращает элемент с наименьшим приоритетом, но не удаляет его.
size(): возвращает количество элементов.

28. Байтовые потоки InputStream и OutputStream. Консольный ввод и вывод Java. Символьные потоки данных. Абстрактные классы Writer, Reader.

InputStream и OutputStream являются базовыми классами для работы с потоками ввода и вывода байтовых данных в Java.

InputStream
Представляет поток для ввода байтовых данных.
Основные методы:
int read(): читает один байт из потока.
int read(byte[] b): читает массив байтов из потока.
void close(): закрывает поток.

OutputStream
Представляет поток для вывода байтовых данных.
Основные методы:
void write(int b): записывает один байт в поток.
void write(byte[] b): записывает массив байтов в поток.
void close(): закрывает поток.

Консольный ввод и вывод
Для работы с консольным вводом и выводом в Java используются потоки:
System.in — для ввода данных.
System.out — для вывода данных.

Символьные потоки данных
Reader и Writer являются абстрактными классами для работы с символьными данными (текстовыми потоками).

Абстрактные классы Writer и Reader:
Writer
Представляет поток для записи символов.
void write(char[] cbuf): записывает массив символов.
void write(String str): записывает строку.

Reader
Представляет поток для чтения символов.
int read(): читает один символ.
int read(char[] cbuf): читает массив символов.

Антон

29. Чтение и запись файлов. Классы FileInputStream, FileOutputStream. Файловый ввод-вывод с использованием символьных потоков. Классы FileReader и FileWriter.

Ответ:

Байтовые потоки (Классы FileInputStream, FileOutputStream)

Класс FileInputStream предназначен для **считывания данных из файла** (байт за байтом).
FileInputStream является наследником класса InputStream

Для создания экземпляра класса нужно передать строковую переменную или файловый объект, представляющий путь к файлу:

```
FileInputStream(String fileName) throws FileNotFoundException
```

Если файл не может быть открыт, то генерируется исключение **FileNotFoundException**

Пример использования:

```java
import java.io.*;

public class Program {

    public static void main(String[] args) {

        try(FileInputStream fin=new FileInputStream("notes.txt"))
        {
            int i;
            while((i=fin.read())!=-1){

                System.out.print((char)i);
            }
        }
        catch(IOException ex){

            System.out.println(ex.getMessage());
        }
    }
}
```

Класс FileOutputStream предназначен **для записи байтов в файл**. Он является производным от класса OutputStream

Через конструктор класса FileOutputStream задается файл, в который производится запись. Класс поддерживает несколько конструкторов:

```
1 FileOutputStream(String filePath)
2 FileOutputStream(File fileObj)
3 FileOutputStream(String filePath, boolean append)
4 FileOutputStream(File fileObj, boolean append)
```

Файл задается либо через строковый путь, либо через объект File. Второй параметр - append задает способ записи:

true - дозаписываются в конец файла

false - файл полностью перезаписывается.

Пример использования:

```java
import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutputStreamExample {
    public static void main(String[] args) {
        String data = "Hello, World!";
        try (FileOutputStream fos = new FileOutputStream("example.txt")) {
            fos.write(data.getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Символьные потоки (Классы FileReader и FileWriter)

Класс FileWriter является производным от класса Writer. Он используется **для записи текстовых файлов**.

Чтобы создать объект FileWriter, можно использовать один из следующих конструкторов:

```
1 FileWriter(File file)
2 FileWriter(File file, boolean append)
3 FileWriter(FileDescriptor fd)
4 FileWriter(String fileName)
```

```
5 FileWriter(String fileName, boolean append)
```

В конструктор передается либо путь к файлу в виде строки, либо объект File, который ссылается на конкретный текстовый файл. Параметр append:

true - дозаписываются в конец файла

false - файл полностью перезаписывается.

Пример использования:

```java
import java.io.FileWriter;
import java.io.IOException;

public class FileWriterExample {
    public static void main(String[] args) {
        String data = "Hello, World!";
        try (FileWriter fw = new FileWriter("example.txt")) {
            fw.write(data);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Класс FileReader наследуется от абстрактного класса Reader и предоставляет функциональность **для чтения текстовых файлов**.

Для создания объекта FileReader мы можем использовать один из его конструкторов:

```
1 FileReader(String fileName)

2 FileReader(File file)

3 FileReader(FileDescriptor fd)
```

Пример использования:

```java
import java.io.FileReader;
import java.io.IOException;

public class FileReaderExample {
    public static void main(String[] args) {
        try (FileReader fr = new FileReader("example.txt")) {
            int content;
            while ((content = fr.read()) != -1) {
                System.out.print((char) content);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Основные различия

Байтовые потоки (FileInputStream, FileOutputStream) работают с байтами и подходят для чтения и записи бинарных данных, таких как изображения, аудиофайлы и т.д.

Символьные потоки (FileReader, FileWriter) работают с символами и подходят для чтения и записи текстовых данных. Они автоматически обрабатывают кодировку символов, что делает их более удобными для работы с текстом.

30. Многопоточное программирование: общие принципы.

Ответ:
Многопоточное программирование — это модель выполнения программ, при которой несколько потоков (threads) выполняются параллельно в рамках одного процесса. Это позволяет эффективно использовать ресурсы многоядерных процессоров и улучшать производительность приложений.
Общие принципы многопоточного программирования в Java:

1. **Поток (Thread)** — наименьшая единица выполнения задачи внутри программы. Создавать потоки и управлять ими можно с помощью класса java.lang.Thread.
2. **Синхронизация** - это концепция многопоточного программирования, обеспечивающая координацию и контроль доступа к общим ресурсам между потоками. Она гарантирует, что только один поток может одновременно получить доступ к критическому разделу кода или общему ресурсу. В Java для создания синхронизированных методов и блоков используется ключевое слово **`synchronized`**.
3. **Пулы потоков (Thread Pools)**. Пулы потоков используются для эффективного управления потоками и их повторного использования. Пул потоков — это набор предварительно инициализированных рабочих потоков, готовых одновременно выполнять задачи. Это устраняет накладные расходы на создание и удаление потоков для каждой задачи.

В основу системы многопоточной обработки в Java положены класс Thread и интерфейс Runnable, входящие в пакет java.lang

31. Класс Thread и интерфейс Runnable: создание потоков, приоритеты потоков.

Ответ:
В Java для создания и управления потоками используются класс **`Thread`** и интерфейс **`Runnable`**.

Класс `Thread`

Класс **`Thread`** предоставляет методы для создания и управления потоками. Потоки можно создавать, наследуя этот класс и переопределяя его метод **`run`**`()`.

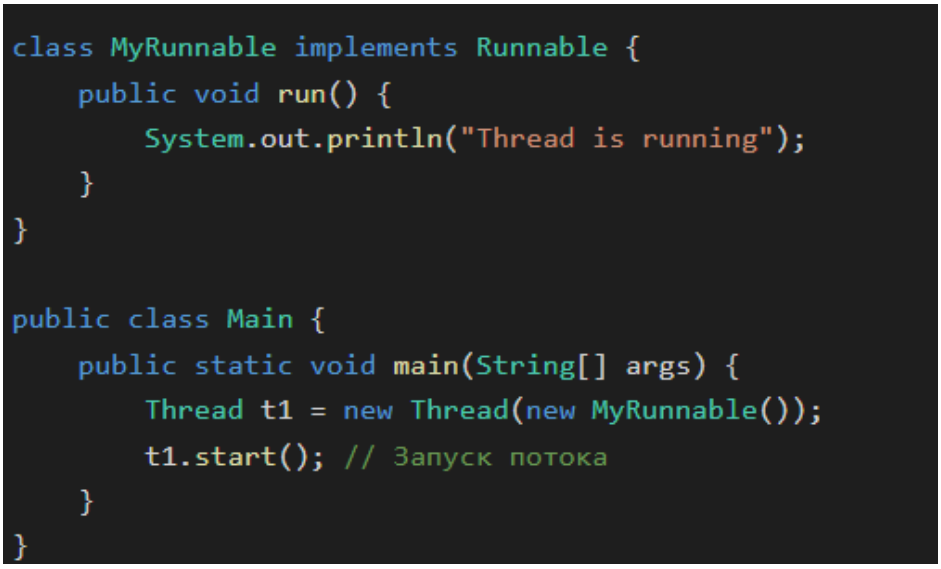
Пример создания потока через наследование `Thread`

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Запуск потока
    }
}
```

Интерфейс `Runnable`

Интерфейс **`Runnable`** предоставляет метод **`run()`**, который должен быть реализован классом, желающим выполняться в отдельном потоке. Это более гибкий способ создания потоков, так как он позволяет наследовать другие классы.

Пример создания потока через реализацию `Runnable`



```java
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start(); // Запуск потока
    }
}
```

Приоритеты потоков

Приоритеты потоков используются для контроля важности и очерёдности работы разных потоков. **Они задаются числами от 1 до 10**:

1. **MIN_PRIORITY (1)** — минимальный приоритет;
2. **NORM_PRIORITY (5)** — нормальный приоритет, который используется по умолчанию;
3. **MAX_PRIORITY (10)** — максимальный приоритет.

Приоритет потока можно установить с помощью метода setPriority(int priority). Этот метод должен быть вызван после создания потока, но до его запуска методом start().

32. Приостановка и прерывание потоков. Определение момента завершения потока.

Ответ:

Приостановка потоков

Приостановка потока позволяет временно остановить его выполнение. Это может быть полезно для синхронизации потоков или для выполнения задач через определенные интервалы времени.

Метод `sleep()`

Метод `sleep()` приостанавливает выполнение потока на указанное количество миллисекунд.

```
Thread.sleep(1000); // Приостановка на 1 секунду
```

Прерывание потоков
Прерывание потока позволяет завершить его выполнение досрочно. Это может быть полезно для отмены длительных операций или для управления ресурсами.

Метод `interrupt()`

Метод `interrupt()` устанавливает флаг прерывания для потока. Поток может проверять этот флаг и завершать свою работу.

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt(); // Восстановление флага прерывания
    System.out.println("Thread was interrupted");
}
```

```
try {
    Thread.sleep(3000); // Даем потоку время выполниться
} catch (InterruptedException e) {
    e.printStackTrace();
}

t1.interrupt(); // Прерывание потока
```

Определение момента завершения потока
Для определения момента завершения потока можно использовать метод `join()` или метод `isAlive()`.

Метод `join()`

Метод `join()` позволяет одному потоку дождаться завершения другого потока.

```java
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("Thread is running: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        t1.start();
        t1.join(); // Ожидание завершения потока t1
        System.out.println("Thread t1 has finished");
    }
}
```

Метод `isAlive()`

Метод `isAlive()` возвращает `true`, если поток все еще выполняется, и `false`, если поток завершил свою работу.

33. Синхронизация потоков.

Ответ:

Синхронизация - это концепция многопоточного программирования, обеспечивающая координацию и контроль доступа к общим ресурсам между потоками. Она гарантирует, что только один поток может одновременно получить доступ к критическому разделу кода или общему ресурсу. В Java для создания синхронизированных методов и блоков используется ключевое слово **`synchronized, a` также следующие методы:** `wait()`, `notify()` **и** `notifyAll()`.

Методы wait(), notify() и notifyAll()

Методы `wait()`, `notify()`, и `notifyAll()` — это инструменты для координации работы между потоками, которые используют общие ресурсы. Они вызываются на объекте класса, который реализует интерфейс `Object`,

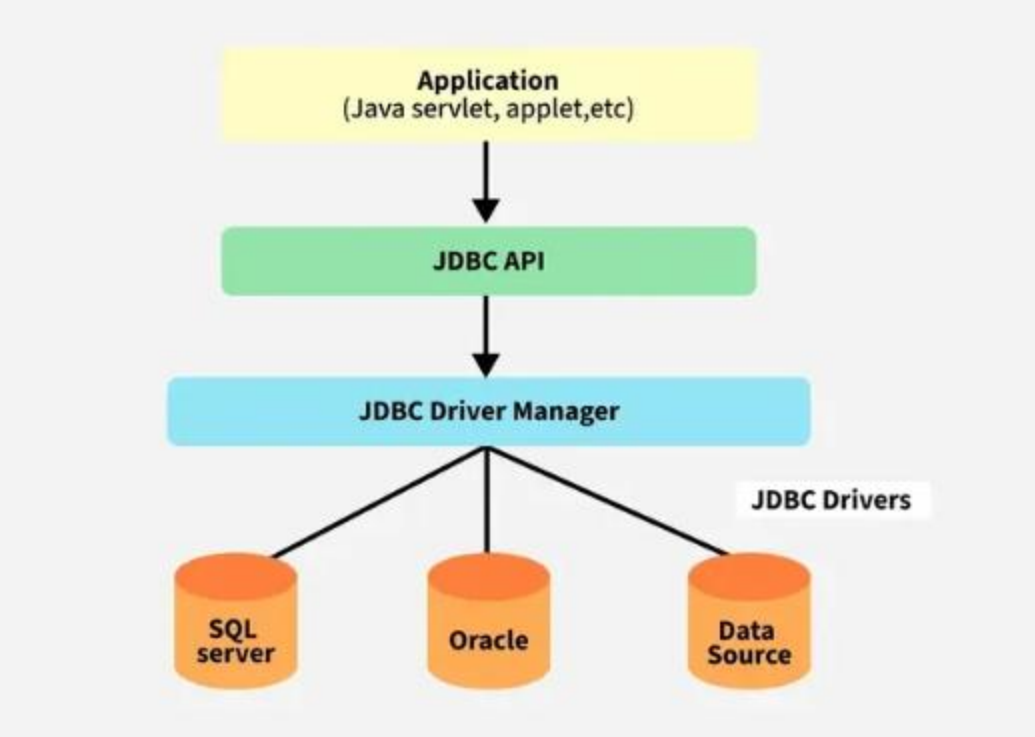
и должны использоваться только в синхронизированных блоках или методах.

- `wait()` заставляет текущий поток ожидать до тех пор, пока другой поток не вызовет `notify()` или `notifyAll()` на том же объекте.
- `notify()` пробуждает один случайно выбранный поток, который ожидает на этом объекте.
- `notifyAll()` пробуждает все потоки, которые ожидают на этом объекте.

34. Архитектура JDBC (Java DataBase Connectivity). Двух и трехуровневые модели доступа к базе данных. Преимущества и недостатки JDBC.

Ответ:
JDBC (Java Database Connectivity) — это стандартный API для взаимодействия с базами данных в Java. Он предоставляет механизмы для выполнения SQL-запросов, обновления данных и управления транзакциями. Рассмотрим архитектуру JDBC, двух- и трехуровневые модели доступа к базе данных, а также преимущества и недостатки JDBC.

Архитектура JDBC



Объяснение:

- **Application:** Это Java-апплет или сервлет, который взаимодействует с источником данных.
- **API JDBC:** Он позволяет Java-программам выполнять SQL-запросы и извлекать результаты. Ключевые интерфейсы включают Driver, ResultSet, RowSet, PreparedStatement и Connection. Важные классы включают DriverManager, Types, Blob и Clob.
- **DriverManager:** Он играет важную роль в архитектуре JDBC. Он использует некоторые драйверы, зависящие от базы данных, для эффективного подключения корпоративных приложений к базам данных.
- **Драйверы JDBC:** Эти драйверы управляют взаимодействием между приложением и базой данных.

Архитектура JDBC состоит из двухуровневых и трехуровневых моделей обработки для доступа к базе данных.

1. Двухуровневая архитектура

Приложение Java взаимодействует напрямую с базой данных с помощью драйвера JDBC. Запросы отправляются в базу данных, а результаты возвращаются непосредственно приложению. При настройке клиент / сервер компьютер пользователя (клиент) взаимодействует с удаленным сервером базы данных.

Структура:

Client Application (Java) -> JDBC Driver -> Database

2. Трехуровневая архитектура

В этом случае запросы пользователя отправляются в службы среднего уровня, которые взаимодействуют с базой данных. Результаты базы данных обрабатываются средним уровнем и затем отправляются обратно пользователю.

Структура:

Client Application -> Application Server -> JDBC Driver -> Database

Преимущества JDBC (Java Database Connectivity):

1. Простота и лёгкость в использовании.
2. Поддержка различных баз данных.
3. Предоставление набора интерфейсов для выполнения операций с базой данных.

Недостатки JDBC:

1. Необходимость написания большого количества кода для выполнения простых операций.
2. Отсутствие проверки типов на этапе компиляции.
3. Отсутствие ORM (Object-Relational Mapping) возможностей.

35. Библиотека Swing, общие черты и особенности.

Ответ:
Библиотека Swing — это мощная библиотека для создания графических пользовательских интерфейсов (GUI) в Java. Она предоставляет богатый набор компонентов, которые позволяют разработчикам создавать кросс-платформенные приложения с привлекательными и удобными интерфейсами.

Общие черты Swing

1. **Кроссплатформенность:** Swing написан полностью на Java, что обеспечивает его кроссплатформенность. Приложения, созданные с использованием Swing, могут работать на любой платформе, поддерживающей Java, без необходимости изменения кода.
2. **Богатый набор компонентов:** Swing предоставляет широкий спектр компонентов, таких как кнопки, текстовые поля, таблицы, деревья, панели и многое другое. Эти компоненты можно легко настраивать и комбинировать для создания сложных интерфейсов.
3. **Модель событий:** Swing использует модель событий для обработки пользовательских действий, таких как нажатия кнопок, ввод текста и т.д. Это позволяет разработчикам легко реагировать на действия пользователя.
4. **Поддержка MVC (Model-View-Controller):** Swing следует архитектурному паттерну MVC, что позволяет отделить логику приложения от его представления и управления. Это упрощает разработку и поддержку сложных приложений.
5. **Настраиваемость:** Компоненты Swing можно настраивать с помощью различных свойств и стилей, что позволяет создавать уникальные и привлекательные интерфейсы.

Особенности библиотеки Swing:

1. **Легковесные компоненты:** В отличие от AWT (Abstract Window Toolkit), компоненты Swing являются "легковесными", то есть они написаны полностью на Java и не зависят от нативных компонентов операционной системы. Это обеспечивает единообразный внешний вид и поведение на всех платформах.
2. **Look and Feel:** Swing поддерживает различные "Look and Feel" (внешний вид и ощущения), что позволяет изменять внешний вид приложения в соответствии с платформой или предпочтениями пользователя. Например, можно использовать внешний вид, имитирующий интерфейсы Windows, macOS или Motif.
3. **Поддержка плагинов:** Swing позволяет создавать плагины и расширять функциональность существующих компонентов, что делает его гибким и расширяемым.
4. **Документированность и сообщество:** Swing хорошо документирован, и существует большое сообщество разработчиков, что облегчает поиск решений и обмен опытом.

36. Виды контейнеров в Swing.

В Swing определены **два вида контейнеров**:

1. **Контейнеры верхнего уровня**. К ним относятся JFrame, JApplet, JWindow и JDialog. Эти классы не являются производными от класса JComponent, но являются производными от классов Component и Container. Контейнеры верхнего уровня должны находиться на вершине иерархии контейнеров и не могут содержаться в других контейнерах.
2. **Облегченные контейнеры**. Являются производными от класса JComponent. В качестве примера легковесных контейнеров можно привести классы JPanel, JScrollPane и JRootPane. Легковесные контейнеры могут содержаться в других контейнерах, и поэтому они нередко используются для объединения группы взаимосвязанных компонентов.

Саша

37. Элементы пользовательского интерфейса Swing.

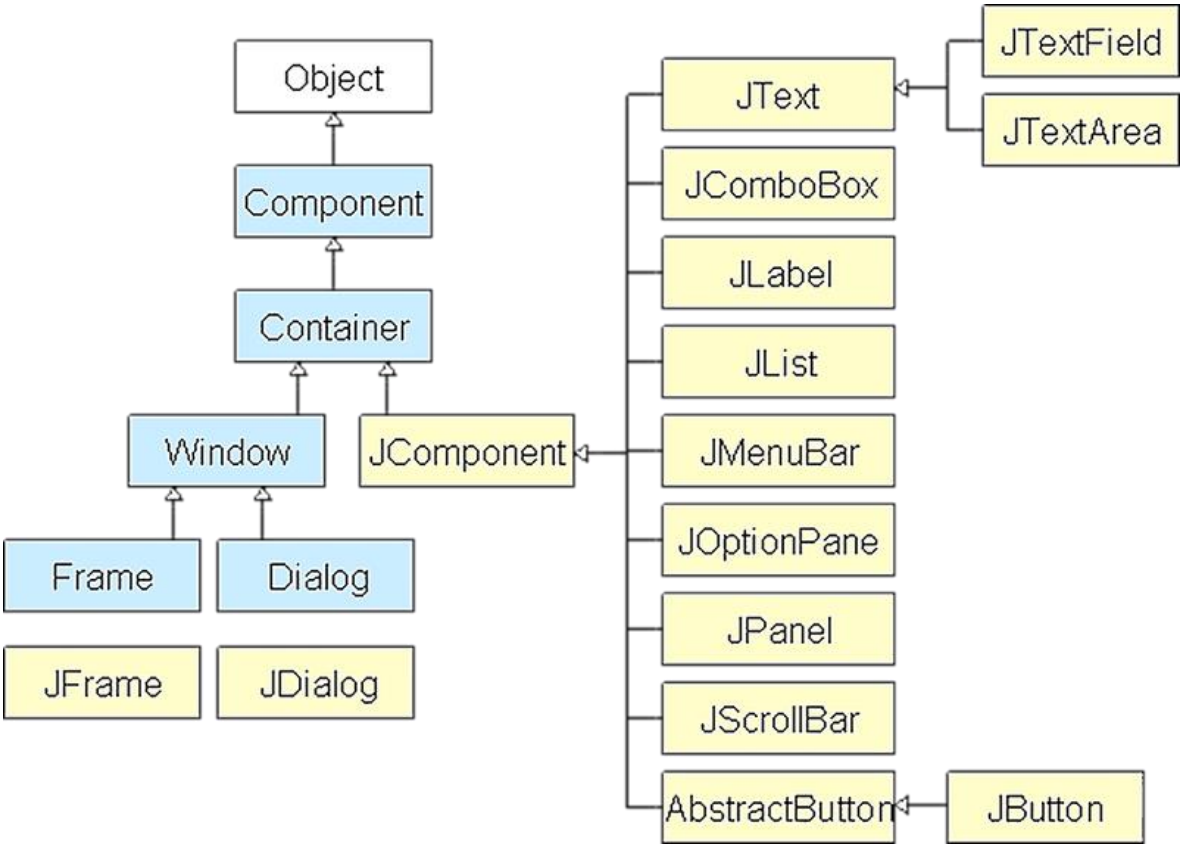
Swing в Java — это инструментарий графического интерфейса пользователя (GUI), включающий компоненты GUI. Она предоставляет множество компонентов и контейнеров, которые позволяют разработчикам создавать интерактивные приложения.

Swing является частью **Java Foundation Classes** (JFC). **JFC** представляет собой API для программирования GUI на Java, обеспечивающий графический интерфейс пользователя.

Библиотека **Java Swing** построена поверх **Java Abstract Widget Toolkit** (AWT). Можно использовать простые компоненты программирования графического интерфейса Java:

- кнопки;
- метки;
- выпадающие списки;
- текстовые поля и т.д.,

из библиотеки, без необходимости создания данных компонентов с нуля.



Для создания графического интерфейса необходим хотя бы один объект-контейнер. Классы-контейнеры – это классы, на которых могут располагаться другие компоненты.

Существует три типа контейнеров Java Swing.

1. **Панель**: Это чистый контейнер, который сам по себе не является окном. Единственная цель панели — разместить компоненты на окне.
2. **Фрейм (Frame)**: Это полностью функционирующее окно с заголовком и значками.
3. **Диалог**: Его можно представить как всплывающее окно, которое появляется, когда необходимо отобразить сообщение. Это не полностью функционирующее окно, как фрейм.

38. Модель событий Swing. Интерфейс EventListener.

Модель событий, поддерживаемые компонентами Swing
Вы можете определить, какие типы событий может запускать компонент, посмотрев на типы прослушивателей событий, которые вы можете зарегистрировать в нем. Например, класс JComboBox определяет эти методы регистрации прослушивателей:

- addActionListener
- addItemListener
- addPopupMenuListener

Таким образом, выпадающий список поддерживает прослушиватели действий, элементов и контекстного меню в дополнение к методам прослушивания, которые он наследует от JComponent.

Прослушиватели, поддерживаемые компонентами Swing, делятся на две категории:

- Прослушиватели, поддерживаемые всеми компонентами Swing
- Другие прослушиватели, поддерживаемые компонентами Swing

Интерфейс EventListener является основным методом обработки событий. Пользователи реализуют интерфейс EventListener и регистрируют свой прослушиватель в целевом объекте события с помощью метода addEventListener. Пользователи также должны удалить свой прослушиватель событий из целевого объекта события после завершения использования прослушивателя.

Когда узел копируется с использованием метода cloneNode, прослушиватели событий, подключенные к исходному узлу, не подключаются к скопированному узлу. Если пользователь хочет, чтобы те же самые прослушиватели событий были добавлены во вновь созданную копию, пользователь должен добавить их вручную.

Модификатор и тип:void
Метод: handleEvent(Event evt)
Описание: Этот метод вызывается всякий раз, когда происходит событие того типа, для которого был зарегистрирован интерфейс EventListener.

39. Менеджеры компоновки Swing.

Менеджер расположения (layout manager) определяет, каким образом на форме будут располагаться компоненты. Независимо от платформы, виртуальной машины, разрешения и размеров экрана менеджер расположения гарантирует, что компоненты будут иметь предпочтительный или близкий к нему размер и располагаться в том порядке, который был указан программистом при создании программы.

В **Swing** менеджер расположения играет еще большую роль, чем обычно. Он позволяет не только сгладить различия между операционными системами, но к тому же дает возможность с легкостью менять внешний вид приложения, не заботясь о том, как при этом изменяются размеры и расположение компонентов.

Поддержка менеджеров расположения встроена в базовый класс контейнеров java.awt.Container. Все компоненты библиотеки Swing унаследованы от базового класса JComponent, который, в свою очередь, унаследован от класса Container. Таким образом, для любого компонента Swing можно установить менеджер расположения или узнать, какой менеджер им используется в данный момент. Для этого предназначены методы **setLayout()** и **getLayout()**.

Конечно, изменять расположение желательно только в контейнерах, которые предназначены для размещения в них компонентов пользовательского интерфейса, то есть в панелях (JPanel) и окнах (унаследованных от класса Window). Не стоит менять расположение в кнопках или флажках, хотя такая возможность имеется. В стандартной библиотеке Java существует несколько готовых менеджеров расположения, и с их помощью можно реализовать абсолютно любое расположение.

Выполнить проверку корректности для любого компонента **Swing**, будь это контейнер или отдельный компонент, позволяет метод **revalidate()**, определенный в базовом классе библиотеки JComponent.

Менеджер расположения размещает добавляемые в контейнер компоненты в некотором порядке согласно реализованного в нем алгоритма, и определяет их некоторый размер. При этом он обычно учитывает определенные свойства компонентов :

- **Предпочтительный размер.** Размер, который идеально подходит компоненту. По умолчанию все размеры компонентов устанавливаются текущим менеджером внешнего вида и поведения (look and feel manager), но можно их изменить. Предпочтительный размер можно изменить с помощью метода setPrefferedSize().
- **Минимальный размер.** Параметр определения минимального размера компонента. После достижения минимального размера всех компонентов контейнер больше не сможет уменьшить свой размер. Минимальный размер также можно изменить, для этого предназначен метод setMinimumSize().
- **Максимальный размер.** Параметр определения максимального размера компонента при увеличении размеров контейнера. Например, максимальный размер текстового поля JTextField не ограничен. Чаще всего оно должно сохранять свой предпочтительный размер. Это можно исправить с помощью метода setMaximumSize(), устанавливающего максимальный размер. Большая часть менеджеров расположения игнорирует максимальный размер, работая в основном с предпочтительным и минимальным размерами.
- **Выравнивание по осям X и Y**. Данные параметры нужны только менеджеру BoxLayout, причем для него они играют важнейшую роль.
- **Границы контейнера.** Эти параметры контейнера определяют размеры отступов от границ контейнера. Значения границ контейнера позволяет получить метод getInsets(). Иногда менеджеру расположения приходится их учитывать.

40. GUI Designer Swing

Плагин UI Designer в IntelliJ IDEA позволяет создавать графические пользовательские интерфейсы (GUI) для ваших приложений с помощью компонентов библиотеки Swing. С помощью UI Designer вы можете быстро создавать диалоговые окна и группы элементов управления для использования в контейнерах верхнего уровня, таких как JFrame. Эти элементы могут сосуществовать с компонентами, которые вы определяете непосредственно в коде Java.

JFormDesigner

Описание: JFormDesigner — это профессиональный GUI-дизайнер для Java Swing, который поддерживает различные макеты, такие как MigLayout и GroupLayout.

Java Swing Designer

Описание: Java Swing Designer — это мощный инструмент для создания современных графических интерфейсов с использованием фреймворка Java Swing. Он предлагает интуитивно понятный интерфейс, позволяющий разработчикам визуально проектировать компоненты GUI, настраивать их свойства и располагать их на экране.

Функции:

- Визуальный дизайн интерфейса с возможностью перетаскивания компонентов.

- Генерация соответствующего Java-кода автоматически.
- Поддержка сложных макетов и управления свойствами компонентов

41. Текстовые поля в Swing

Текстовое поле — это базовый элемент управления текстом, который позволяет пользователю вводить небольшой объём текста. Когда пользователь указывает, что ввод текста завершён (обычно нажатием клавиши Enter), текстовое поле запускает событие действия. Если вам нужно получить от пользователя более одной строки ввода, используйте текстовую область.

API Swing предоставляет несколько классов для компонентов, которые являются разновидностями текстовых полей или включают в себя текстовые поля.

JTextField	Что рассматривается в этом разделе: основные текстовые поля.
JFormattedTextField	Подкласс JTextField позволяет указать допустимый набор символов, которые может вводить пользователь.
JPasswordField	Подкласс JTextField, который не отображает вводимые пользователем символы.
JComboBox	Можно редактировать и предоставляет меню строк для выбора.
JSpinner	Сочетает в себе отформатированное текстовое поле с парой небольших кнопок, которые позволяют пользователю выбрать предыдущее или следующее доступное значение.

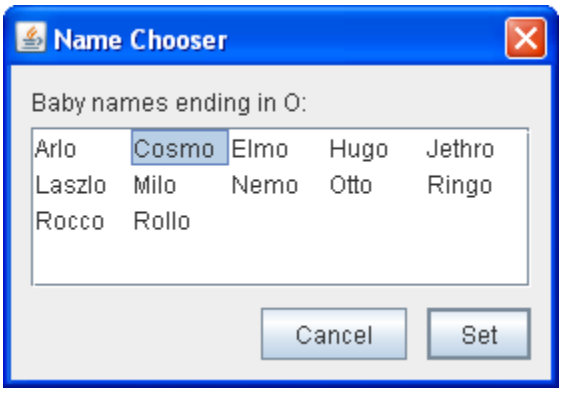
В следующем примере показано базовое текстовое поле и текстовая область. Текстовое поле можно редактировать. Текстовую область нельзя редактировать. Когда пользователь нажимает Enter в текстовом поле, программа копирует содержимое текстового поля в текстовую область, а затем выделяет весь текст в текстовом поле.

42. Компонент управления JButton в Swing

https://docs.oracle.com/javase/tutorial/uiswing/components/button.html#jbutton
JButton, JGroupButton, Action

JButton в Swing — это **компонент управления для создания кнопок**.
Обычные кнопки — объекты JButton — обладают чуть большей функциональностью, чем класс AbstractButton: вы можете использовать JButton в качестве кнопки по умолчанию.

Кнопкой по умолчанию может быть не более одной кнопки в контейнере верхнего уровня. Кнопка по умолчанию, как правило, подсвечивается и активируется при нажатии всякий раз, когда контейнер верхнего уровня находится в фокусе клавиатуры и пользователь нажимает клавишу Return или Enter. Вот изображение диалогового окна, реализованного в примере ListDialog, в котором кнопка Set является кнопкой по умолчанию:



В Java Look & Feel кнопка по умолчанию имеет плотную рамку
Вы устанавливаете кнопку по умолчанию, вызывая метод setDefaultButton на корневой панели контейнера верхнего уровня. Вот код, который устанавливает кнопку по умолчанию для примера ListDialog:

```
//В конструкторе подкласса JDialog.:
getRootPane().setDefaultButton(Установить кнопку);
```

Точная реализация функции кнопки по умолчанию зависит от внешнего вида. Например, во внешнем виде Windows кнопка по умолчанию используется в зависимости от того, какая кнопка имеет фокус, так что нажатие клавиши Enter приводит к нажатию сфокусированной кнопки. Если ни одна кнопка не имеет фокуса, кнопка, которую вы изначально указали в качестве кнопки по умолчанию, снова становится кнопкой по умолчанию.

еще нашел вот что, касаемо данного материала:

Кнопки **JButton** кроме собственного внешнего вида не включают практически ничего уникального. Поэтому всё, что верно для кнопок, будет верно и для остальных элементов управления. Пример кода создания обычной кнопки :

```
JButton button = new JButton("Кнопка");
button.addActionListener(new ButtonAction());
```

Основное время работы с кнопками связано не столько с их созданием и настройкой, сколько с размещением в контейнере и написанием обработчиков событий.

Внешний вид кнопок **JButton** можно легко изменить, не меняя менеджера внешнего вида и поведения. С интерфейсом кнопок можно делать практически все — сопоставлять каждому действию пользователя своё изображение, убирать рамку, закрашивать в любой цвет, перемещать содержимое по разным углам, не рисовать фокус.

43. Платформа JavaFX, особенности, компоненты

https://javarush.com/groups/posts/2560-vvedenie-v-java-fx

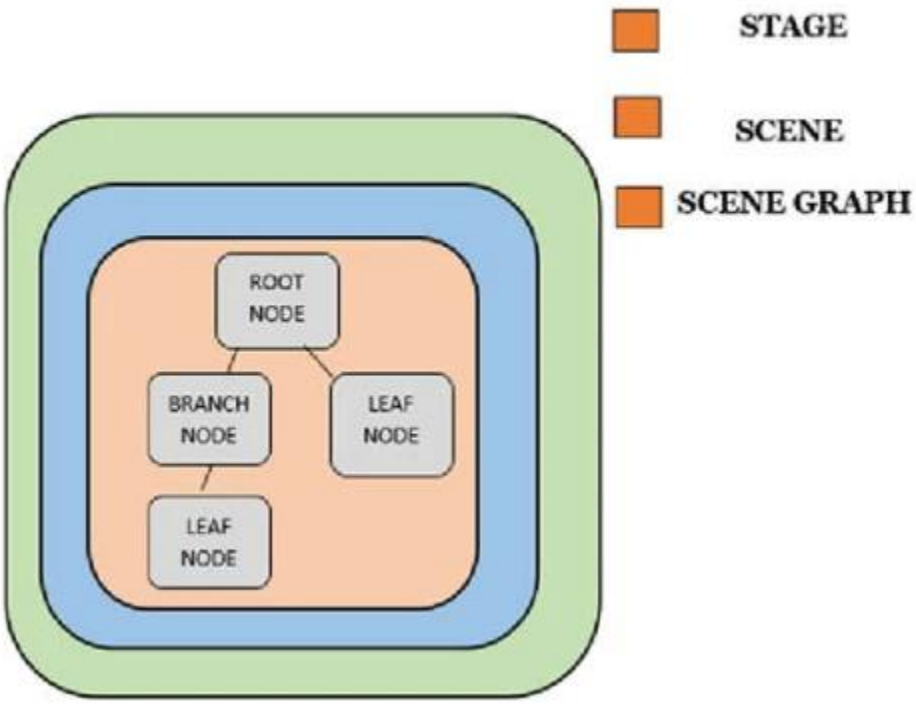
JavaFX — это опенсорсная платформа, содержащая пакеты с графическими и медиа-инструментами для разработки и развёртывания клиентских приложений с насыщенным графическим пользовательским интерфейсом (GUI)

Особенности JavaFX:

- JavaFX изначально поставляется с большим набором частей графического интерфейса, таких как всякие там кнопки, текстовые поля, таблицы, деревья, меню, диаграммы и т.д., что в свою очередь сэкономит нам вагон времени.
- JavaFX часто юзает стили CSS, и мы сможем использовать специальный формат FXML для создания GUI, а не делать это в коде Java. Это облегчает быстрое размещение графического интерфейса пользователя или изменение внешнего вида или композиции без необходимости долго играться в коде Java.
- JavaFX имеет готовые к использованию части диаграммы, поэтому нам не нужно писать их с нуля в любое время, когда вам нужна базовая диаграмма.
- JavaFX дополнительно поставляется с поддержкой 3D графики, которая часто полезна, если мы разрабатываем какую-то игру или подобные приложения.

основные составляющие окна (компоненты):

- Stage — по сути это окружающее окно, которое используется как начальное полотно и содержит в себе остальные компоненты. У приложения может быть несколько stage, но один такой компонент должен быть в любом случае. По сути Stage является основным контейнером и точкой входа.
- Scene — отображает содержание stage (прям матрёшка). Каждый stage может содержать несколько компонентов — scene, которые можно между собой переключать. Внутри это реализуется графом объектов, который называется — Scene Graph (где каждый элемент — узел, ещё называемый как Node).
- Node — это элементы управления, например, кнопки метки, или даже макеты (layout), внутри которых может быть несколько вложенных компонентов. У каждой сцены (scene) может быть один вложенный узел (node), но это может быть макет (layout) с несколькими компонентами. Вложенность может быть многоуровневой, когда макеты содержат другие макеты и обычные компоненты. У каждого такого узла есть свой идентификатор, стиль, эффекты, состояние, обработчики событий.



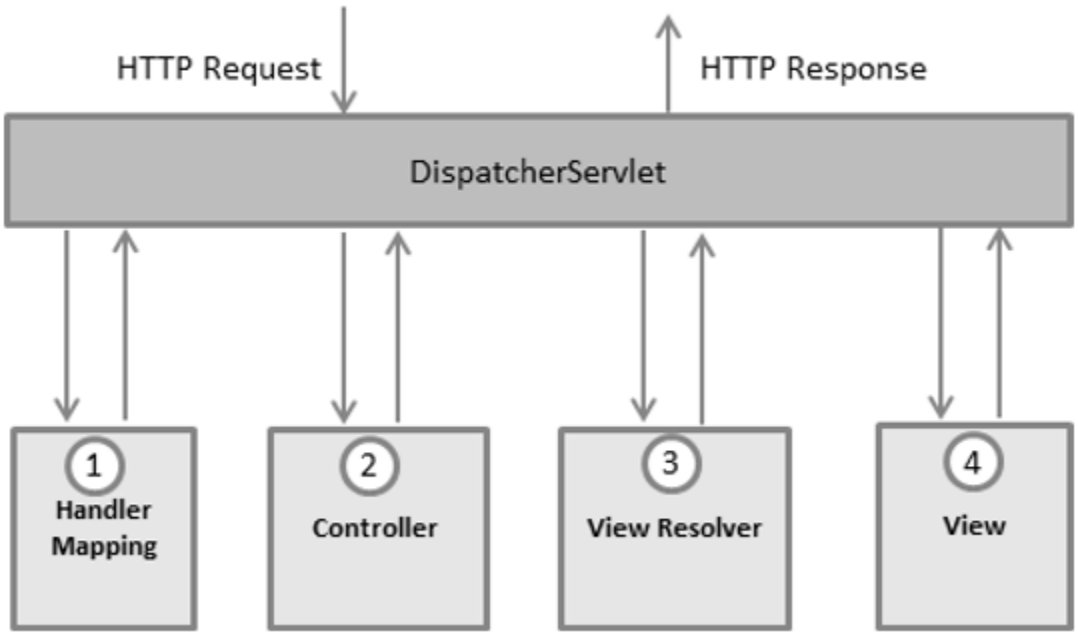
44. Шаблон MVC (Model-View-Controller) в Spring.

Фреймворк Spring MVC обеспечивает архитектуру паттерна Model — View — Controller (Модель — Отображение (далее — Вид) — Контроллер) при помощи слабо связанных готовых компонентов. Паттерн MVC разделяет аспекты приложения (логику ввода, бизнес-логику и логику UI), обеспечивая при этом свободную связь между ними.

- Model (Модель) инкапсулирует (объединяет) данные приложения, в целом они будут состоять из POJO («Старых добрых Java-объектов», или бинов).
- View (Отображение, Вид) отвечает за отображение данных Модели, — как правило, генерируя HTML, которые мы видим в своём браузере.
- Controller (Контроллер) обрабатывает запрос пользователя, создаёт соответствующую Модель и передаёт её для отображения в Вид.

DispatcherServlet

Вся логика работы Spring MVC построена вокруг DispatcherServlet, который принимает и обрабатывает все HTTP-запросы (из UI) и ответы на них. Рабочий процесс обработки запроса DispatcherServlet'ом проиллюстрирован на следующей диаграмме:



ещё такой вариант для запоминания:
Spring Модель-Представление-Контроллер

- **Модель** – Модель содержит данные приложения. Набор данных может быть отдельным объектом или группой вещей.

- **Контроллер** – Контроллер содержит бизнес-логику приложения. Аннотация @Controller используется здесь для идентификации класса как контроллера.
- **View** – View – это представление доставленной информации в определенном формате. В большинстве случаев для построения страницы представления используется JSP+JSTL. Однако Spring поддерживает дополнительные технологии представления, такие как Apache Velocity, Thymeleaf и FreeMarker.
- **Front Controller** – Класс DispatcherServlet служит в качестве front controller в Spring Web MVC. Он отвечает за управление потоком приложения Spring MVC.

https://www.geeksforgeeks.org/spring-mvc/

Лера

45. Классы StringBuffer и StringBuilder.

StringBuffer и StringBuilder — классы для работы с изменяемыми строками, которые позволяют выполнять операции без создания новых объектов (в отличие от String).

Общие черты:

- Оба класса изменяемы, то есть поддерживают операции вставки, удаления, замены символов.
- Методы:
    - append(String s): добавляет строку к текущему объекту.
    - insert(int offset, String s): вставляет строку в указанную позицию.
    - delete(int start, int end): удаляет подстроку.
    - reverse(): разворачивает строку.

Отличия:

- StringBuffer:
    - потокобезопасен, синхронизирован.
    - Подходит для многопоточных приложений.
    - Медленнее, из-за накладных расходов на синхронизацию.

    - Использовать, если строки будут изменяться в многопоточной среде.

- StringBuilder:
    - не потокобезопасен
    - быстрее в однопоточных задачах.
    - Использовать, если приложение работает в однопоточном режиме и важна высокая производительность.

46. Архитектура JDBC (Java DataBase Connectivity). Двух и трехуровневые модели доступа к базе данных. Преимущества и недостатки JDBC.

Архитектура JDBC:

- JDBC API: предоставляет интерфейсы и классы для работы с БД (Connection, Statement, ResultSet и др.).
- JDBC Driver: драйвер, обеспечивающий связь с конкретной СУБД.

Этапы подключения к базе данных

1. Установка базы данных на сервер или выбор облачного сервиса, к которому нужно получить доступ.
2. Подключение библиотеки JDBC.
3. Проверка факта нахождения необходимого драйвера JDBC в classpath.
4. Установление соединения с базой данных с помощью библиотеки JDBC.
5. Использование установленного соединения для выполнения команд SQL.
6. Закрытие соединения после окончания сеанса.

Преимущества и недостатки JDBC:

- Преимущества:
    - Независимость от платформы и СУБД.
    - Широкий набор возможностей для работы с БД.
- Недостатки:
    - Сложность ручного управления ресурсами.
    - Меньшая гибкость по сравнению с ORM (например, Hibernate).

47. Массивы Java: объявление, инициализация. Основные методы класса Arrays. Доступ к элементам массивов, итерация массивов. Двумерные массивы.

Массив — это структура данных, в которой хранятся элементы одного типа. Его можно представить, как набор пронумерованных ячеек, в каждую из которых можно поместить какие-то данные (один элемент данных в одну ячейку). Доступ к конкретной ячейке осуществляется через её номер. Номер элемента в массиве также называют индексом. Во всех ячейках будут храниться элементы одного типа.

- Одномерные массивы:
    - int[] array = new int[5]; // Объявление и выделение памяти

    int[] array2 = {1, 2, 3, 4, 5}; // Инициализация

- Двумерные массивы:
    - int[][] matrix = new int[3][3]; // 3 строки, 3 столбца

    int[][] matrix2 = {{1, 2}, {3, 4}, {5, 6}}; // Инициализация

Основные методы класса Arrays:

- sort(array): сортировка массива.
- binarySearch(array, key): поиск элемента (требуется отсортированный массив).
- fill(array, value): заполнение массива значением.
- toString(array): преобразование массива в строку.

Итерация по массивам:

- Через цикл for:
    - for (int i = 0; i < array.length; i++) {

    System.out.println(array[i]);

    }

- Через цикл for-each:
    - for (int num : array) {

    System.out.println(num);

}

48. Иерархия наследования Java. Преобразование типов при наследовании. Ключевое слово instanceof.

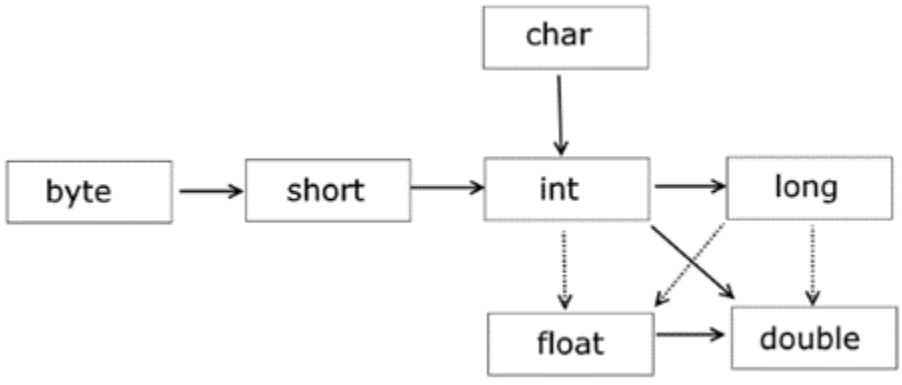
- один класс может служить родительским классом или базой для нескольких дочерних классов. каждый из них может иметь свои дочерние классы.
- В Java все классы наследуются от Object.
- Ключевое слово extends используется для наследования.
- Дочерний класс наследует поля и методы родительского класса.
- Java не поддерживает множественное наследование классов. Тем не менее множественное наследование можно реализовать с помощью интерфейсов

Преобразование типов:

Каждый базовый тип данных занимает определенное количество байт памяти. Это накладывает ограничение на операции, в которые вовлечены различные типы данных.

```
int a = 4;

byte b = a; // ! Ошибка
```

Автоматическое преобразование:



Пунктирными стрелками показаны автоматические преобразования с потерей точности.

Автоматически без каких-либо проблем производятся расширяющие преобразования (widening) - они расширяют представление объекта в памяти.

```
byte b = 7;
```

```
int d = b;  // преобразование от byte к int
```

Расширяющие автоматические преобразования представлены следующими цепочками:

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

Явное преобразование:

```
long a = 4;

int b = (int) a;
```

Ключевое слово instanceof:

Оператор *instanceof* возвращает *true* или *false* в зависимости от того принадлежит ли экземпляр проверяемому классу. работает также с наследованием.

```
if (a instanceof Dog) {
    System.out.println("a is a Dog");
}
```

Другими словами, оператор instanceof вернет значение true, если:

1) переменная a хранит ссылку на объект типа B

2) переменная a хранит ссылку на объект, класс которого унаследован от B

3) переменная a хранит ссылку на объект реализующий интерфейс B

49. Интерфейсы Java: определение интерфейса, реализация интерфейса. Преимущества применения интерфейсов. Переменные интерфейсов. Наследование интерфейсов. Методы по умолчанию. Статические методы интерфейсов.

Интерфейсы определяют некоторый функционал, не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. И один класс может применить множество интерфейсов.

Все методы интерфейса не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

```
interface Printable{

	void print();

}
```

Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Так, в данном случае объявлен один метод, который не имеет реализации.

Чтобы класс применил интерфейс, надо использовать ключевое слово implements

```
public class Dog implements Animal {

	@Override

	public void sound() {

	System.out.println("Dog barks");

	}


	@Override

	public void eat() {

	System.out.println("Dog eats");

	}

}
```

Если класс применяет интерфейс, то он должен реализовать все методы интерфейса.

Преимущества интерфейсов:

- Множественное наследование: класс может реализовать несколько интерфейсов, чего нельзя достичь с помощью наследования классов.
- Слабая связность: интерфейсы отделяют определение от реализации, что улучшает поддержку и тестируемость кода.
- Полиморфизм: объекты интерфейсов можно использовать для работы с любыми реализациями.

Переменные интерфейсов:

- неявно всегда являются полями с модификаторами public (доступны везде), static (относятся к интерфейсу, а не к объекту) и fina (значение переменной неизменно после инициализации). То есть константами.

```
public interface Constants {

int maxValue = 100; // Неявно public static final

}
```

Наследование интерфейсов:

Интерфейсы, как и классы, могут наследоваться с помощью ключевого слова extends:

```
interface BookPrintable extends Printable{

        void paint();

}
```

Методы по умолчанию:

интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Например, создадим метод по умолчанию в интерфейсе Printable:

```
interface Printable {

        default void print(){

    System.out.println("Undefined printable");

        }

}
```

Метод по умолчанию - это обычный метод без модификаторов, который помечается ключевым словом default.

Статистические методы:

аналогичны методам класса - относятся к интерфейсу, а не объекту:

```
interface Printable {

        void print();
```

```
        static void read(){
    System.out.println("Read printable");
        }
}
```

Чтобы обратиться к статическому методу интерфейса также, как и в случае с классами, пишут название интерфейса и метод:

```
public static void main(String[] args) {
        Printable.read();
}
```

50. Байтовые потоки InputStream и OutputStream. Консольный ввод и вывод Java. Символьные потоки данных. Абстрактные классы Writer, Reader.

InputStream и OutputStream — базовые классы для работы с байтовыми потоками.

InputStream — это класс в Java, предоставляющий возможность чтения байтов из различных источников, таких как файлы или сетевые соединения.

OutputStream предназначен для записи байтов в разнообразные приемники, будь то файлы или сетевые сокеты.

```
import java.io.*;
public class ByteStreamExample {
        public static void main(String[] args) throws IOException {
        // Запись в файл
        try (OutputStream os = new FileOutputStream("output.txt")) {
            os.write("Hello, Java!".getBytes());
        }
        // Чтение из файла
        try (InputStream is = new FileInputStream("output.txt")) {
        int data;
```

```
        while ((data = is.read()) != -1) {

        System.out.print((char) data); } }  } }
```

Консольный ввод и вывод:

Консольный ввод: System.in через Scanner или InputStreamReader.

Консольный вывод: System.out.

```
import java.util.Scanner;

public class ConsoleIOExample {

        public static void main(String[] args) {

        Scanner scanner =new Scanner(System.in);

        System.out.print("Enter your name: ");

        String name = scanner.nextLine();

        System.out.println("Hello, " + name + "!"); } }
```

Основные классы символьных потоков данных:

Объект, из которого можно считать данные, называется потоком ввода, а объект, в который можно записывать данные, - потоком вывода. Например, если надо считать содержимое файла, то применяется поток ввода, а если надо записать в файл - то поток вывода.

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: InputStream (представляющий потоки ввода) и OutputStream (представляющий потоки вывода)

Но поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы Reader (для чтения потоков символов) и Writer (для записи потоков символов).

Все остальные классы, работающие с потоками, являются наследниками абстрактных классов Reader и Writer, OutputStream и InputStream.

Классы Reader и Writer:

Основные методы Reader:

- *int read()* - возвращает представление очередного доступного символа во входном потоке в виде целого числа.
- *int read(char[] buffer)* - пытается прочесть максимум *buffer.length* символов из входного потока в массив *buffer*. Возвращает количество символов, в действительности прочитанных из потока.
- *int read(char[] buffer, int offset, int length)* - пытается прочесть максимум *length* символов, расположив их в массиве *buffer*, начиная с элемента *offset*. Возвращает количество реально прочитанных символов.
- *close()* – метод закрывает поток.

Основные методы Writer:

- *void write(int c)* – записывает один символ в поток.
- *void write(char[] buffer)* – записывает массив символов в поток.
- *void write(char[] buffer, int offset, int length)* – записывает в поток подмассив символов длиной *length*, начиная с позиции *offset*.
- *void write(String aString)* – записывает строку в поток.
- *void write(String aString, int offset, int length)* – записывает в поток подстроку символов длиной *length*, начиная с позиции *offset.*

51. Основные фреймворки и задачи, решаемые Spring.

Spring — это мощный фреймворк для создания Java-приложений. Он упрощает разработку корпоративных приложений за счет инверсии управления (IoC) и внедрения зависимостей (DI).

Основные модули Spring:

- Spring Core: основа фреймворка, предоставляет контейнер IoC для управления зависимостями.
- Spring MVC: фреймворк для создания веб-приложений с использованием шаблона MVC.
- Spring Data: упрощает работу с базами данных (например, через JPA, Hibernate).
- Spring Security: обеспечивает аутентификацию, авторизацию и защиту приложений.
- Spring Boot: упрощает настройку и развертывание приложений благодаря встроенному серверу и автонастройке.

Задачи, решаемые Spring:

- Управление зависимостями:
    - Использование IoC-контейнера для создания и управления объектами.
- Создание REST API:
    - Легкая разработка API через Spring MVC и Spring Boot.
- Доступ к данным:
    - Использование Spring Data для взаимодействия с реляционными и NoSQL базами данных.
- Безопасность:
    - Реализация ролей, токенов и шифрования через Spring Security.
- Масштабируемость:
    - Поддержка распределенных систем и микросервисов через Spring Cloud.

52. Spring Inversion of Control (IoC) контейнер Spring.

Inversion of Control (IoC) — это принцип проектирования, который предполагает передачу управления созданием объектов и их зависимостями от приложения к специальному контейнеру. В контексте фреймворка Spring это реализуется с помощью IoC-контейнера, который отвечает за управление жизненным циклом объектов (бинов) и их зависимостями.

Основные аспекты IoC-контейнера Spring:

Бины:

Бин — это объект, который управляется IoC-контейнером. Бины могут быть созданы, настроены и управлены контейнером в соответствии с определённой конфигурацией.

Контейнер:

IoC-контейнер Spring принимает на себя ответственность за создание и связывание бинов. Он может быть настроен с помощью XML-файлов, аннотаций или Java-конфигурации.

Внедрение зависимостей:

IoC-контейнер Spring использует паттерн "внедрение зависимостей" (Dependency Injection, DI) для автоматического предоставления объектам (бинам) необходимых зависимостей. Это можно сделать через конструктор, сеттер или метод-инициализатор.

Конфигурация:

Вы можете конфигурировать контейнер с помощью:

XML-описаний, где указываются бины и их зависимости.

Аннотаций, таких как @Component, @Autowired, @Configuration, @Bean, которые позволяют более удобно и гибко описывать бины и их зависимости прямо в Java-коде.

Java-классов конфигурации, в которых используются аннотации для определения бинов.

Жизненный цикл бинов:

IoC-контейнер управляет жизненным циклом бинов, включая создание, инициализацию, использование и уничтожение. Вы можете использовать коллбеки, такие как @PostConstruct и @PreDestroy, для выполнения логики после создания и перед уничтожением бина.

CScope (Область видимости):

Бины могут иметь разную область видимости: Singleton (один экземпляр на контейнер), Prototype (новый экземпляр каждый раз), Request, Session и другие.

Преимущества использования IoC:

Упрощение тестирования: Благодаря инверсии контроля, классы можно легче тестировать, подставляя моки для зависимостей.

Снижение связанности: Объекты менее зависимы от конкретных реализаций своих зависимостей.

Упрощение управления зависимостями: Контейнер берет на себя управление созданием и связыванием объектов.

Даня

53. Dependency Injection (DI) в Spring.

Dependency Injection (DI) — это шаблон проектирования, который используется для реализации принципа инверсии управления (IoC) в объектно-ориентированном программировании. В Spring Framework DI является одним из ключевых компонентов, который позволяет управлять зависимостями между объектами (beans) и обеспечивает гибкость и тестируемость кода.

Основные концепции DI в Spring:

1. Bean: Это объект, который управляется контейнером Spring. Bean может быть любым объектом, который вы хотите использовать в вашем приложении.

2. Application Context: Это центральный интерфейс для доступа к конфигурации приложения. Он читает конфигурационные метаданные и создает, настраивает и управляет бинами.

3. Configuration Metadata: Это информация, которая описывает, как создавать, настраивать и связывать бины. Конфигурационные метаданные могут быть предоставлены в виде XML-файлов, аннотаций или Java-кода.

Типы DI в Spring:

1. *Constructor-based DI*: Зависимости внедряются через конструктор класса.

```
public class MyService {
    private final MyRepository myRepository;

    @Autowired
    public MyService(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
```

2. *Setter-based DI:* Зависимости внедряются через сеттеры.

```
public class MyService {
    private MyRepository myRepository;

    @Autowired
    public void setMyRepository(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
```

3. *Field-based DI:* Зависимости внедряются непосредственно в поля класса.

```
public class MyService {
    @Autowired
    private MyRepository myRepository;
}
```

54. Жизненный цикл объекта Bean Spring.

Жизненный цикл объекта Bean в Spring Framework включает несколько этапов, начиная с его создания и заканчивая уничтожением. Понимание этих этапов важно для эффективного управления ресурсами и правильного использования бинов в приложении. Вот основные этапы жизненного цикла бина в Spring:

1. Инстанцирование (Instantiation)
Когда контейнер Spring загружает конфигурацию приложения, он создает экземпляры бинов. Это может быть сделано через конструктор, фабричный метод или статический фабричный метод.

2. Наполнение свойств (Dependency Injection)
После создания экземпляра бина, Spring внедряет зависимости через конструктор, сеттеры или поля. Этот процесс называется Dependency Injection (DI).

3. Название бина (Bean Naming)
Каждый бин в контейнере Spring имеет уникальное имя. Это имя может быть явно задано в конфигурации или автоматически сгенерировано на основе имени класса.

4. Инициализация бина (Bean Initialization)
После того как все зависимости внедрены, Spring вызывает методы инициализации бина. Это могут быть методы, аннотированные `@PostConstruct`, или методы, указанные в конфигурации через атрибут `init-method`.

5. Использование бина (Bean Usage)
Бин готов к использованию. Контейнер Spring предоставляет его другим бинам или компонентам приложения по мере необходимости.

6. Уничтожение бина (Bean Destruction)
Когда контейнер Spring завершает работу или бин больше не нужен, он вызывает методы уничтожения бина. Это могут быть методы, аннотированные `@PreDestroy`, или методы, указанные в конфигурации через атрибут `destroy-method`.

55. Конфигурация ApplicationContext с помощью xml в Spring.

Конфигурация `ApplicationContext` с помощью XML в Spring Framework является одним из традиционных способов настройки контейнера Spring. XML-конфигурация позволяет определить бины, их зависимости и другие настройки приложения. Вот основные шаги и примеры, как это сделать:

1. Создание XML-файла конфигурации

XML-файл конфигурации обычно называется `applicationContext.xml` или `beans.xml`. В этом файле определяются бины и их зависимости.

2. Определение бинов

В XML-файле конфигурации бины определяются с помощью тега `<bean>`. Каждый бин имеет уникальное имя и класс, который будет использоваться для создания экземпляра бина.

3. Загрузка конфигурации в приложении

Для загрузки XML-конфигурации и создания `ApplicationContext` используется класс `ClassPathXmlApplicationContext` или `FileSystemXmlApplicationContext`.

4. Внедрение зависимостей

Зависимости могут быть внедрены через конструктор, сеттеры или поля. В XML-конфигурации это делается с помощью тегов `<constructor-arg>`, `<property>` и `<value>`.

5. Использование аннотаций

Хотя XML-конфигурация является традиционным способом, Spring также поддерживает использование аннотаций для конфигурации бинов. Для этого необходимо включить поддержку аннотаций в XML-конфигурации.

56. Область видимости Bean в Spring.

В Spring Framework область видимости (scope) бина определяет, как и когда создаются экземпляры бина. Spring предоставляет несколько встроенных областей видимости, которые можно использовать для управления жизненным циклом бинов. Вот основные области видимости бинов в Spring:

1. Singleton (по умолчанию)
Описание: Один экземпляр бина создается на весь контейнер Spring.
Использование: Подходит для статических данных или сервисов, которые не хранят состояние.

2. Prototype
Описание: Новый экземпляр бина создается каждый раз, когда он запрашивается.
Использование: Подходит для бинов, которые должны иметь уникальное состояние для каждого запроса.

3. Request
Описание: Один экземпляр бина создается на каждый HTTP-запрос.
Использование: Подходит для веб-приложений, где бин должен существовать только в рамках одного запроса.

4. Session
Описание: Один экземпляр бина создается на каждую HTTP-сессию.
Использование: Подходит для веб-приложений, где бин должен существовать в рамках одной сессии пользователя.

5. Global Session
Описание: Один экземпляр бина создается на глобальную HTTP-сессию.
Использование: Подходит для порталов, где бин должен существовать в рамках глобальной сессии.

6. Application
Описание: Один экземпляр бина создается на весь контекст сервлета.
Использование: Подходит для веб-приложений, где бин должен существовать в рамках всего приложения.

7. Websocket
Описание: Один экземпляр бина создается на каждую WebSocket-сессию.
Использование: Подходит для приложений, использующих WebSocket, где бин должен существовать в рамках одной WebSocket-сессии.

Кастомные области видимости
Spring также позволяет создавать кастомные области видимости, реализуя интерфейс `Scope`. Это может быть полезно для специфических требований приложения.

57. Фабричные или factory-методы в Spring.

В Spring Framework фабричные методы (factory methods) используются для создания объектов (бинов) с помощью методов, которые возвращают экземпляры этих объектов. Это позволяет более гибко управлять процессом создания бинов и может быть полезно в случаях, когда создание объекта требует сложной логики или дополнительных параметров.

Основные типы фабричных методов в Spring:

1. Статические фабричные методы
Статические фабричные методы — это методы, которые не требуют создания экземпляра класса для вызова. Они вызываются напрямую через имя класса.

2. Нестатические фабричные методы
Нестатические фабричные методы — это методы, которые требуют создания экземпляра класса для вызова. Они вызываются через экземпляр класса.

Использование фабричных методов

Фабричные методы могут быть полезны в следующих случаях:

1. Сложная логика создания объектов: Если создание объекта требует сложной логики или дополнительных параметров, фабричные методы позволяют инкапсулировать эту логику.

2. Создание объектов с ограниченным доступом: Если конструктор объекта имеет ограниченный доступ (например, private), фабричные методы могут предоставить альтернативный способ создания объекта.

3. Поддержка различных конфигураций: Фабричные методы могут использоваться для создания объектов с различными конфигурациями в зависимости от параметров.

58. Конфигурация ApplicationContext с помощью аннотаций в Spring.

Конфигурация `ApplicationContext` с помощью аннотаций в Spring Framework является современным и предпочтительным способом настройки контейнера Spring. Аннотации позволяют более компактно и чисто описывать конфигурацию бинов и их зависимостей, делая код более читаемым и удобным для поддержки.

Основные аннотации для конфигурации:

1. `@Configuration`: Указывает, что класс содержит конфигурацию бинов.
2. `@Bean`: Указывает, что метод возвращает бин, который должен быть управляемым контейнером Spring.
3. `@ComponentScan`: Указывает, что Spring должен сканировать указанные пакеты для автоматического обнаружения и регистрации бинов.
4. `@Autowired`: Указывает, что Spring должен автоматически внедрить зависимость.
5. `@Value`: Указывает, что значение должно быть внедрено из внешнего источника, такого как файл свойств.

Конфигурация `ApplicationContext` с помощью аннотаций в Spring предоставляет мощный и гибкий способ настройки контейнера и управления бинами. Аннотации делают код более чистым и удобным для поддержки, что особенно важно в больших и сложных приложениях.

59. Связывание в Spring, аннотация @Autowired.

В Spring Framework аннотация `@Autowired` используется для автоматического внедрения зависимостей в бины. Она позволяет Spring автоматически находить и внедрять зависимости, что упрощает управление зависимостями и делает код более чистым и удобным для поддержки.

Основные способы использования `@Autowired`:

1. Внедрение через конструктор
Внедрение через конструктор является предпочтительным способом, так как оно обеспечивает неизменяемость бина и явно показывает зависимости, необходимые для создания бина.

2. Внедрение через сеттер
Внедрение через сеттер позволяет изменять зависимости после создания бина. Это может быть полезно в некоторых случаях, но менее предпочтительно по сравнению с внедрением через конструктор.

3. Внедрение через поле
Внедрение через поле является наиболее простым способом, но оно скрывает зависимости и делает бин изменяемым. Этот способ менее предпочтителен для тестирования и поддержки кода.

Использование `@Autowired` с коллекциями и массивами
Spring также поддерживает внедрение коллекций и массивов с помощью `@Autowired`.

Использование `@Autowired` с `@Qualifier`
Если у вас есть несколько бинов одного типа, вы можете использовать аннотацию `@Qualifier` для указания конкретного бина, который должен быть внедрен.

Использование `@Autowired` с `@Value`
Аннотация `@Value` используется для внедрения значений из внешних источников, таких как файлы свойств или переменные окружения.

60. Архитектурный стиль REST.

REST (Representational State Transfer) — это архитектурный стиль для создания веб-сервисов, который использует стандартные протоколы HTTP для обмена данными между клиентом и сервером. RESTful веб-сервисы основаны на принципах REST и обеспечивают гибкость, масштабируемость и простоту взаимодействия.

Основные принципы REST:

1. Клиент-серверная архитектура:
Клиент и сервер взаимодействуют через стандартные протоколы (обычно HTTP).
Клиент отправляет запросы, а сервер обрабатывает их и возвращает ответы.

2. Безсостоятельность (Statelessness):
Каждый запрос от клиента к серверу должен содержать всю необходимую информацию для его обработки.
Сервер не хранит состояние клиента между запросами.

3. Кэширование:
Ответы сервера могут быть кэшированы клиентом или промежуточными серверами для улучшения производительности.

4. Единообразие интерфейса (Uniform Interface):
Использование стандартных методов HTTP (GET, POST, PUT, DELETE и т.д.) для выполнения операций.
Использование стандартных форматов данных (JSON, XML) для обмена информацией.

5. Слои (Layered System):
Архитектура может состоять из нескольких слоев, каждый из которых выполняет свою роль (например, балансировщик нагрузки, кэш, база данных).

6. Код по требованию (Code on Demand, опционально):
Сервер может отправлять клиенту исполняемый код (например, JavaScript), который клиент может выполнить.

Основные компоненты RESTful веб-сервиса:

1. Ресурсы:
Основные единицы данных, с которыми взаимодействует клиент.

Ресурсы идентифицируются с помощью URI (Uniform Resource Identifier).

2. Методы HTTP:
GET: Получение данных.
POST: Создание нового ресурса.
PUT: Обновление существующего ресурса.
DELETE: Удаление ресурса.
PATCH: Частичное обновление ресурса.

3. Статусные коды HTTP:
200 OK: Запрос успешно выполнен.
201 Created: Ресурс успешно создан.
204 No Content: Запрос успешно выполнен, но нет контента для возврата.
400 Bad Request: Неверный запрос.
401 Unauthorized: Неавторизованный доступ.
403 Forbidden: Доступ запрещен.
404 Not Found: Ресурс не найден.
500 Internal Server Error: Внутренняя ошибка сервера.

Преимущества REST:

1. Простота: Использование стандартных протоколов и форматов данных.
2. Масштабируемость: Безсостоятельность и кэширование позволяют легко масштабировать приложение.
3. Гибкость: Поддержка различных форматов данных и методов HTTP.
4. Интероперабельность: Легко интегрируется с различными системами и платформами.

Катя

61. Spring Web-MVC, основная схема и логика работы.

Spring Web MVC (Model-View-Controller) — это фреймворк для создания веб-приложений на базе Java, который следует архитектурному паттерну MVC.

Основные компоненты:

1. DispatcherServlet:
    - Центральный компонент, который обрабатывает все входящие HTTP-запросы.
    - Перенаправляет запросы к соответствующим контроллерам на основе URL-маппинга.
2. HandlerMapping:

    - Определяет, какой контроллер (handler) должен обработать запрос на основе URL и других параметров.
3. Controller (Handler):
    - Обрабатывает бизнес-логику и взаимодействует с моделью.
    - Возвращает имя представления (view) и модель (data).
4. ModelAndView:
    - Объект, который содержит данные модели и имя представления.
5. ViewResolver:
    - Определяет, какое представление (view) должно быть использовано для рендеринга ответа на основе имени представления.
6. View:
    - Отвечает за рендеринг ответа, обычно в формате HTML, но может быть и в других форматах (JSON, XML и т.д.).

Логика работы:

1. Получение запроса:
    - Клиент (например, веб-браузер) отправляет HTTP-запрос на сервер.
    - DispatcherServlet принимает запрос.
2. Определение контроллера:
    - DispatcherServlet использует HandlerMapping для определения, какой контроллер должен обработать запрос.
    - HandlerMapping возвращает соответствующий контроллер на основе URL и других параметров.
3. Обработка запроса контроллером:
    - DispatcherServlet вызывает метод контроллера, который соответствует запросу.
    - Контроллер обрабатывает бизнес-логику, взаимодействует с моделью (например, получает данные из базы данных) и возвращает ModelAndView или String (имя представления) и Model.
4. Определение представления:
    - DispatcherServlet использует ViewResolver для определения, какое представление должно быть использовано для рендеринга ответа.
    - ViewResolver возвращает соответствующее представление на основе имени представления.
5. Рендеринг ответа:
    - DispatcherServlet передает модель и представление в View.
    - View рендерит ответ (например, генерирует HTML-страницу) и отправляет его обратно клиенту.

62. Класс DispatcherServlet, его функции.

`DispatcherServlet` — это центральный компонент в Spring Web MVC, который обрабатывает все входящие HTTP-запросы и управляет их маршрутизацией к соответствующим контроллерам. Вот основные функции `DispatcherServlet`:

Основные функции `DispatcherServlet`:

1. Прием запросов:
   - `DispatcherServlet` принимает все входящие HTTP-запросы от клиентов
   - Он служит единой точкой входа для всех запросов, что упрощает управление и конфигурацию.
2. Маршрутизация запросов:
   - `DispatcherServlet` использует `HandlerMapping` для определения, какой контроллер (handler) должен обработать запрос.
   - `HandlerMapping` отображает URL-запросы на соответствующие контроллеры на основе конфигурации.
3. Вызов контроллеров:
   - После определения соответствующего контроллера, `DispatcherServlet` вызывает метод контроллера, который соответствует запросу.
   - Контроллер обрабатывает бизнес-логику и возвращает `ModelAndView` или `String` (имя представления) и `Model`.
4. Обработка исключений:
   - `DispatcherServlet` может обрабатывать исключения, возникающие в процессе выполнения запроса.
   - Он использует `HandlerExceptionResolver` для обработки исключений и возврата соответствующих ответов клиенту.
5. Определение представлений:
   - `DispatcherServlet` использует `ViewResolver` для определения, какое представление должно быть использовано для рендеринга ответа.
   - `ViewResolver` возвращает соответствующее представление на основе имени представления.
6. Рендеринг ответа:
   - `DispatcherServlet` передает модель и представление в `View`.
   - `View` рендерит ответ (например, генерирует HTML-страницу) и отправляет его обратно клиенту.
7. Поддержка интернационализации:

- `DispatcherServlet` поддерживает интернационализацию (i18n) и локализацию (l10n), что позволяет создавать многоязычные приложения.

8. Поддержка тематизации:
    - `DispatcherServlet` поддерживает тематизацию, что позволяет изменять внешний вид приложения без изменения бизнес-логики.

63. Маппинг в Spring.

В Spring MVC маппинг — это процесс сопоставления входящих HTTP-запросов с соответствующими контроллерами и методами, которые должны их обработать. Маппинг позволяет определить, какой контроллер и метод будут вызваны для обработки запроса на основе URL, HTTP-метода, параметров запроса и других условий.

Основные типы маппинга в Spring MVC:

1. URL-маппинг:
    - Сопоставление запросов на основе URL-пути.
    - Используется аннотация `@RequestMapping` или более специализированные аннотации, такие как `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`, `@PatchMapping`.
2. HTTP-метод маппинг:
    - Сопоставление запросов на основе HTTP-метода (GET, POST, PUT, DELETE и т.д.).
    - Также используется аннотация `@RequestMapping` с атрибутом `method` или специализированные аннотации.
3. Параметры запроса:
    - Сопоставление запросов на основе параметров запроса.
    - Используется атрибут `params` в аннотации `@RequestMapping`.
4. Заголовки запроса:
    - Сопоставление запросов на основе заголовков HTTP-запроса.
    - Используется атрибут `headers` в аннотации `@RequestMapping`.
5. Контент-тип:
    - Сопоставление запросов на основе типа содержимого (Content-Type).
    - Используется атрибут `consumes` в аннотации `@RequestMapping`.
6. Принимаемый тип:
    - Сопоставление запросов на основе типа, который клиент ожидает получить в ответе (Accept).
    - Используется атрибут `produces` в аннотации `@RequestMapping`.

Для того чтобы Spring MVC мог корректно обрабатывать маппинг, необходимо настроить компонент-сканирование и аннотационную обработку в конфигурационном файле.Маппинг в Spring MVC позволяет гибко и точно определять, какой контроллер и метод будут обрабатывать входящие HTTP-запросы, что делает разработку веб-приложений более структурированной и управляемой.

64. Интерфейсы HttpServletRequest и HttpServletResponse.

Интерфейсы являются частью Java Servlet API и используются для обработки HTTP-запросов и ответов в веб-приложениях. Вот краткое описание каждого из них:

HttpServletRequest

`HttpServletRequest` предоставляет методы для получения информации о HTTP-запросе, который был отправлен клиентом (например, веб-браузером) на сервер. Основные методы включают:

- Получение параметров запроса:
  - `getParameter(String name)`: Возвращает значение указанного параметра запроса.
  - `getParameterValues(String name)`: Возвращает все значения указанного параметра запроса в виде массива строк.
  - `getParameterMap()`: Возвращает карту всех параметров запроса.
- Получение информации о запросе:
  - `getMethod()`: Возвращает метод HTTP-запроса (например, "GET", "POST").
  - `getRequestURI()`: Возвращает URI запроса.
  - `getRequestURL()`: Возвращает полный URL запроса.
  - `getHeader(String name)`: Возвращает значение указанного заголовка запроса.
  - `getHeaders(String name)`: Возвращает все значения указанного заголовка запроса в виде перечисления.
- Получение информации о сессии и куки:
  - `getSession()`: Возвращает текущую сессию или создает новую, если она не существует.
  - `getCookies()`: Возвращает массив объектов `Cookie`, связанных с этим запросом.

HttpServletResponse

`HttpServletResponse` предоставляет методы для создания HTTP-ответа, который будет отправлен клиенту. Основные методы включают:

- Установка статуса ответа:
  - `setStatus(int sc)`: Устанавливает статусный код HTTP-ответа (например, 200 для OK, 404 для Not Found).
- Установка заголовков ответа:
  - `setHeader(String name, String value)`: Устанавливает значение указанного заголовка ответа.
  - `addHeader(String name, String value)`: Добавляет значение к указанному заголовку ответа.
- Отправка данных клиенту:
  - `getWriter()`: Возвращает объект `PrintWriter` для отправки текстовых данных клиенту.
  - `getOutputStream()`: Возвращает объект `ServletOutputStream` для отправки бинарных данных клиенту.
- Управление сессиями и куки:
  - `addCookie(Cookie cookie)`: Добавляет указанный объект `Cookie` в ответ.

65. Архитектурный стиль CRUD, его соответствие REST и HTTP.

Архитектурный стиль CRUD (Create, Read, Update, Delete) часто используется в контексте веб-приложений и API для выполнения базовых операций с ресурсами. CRUD тесно связан с принципами REST (Representational State Transfer) и HTTP (HyperText Transfer Protocol). Давайте рассмотрим, как эти концепции соотносятся друг с другом.

CRUD Операции

CRUD операции представляют собой четыре основные операции, которые можно выполнять с ресурсами:

1. Create (Создание): Создание нового ресурса.
2. Read (Чтение): Получение существующего ресурса.
3. Update (Обновление): Обновление существующего ресурса.
4. Delete (Удаление): Удаление существующего ресурса.

REST и HTTP

REST — это архитектурный стиль для создания веб-сервисов, который использует HTTP-методы для выполнения операций над ресурсами. HTTP-методы соответствуют CRUD операциям следующим образом:

1. Create:
    - HTTP-метод: POST
    - Описание: Отправляет данные на сервер для создания нового ресурса.
    - Пример: `POST /resources`
2. Read:
    - HTTP-метод: GET
    - Описание: Запрашивает данные с сервера для получения существующего ресурса.
    - Пример: `GET /resources/{id}`
3. Update:
    - HTTP-метод: PUT или PATCH
    - Описание: Отправляет данные на сервер для обновления существующего ресурса.
        - PUT: Заменяет ресурс полностью.
        - PATCH: Обновляет только указанные поля ресурса.
    - Пример: `PUT /resources/{id}` или `PATCH /resources/{id}`
4. Delete:
    - HTTP-метод: DELETE
    - Описание: Отправляет запрос на сервер для удаления существующего ресурса.
    - Пример: `DELETE /resources/{id}`

Соответствие CRUD, REST и HTTP

- Create: POST запрос используется для создания нового ресурса.
- Read: GET запрос используется для получения существующего ресурса.
- Update: PUT или PATCH запросы используются для обновления существующего ресурса.
- Delete: DELETE запрос используется для удаления существующего ресурса.

66. Шаблон Data Access Object (DAO).

Шаблон Data Access Object (DAO) — это паттерн проектирования, который используется для абстракции и инкапсуляции доступа к источнику данных. Он позволяет отделить бизнес-логику от логики доступа к данным, что делает код

более модульным, тестируемым и поддерживаемым. DAO предоставляет интерфейс для выполнения операций создания, чтения, обновления и удаления (CRUD) данных.

Основные компоненты шаблона DAO

1. Интерфейс DAO: Определяет методы для выполнения операций CRUD.
2. Реализация DAO: Конкретная реализация интерфейса DAO, которая взаимодействует с источником данных (например, базой данных).
3. Модель данных: Классы, представляющие данные, с которыми работает DAO.

Пример интерфейса DAO

```
public interface UserDao {
    void create(User user);
    User read(int id);
    void update(User user);
    void delete(int id);
    List<User> findAll();
}
```

Преимущества использования шаблона DAO

1. Абстракция доступа к данным: Логика доступа к данным инкапсулируется в DAO, что позволяет изменять источник данных без изменения бизнес-логики.
2. Упрощение тестирования: DAO можно легко заменить моками или заглушками для тестирования бизнес-логики.
3. Повышение поддерживаемости: Код становится более модульным и легче поддерживается.

Заключение

Шаблон Data Access Object (DAO) является мощным инструментом для организации доступа к данным в приложениях. Он позволяет отделить бизнес-логику от логики доступа к данным, что делает код более гибким, тестируемым и поддерживаемым

67. Основные понятия Объектно-реляционного отображения (ORM - Object-Relational Mapping).

Объектно-реляционное отображение (ORM, Object-Relational Mapping) — это техника программирования, которая позволяет преобразовывать данные между несовместимыми типами систем хранения данных в объектно-ориентированных

языках программирования. ORM используется для упрощения работы с реляционными базами данных, предоставляя разработчикам возможность работать с данными как с объектами в коде, а не с SQL-запросами.

Основные понятия ORM

1. Объект (Object):
    - В контексте ORM объект представляет собой экземпляр класса, который отображается на запись в таблице базы данных.
    - Объекты имеют свойства (поля), которые соответствуют столбцам таблицы.
2. Класс (Class):
    - Класс в ORM отображается на таблицу в базе данных.
    - Свойства класса (поля) соответствуют столбцам таблицы.
3. Сессия (Session):
    - Сессия представляет собой временный контекст, в котором выполняются операции с объектами.
    - Сессия отслеживает изменения объектов и синхронизирует их с базой данных.
4. Контекст (Context):
    - Контекст — это объект, который управляет жизненным циклом объектов и их состоянием.
    - Контекст отслеживает изменения объектов и выполняет операции сохранения, обновления и удаления в базе данных.
5. Маппинг (Mapping):
    - Маппинг — это процесс отображения объектов на таблицы базы данных и наоборот.
    - Маппинг может быть определен с помощью аннотаций, XML-конфигураций или программного кода.
6. Аннотации (Annotations):
    - Аннотации используются для определения маппинга между объектами и таблицами базы данных.
    - Примеры аннотаций включают `@Entity`, `@Table`, `@Column`, `@Id`, `@OneToMany`, `@ManyToOne` и т.д.
7. Конфигурация (Configuration):
    - Конфигурация ORM включает настройки подключения к базе данных, параметры кэширования, настройки логирования и т.д.
    - Конфигурация может быть задана в файле конфигурации (например, `hibernate.cfg.xml` для Hibernate) или программно.
8. Ленивая загрузка (Lazy Loading):

- Ленивая загрузка — это стратегия загрузки данных, при которой данные загружаются только тогда, когда они действительно нужны.
- Ленивая загрузка помогает уменьшить количество запросов к базе данных и улучшить производительность.

9. Жадная загрузка (Eager Loading):
    - Жадная загрузка — это стратегия загрузки данных, при которой данные загружаются сразу при первом обращении к объекту.
    - Жадная загрузка может увеличить количество запросов к базе данных, но обеспечивает доступ ко всем данным сразу.
10. Кэширование (Caching):
    - Кэширование используется для улучшения производительности путем сохранения часто используемых данных в памяти.
    - ORM может поддерживать различные уровни кэширования, такие как кэш первого уровня (сессионный кэш) и кэш второго уровня (глобальный кэш)

68. Спецификация Java Persistence API (JPA).

Java Persistence API (JPA) — это спецификация для управления реляционными данными в приложениях на языке Java. JPA предоставляет стандартизированный способ взаимодействия с базами данных, используя объектно-реляционное отображение (ORM). JPA является частью Java EE (Enterprise Edition) и Java SE (Standard Edition), что делает её доступной для широкого круга приложений.

Основные компоненты JPA

1. Entity (Сущность):
    - Сущность — это легковесный, управляемый, постоянный объект, который представляет данные в базе данных.
    - Сущности аннотируются с помощью `@Entity`.
2. EntityManager (Менеджер сущностей):
    - `EntityManager` — это интерфейс, который используется для выполнения операций CRUD (создание, чтение, обновление, удаление) с сущностями.
    - `EntityManager` управляет жизненным циклом сущностей и их состоянием.
3. Persistence Context (Контекст постоянства):
    - Контекст постоянства — это набор управляемых сущностей в определенный момент времени.

- `EntityManager` управляет контекстом постоянства и отслеживает изменения сущностей.

4. Persistence Unit (Единица постоянства):
    - Единица постоянства — это набор сущностей, которые управляются одним `EntityManagerFactory`.
    - Единица постоянства определяется в файле конфигурации `persistence.xml`.
5. Query Language (Язык запросов):
    - JPA предоставляет свой собственный язык запросов, называемый JPQL (Java Persistence Query Language), который похож на SQL, но работает с сущностями вместо таблиц.

Олеся

69. Архитектура ORM Java Persistence API (JPA).

Ответ:
Java Persistence API (JPA) — это спецификация Java API для управления сохранением данных в Java-приложениях. Она предоставляет стандартизированный способ взаимодействия с базами данных, позволяя разработчикам писать код, который не привязан к конкретной реализации базы данных. JPA базируется на архитектуре Object-Relational Mapping (ORM), то есть сопоставляет объекты Java с реляционными таблицами базы данных.

Ключевые компоненты архитектуры JPA:
1. Интерфейс EntityManager: Это основной интерфейс JPA, который отвечает за взаимодействие с персистентными сущностями. Он предоставляет методы для создания (persist()) , чтения (find(), getReference()), обновления (merge()), удаления (remove()) и выполнения запросов
2. Интерфейс EntityManagerFactory: Фабрика для создания экземпляров EntityManager. Она настраивается на основе конфигурации приложения (например, через файл persistence.xml) и отвечает за установление соединения с базой данных.
3. JPA-провайдер (реализация JPA): Это конкретная реализация интерфейсов JPA, предоставляемая сторонними библиотеками (например, Hibernate). JPA-провайдер берет на себя фактическое взаимодействие с базой данных и реализацию ORM.
4. Конфигурация (persistence.xml): XML-файл, содержащий настройки для JPA-провайдера.
5. Сущности (Entities): Классы Java, которые представляют таблицы в базе данных. JPA аннотации включают:

- @Entity: Указывает, что класс является сущностью.

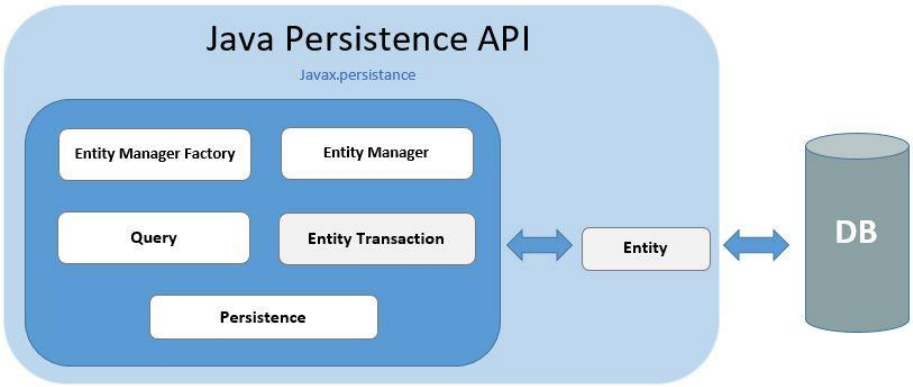
- @Table: Указывает имя таблицы в базе данных.
- @Id: Указывает поле, которое является первичным ключом.
- @Column: Указывает на соответствие поля колонке в базе данных.
- @GeneratedValue: Указывает на стратегию генерации первичного ключа.
- @ManyToOne, @OneToMany, @ManyToMany, @OneToOne: Аннотации для определения связей между сущностями.

Преимущества использования JPA:

- Независимость от базы данных
- Объектно-ориентированный подход
- Упрощение разработки
- Переносимость

Недостатки использования JPA:

- Увеличение сложности
- Потенциальные проблемы производительности
- Необходимость понимания ORM



70. Основные аннотации Java Persistence API (JPA).

Ответ:

Аннотации JPA являются неотъемлемой частью спецификации и играют ключевую роль в определении того, как объекты Java сопоставляются с реляционными таблицами базы данных. Они позволяют разработчикам определять поведение сущностей.

Основные:

Аннотации для сущностей (Entities):

1. @Entity: обозначает класс как сущность, т.е. как объект, который должен быть сохранен в базе данных. Каждый класс, представляющий таблицу в базе данных, должен быть аннотирован
2. @Table: указывает имя таблицы в базе данных, с которой связана данная сущность.

Аннотации для идентификаторов (Primary Keys):

1. @Id: обозначает поле (или метод), которое является первичным ключом в таблице. Каждая сущность должна иметь хотя бы одно поле, аннотированное @Id
2. @GeneratedValue: определяет стратегию генерации значений для первичного ключа. Например, GenerationType.AUTO, GenerationType.IDENTITY Может быть использована совместно с @Id.

Аннотации для отображения полей (Column Mapping):

1. @Column: отображает поле сущности на столбец в таблице базы данных. Можно настраивать различные параметры столбца (например, имя, длину, nullable, unique).
2. @Basic: по умолчанию используется для отображения простых типов данных на соответствующие столбцы.

Аннотации для связей (Relationships):

1. @ManyToOne: отображает связь "многие к одному" между двумя сущностями.
2. @OneToMany: отображает связь "один ко многим" между двумя сущностями.
3. @OneToOne: отображает связь "один к одному" между двумя сущностями.
4. @ManyToMany: отображает связь "многие ко многим" между двумя сущностями
5. @JoinColumn: указывает имя столбца в таблице, который используется для связывания двух сущностей через внешний ключ.
6. @JoinTable: используется для связи "многие ко многим" и определяет имя промежуточной таблицы.

Аннотации для запросов (Queries):

1. @NamedQuery: определяет статический именованный запрос, который можно вызывать по имени.
2. @NamedNativeQuery: определяет статический именованный запрос, который написан на языке SQL (а не JPQL).

71. Библиотека Hibernate, основные аннотации.

Ответ:

Hibernate – это популярный Java-фреймворк. Он предоставляет инструменты для управления персистентностью данных в приложениях, работающих с базами данных, обеспечивая объектно-реляционное отображение и облегчая взаимодействие с различными СУБД.

В основе Hibernate лежит идея сопоставления Java-объектов с таблицами реляционных баз данных. Он позволяет разработчикам работать с данными, используя объектно-ориентированный подход, при этом детали SQL-запросов и взаимодействия с базами данных абстрагируются.

Основные аннотации Hibernate:

Hibernate, будучи JPA-провайдером, использует JPA-аннотации для настройки ORM.

Аннотации для сущностей (Entities):

1. @Entity: обозначает класс как сущность, т.е. как объект, который должен быть сохранен в базе данных. Каждый класс, представляющий таблицу в базе данных, должен быть аннотирован
2. @Table: указывает имя таблицы в базе данных, с которой связана данная сущность.

Аннотации для идентификаторов (Primary Keys):

1. @Id: обозначает поле (или метод), которое является первичным ключом в таблице. Каждая сущность должна иметь хотя бы одно поле, аннотированное @Id
2. @GeneratedValue: определяет стратегию генерации значений для первичного ключа. Например, GenerationType.AUTO, GenerationType.IDENTITY Может быть использована совместно с @Id.

Аннотации для отображения полей (Column Mapping):

1. @Column: отображает поле сущности на столбец в таблице базы данных. Можно настраивать различные параметры столбца (например, имя, длину, nullable, unique).

Аннотации для связей (Relationships):

1. @ManyToOne: отображает связь "многие к одному" между двумя сущностями.
2. @OneToMany: отображает связь "один ко многим" между двумя сущностями.
3. @OneToOne: отображает связь "один к одному" между двумя сущностями.
4. @ManyToMany: отображает связь "многие ко многим" между двумя сущностями
5. @JoinColumn: указывает имя столбца в таблице, который используется для связывания двух сущностей через внешний ключ.
6. @JoinTable: используется для связи "многие ко многим" и определяет имя промежуточной таблицы.

Аннотации для запросов (Queries):

1. @NamedQuery: определяет статический именованный запрос, который можно вызывать по имени.
2. @NamedNativeQuery: определяет статический именованный запрос, который написан на языке SQL (а не JPQL).

Дополнительные аннотации (специфичные для Hibernate):

1. @Type: позволяет указать, как Hibernate должен отображать определенный тип данных на столбец базы данных. Применяется для работы со сложными типами или для указания конкретных SQL-типов
2. @DynamicInsert, @DynamicUpdate: позволяют Hibernate генерировать INSERT и UPDATE запросы, включающие только измененные поля.

72. Объявление сущности и таблицы в Hibernate.

Ответ:

В Hibernate сущности и таблицы представляют собой ключевые концепции, которые позволяют разработчикам работать с базами данных в объектно-ориентированном стиле.

Hibernate — это фреймворк для объектно-реляционного отображения (ORM), который упрощает взаимодействие между Java-приложениями и реляционными базами данных.

Сущность в Hibernate — это Java-класс, который представляет собой таблицу в базе данных. Каждый экземпляр этого класса соответствует строке в таблице. Для объявления сущности необходимо использовать аннотацию @Entity. Также важно указать, какая таблица будет соответствовать данной сущности, с помощью аннотации @Table.

Пример:

```
@Entity
@Table(name = "users")
    public class User {
   @Id
   private Long id;
   private String name;
   private String email;
```

Дальше идут конструкторы, геттеры и сеттеры

В приведенном выше примере аннотация @Table(name = "users") указывает, что данная сущность соответствует таблице users в базе данных. Поле id помечено аннотацией @Id, что обозначает, что это первичный ключ таблицы.

Hibernate также поддерживает различные типы связей между сущностями, такие как @OneToOne, @OneToMany, @ManyToOne, и @ManyToMany. Эти аннотации позволяют моделировать сложные структуры данных и управлять ими.

73. Интерфейс Session в Hibernate.

Ответ:

В Hibernate, интерфейс Session является ключевым компонентом, предоставляющим доступ к базе данных и позволяющим выполнять операции по сохранению, чтению, обновлению и удалению сущностей. Session является основным интерфейсом для взаимодействия с персистентным слоем и предоставляет возможности для работы с транзакциями, запросами и кэшированием.

Основные функции интерфейса Session:

1. Управление жизненным циклом сущностей:
   - save()
   - persist()
   - remove()
   - update()

- merge()
- refresh()

2. Загрузка сущностей:
   - get()
   - load():
3. Выполнение запросов:
   - createQuery(): Создает объект Query для выполнения HQL (Hibernate Query Language) запросов
   - createSQLQuery(): Создает объект Query для выполнения нативных SQL-запросов.
4. Управление транзакциями:
   - beginTransaction()
   - getTransaction()
   - clear()
   - close()

Состояния сущностей в Hibernate:

Session управляет жизненным циклом сущностей, которые могут находиться в одном из следующих состояний:

- Transient (временный): Сущность создана, но еще не связана с сессией и не имеет идентификатора.
- Persistent (персистентный): Сущность связана с сессией, ее изменения отслеживаются, и она синхронизируется с базой данных.
- Detached (отсоединенный): Сущность не связана с сессией, ее изменения не отслеживаются.
- Removed (удаленный): Сущность запланирована на удаление из базы данных.

Session создается с помощью SessionFactory:

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
```

74. Ассоциация сущностей в Hibernate.

Ответ:

В Hibernate ассоциации сущностей представляют собой связи между таблицами в реляционной базе данных, которые выражаются в виде связей между объектами в объектно-ориентированном приложении. Hibernate предоставляет различные типы ассоциаций:

1. Один к одному (One-to-One):
   - Представляет связь, где одна сущность связана с *одной* другой сущностью.
   - Пример: Один пользователь имеет один профиль.

- Аннотации: @OneToOne

2. Один ко многим (One-to-Many):
   - Представляет связь, где одна сущность связана с *несколькими* другими сущностями.
   - Пример: Один магазин имеет несколько работников.
   - Аннотации: @OneToMany, @ManyToOne
   - Владеющая сторона: Сущность с @ManyToOne является владеющей стороной связи.
   - Инверсная сторона: Сущность с @OneToMany является инверсной стороной связи, обычно со ссылкой mappedBy.
3. Многие к одному (Many-to-One):
   - Представляет связь, где *несколько* сущностей связаны с *одной* другой сущностью.
   - Пример: Много работников принадлежат к одному магазину.
   - Аннотации: @ManyToOne, @JoinColumn
   - Связь многие к одному всегда является *владеющей стороной*.
4. Многие ко многим (Many-to-Many):
   - Представляет связь, где *несколько* сущностей связаны с *несколькими* другими сущностями.
   - Пример: Много студентов могут посещать много курсов.
   - Аннотации: @ManyToMany, @JoinTable
   - Требует промежуточную таблицу для хранения связей.

Ассоциации сущностей в Hibernate позволяют моделировать сложные отношения между объектами и автоматически отображать их на реляционную базу данных. Понимание различных типов ассоциаций и связанных аннотаций является ключевым для создания гибких и эффективных приложений на Hibernate. Выбор правильной ассоциации и грамотное ее использование способствуют созданию удобной и поддерживаемой архитектуры приложения.

75. Spring Boot: определение, характеристики, преимущества.

Ответ:

Spring Boot - это фреймворк для создания микросервисов и самостоятельных приложений на платформе Spring. Он предлагает подход, который позволяет разработчикам сосредоточиться на бизнес-логике, а не на рутинных настройках. Spring Boot инкапсулирует в себе все необходимые зависимости и настройки, избавляя от необходимости ручного управления конфигурацией и зависимостями.

Основные характеристики Spring Boot:

1. Автоконфигурация:
   - Spring Boot автоматически настраивает Spring-приложение, основываясь на зависимостях, присутствующих в classpath.

    - Он определяет необходимые компоненты и настройки, устраняя необходимость ручного конфигурирования.
2. Запуск как самостоятельное приложение:
    - Spring Boot позволяет создавать приложения, которые можно запускать как обычные Java-программы с помощью встроенного веб-сервера.
3. Наличие стартеров:
    - Spring Boot предлагает "стартеры" – набор зависимостей, объединенных по функциональности. Стартеры упрощают добавление необходимых библиотек в проект, автоматически разрешая конфликты версий и обеспечивая совместимость.
4. Встроенные инструменты:
    - Spring Boot предоставляет встроенные инструменты для мониторинга, управления и тестирования приложения.
5. Конфигурация через properties и YAML файлы:
    - Spring Boot позволяет настраивать приложение, используя файлы application.properties или application.yml. Эти файлы позволяют переопределить значения по умолчанию, настроить соединение с базой данных, задать порты и т.д.

Преимущества использования Spring Boot:

1. Ускорение разработки
2. Упрощение развертывания
3. Повышение производительности
4. Микросервисная архитектура
5. Гибкость и расширяемость

76. Spring Initializr, особенности и преимущества применения.

Ответ:

Spring Initializr - это веб-сервис и инструмент, предоставляемый Spring Framework, который значительно упрощает начальную настройку Spring Boot проектов. Он позволяет быстро создавать скелеты проектов с необходимыми зависимостями и конфигурацией, избавляя разработчиков от рутинной работы по настройке базового проекта.

Особенности Spring Initializr

1. Веб-интерфейс и API:
    - Spring Initializr доступен через веб-интерфейс, а также через REST API. Это позволяет использовать его как в браузере, так и интегрировать с инструментами разработки
2. Простота использования:

    - Разработчику нужно выбрать лишь основные параметры, такие как язык, тип сборки, версия Spring Boot, и добавить необходимые зависимости.
3. Настройка проекта:
    - Initializr позволяет настраивать основные параметры проекта, такие как тип проекта, язык программирования, версия Spring Boot, группу, артефакт и тд.
4. Выбор зависимостей (Starters):
    - Initializr позволяет выбрать необходимые стартеры Spring Boot, которые автоматически добавляют нужные зависимости в проект
5. Генерация проекта:
    - После настройки параметров и выбора зависимостей, Initializr генерирует ZIP-архив с готовым проектом.
6. Интеграция с IDE
7. Поддержка REST API

Преимущества применения Spring Initializr:

1. Быстрая настройка проекта
2. Избежание ошибок
3. Сокращение boilerplate-кода
4. Унифицированный подход
5. Актуальные версии зависимостей
6. Интеграция с IDE
7. Улучшение продуктивности

Никита

77. Структура фреймворка JUnit.

JUnit — это популярный фреймворк для тестирования на языке Java. Он предоставляет инструменты для написания и выполнения автоматических тестов, что помогает разработчикам убедиться в корректности их кода. Вот основная структура и компоненты JUnit:

1. **Аннотации**
JUnit использует аннотации для обозначения тестовых методов и конфигураций. Вот основные аннотации:

@Test: Обозначает метод как тестовый метод.
@Before: Метод, аннотированный @Before, выполняется перед каждым тестовым методом.
@After: Метод, аннотированный @After, выполняется после каждого тестового метода.

2. **Assertions** (Утверждения)

Утверждения используются для проверки ожидаемых результатов. Вот некоторые из них:

assertEquals(expected, actual): Проверяет, что два значения равны.
assertTrue(condition): Проверяет, что условие истинно.
assertFalse(condition): Проверяет, что условие ложно.

3. **Test Runners (Запускатели тестов)**
Запускатели тестов отвечают за выполнение тестов. В JUnit 4 используется JUnit4 запускатель, а в JUnit 5 — JUnitPlatform.

4. **Test Suites** (Наборы тестов)
Наборы тестов позволяют группировать несколько тестовых классов для совместного выполнения.
5. **Parameterized Tests** (Параметризованные тесты)
Параметризованные тесты позволяют выполнять один и тот же тест с разными наборами данных.
6. **Rules** (Правила)
Правила позволяют изменять поведение тестов. Например, TemporaryFolder правило создает временную папку, которая удаляется после выполнения тестов.

78. JUnit аннотации @Test, @DisplayName.

Аннотация @Test
Аннотация @Test используется для обозначения метода как тестового метода. В JUnit 5 эта аннотация входит в пакет org.junit.jupiter.api.

Пример использования:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MyTest {

    @Test
    public void testAddition() {
        assertEquals(4, 2 + 2);
    }

    @Test
    public void testSubtraction() {
        assertEquals(2, 4 - 2);
    }
}
```

Аннотация @DisplayName
Аннотация @DisplayName используется для задания отображаемого имени теста или тестового класса. Это полезно для улучшения читаемости отчетов о тестах, так как позволяет давать более описательные имена тестам.

Пример использования:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MyTest {

    @Test
    @DisplayName("Тест сложения двух чисел")
    public void testAddition() {
        assertEquals(4, 2 + 2);
    }

    @Test
    @DisplayName("Тест вычитания двух чисел")
    public void testSubtraction() {
        assertEquals(2, 4 - 2);
    }
}
```

79. JUnit аннотации @BeforeEach, @AfterEach.

Аннотация @BeforeEach
Аннотация @BeforeEach используется для обозначения метода, который должен выполняться перед каждым тестовым методом в классе. Это полезно для инициализации состояния, которое требуется для выполнения тестов.

Пример использования:

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MyTest {

    private int value;

    @BeforeEach
```

```
    public void setUp() {
        value = 0; // Инициализация состояния перед каждым тестом
    }

    @Test
    public void testIncrement() {
        value++;
        assertEquals(1, value);
    }

    @Test
    public void testDecrement() {
        value--;
        assertEquals(-1, value);
    }
}
```

Аннотация @AfterEach

Аннотация @AfterEach используется для обозначения метода, который должен выполняться после каждого тестового метода в классе. Это полезно для очистки состояния или освобождения ресурсов после выполнения тестов.

Пример использования:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MyTest {

    private int value;

    @BeforeEach
    public void setUp() {
        value = 0; // Инициализация состояния перед каждым тестом
    }

    @AfterEach
    public void tearDown() {
        value = -1; // Очистка состояния после каждого теста
    }

    @Test
    public void testIncrement() {
```

```
        value++;
        assertEquals(1, value);
    }

    @Test
    public void testDecrement() {
        value--;
        assertEquals(-1, value);
    }
}
```

80. Тестовые классы и методы JUnit.

Тестовые классы

Тестовый класс — это класс, содержащий тестовые методы. В JUnit 5 тестовый класс не требует специальных аннотаций на уровне класса, но может содержать аннотации для настройки и очистки состояния, а также для определения отображаемого имени класса.

Пример тестового класса:

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

@DisplayName("Тесты для математических операций")
public class MathOperationsTest {

    @Test
    @DisplayName("Тест сложения двух чисел")
    public void testAddition() {
        assertEquals(4, 2 + 2);
    }

    @Test
    @DisplayName("Тест вычитания двух чисел")
    public void testSubtraction() {
        assertEquals(2, 4 - 2);
    }
}
```

Тестовые методы

Тестовый метод — это метод, который содержит код для проверки определенного аспекта функциональности. В JUnit 5 тестовый метод обозначается аннотацией @Test.

Пример тестового метода:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MathOperationsTest {

    @Test
    public void testAddition() {
        assertEquals(4, 2 + 2);
    }

    @Test
    public void testSubtraction() {
        assertEquals(2, 4 - 2);
    }
}
```

81. Утверждения JUnit. Класс Assert.

В JUnit утверждения (assertions) используются для проверки ожидаемых результатов в тестах. Класс Assert предоставляет различные методы для выполнения этих проверок. В JUnit 5 класс Assert был заменен на Assertions, который предоставляет аналогичные и дополнительные методы.

Основные методы класса Assertions в JUnit 5

1) *assertEquals(expected, actual):*

Проверяет, что два значения равны.

```
assertEquals(4, 2 + 2);
```

2) assertTrue(condition):

Проверяет, что условие истинно.

```
assertTrue(2 + 2 == 4);
```

3) assertFalse(condition):

Проверяет, что условие ложно.

```
assertFalse(2 + 2 == 5);
```

4) *assertNull(object):*

Проверяет, что объект равен null.

assertNull(null);

5) *assertNotNull(object):*

Проверяет, что объект не равен null.

assertNotNull("Hello");

82. Тестирование исключений JUnit.

Тестирование исключений в JUnit позволяет проверять, что определенный код вызывает ожидаемое исключение. Это важно для убедиться, что код правильно обрабатывает исключительные ситуации. В JUnit 5 для тестирования исключений используется метод assertThrows из класса Assertions.

Пример использования assertThrows

Метод assertThrows позволяет указать тип исключения, которое вы ожидаете, и лямбда-выражение, содержащее код, который должен вызвать это исключение.

Пример:

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class ExceptionTest {

    @Test
    public void testDivideByZero() {
        ArithmeticException exception = assertThrows(ArithmeticException.class, () -> {
            int result = 1 / 0;
        });
        assertEquals("/ by zero", exception.getMessage());
    }

    @Test
    public void testNullPointerException() {
        NullPointerException exception = assertThrows(NullPointerException.class, () -> {
            String str = null;
```

```
            str.length();
        });
        assertNull(exception.getMessage());
    }
}
```

83. Генератор документирования Javadoc. Виды комментариев.

Javadoc — это инструмент для генерации документации из исходного кода Java. Он анализирует комментарии в исходном коде и генерирует HTML-документацию, которая может быть использована для описания классов, методов, полей и других элементов кода.

Виды комментариев в Javadoc
В Javadoc существует несколько видов комментариев, которые используются для документирования различных элементов кода:

1) **Комментарии класса:**

Описывают класс в целом.
Размещаются непосредственно перед объявлением класса.

```
/**
 * Класс для выполнения математических операций.
 */
public class MathOperations {
    // ...
}
```

2) **Комментарии метода:**

Описывают метод, его параметры, возвращаемое значение и возможные исключения.
Размещаются непосредственно перед объявлением метода.

```
/**
 * Сложение двух чисел.
 */
public int add(int a, int b) {
    return a + b;
}
```

3) **Комментарии поля:**

Описывают поле класса.
Размещаются непосредственно перед объявлением поля.

```
/**
 * Значение по умолчанию.
 */
private int defaultValue = 0;
```

4) **Комментарии конструктора:**

Описывают конструктор класса.
Размещаются непосредственно перед объявлением конструктора.

```
/**
 * Конструктор класса MathOperations.
 */
public MathOperations(int initialValue) {
    this.defaultValue = initialValue;
}
```

84. Дескрипторы Javadoc.

Дескрипторы (или теги) Javadoc используются для документирования различных аспектов кода, таких как параметры методов, возвращаемые значения, исключения и т.д. Эти теги помогают создавать более информативную и структурированную документацию. Вот некоторые из наиболее часто используемых дескрипторов Javadoc:

Основные дескрипторы Javadoc

1) @author:

Указывает автора класса или интерфейса.

2) @version:

Указывает версию класса или интерфейса.

3) @param:

Описывает параметр метода.

4) @return:

Описывает возвращаемое значение метода.

5) @throws:

Описывает исключение, которое может быть выброшено методом.

6) @deprecated:

Указывает, что метод или класс устарел и не рекомендуется к использованию.

7) @see:

Ссылается на другой метод, класс или документ.

8) @link:

Вставляет гиперссылку на другой метод, класс или документ.

Задачи

Настя - 1, 46 Простые: 2, 13
Соня - 4, 18 П: 3, 14
Антон - Сложные: 16, 28 | Простые: 5, 15
Гриша - 17, 41 П: 6, 19
Катя - 21, 26 П: 7, 20
Олеся - 22, 42 П: 8, 23
Саша - 24, 43 П: 9, 25
Лера - 36, 44 П: 10, 27
Даня - 40 П: 11, 29, 31, 33, 35, 38
Никита - 45 П: 12, 30, 32, 34, 37, 39

Простые:

2. Напишите программу, в которой из строки "I have 3 cats, 4 dogs, and 1 turtle" отбираются цифры. Из этих цифр формируется массив.

```
// Объект для получения строки с консоли
Scanner scan = new Scanner(System.in);
// Список для хранения цифр
List<Integer> arrayDigits = new ArrayList<>();
// Получение строки с консоли
String inputString = scan.nextLine();
```

```
// Проход по каждому символу строки
for (char c : inputString.toCharArray()) {
   // Проверяем, является ли символ цифрой
   if (Character.isDigit(c)) {
       // Преобразуем символ в число и добавляем в список
       arrayDigits.add(Character.getNumericValue(c));
   }
}
// Вывод результата
System.out.println("Найденные цифры: " + arrayDigits);
```

3. Разработайте программу, которая выводит в консоль все цифры, входящие в натуральное число n. К примеру, если дано число 2359, то в консоль выводятся отдельно числа 2, 3, 5, 9.

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

       Scanner scanner = new Scanner(System.in);
       int userInt = scanner.nextInt();
       if (userInt <= 0) {
           System.out.println("Данное число ненатуральное!");
       } else {
           String stringNumber = Integer.toString(userInt);
           char[] sctringChar = stringNumber.toCharArray();
           System.out.println("Цифры в числе:");
           for (char digit : sctringChar) {
               System.out.println(digit);
           }
       }
   }
}
```

5. Напишите программную реализацию бинарного дерева поиска.

```
public class task5 {

   static class Node {
      int key;
      Node left, right;

      public Node (int item) {
         key = item;
```

```
        left = right = null;
    }
}

static class BinarySearchTree {
    Node root;

    // Конструктор
    BinarySearchTree() {
        root = null;
    }

    // Вставка нового ключа
    void insert(int key) {
        root = insertRec(root, key);
    }

    /* Функция для вставки нового узла с заданным
    ключом */
    Node insertRec(Node root, int key) {
        //Если дерево пусто, возвращаем новый узел
        if (root == null) {
            root = new Node(key);
            return root;
        }

        // Иначе, рекурсивно вызываем функцию вставки
        if (key < root.key)
            root.left = insertRec(root.left, key);
        else if (key > root.key)
            root.right = insertRec(root.right, key);

        // Возвращаем неизменный узел
        return root;
    }

    // Поиск ключа в BST
    boolean search(int key) {
        return searchRec(root, key);
    }

    /* Функция для поиска узла с заданным ключом */
    boolean searchRec(Node root, int key) {
        // Базовый случай: пустое дерево
```

```
    if (root == null)
        return false;

    // Если ключ совпадает с корневым
    if (root.key == key)
        return true;

    // Если ключ меньше корневого, рекурсивно ищем в левом поддереве
    if (key < root.key)
        return searchRec(root.left, key);

    // Если ключ больше корневого, рекурсивно ищем в правом поддереве
    return searchRec(root.right, key);
}

// Обход дерева в порядке inorder
void inorder() {
    inorderRec(root);
}

/* Функция для выполнения обхода дерева inorder */
void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.print(root.key + " ");
        inorderRec(root.right);
    }
}

// Обход дерева в порядке preorder
void preorder() {
    preorderRec(root);
}

/* Функция для выполнения обхода дерева preorder */
void preorderRec(Node root) {
    if (root != null) {
        System.out.print(root.key + " ");
        preorderRec(root.left);
        preorderRec(root.right);
    }
}

// Обход дерева в порядке postorder
```

```
    void postorder() {
        postorderRec(root);
    }

    /* Функция для выполнения обхода дерева postorder */
    void postorderRec(Node root) {
        if (root != null) {
            postorderRec(root.left);
            postorderRec(root.right);
            System.out.print(root.key + " ");
        }

    }
}

// Основная функция для тестирования
public static void main (String[] args) {
    BinarySearchTree tree = new BinarySearchTree();

    /* Вставка узлов */
    tree.insert(50);
    tree.insert(30);
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    /* Поиск узлов */
    System.out.println("Поиск 60: " + tree.search(60));
    System.out.println("Поиск 90: " + tree.search(90));

    /* Обходы дерева */
    System.out.println("Обход inorder:");
    tree.inorder();
    System.out.println();

    System.out.println("Обход preorder:");
    tree.preorder();
    System.out.println();

    System.out.println("Обход postorder:");
    tree.postorder();
    System.out.println();
```

```
    }
}
```

6. Разработайте программу, которая выводит буквы английского алфавита, используя цикл while в MySQL/PostgreSQL.

ЭТО ЕСЛИ ОН РАЗРЕШИТ ДЕЛАТЬ БЕЗ ДЖАВЫ.

```
CREATE TABLE alphabet_table (
    letter CHAR(1)
);

DO $$
DECLARE
    alphabet_char CHAR(1);
    start INTEGER := 65; -- кодировка символа 'A'
BEGIN
    WHILE start <= 90 LOOP
        SELECT chr(start) INTO alphabet_char;
        INSERT INTO alphabet_table (letter) VALUES (alphabet_char);
        start := start + 1;
    END LOOP;
END $$;

SELECT * FROM alphabet_table;
```

НА ДЖАВЕ:

```
package org.example;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Main {
    public static void main(String[] args) {
        String url = "jdbc:postgresql://localhost:5432/postgres";
        String user = "postgres";
        String password = "postgres";
        String query = "SELECT CHR(x) AS letter FROM
generate_series(65, 90) AS x";

        try (Connection connection = DriverManager.getConnection(url,
user, password);
             Statement statement = connection.createStatement();
```

```
             ResultSet resultSet = statement.executeQuery(query)) {

            System.out.println("Алфавит:");

            while (resultSet.next()) {
                String letter = resultSet.getString("letter");
                System.out.print(letter + " ");
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

POM.XML:

```
<dependencies>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.6.0</version> <!-- Замените на актуальную версию -->
    </dependency>
</dependencies>
```

7. Напишите программу, которая будет выводить в консоль введённое слово 6 раз и сохранять в MySQL/PostgreSQL.

сначала создаём бд postgres

```
CREATE DATABASE word_db;
\c word_db;

CREATE TABLE words (
    id SERIAL PRIMARY KEY,
    word VARCHAR(255) NOT NULL
);
```

Далее настраиваем pom.xml:

```
<dependencies>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <version>42.2.23</version>
    </dependency>
</dependencies>
```

код:

```
import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Scanner;

public class WordSaver {
     private static final String DB_URL = "jdbc:postgresql://localhost:5432/word_db";
    private static final String DB_USER = "postgres";
    private static final String DB_PASSWORD = "postgres";

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Введите слово: ");
        String word = scanner.nextLine();

        for (int i = 0; i < 6; i++) {
            System.out.println(word);
        }

        saveWordToDatabase(word);
    }

    private static void saveWordToDatabase(String word) {
        String insertSQL = "INSERT INTO words (word) VALUES (?)";

        try (Connection connection = DriverManager.getConnection(DB_URL, DB_USER,
DB_PASSWORD);
             PreparedStatement preparedStatement = connection.prepareStatement(insertSQL)) {

            preparedStatement.setString(1, word);
            preparedStatement.executeUpdate();

            System.out.println("Слово успешно сохранено в базу данных.");

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

8. Разработать программу для вывода на экран кубов первых десяти положительных чисел.

```
public class Cubes {
```

```
    public static void main(String[] args) {
        System.out.println("Кубы первых десяти положительных чисел:");
        for (int i = 1; i <= 10; i++) {
            int cube = i * i * i;
            System.out.println("Куб числа " + i + " равен " + cube);
        }
    }
}
```

9. Напишите программу, которая по дате определяет день недели, на который эта дата приходится.

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/time/DayOfWeek.html
https://metanit.com/java/tutorial/12.3.php

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.DayOfWeek;
import java.util.Scanner;

public class DayOfWeekCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Введите дату в формате ГГГГ-ММ-ДД: ");
        String inputDate = scanner.nextLine();
        //форматирование даты
        DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd");

        try {
            LocalDate date = LocalDate.parse(inputDate, formatter);

            //получаем день недели
            DayOfWeek dayOfWeek = date.getDayOfWeek();


            System.out.println("День недели для даты " + inputDate + ":
" + dayOfWeek);
        } catch (Exception e) {
            System.out.println("Ошибка: неверный формат даты.
Пожалуйста, используйте ГГГГ-ММ-ДД.");
        } finally {
            scanner.close();
```

```
        }
    }
}
```

10. Написать класс, который при введении даты в формате ДД.ММ.ГГ (к примеру, 22.10.20) выводит номер недели. Даты начиная с 2020 по 2022 годы. К примеру, первая неделя в 2020 году: 1-5 января, вторая неделя – 6-12 января. Значит при вводе 08.01.20 вывод должен быть: Неделя 2.

11. Разработайте программу, реализующую рекурсивное вычисление факториала.

```
public class Factorial {

    // Метод для рекурсивного вычисления факториала
    public static int factorial(int n) {
        // Базовый случай: факториал 0 и 1 равен 1
        if (n == 0 || n == 1) {
            return 1;
        }
        // Рекурсивный случай: n * факториал (n-1)
        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        // Пример использования метода factorial
        int number = 10;
        int result = factorial(number);
        System.out.println("Факториал числа " + number + " равен " +
result);
    }
}
```

12. Разработать класс-оболочку для числового типа double. Реализовать статические методы сложения, деления, возведения в степень.

```
public class DoubleWrapper {

   private double value;
```

```
    public DoubleWrapper(double value) {
        this.value = value;
    }

    public double getValue() {
        return value;
    }

    public void setValue(double value) {
        this.value = value;
    }

    public static double add(double a, double b) {
        return a + b;
    }

    public static double divide(double a, double b) throws
ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Делитель не может быть равен
нулю");
        }
        return a / b;
    }

    public static double power(double base, double exponent) {
        return Math.pow(base, exponent);
    }

    public static void main(String[] args) {
        DoubleWrapper dw1 = new DoubleWrapper(5.0);
        DoubleWrapper dw2 = new DoubleWrapper(3.0);

        System.out.println("Сложение: " +
DoubleWrapper.add(dw1.getValue(), dw2.getValue()));
        System.out.println("Деление: " +
DoubleWrapper.divide(dw1.getValue(), dw2.getValue()));
        System.out.println("Возведение в степень: " +
DoubleWrapper.power(dw1.getValue(), dw2.getValue()));
    }
}
```

13. Разработать программу, которая заполняет двумерный массив случайными положительными числами в диапазоне от 1 до 100 до тех пор, пока сумма граничных элементов не станет равной 666. Пользователь вначале вводит размер матрицы.

```
// Объект для получения строки с консоли
Scanner scan = new Scanner(System.in);
Random random = new Random();

// Получаем параметры размерности матрицы
System.out.println("Введите количество строк: ");
int rows = Integer.parseInt(scan.nextLine());
System.out.println("Введите количество столбцов: ");
int cols = Integer.parseInt(scan.nextLine());

// создаем пустую матрицу и переменную для сохранения суммы граничных
элементов
int[][] matrix = new int[rows][cols];
int sum = 0;

// Повторяем, пока сумма граничных элементов не станет равной 666
do {
    sum = 0;

    // Заполняем матрицу случайными числами
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = random.nextInt(100) + 1;
        }
    }
    // Вычисляем сумму граничных элементов
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            // Верхняя и нижняя границы
            if (i == 0 || i == rows - 1) {
                sum += matrix[i][j];
            }
            // Левые и правые границы (кроме уже учтенных углов)
            else if (j == 0 || j == cols - 1) {
                sum += matrix[i][j];
            }
        }
    }
} while (sum != 666);
```

```
// Выводим результат
System.out.println("Матрица с суммой граничных элементов 666:");
for (int i = 0; i < rows; i++) {
   for (int j = 0; j < cols; j++) {
       System.out.print(matrix[i][j] + "\t");
   }
   System.out.println();
}
System.out.println("Сумма граничных элементов: " + sum);
```

14. Разработать программу, в которой требуется создать класс, описывающий геометрическую фигуру – треугольник. Методами класса должны быть – вычисление площади, периметра. Создать класс-наследник, определяющий прямоугольный треугольник.

```
public class Triangle {

   int firstSide;
   int secondSide;
   int thirdSide;
   int sumSide;

   public Triangle(){}
   public Triangle(int firstSide, int secondSide, int thirdSide) {
       this.firstSide = firstSide;
       this.secondSide = secondSide;
       this.thirdSide = thirdSide;
   }

   public int P(){
       return firstSide +secondSide + thirdSide;
   }

   public double S(){
       int p = P()/2;
       return Math.sqrt(p*(p-firstSide)*(p-secondSide)*(p-thirdSide));
   }
}
```

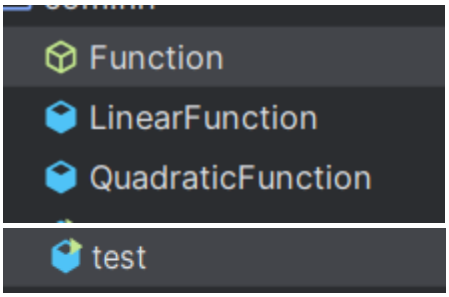
```
import java.awt.*;
import java.util.ArrayList;

public class TriS extends Triangle{
```

```
    private double base;
    private double height;

    public TriS(int firstSide, int secondSide, int thirdSide, double
base, double height) {
        super(firstSide, secondSide, thirdSide);
        this.base = base;
        this.height = height;
    }

    @Override
    public double S(){
        return (base * height) / 2;
    }

}
```

```
public class Main {

    public static void main(String[] args) {

        Triangle tri = new Triangle(3,4,5);
         System.out.println(tri.P());
        System.out.println(tri.S());

        TriS triS = new TriS(10,10,10,10, 20);
         System.out.println( triS.S());
        System.out.println(triS.P());

}
}
```

15. Разработать программу, в которой требуется создать абстрактный класс. В этом абстрактном классе определить абстрактные методы вычисления функции в определенной точке. Создать классы-наследники абстрактного класса, описывающими уравнения прямой и параболы. Программа должна выводить в консоль значение функции при вводе определенного значения.

Структура следующая:



1. Function - это главный абстрактный класс, в котором мы определяем абстрактный метод вычисления функции в точке x

```
abstract class Function {  1 usage  2 inheritors

    // Абстрактный метод для вычисления значения функции в определенной точке
    public abstract double evaluate(double x);  no usages  2 implementations
}
```

2. LinearFunction - это дочерний класс. Он наследует метод класса Function.

```
public class LinearFunction extends Function{  1 usage
    private final double a; // Коэффициент a в уравнении y = ax + b  2 usages
    private final double b; // Свободный член b в уравнении y = ax + b  2 usages

    public LinearFunction(double a, double b) {  1 usage
        this.a = a;
        this.b = b;
    }

    @Override  2 usages
    public double evaluate(double x) {
        return a * x + b;
    }
}
```

3. QuadraticFunction - это дочерний класс. Он наследует метод класса Function.

```java
public class QuadraticFunction extends Function{  1 usage
    private final double a; // Коэффициент a в уравнении y = ax^2 + bx + c  2 usages
    private final double b; // Коэффициент b в уравнении y = ax^2 + bx + c  2 usages
    private final double c; // Свободный член c в уравнении y = ax^2 + bx + c  2 usages

    public QuadraticFunction(double a, double b, double c) {  1 usage
        this.a = a;
        this.b = b;
        this.c = c;
    }

    @Override  2 usages
    public double evaluate(double x) {
        return a * Math.pow(x, 2) + b * x + c;
    }
}
```

4. test - здесь запускаем программу

```java
import java.util.Scanner;

public class test {
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);

        Function linearFunction = new LinearFunction( a: 2,  b: 3); // y = 2x + 3
        Function quadraticFunction = new QuadraticFunction( a: 1,  b: -3,  c: 2); // y = x^2 - 3x + 2

        System.out.print("Введите значение x: ");
        double x = sc.nextDouble();

        // Вычисление и вывод значений функций в точке x
        System.out.println("Значение линейной функции в точке " + x + ": " + linearFunction.evaluate(x));
        System.out.println("Значение квадратичной функции в точке " + x + ": " + quadraticFunction.evaluate
    }

}
```

19. Создать класс Matrix для работы с двумерными матрицами. Создать методы для генерации нулевой матрицы, а также для генерации матрицы со случайными величинами – применить Math.random(). Реализовать метод сложения матриц.

package org.example;

```
import java.util.Random;

public class Matrix {
    private int rows;
    private int cols;
    private int[][] matrix;

    public Matrix(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        this.matrix = new int[rows][cols];
    }

    public void generateZeroMatrix() {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = 0;
            }
        }
    }

    public void generateRandomMatrix() {
        Random random = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = random.nextInt(100);
            }
        }
    }

    public Matrix add(Matrix other) {
        if (this.rows != other.rows || this.cols != other.cols) {
            throw new IllegalArgumentException("Размеры матриц должны совпадать.");
        }

        Matrix result = new Matrix(this.rows, this.cols);
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result.matrix[i][j] = this.matrix[i][j] + other.matrix[i][j];
            }
        }
        return result;
    }
```

```
    public void displayMatrix() {
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(matrix[i][j] + "\t");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Matrix mat0 = new Matrix(5, 5);
        mat0.generateZeroMatrix();
        System.out.println("Нулевая матрица");
        mat0.displayMatrix();

        Matrix mat1 = new Matrix(3, 3);
        mat1.generateRandomMatrix();

        Matrix mat2 = new Matrix(3, 3);
        mat2.generateRandomMatrix();

        System.out.println("\nМатрица 1:");
        mat1.displayMatrix();

        System.out.println("\nМатрица 2:");
        mat2.displayMatrix();

        Matrix sumMatrix = mat1.add(mat2);
        System.out.println("\nСумма Матриц:");
        sumMatrix.displayMatrix();
    }
}
```

20. Реализовать класс MyMath для работы с числами. Реализовать статический метод класса MyMath.round(), который округляет дробь до целого числа. Также статический метод abs(), который находит модуль числа. Статический метод MyMath.pow() для нахождения степени числа. Библиотеку Math не использовать.

```
public class MyMath {

    // Метод для округления дроби до целого числа
    public static int round(double number) {
        int intPart = (int) number;
        double fractionPart = number - intPart;
```

```java
        if (fractionPart > 0.5) {
            return intPart + 1;
        } else if (fractionPart < -0.5) {
            return intPart - 1;
        } else {
            return intPart;
        }
    }

    // Метод для нахождения модуля числа
    public static int abs(int number) {
        return number < 0 ? -number : number;
    }

    // Метод для нахождения степени числа
    public static double pow(double base, int exponent) {
        double result = 1;
        boolean isNegative = exponent < 0;
        int absExponent = isNegative ? -exponent : exponent;

        for (int i = 0; i < absExponent; i++) {
            result *= base;
        }

        return isNegative ? 1 / result : result;
    }

    public static void main(String[] args) {
        // Примеры использования методов
        System.out.println("Round(2.3): " + MyMath.round(2.3)); // 2
        System.out.println("Round(2.7): " + MyMath.round(2.7)); // 3
        System.out.println("Round(-2.3): " + MyMath.round(-2.3)); // -2
        System.out.println("Round(-2.7): " + MyMath.round(-2.7)); // -3

        System.out.println("Abs(-5): " + MyMath.abs(-5)); // 5
        System.out.println("Abs(5): " + MyMath.abs(5)); // 5

        System.out.println("Pow(2, 3): " + MyMath.pow(2, 3)); // 8
        System.out.println("Pow(2, -3): " + MyMath.pow(2, -3)); //
0.125
    }
}
```

23. Разработать класс для представления комплексных чисел с возможностью задания вещественной и мнимой частей в виде массива из двух чисел типа int. Определить методы для выполнения операций сложения, вычитания и умножения комплексных чисел.

```
public class ComplexNumber {
   private int[] parts; // Массив для хранения вещественной и мнимой
частей

   // Конструктор
   public ComplexNumber(int real, int imaginary) {
       parts = new int[2];
       parts[0] = real; // Вещественная часть
       parts[1] = imaginary; // Мнимая часть
   }

   public int getReal() {
       return parts[0];
   }

   public int getImaginary() {
       return parts[1];
   }

   // Метод для сложения комплексных чисел
   public ComplexNumber add(ComplexNumber other) {
       int realPart = this.getReal() + other.getReal();
       int imaginaryPart = this.getImaginary() + other.getImaginary();
       return new ComplexNumber(realPart, imaginaryPart);
   }

   // Метод для вычитания комплексных чисел
   public ComplexNumber subtract(ComplexNumber other) {
       int realPart = this.getReal() - other.getReal();
       int imaginaryPart = this.getImaginary() - other.getImaginary();
       return new ComplexNumber(realPart, imaginaryPart);
   }

   // Метод для умножения комплексных чисел
   public ComplexNumber multiply(ComplexNumber other) {
       int realPart = this.getReal() * other.getReal() -
this.getImaginary() * other.getImaginary();
       int imaginaryPart = this.getReal() * other.getImaginary() +
this.getImaginary() * other.getReal();
       return new ComplexNumber(realPart, imaginaryPart);
```

```
    }

    // Переопределение метода toString для удобного вывода
    @Override
    public String toString() {
        return parts[0] + " + " + parts[1] + "i";
    }

    // Пример использования класса
    public static void main(String[] args) {
        ComplexNumber num1 = new ComplexNumber(3, 2); // 3 + 2i
        ComplexNumber num2 = new ComplexNumber(1, 7); // 1 + 7i

        ComplexNumber sum = num1.add(num2);
        ComplexNumber difference = num1.subtract(num2);
        ComplexNumber product = num1.multiply(num2);

        System.out.println("Сумма: " + sum);
        System.out.println("Разность: " + difference);
        System.out.println("Произведение: " + product);
    }
}
```

25. Сделайте класс User, в котором будут следующие protected поля - name (имя), age (возраст), public методы setName, getName, setAge, getAge. Сделайте класс Worker, который наследует от класса User и вносит дополнительное private поле salary (зарплата), а также методы public getSalary и setSalary. Создайте объект этого класса 'Иван', возраст 25, зарплата 1000. Создайте второй объект этого класса 'Вася', возраст 26, зарплата 2000. Найдите сумму зарплата Ивана и Васи. Сделайте класс Student, который наследует от класса User и вносит дополнительные private поля стипендия, курс, а также геттеры и сеттеры для них.

```
class User {
    protected String name;
    protected int age;

    //сеттеры и геттеры
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
```

```
        return name;
    }

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

    public int getAge() {
        return age;
    }
}

//класс Worker, наследующий от User
class Worker extends User {
    private double salary;

    //сеттеры и геттеры
    public void setSalary(double salary) {
        this.salary = salary;
    }
    public double getSalary() {
        return salary;
    }
}

//класс Student, наследующий от User
class Student extends User {
    private double scholarship;
    private int course;

    public void setScholarship(double scholarship) {
        this.scholarship = scholarship;
    }

    public double getScholarship() {
        return scholarship;
    }

    public void setCourse(int course) {
        this.course = course;
    }
    public int getCourse() {
        return course;
    }
}
```

```
public class Main {
   public static void main(String[] args) {
       Worker ivan = new Worker();
       ivan.setName("Иван");
       ivan.setAge(25);
       ivan.setSalary(1000);

       Worker vasya = new Worker();
       vasya.setName("Вася");
       vasya.setAge(26);
       vasya.setSalary(2000);

       //сумма зарплат Ивана и Васи
       double totalSalary = ivan.getSalary() + vasya.getSalary();

       System.out.println("Сумма зарплат Ивана и Васи: " +
totalSalary);

       Student student = new Student();
       student.setName("Алексей");
       student.setAge(20);
       student.setScholarship(500);
       student.setCourse(2);

       System.out.println("Студент: " + student.getName() + ",
возраст: " + student.getAge() +
               ", стипендия: " + student.getScholarship() + ", курс: "
+ student.getCourse());
   }
}
```

27. Создайте класс Number для конвертации десятичного числа в бинарный, восьмеричный, шестнадцатеричный вид. Реализовать в виде статических методов класса. Числа вводятся с клавиатуры с запросом в какой численный вид конвертировать.

29. Напишите программу, которая заполняет списочный массив случайными числами типа Integer (значения этих чисел были от 1 до 100). Список должен содержать 100 элементов. Затем отсортируйте по убыванию список и выведите первые 10 значений в консоль. Результаты сохраните в MySQL/PostgreSQL.

```
CREATE DATABASE number;
\c number;

CREATE TABLE random_numbers (
    id SERIAL PRIMARY KEY,
    value INT NOT NULL
);
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;

public class RandomNumberSorter {

    // Метод для заполнения списка случайными числами
    public static List<Integer> generateRandomNumbers(int size, int min,
int max) {
        List<Integer> numbers = new ArrayList<>();
        Random random = new Random();
        for (int i = 0; i < size; i++) {
            numbers.add(random.nextInt((max - min) + 1) + min);
        }
        return numbers;
```

```
    }

    // Метод для сохранения результатов в базе данных
    public static void saveToDatabase(List<Integer> numbers) {
        String url = "jdbc:postgresql://localhost:5432/number";
        String user = "postgres";
        String password = "postgres";

        String sql = "INSERT INTO random_numbers (value) VALUES (?)";

        try (Connection conn = DriverManager.getConnection(url, user,
password);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            for (int number : numbers) {
                pstmt.setInt(1, number);
                pstmt.addBatch();
            }
            pstmt.executeBatch();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        // Генерация списка случайных чисел
        List<Integer> numbers = generateRandomNumbers(100, 1, 100);

        // Сортировка списка по убыванию
        Collections.sort(numbers, Collections.reverseOrder());

        // Вывод первых 10 значений в консоль
        System.out.println("Первые 10 значений отсортированного списка:");
        for (int i = 0; i < 10; i++) {
            System.out.println(numbers.get(i));
        }

        // Сохранение результатов в базе данных
        saveToDatabase(numbers);
```

```
    }
}
```

30. Разработайте программу, которая заполняет список случайными числами. Количество элементов и числовой диапазон вводятся пользователем. Программа должна проверять, входит ли число (также вводится пользователем) в данный список. Должен быть реализован бинарный поиск. Результаты должны сохраняться в MySQL/PostgreSQL и выводиться оттуда же.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Random;
import java.util.Scanner;

public class RandomNumberSearch {

   private static final String DB_URL =
"jdbc:postgresql://localhost:5432/randomnumbersdb";
   private static final String DB_USER = "postgres";
   private static final String DB_PASSWORD = "your_password";

   public static void main(String[] args) {
       Scanner scanner = new Scanner(System.in);

       System.out.print("Введите количество элементов: ");
       int numElements = scanner.nextInt();

       System.out.print("Введите минимальное значение диапазона: ");
       int minRange = scanner.nextInt();

       System.out.print("Введите максимальное значение диапазона: ");
       int maxRange = scanner.nextInt();

       List<Integer> randomNumbers =
generateRandomNumbers(numElements, minRange, maxRange);
       Collections.sort(randomNumbers);
```

```
        System.out.print("Введите число для поиска: ");
        int searchNumber = scanner.nextInt();

        boolean found = binarySearch(randomNumbers, searchNumber);
        saveResultToDatabase(searchNumber, found);

        System.out.println("Результаты поиска:");
        printResultsFromDatabase();

        scanner.close();
    }

    private static List<Integer> generateRandomNumbers(int numElements,
int minRange, int maxRange) {
        List<Integer> randomNumbers = new ArrayList<>();
        Random random = new Random();
        for (int i = 0; i < numElements; i++) {
            randomNumbers.add(random.nextInt((maxRange - minRange) + 1)
+ minRange);
        }
        return randomNumbers;
    }

    private static boolean binarySearch(List<Integer> list, int target)
{
        int left = 0;
        int right = list.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (list.get(mid) == target) {
                return true;
            }
            if (list.get(mid) < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return false;
    }

    private static void saveResultToDatabase(int number, boolean found)
{
        String insertSQL = "INSERT INTO SearchResults (number, found)
VALUES (?, ?)";
```

```
        try (Connection conn = DriverManager.getConnection(DB_URL,
DB_USER, DB_PASSWORD);
             PreparedStatement pstmt =
conn.prepareStatement(insertSQL)) {
            pstmt.setInt(1, number);
            pstmt.setBoolean(2, found);
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void printResultsFromDatabase() {
        String selectSQL = "SELECT number, found FROM SearchResults";
        try (Connection conn = DriverManager.getConnection(DB_URL,
DB_USER, DB_PASSWORD);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(selectSQL)) {
            while (rs.next()) {
                int number = rs.getInt("number");
                boolean found = rs.getBoolean("found");
                System.out.println("Число: " + number + ", Найдено: " +
found);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Создание бд и таблицы:

```
CREATE DATABASE RandomNumbersDB;
CREATE TABLE SearchResults (
    id SERIAL PRIMARY KEY,
    number INT,
    found BOOLEAN
);
```

31. На основе класса BitSet разработайте программу для реализации битовых операций AND, OR, XOR, а также маскирования.

```java
import java.util.BitSet;

public class BitSetOperations {

    public static void main(String[] args) {
        // Создание и инициализация BitSet
        BitSet bitSet1 = new BitSet();
        bitSet1.set(0);
        bitSet1.set(1);
        bitSet1.set(2);
        bitSet1.set(3);

        BitSet bitSet2 = new BitSet();
        bitSet2.set(0);
        bitSet2.set(1);
        bitSet2.set(2);
        bitSet2.set(3);

        // Выполнение битовых операций
        BitSet andResult = (BitSet) bitSet1.clone();
        andResult.and(bitSet2);
        System.out.println("AND: " + andResult);

        BitSet orResult = (BitSet) bitSet1.clone();
        orResult.or(bitSet2);
        System.out.println("OR: " + orResult);

        BitSet xorResult = (BitSet) bitSet1.clone();
        xorResult.xor(bitSet2);
        System.out.println("XOR: " + xorResult);

        // Маскирование битов
        BitSet mask = new BitSet();
        mask.set(0);
        mask.set(1);
        mask.set(2);
        mask.set(3);

        BitSet maskedResult = (BitSet) bitSet1.clone();
        maskedResult.and(mask);
```

```
        System.out.println("Masked: " + maskedResult);
    }
}
```

32. Напишите программу, которая получает в качестве входных данных два числа. Эти числа являются количество строк и столбцов двумерной коллекции целых чисел. Далее элементы заполняются случайными числами и выводятся в консоль в виде таблицы.

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

public class RandomMatrix {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();

        // Получаем количество строк и столбцов от пользователя
        System.out.print("Введите количество строк: ");
        int rows = scanner.nextInt();
        System.out.print("Введите количество столбцов: ");
        int cols = scanner.nextInt();

        // Создаем двумерный массив
        int[][] matrix = new int[rows][cols];

        // Заполняем массив случайными числами
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                matrix[i][j] = random.nextInt(100); // Случайные числа
от 0 до 99
            }
        }

        // Выводим массив в виде таблицы
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(matrix[i][j] + "\t");
            }
            System.out.println();
        }

        scanner.close();
    }
```

```
}
```

33. Разработайте программу, которая получает в качестве параметра два числа – количество строк и столбцов двумерной коллекции целых чисел. Коллекция заполняется случайными числами, после чего на экран выводятся максимальное и минимальное значения с индексами ячеек.

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

public class TwoDimensionalArray {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Получение количества строк и столбцов от пользователя
        System.out.print("Введите количество строк: ");
        int rows = scanner.nextInt();
        System.out.print("Введите количество столбцов: ");
        int cols = scanner.nextInt();

        // Создание и заполнение двумерного массива случайными числами
        int[][] array = new int[rows][cols];
        Random random = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                array[i][j] = random.nextInt(100);
            }
        }

        // Поиск максимального и минимального значений и их индексов
        int maxValue = Integer.MIN_VALUE;
        int minValue = Integer.MAX_VALUE;
        int maxRow = -1, maxCol = -1;
        int minRow = -1, minCol = -1;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (array[i][j] > maxValue) {
                    maxValue = array[i][j];
```

```
                    maxRow = i;
                    maxCol = j;
                }
                if (array[i][j] < minValue) {
                    minValue = array[i][j];
                    minRow = i;
                    minCol = j;
                }
            }
        }

        // Вывод результатов
        System.out.println("Максимальное значение: " + maxValue + "
(строка: " + maxRow + ", столбец: " + maxCol + ")");
        System.out.println("Минимальное значение: " + minValue + "
(строка: " + minRow + ", столбец: " + minCol + ")");
    }
}
```

34. Разработайте программу, в которой создайте две коллекции с именами людей (строковые переменные). Результат сохранить в MySQL/PostgreSQL. Затем последовательно выводите в консоль имена.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class SaveNamesToPostgreSQL {
   // JDBC URL, username and password of MySQL server
   private static final String URL =
"jdbc:postgresql://localhost:5432/yourdatabase";
   private static final String USER = "postgres";
   private static final String PASSWORD = "postgres";

   public static void main(String[] args) {
       // Создаем две коллекции с именами людей
       List<String> namesList1 = new ArrayList<>();
```

```
        namesList1.add("Alice");
        namesList1.add("Bob");
        namesList1.add("Charlie");

        List<String> namesList2 = new ArrayList<>();
        namesList2.add("David");
        namesList2.add("Eve");
        namesList2.add("Frank");

        // Сохраняем имена в базу данных
        saveNamesToDatabase(namesList1);
        saveNamesToDatabase(namesList2);

        // Выводим имена в консоль
        printNamesFromDatabase();
    }

    private static void saveNamesToDatabase(List<String> names) {
        String sql = "INSERT INTO people (name) VALUES (?)";

        try (Connection conn = DriverManager.getConnection(URL, USER,
PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            for (String name : names) {
                pstmt.setString(1, name);
                pstmt.addBatch();
            }
            pstmt.executeBatch();

        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }

    private static void printNamesFromDatabase() {
        String sql = "SELECT name FROM people";

        try (Connection conn = DriverManager.getConnection(URL, USER,
PASSWORD);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {
                System.out.println(rs.getString("name"));
```

```
            }

        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

35. Напишите программу, которая реализует класс Matrix и следующие методы:
    a. Сложение и вычитание матриц.
    b. Умножение матрицы на число.
    c. Произведение двух матриц.
    d. Транспонированная матрица.
    e. Возведение матрицы в степень.
    f. Если метод, возвращает матрицу, то он должен возвращать новый объект, а не менять базовый.

```
public class Matrix {
    private int[][] data;

    // Конструктор для создания матрицы
    public Matrix(int[][] data) {
        this.data = data;
    }

    // Метод для получения размеров матрицы
    public int getRows() {
        return data.length;
    }

    public int getCols() {
        return data[0].length;
    }

    // Метод для сложения двух матриц
    public Matrix add(Matrix other) {
        int rows = getRows();
        int cols = getCols();
        int[][] result = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
```

```
            for (int j = 0; j < cols; j++) {
                result[i][j] = data[i][j] + other.data[i][j];
            }
        }
        return new Matrix(result);
    }

    // Метод для вычитания двух матриц
    public Matrix subtract(Matrix other) {
        int rows = getRows();
        int cols = getCols();
        int[][] result = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = data[i][j] - other.data[i][j];
            }
        }
        return new Matrix(result);
    }

    // Метод для умножения матрицы на число
    public Matrix multiply(int scalar) {
        int rows = getRows();
        int cols = getCols();
        int[][] result = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[i][j] = data[i][j] * scalar;
            }
        }
        return new Matrix(result);
    }

    // Метод для произведения двух матриц
    public Matrix multiply(Matrix other) {
        int rows = getRows();
        int cols = other.getCols();
        int common = getCols();
        int[][] result = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
```

```
            for (int j = 0; j < cols; j++) {
                for (int k = 0; k < common; k++) {
                    result[i][j] += data[i][k] * other.data[k][j];
                }
            }
        }
        return new Matrix(result);
    }

    // Метод для транспонирования матрицы
    public Matrix transpose() {
        int rows = getRows();
        int cols = getCols();
        int[][] result = new int[cols][rows];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                result[j][i] = data[i][j];
            }
        }
        return new Matrix(result);
    }

    // Метод для возведения матрицы в степень
    public Matrix power(int exponent) {
        Matrix result = this;
        Matrix base = this;
        for (int i = 1; i < exponent; i++) {
            result = result.multiply(base);
        }
        return result;
    }

    // Метод для вывода матрицы
    public void print() {
        int rows = getRows();
        int cols = getCols();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(data[i][j] + " ");
            }
```

```
            System.out.println();
        }
    }

    public static void main(String[] args) {
        int[][] data1 = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int[][] data2 = {
            {9, 8, 7},
            {6, 5, 4},
            {3, 2, 1}
        };

        Matrix matrix1 = new Matrix(data1);
        Matrix matrix2 = new Matrix(data2);

        System.out.println("Matrix 1:");
        matrix1.print();

        System.out.println("Matrix 2:");
        matrix2.print();

        System.out.println("Matrix 1 + Matrix 2:");
        matrix1.add(matrix2).print();

        System.out.println("Matrix 1 - Matrix 2:");
        matrix1.subtract(matrix2).print();

        System.out.println("Matrix 1 * 2:");
        matrix1.multiply(2).print();

        System.out.println("Matrix 1 * Matrix 2:");
        matrix1.multiply(matrix2).print();

        System.out.println("Transpose of Matrix 1:");
        matrix1.transpose().print();
```

```
        System.out.println("Matrix 1 ^ 2:");
        matrix1.power(2).print();
    }
}
```

37. Написать приложение для сложения, вычитания, умножения, деления, возведения в степень логарифмов. Программа должна выполнять ввод данных, проверку правильности введенных данных, выдачу сообщений в случае ошибок. Результат выводится на экран и записывается в файл.

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;

public class Calculator {
   public static void main(String[] args) {
       Scanner scanner = new Scanner(System.in);
       String fileName = "results.txt";

       try (PrintWriter writer = new PrintWriter(new
FileWriter(fileName, true))) {
           while (true) {
               System.out.println("Выберите операцию:");
               System.out.println("1. Сложение");
               System.out.println("2. Вычитание");
               System.out.println("3. Умножение");
               System.out.println("4. Деление");
               System.out.println("5. Возведение в степень");
               System.out.println("6. Логарифм");
               System.out.println("7. Выход");

               int choice = scanner.nextInt();

               if (choice == 7) {
                   break;
               }

               double result = 0;
               String operation = "";

               switch (choice) {
                   case 1:
```

```
                        result = add(scanner);
                        operation = "Сложение";
                        break;
                    case 2:
                        result = subtract(scanner);
                        operation = "Вычитание";
                        break;
                    case 3:
                        result = multiply(scanner);
                        operation = "Умножение";
                        break;
                    case 4:
                        result = divide(scanner);
                        operation = "Деление";
                        break;
                    case 5:
                        result = power(scanner);
                        operation = "Возведение в степень";
                        break;
                    case 6:
                        result = logarithm(scanner);
                        operation = "Логарифм";
                        break;
                    default:
                        System.out.println("Неверный выбор. Пожалуйста,
попробуйте снова.");
                        continue;
                }

                System.out.println("Результат: " + result);
                writer.println(operation + ": " + result);
            }
        } catch (IOException e) {
            System.out.println("Ошибка записи в файл: " +
e.getMessage());
        }

        scanner.close();
    }

    private static double add(Scanner scanner) {
        System.out.print("Введите первое число: ");
        double a = scanner.nextDouble();
        System.out.print("Введите второе число: ");
        double b = scanner.nextDouble();
```

```
        return a + b;
    }

    private static double subtract(Scanner scanner) {
        System.out.print("Введите первое число: ");
        double a = scanner.nextDouble();
        System.out.print("Введите второе число: ");
        double b = scanner.nextDouble();
        return a - b;
    }

    private static double multiply(Scanner scanner) {
        System.out.print("Введите первое число: ");
        double a = scanner.nextDouble();
        System.out.print("Введите второе число: ");
        double b = scanner.nextDouble();
        return a * b;
    }

    private static double divide(Scanner scanner) {
        System.out.print("Введите первое число: ");
        double a = scanner.nextDouble();
        System.out.print("Введите второе число: ");
        double b = scanner.nextDouble();
        if (b == 0) {
            System.out.println("Ошибка: деление на ноль.");
            return 0;
        }
        return a / b;
    }

    private static double power(Scanner scanner) {
        System.out.print("Введите основание: ");
        double base = scanner.nextDouble();
        System.out.print("Введите показатель: ");
        double exponent = scanner.nextDouble();
        return Math.pow(base, exponent);
    }

    private static double logarithm(Scanner scanner) {
        System.out.print("Введите число: ");
        double number = scanner.nextDouble();
        if (number <= 0) {
            System.out.println("Ошибка: логарифм от неположительного
числа не определен.");
```

```
            return 0;
        }
        return Math.log(number);
    }
}
```

38. Разработать программу шифровки-дешифровки по алгоритму AES-128. Данные берутся из файла, зашифрованные данные сохраняются в указанный файл.

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;

public class AES128Encryption {

    // Метод для генерации ключа AES-128
    public static SecretKey generateKey() throws NoSuchAlgorithmException
{
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(128);
        return keyGen.generateKey();
    }

    // Метод для шифрования данных
    public static byte[] encrypt(byte[] data, SecretKey key) throws
Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        return cipher.doFinal(data);
    }
```

```
    // Метод для дешифрования данных
    public static byte[] decrypt(byte[] data, SecretKey key) throws
Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.DECRYPT_MODE, key);
        return cipher.doFinal(data);
    }

    // Метод для чтения данных из файла
    public static byte[] readFile(String filePath) throws IOException {
        return Files.readAllBytes(Paths.get(filePath));
    }

    // Метод для записи данных в файл
    public static void writeFile(String filePath, byte[] data) throws
IOException {
        Files.write(Paths.get(filePath), data);
    }

    public static void main(String[] args) {
        try {
            // Генерация ключа
            SecretKey key = generateKey();

            // Чтение данных из файла
            String inputFilePath = "input.txt";
            byte[] inputData = readFile(inputFilePath);

            // Шифрование данных
            byte[] encryptedData = encrypt(inputData, key);

            // Сохранение зашифрованных данных в файл
            String encryptedFilePath = "encrypted.txt";
            writeFile(encryptedFilePath, encryptedData);

            // Дешифрование данных
            byte[] decryptedData = decrypt(encryptedData, key);

            // Сохранение дешифрованных данных в файл
```

```
            String decryptedFilePath = "decrypted.txt";
            writeFile(decryptedFilePath, decryptedData);

            System.out.println("Шифрование и дешифрование завершены
успешно.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

39. Разработать программу нахождения наибольшего общего делителя двух натуральных чисел. Требуется реализовать рекурсивный и без рекурсии варианты. Результат сохранить в MySQL/PostgreSQL.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Scanner;

public class GCDCalculator {
   // JDBC URL, username and password of PostgreSQL server
   private static final String URL =
"jdbc:postgresql://localhost:5432/yourdatabase";
   private static final String USER = "postgres";
   private static final String PASSWORD = "postgres";

   public static void main(String[] args) {
       Scanner scanner = new Scanner(System.in);

       System.out.print("Введите первое натуральное число: ");
       int a = scanner.nextInt();
       System.out.print("Введите второе натуральное число: ");
       int b = scanner.nextInt();

       int gcdRecursive = gcdRecursive(a, b);
       int gcdIterative = gcdIterative(a, b);

       System.out.println("НОД (рекурсивный метод): " + gcdRecursive);
       System.out.println("НОД (итеративный метод): " + gcdIterative);
```

```
        saveResultToDatabase(a, b, gcdRecursive, gcdIterative);

        scanner.close();
    }

    // Рекурсивный метод для нахождения НОД
    private static int gcdRecursive(int a, int b) {
        if (b == 0) {
            return a;
        }
        return gcdRecursive(b, a % b);
    }

    // Итеративный метод для нахождения НОД
    private static int gcdIterative(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    // Метод для сохранения результатов в базу данных
    private static void saveResultToDatabase(int a, int b, int
gcdRecursive, int gcdIterative) {
        String sql = "INSERT INTO gcd_results (a, b, gcd_recursive,
gcd_iterative) VALUES (?, ?, ?, ?)";

        try (Connection conn = DriverManager.getConnection(URL, USER,
PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setInt(1, a);
            pstmt.setInt(2, b);
            pstmt.setInt(3, gcdRecursive);
            pstmt.setInt(4, gcdIterative);
            pstmt.executeUpdate();

        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
CREATE TABLE gcd_results (
    id SERIAL PRIMARY KEY,
    a INT NOT NULL,
    b INT NOT NULL,
    gcd_recursive INT NOT NULL,
    gcd_iterative INT NOT NULL
);
```

Сложные:

1. Условие: «Реализовать программу для выполнения следующих математических операций с целочисленным, байтовым и вещественным типами данных: сложение, вычитание, умножение, деление, деление по модулю (остаток), модуль числа, возведение в степень. Все данные вводятся с клавиатуры (класс Scanner, System.in, nextint).» По данному условию необходимо реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). При этом в программе данные пункты должны называться следующим образом:

1. Вывести все таблицы из MySQL.
2. Создать таблицу в MySQL.
3. Сложение чисел, результат сохранить в MySQL с последующим выводом в консоль.
4. Вычитание чисел, результат сохранить в MySQL с последующим выводом в консоль.
5. Умножение чисел, результат сохранить в MySQL с последующим выводом в консоль.
6. Деление чисел, результат сохранить в MySQL с последующим выводом в консоль.
7. Деление чисел по модулю (остаток), результат сохранить в MySQL с последующим выводом в консоль.
8. Возведение числа в модуль, результат сохранить в MySQL с последующим выводом в консоль.
9. Возведение числа в степень, результат сохранить в MySQL с последующим выводом в консоль.
10. Сохранить все данные (вышеполученные результаты) из MySQL в Excel и вывести на экран. (https://www.geeksforgeeks.org/working-with-microsoft-excel-using-apache-poi-and-jexcel-api-with-a-maven-project-in-java/)

Объяснение некоторых методов и типов:

1) ResultSet возвращает **результаты выполнения SQL-запроса над базой данных**. Он представляет собой таблицу данных, где каждая строка представляет запись, а каждый столбец — поле в базе данных. Используется для возвращения метаданных бд. Метод `getTables` из класса `DatabaseMetaData` возвращает информацию о таблицах в базе данных. Синтаксис:

ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[]

types), где
`catalog`: Каталог базы данных. Если не нужно фильтровать по каталогу, можно передать `null`.
`schemaPattern`: Шаблон схемы базы данных. Если фильтр не нужен, тоже можно передать `null`.
`tableNamePattern`: Шаблон названия таблицы. Символ `%` означает «все таблицы» (аналогично `LIKE '%'` в SQL).
`types`: Массив строк, указывающий, какие типы объектов нужны. Например, `"TABLE"` для таблиц, `"VIEW"` для представлений.
Первое `null`: Мы не фильтруем по каталогу, поэтому указываем `null`.
Второе `null`: Мы не фильтруем по схеме, поэтому тоже указываем `null`.
`\"%\"`: Указывает, что мы хотим получить все таблицы (аналог `SELECT *`).
`new String[]{\"TABLE\"}`: Указывает, что нас интересуют только объекты типа «таблицы».
2) Использование BinaryOperator. это **функциональный интерфейс в Java**, который представляет операцию, принимающую два параметра и возвращающую одно значение. Оба параметра и тип возврата должны быть одного типа.
`.apply()` — это абстрактный метод функционального интерфейса `BinaryOperator` из Java, который применяется для выполнения операции над двумя однотипными аргументами и возвращает результат того же типа. Лямбда-функции: (один из параметров метода apply) (a, b) -> a * b;

```
package org.example;

import java.sql.*;
import java.util.Scanner;

public class MathOperations {

   private static final String URL =
"jdbc:postgresql://localhost:5434/exam";
   private static final String USER = "postgres";
   private static final String PASSWORD = "root";

   public static void main(String[] args) {
       try (Connection connection = DriverManager.getConnection(URL,
USER, PASSWORD);
            Scanner scanner = new Scanner(System.in)) {

           System.out.println("Подключение к базе данных
установлено.");
```

```
            boolean running = true;

            while (running) {
                System.out.println("\nВыберите действие:");
                System.out.println("1. Вывести все таблицы");
                System.out.println("2. Создать таблицу");
                System.out.println("3. Сложение чисел");
                System.out.println("4. Вычитание чисел");
                System.out.println("5. Умножение чисел");
                System.out.println("6. Деление чисел");
                System.out.println("7. Деление по модулю");
                System.out.println("8. Модуль числа");
                System.out.println("9. Возведение в степень");
                System.out.println("0. Выход");

                int choice = scanner.nextInt();
                switch (choice) {
                    case 1:
                        listTables(connection);
                        break;
                    case 2:
                        createTable(connection);
                        break;
                    case 3:
                        performOperation(connection, scanner,
"Сложение", Double::sum);
                        break;
                    case 4:
                        performOperation(connection, scanner,
"Вычитание", (a, b) -> a - b);
                        break;
                    case 5:
                        performOperation(connection, scanner,
"Умножение", (a, b) -> a * b);
                        break;
                    case 6:
                        performOperation(connection, scanner,
"Деление", (a, b) -> a / b);
                        break;
                    case 7:
                        performOperation(connection, scanner, "Деление
по модулю", (a, b) -> a % b);
                        break;
                    case 8:
```

```
                        performSingleOperation(connection, scanner,
"Модуль", Math::abs);
                        break;
                    case 9:
                        performOperation(connection, scanner,
"Возведение в степень", Math::pow);
                        break;
                    case 0:
                        running = false;
                        break;
                    default:
                        System.out.println("Некорректный выбор.
Попробуйте снова.");
                }
            }
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    private static void listTables(Connection connection) throws
SQLException {
        DatabaseMetaData metaData = connection.getMetaData();
        ResultSet tables = metaData.getTables(null, null, "%", new
String[]{"TABLE"});
        System.out.println("Список таблиц в базе данных:");
        while (tables.next()) {
            System.out.println(tables.getString("TABLE_NAME"));
        }
    }
    private static void createTable(Connection connection) throws
SQLException {
        String createTableSQL = "CREATE TABLE IF NOT EXISTS
math_results ("
                + "id SERIAL PRIMARY KEY,"
                + "operation VARCHAR(50),"
                + "operand1 NUMERIC,"
                + "operand2 NUMERIC,"
                + "result NUMERIC"
                + ");";
        try (Statement statement = connection.createStatement()) {
            statement.execute(createTableSQL);
            System.out.println("Таблица math_results успешно создана
(или уже существует).");
        }
```

```
    }

    private static void saveResult(Connection connection, String
operationName, Double operand1, Double operand2, Double result) throws
SQLException {
        String insertSQL = "INSERT INTO math_results (operation,
operand1, operand2, result) VALUES (?, ?, ?, ?);";
        try (PreparedStatement preparedStatement =
connection.prepareStatement(insertSQL)) {
            preparedStatement.setString(1, operationName);
            preparedStatement.setObject(2, operand1);
            preparedStatement.setObject(3, operand2);
            preparedStatement.setObject(4, result);
            preparedStatement.executeUpdate();
        }
    }

    private static void performOperation(Connection connection, Scanner
scanner, String operationName, BinaryOperator operation) throws
SQLException {
        System.out.println("Введите первое число:");
        double operand1 = scanner.nextDouble();
        System.out.println("Введите второе число:");
        double operand2 = scanner.nextDouble();
        double result = operation.apply(operand1, operand2);

        saveResult(connection, operationName, operand1, operand2,
result);
        System.out.println(operationName + " результат: " + result);
    }

    private static void performSingleOperation(Connection connection,
Scanner scanner, String operationName, UnaryOperator operation) throws
SQLException {
        System.out.println("Введите число:");
        double operand = scanner.nextDouble();
        double result = operation.apply(operand);

        saveResult(connection, operationName, operand, null, result);
        System.out.println(operationName + " результат: " + result);
    }

    @FunctionalInterface
    interface BinaryOperator {
        double apply(double a, double b);
```

```
    }

    @FunctionalInterface
    interface UnaryOperator {
        double apply(double a);
    }
}
```

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>Exam</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
    </properties>
    <dependencies>
        <!--
https://mvnrepository.com/artifact/org.postgresql/postgresql -->
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.7.3</version>
        </dependency>
    </dependencies>
</project>
```

4. Написать калькулятор для строковых выражений вида "<число> <операция> <число>", где <число> - положительное целое число меньшее 10, записанное словами, например, "четыре", <арифметическая операция> - одна из операций "плюс", "минус", "умножить". Результат выполнения операции вернуть в виде текстового представления числа. Пример: "пять плюс четыре" --> "девять".

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

```
public class Main {
   static int number = 0;
   static String[] numbers = new String[]{"ноль", "один", "два",
"три", "четыре", "пять", "шесть", "семь", "восемь", "девять"};
  static Map<String, String> numberWords = new HashMap<>();
   public static int getNumber(String num) {
       for (int i = 0; i < 10; i++) {
           if (numbers[i].equals(num)) {
               return i;
           }
       }
       System.out.println("Такого числа нет!");
       return -1;

   }

   public static void getResult(int result, String num1, String num2,
String oper){
       if (result > 20) {//если больше 20 так как потом едет сложение
20 и 1 = 21 30 и 5 =35
           char[] nums = Integer.toString(result).toCharArray();
//разбиваем число на цифры - первое деястки
           String firstNum = nums[0] + "0"; //прибавляем к строке с
десятком ноль ->  получаем 20, 30 и тд
           String secondNum = String.valueOf(nums[1]); // цифру
сохраянем как есть, 4, 5 или 9
           String firstN = numberWords.get(firstNum);
           String secondN = numberWords.get(secondNum);
           System.out.println(num1 + " " + oper + " на " + num2 + " =
" + firstN + " " + secondN);
       } else {
           String resultStr = Integer.toString(result);
           String res = numberWords.get(resultStr);
           System.out.println(num1 + " " + oper + " на " + num2 + " =
" + res);
       }
   }


   public static void main(String[] args) {

       int result = 0;

       numberWords.put("0", "ноль");
```

```
        numberWords.put("1", "один");
        numberWords.put("2", "два");
        numberWords.put("3", "три");
        numberWords.put("4", "четыре");
        numberWords.put("5", "пять");
        numberWords.put("6", "шесть");
        numberWords.put("7", "семь");
        numberWords.put("8", "восемь");
        numberWords.put("9", "девять");
        numberWords.put("10", "десять");
        numberWords.put("11", "одиннадцать");
        numberWords.put("12", "двенадцать");
        numberWords.put("13", "тринадцать");
        numberWords.put("14", "четырнадцать");
        numberWords.put("15", "пятнадцать");
        numberWords.put("16", "шестнадцать");
        numberWords.put("17", "семнадцать");
        numberWords.put("18", "восемнадцать");
        numberWords.put("19", "девятнадцать");
        numberWords.put("20", "двадцать");
        numberWords.put("30", "тридцать");
        numberWords.put("40", "сорок");
        numberWords.put("50", "пятьдесят");
        numberWords.put("60", "шестьдесят");
        numberWords.put("70", "семьдесят");
        numberWords.put("80", "восемьдесят");

        Scanner scanner = new Scanner(System.in);
        System.out.println("Введите первое число(строчно)");
        String num1 = scanner.nextLine().toLowerCase();
        System.out.println("Введите второе число(строчно)");
        String num2 = scanner.nextLine().toLowerCase();
        System.out.println("Введите операнда:");
        String oper = scanner.nextLine().toLowerCase();

        int numberOne = getNumber(num1);
        int numberTwo = getNumber(num2);

        if (Objects.equals(oper, "умножить")) {
            result = numberOne * numberTwo; // получаем числовой резалт
            getResult(result, num1, num2, oper);
        }
        if (Objects.equals(oper, "плюс")) {
            result = numberOne + numberTwo; // получаем числовой резалт
            getResult(result, num1, num2, oper);
```

```
        }
        if (Objects.equals(oper, "минус")) {
            result = numberOne - numberTwo; // получаем числовой резалт
            if (result < 0) {
                System.out.println("С минусами не работаем.");
            } else {
                getResult(result, num1, num2, oper);
            }
        }
    }
}
```

16. Создать интерфейс Progress с методами вычисления любого элемента прогрессии и суммы прогрессии. Разработать классы арифметической и геометрической прогрессии, которые имплементируют интерфейс Progress.

17. Разработать интерфейс InArray, в котором предусмотреть метод сложения двух массивов. Создать класс ArraySum, в котором имплементируется метод сложения массивов. Создать класс OrArray, в котором метод сложения массивов имплементируется как логическая операция ИЛИ между элементами массива.

InArray:

```
public interface InArray {
    int[] addArrays(int[] array1, int[] array2);
}
```

OrArray:

```
public class OrArray implements InArray {
    @Override
    public int[] addArrays(int[] array1, int[] array2) {
        int length = Math.min(array1.length, array2.length);
        int[] result = new int[length];
        for (int i = 0; i < length; i++) {
            result[i] = array1[i] | array2[i];
        }
        return result;
    }
}
```

ArraySum:

```
public class ArraySum implements InArray {
    @Override
```

```
    public int[] addArrays(int[] array1, int[] array2) {
        int length = Math.min(array1.length, array2.length);
        int[] result = new int[length];
        for (int i = 0; i < length; i++) {
            result[i] = array1[i] + array2[i];
        }
        return result;
    }
}
```

Main:

```
public class Main {
    public static void main(String[] args) {
        int[] array1 = {1, 2, 3};
        int[] array2 = {4, 5, 6};

        // ArraySum
        InArray sumImpl = new ArraySum();
        int[] sumResult = sumImpl.addArrays(array1, array2);
        System.out.println("Результат сложения массивов:");
        for (int num : sumResult) {
            System.out.print(num + " ");
        }

        System.out.println();

        // OrArray
        InArray orImpl = new OrArray();
        int[] orResult = orImpl.addArrays(array1, array2);
        System.out.println("Результат побитового сложения:");
        for (int num : orResult) {
            System.out.print(num + " ");
        }
    }
}
```

21. Разработать программу для игры «Угадайка». Программа загадывает случайное число от 1 до 10, требуется его отгадать с трех попыток. После каждой попытки, если результат неверен, игроку выводится сообщение, меньше или больше названное игроком число, чем загаданное. Сет заканчивается или если игрок угадывает число, или если исчерпывает три попытки, не угадав. Игра должна быть выполнена в бесконечном цикле, и продолжается до тех пор, пока на предложение «Сыграем еще раз?» игрок не напишет «Нет».

```
import java.util.Random;
```

```java
import java.util.Scanner;

public class GuessingGame {

   public static void main(String[] args) {
       Scanner scanner = new Scanner(System.in);
       Random random = new Random();
       String playAgain;

       do {
           int secretNumber = random.nextInt(10) + 1; // Генерация
случайного числа от 1 до 10
           int attempts = 3;
           boolean guessedCorrectly = false;

           System.out.println("Я загадал число от 1 до 10. Попробуй
угадать его за 3 попытки.");

           for (int i = 0; i < attempts; i++) {
               System.out.print("Попытка " + (i + 1) + ": Введите ваше
предположение: ");
               int guess = scanner.nextInt();

               if (guess == secretNumber) {
                   System.out.println("Поздравляю! Вы угадали
число!");
                   guessedCorrectly = true;
                   break;
               } else if (guess < secretNumber) {
                   System.out.println("Загаданное число больше.");
               } else {
                   System.out.println("Загаданное число меньше.");
               }
           }

           if (!guessedCorrectly) {
               System.out.println("К сожалению, вы не угадали число.
Загаданное число было: " + secretNumber);
           }

           System.out.print("Сыграем еще раз? (Введите 'Нет' для
выхода): ");
           playAgain = scanner.next();

       } while (!playAgain.equalsIgnoreCase("Нет"));
```

```
        System.out.println("Спасибо за игру! До свидания!");
        scanner.close();
    }
}
```

22. Разработайте программу-генератор рабочего календаря. Слесарь механосборочного цеха работает сутки через трое. Если смена попадает на воскресенье, то переносится на понедельник. По введенной дате программа должна генерировать расписание из дат на текущий месяц на 2022 год.

```
public class WorkScheduleGenerator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Введите дату (в формате ГГГГ-ММ-ДД): ");
        String inputDate = scanner.nextLine();

        // Парсим введенную дату
        LocalDate startDate = LocalDate.parse(inputDate);
        int year = 2022; // Устанавливаем год на 2022
        Month month = startDate.getMonth();

        // Генерируем расписание
        List<LocalDate> schedule = generateWorkSchedule(year, month);

        // Выводим расписание
        System.out.println("Рабочий календарь на " + month + " " + year
+ ":");
        for (LocalDate date : schedule) {
            System.out.println(date);
        }

        scanner.close();
    }

    private static List<LocalDate> generateWorkSchedule(int year, Month
month) {
        List<LocalDate> schedule = new ArrayList<>();
        LocalDate currentDate = LocalDate.of(year, month, 1);

        // Находим последний день месяца
        LocalDate lastDate =
currentDate.withDayOfMonth(currentDate.lengthOfMonth());
```

```
        // Начинаем с первого рабочего дня
        LocalDate workDay = currentDate;

        while (!workDay.isAfter(lastDate)) {
            // Если это воскресенье, переносим на понедельник
            if (workDay.getDayOfWeek() == DayOfWeek.SUNDAY) {
                workDay = workDay.plusDays(1);
            }

            // Добавляем рабочий день в расписание
            schedule.add(workDay);

            // Переходим к следующему рабочему дню (через 3 дня)
            workDay = workDay.plusDays(3);
        }

        return schedule;
    }
}
```

24. Создайте класс Form - оболочку для создания и ввода пароля. Он должен иметь методы input, submit, password. Создайте класс SmartForm, который будет наследовать от Form и сохранять значения password.

```
public class Form {
    private String input; // Поле для хранения ввода
    private String password; // Поле для хранения пароля

    //Метод для ввода данных
    public void input(String data) {
        this.input = data;
        System.out.println("Введенные данные: " + this.input);
    }

    //Метод для установки пароля
    public void password(String password) {
        this.password = password;
        System.out.println("Пароль установлен.");
    }

    //Метод для отправки формы
    public void submit() {
        System.out.println("Форма отправлена с данными: " +
this.input);
```

```
    }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class SmartForm extends Form {
    private List<String> savedPasswords; //Список для хранения паролей

    public SmartForm() {
        savedPasswords = new ArrayList<>(); //Инициализация списка
    }

    @Override
    public void password(String password) {
        super.password(password); //Вызов метода родительского класса
        savedPasswords.add(password); //Сохранение пароля в список
        System.out.println("Пароль сохранен: " + password);
    }

    //Метод для получения сохраненных паролей
    public List<String> getSavedPasswords() {
        return savedPasswords;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Form form = new Form();
        form.input("Пример ввода");
        form.password("12345");
        form.submit();

        System.out.println();

        SmartForm smartForm = new SmartForm();
        smartForm.input("Данные для умной формы");
        smartForm.password("password1");
        smartForm.password("password2");
        smartForm.submit();
```

```
        System.out.println("Сохраненные пароли: " +
smartForm.getSavedPasswords());
    }
}
```

36. Разработать программу для поочередной обработки текстовых файлов. Файлы созданы со следующими именами: n.txt, где n – натуральное число. В файлах записаны: в первой строке одно число с плавающей запятой, во второй строке – второе число. Пользователь вводит название файла и требуемую операцию над числами (сложение, умножение, разность). Результат выводится на экран и файл n_out.txt.

40. Напишите программу, которая каждые 5 секунд отображает на экране данные о времени, прошедшем от начала запуска программы, а другой её поток выводит сообщение каждые 7 секунд. Третий поток выводит на экран сообщение каждые 10 секунд. Программа работает одну минуту, затем останавливается. Все результаты после вывода необходимо сохранить в MySQL/PostgreSQL.

```
CREATE TABLE results (
    id SERIAL PRIMARY KEY,
    message VARCHAR(255),
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);


import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class MultiThreadedApp {

    private static final String DB_URL =
"jdbc:postgresql://localhost:5432/result";
    private static final String DB_USER = "postgres";
    private static final String DB_PASSWORD = "postgres";
```

```
    public static void main(String[] args) {
        ScheduledExecutorService scheduler =
Executors.newScheduledThreadPool(3);

        // Выводит время каждые 5 секунд
        scheduler.scheduleAtFixedRate(() -> {
            long currentTime = System.currentTimeMillis() - startTime;
            String message = "Время прошло: " + currentTime / 1000 + "
секунд";
            System.out.println(message);
            saveToDatabase(message);
        }, 0, 5, TimeUnit.SECONDS);

        // Выводит сообщение каждые 7 секунд
        scheduler.scheduleAtFixedRate(() -> {
            String message = "Сообщение каждые 7 секунд";
            System.out.println(message);
            saveToDatabase(message);
        }, 0, 7, TimeUnit.SECONDS);

        // Выводит сообщение каждые 10 секунд
        scheduler.scheduleAtFixedRate(() -> {
            String message = "Сообщение каждые 10 секунд";
            System.out.println(message);
            saveToDatabase(message);
        }, 0, 10, TimeUnit.SECONDS);

        // Останавливаем программу через 1 минуту
        scheduler.schedule(() -> {
            scheduler.shutdown();
            System.out.println("Программа остановлена");
        }, 60, TimeUnit.SECONDS);
    }

    private static long startTime = System.currentTimeMillis();

    private static void saveToDatabase(String message) {
        String sql = "INSERT INTO results (message) VALUES (?)";
        try (Connection conn = DriverManager.getConnection(DB_URL,
DB_USER, DB_PASSWORD);
```

```
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setString(1, message);
            pstmt.executeUpdate();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

45. Разработка веб-MVC приложения на основе Spring Boot. Приложение должно генерировать последовательность из 1000 случайных чисел в диапазоне, заданном пользователем, и выводит эти числа на экран и вычисляет их среднее арифметическое.

Основной класс:

```
package org.example.thousandgeneration;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ThousandGenerationApplication {

   public static void main(String[] args) {
       SpringApplication.run(ThousandGenerationApplication.class,
args);
   }

}
```

Контроллер:

```
package org.example.thousandgeneration;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

import java.util.Random;

@Controller
```

```java
public class RandomNumberController {

    @GetMapping("/")
    public String index() {
        return "index";
    }

    @GetMapping("/generate")
    public String generateRandomNumbers(@RequestParam int min,
@RequestParam int max, Model model) {
        int[] randomNumbers = new int[1000];
        Random random = new Random();
        double sum = 0;

        for (int i = 0; i < 1000; i++) {
            randomNumbers[i] = random.nextInt((max - min) + 1) + min;
            sum += randomNumbers[i];
        }

        double average = sum / 1000;

        model.addAttribute("randomNumbers", randomNumbers);
        model.addAttribute("average", average);

        return "result";
    }
}
```

index.html:

```html
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Генератор случайных чисел</title>
</head>
<body>
<h1>Генератор случайных чисел</h1>
<form action="/generate" method="get">
    <label for="min">Нижняя граница:</label>
    <input type="number" id="min" name="min" required><br><br>
    <label for="max">Верхняя граница:</label>
    <input type="number" id="max" name="max" required><br><br>
    <input type="submit" value="Сгенерировать">
</form>
</body>
</html>
```

result.html:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
   <title>Результат генерации</title>
</head>
<body>
<h1>Сгенерированные числа</h1>
<p>Среднее: <span th:text="${average}"></span></p>
<ul>
   <li th:each="number : ${randomNumbers}" th:text="${number}"></li>
</ul>
<a href="/">Generate New Numbers</a>
</body>
</html>
```

46. Разработать приложение для работы с локальной базой данных MySQL. Создайте базу данных мобильных телефонов (не менее 10 позиций), со следующими полями: производитель, модель, год выпуска, диагональ экрана. Напишите методы для выполнения запросов к базе данных. Все данные должны выводиться в консоли на экран.

```
CREATE DATABASE mobile_store;

USE mobile_store;

CREATE TABLE mobile_phones (
    id SERIAL PRIMARY KEY,
    manufacturer VARCHAR(50),
    model VARCHAR(50),
    release_year INT,
    screen_size DECIMAL(3,1)
);

INSERT INTO mobile_phones (manufacturer, model, release_year,
screen_size) VALUES
('Apple', 'iPhone 14', 2022, 6.1),
('Samsung', 'Galaxy S23', 2023, 6.2),
('Xiaomi', 'Mi 11', 2021, 6.8),
('Google', 'Pixel 7', 2022, 6.3),
('OnePlus', '10 Pro', 2022, 6.7),
('Sony', 'Xperia 1 IV', 2022, 6.5),
('Huawei', 'P50 Pro', 2021, 6.6),
```

```
('Nokia', 'G60', 2022, 6.6),
('Realme', 'GT 2 Pro', 2022, 6.7),
('Asus', 'ROG Phone 6', 2022, 6.8);

Это просто код для постгреса, чтобы были данные

package org.example;

import java.sql.*;

public class PhonesBD {

   private static final String URL =
"jdbc:postgresql://localhost:5434/exam";
   private static final String USER = "postgres";
   private static final String PASSWORD = "root";

   public static void main(String[] args) {
       try (Connection connection = DriverManager.getConnection(URL,
USER, PASSWORD)) {
           System.out.println("Соединение с базой данных
установлено!");

           // Вывод всех записей из таблицы
           printAllPhones(connection);

           // Добавление нового телефона
           addPhone(connection, "Motorola", "Edge 30", 2022, 6.55);

           // Обновление записи
           updatePhone(connection, 1, "Apple", "iPhone 15", 2023,
6.7);

           // Удаление записи
           deletePhone(connection, 2);

           // Повторный вывод всех записей
           printAllPhones(connection);

       } catch (SQLException e) {
           System.err.println("Ошибка при работе с базой данных: " +
e.getMessage());
       }
   }
```

```
    // Вывод всех телефонов
    public static void printAllPhones(Connection connection) throws
SQLException {
        String query = "SELECT * FROM mobile_phones";
        try (Statement statement = connection.createStatement();
             ResultSet resultSet = statement.executeQuery(query)) {

            System.out.println("Список мобильных телефонов:");
            while (resultSet.next()) {
                        int id = resultSet.getInt("id");
                        String manf =
resultSet.getString("manufacturer");
                        String model = resultSet.getString("model");
                        int year = resultSet.getInt("release_year");
                        double size =
resultSet.getDouble("screen_size");
                System.out.println("ID:" + id + " Производитель: " +
manf + " Модель: " + model + " Год выпуска: " + year + " Диагональ: "+
size);
            }
        }
    }

    // Добавление нового телефона
    public static void addPhone(Connection connection, String
manufacturer, String model, int releaseYear, double screenSize) throws
SQLException {
        String query = "INSERT INTO mobile_phones (manufacturer, model,
release_year, screen_size) VALUES (?, ?, ?, ?)";
        try (PreparedStatement preparedStatement =
connection.prepareStatement(query)) {
            preparedStatement.setString(1, manufacturer);
            preparedStatement.setString(2, model);
            preparedStatement.setInt(3, releaseYear);
            preparedStatement.setDouble(4, screenSize);
            preparedStatement.executeUpdate();
            System.out.println("Телефон добавлен: " + manufacturer + "
" + model);
        }
    }

    // Обновление записи
    public static void updatePhone(Connection connection, int id,
String manufacturer, String model, int releaseYear, double screenSize)
throws SQLException {
```

```
        String query = "UPDATE mobile_phones SET manufacturer = ?,
model = ?, release_year = ?, screen_size = ? WHERE id = ?";
        try (PreparedStatement preparedStatement =
connection.prepareStatement(query)) {
            preparedStatement.setString(1, manufacturer);
            preparedStatement.setString(2, model);
            preparedStatement.setInt(3, releaseYear);
            preparedStatement.setDouble(4, screenSize);
            preparedStatement.setInt(5, id);
            preparedStatement.executeUpdate();
            System.out.println("Телефон обновлён: ID " + id);
        }
    }

    // Удаление записи
    public static void deletePhone(Connection connection, int id)
throws SQLException {
        String query = "DELETE FROM mobile_phones WHERE id = ?";
        try (PreparedStatement preparedStatement =
connection.prepareStatement(query)) {
            preparedStatement.setInt(1, id);
            preparedStatement.executeUpdate();
            System.out.println("Телефон с ID " + id + " удалён.");
        }
    }
}
```

18. Создать класс Binary для работы с двоичными числами фиксированной длины. Число должно быть массивом тип char, каждый элемент которого принимает значение 0 или 1. Младший бит имеет младший индекс. Отрицательные числа представляются в дополнительном коде. Дополнительный код получается инверсией всех битов с прибавлением 1 к младшему биту. Например, +1 – это в двоичном коде будет выглядеть, как 0000 0001. А -1 в двоичном коде будет выглядеть, как 1111 1110 + 0000 0001 = 1111 1111. Создать методы конвертации десятичного числа в массив и обратно.

```
public class Main {

    public static void main(String[] args) {
        Binary binaryNumber = new Binary(8); // Длина числа — 8 бит

        int decimalNumber = -5;

        binaryNumber.fromDecimal(decimalNumber);
```

```
        System.out.println("Двоичное представление: " +
binaryNumber.display());

        int convertedBackToDecimal = binaryNumber.toDecimal(); //
Предполагается, что метод toDecimal реализован

        System.out.println("Преобразованное обратно в десятичное: " +
convertedBackToDecimal);

    }
}
```

```
public class Binary  {

    int length;
    char[] number;


    public Binary(int length) {
        this.length = length;
        this.number = new char[length];
    }

    public char[] fromDecimal(int num) {

        boolean isNegative = num < 0;
        if (isNegative) {
            num = Math.abs(num);
        }

        for (int i = 0; i < length; i++) {
            number[i] = (char) ('0' + (num % 2)); // Остаток от деления
на 2 (0 или 1)
            num /= 2; // Делим число на 2
        }

        if (isNegative) {
            toTwoComplement();
        }
        return number;
    }

    public int toDecimal() {
        int decimal = 0;
```

```
        // Проверяем, является ли число отрицательным (старший бит
равен '1')
        boolean isNegative = number[length - 1] == '1';

        // Если число отрицательное, преобразуем из дополнительного
кода
        if (isNegative) {
            fromTwoComplement(); // Преобразуем из дополнительного кода
        }

        // Вычисляем десятичное значение
        for (int i = length - 1; i >= 0; i--) {
            decimal = decimal * 2 + (number[i] - '0'); // Умножаем на
основание (2) и добавляем текущий бит
        }

        // Если число было отрицательным, возвращаем его с минусом
        return isNegative ? -decimal : decimal;
    }
    private void toTwoComplement() {
        // Инверсия всех битов
        for (int i = 0; i < length; i++) {
            number[i] = (number[i] == '0') ? '1' : '0';
        }

        // Прибавляем 1 к младшему биту
        for (int i = 0; i < length; i++) {
            if (number[i] == '0') {
                number[i] = '1';
                break;
            } else {
                number[i] = '0'; // Переносим единицу дальше
            }
        }
    }
    private void fromTwoComplement() {
        // Прибавляем 1 к числу
        for (int i = 0; i < length; i++) {
            if (number[i] == '1') {
                number[i] = '0';
            } else {
                number[i] = '1';
                break;
            }
```

```
        }

        // Инверсия всех битов
        for (int i = 0; i < length; i++) {
            number[i] = (number[i] == '0') ? '1' : '0';
        }
    }
    public String display() {
        StringBuilder sb = new StringBuilder();
        for (int i = length - 1; i >= 0; i--) { // Обратный порядок для
отображения
            sb.append(number[i]);
        }
        return sb.toString();
    }
}
```

26. Создайте класс ColorModel для определения цветовой модели. Разработайте подклассы RGBconverter и CMYKconverter для конвертации цвета из одной модели в другую. Конвертация CMYK в RGB производится по следующим формулам: R = 255 × (1-C) × (1-K), G = 255 × (1-M) × (1-K), B = 255 × (1-Y) × (1-K) (где R – red, G – green, B – black, C – Cyan, M - Magenta, Y - Yellow, K- Black))

Базовый класс `ColorModel`

```
public abstract class ColorModel {
    public abstract void convert();
}
```

Подкласс `RGBconverter`

```
public class RGBconverter extends ColorModel {
    private int red;
    private int green;
    private int blue;

    public RGBconverter(int red, int green, int blue) {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }
```

```
    @Override
    public void convert() {
        double r = red / 255.0;
        double g = green / 255.0;
        double b = blue / 255.0;

        double k = 1 - Math.max(Math.max(r, g), b);
        double c = (1 - r - k) / (1 - k);
        double m = (1 - g - k) / (1 - k);
        double y = (1 - b - k) / (1 - k);

        System.out.println("CMYK: C=" + c + ", M=" + m + ", Y=" + y +
", K=" + k);
    }
}
```

Подкласс `CMYKconverter`

```
public class CMYKconverter extends ColorModel {
    private double cyan;
    private double magenta;
    private double yellow;
    private double black;

    public CMYKconverter(double cyan, double magenta, double yellow,
double black) {
        this.cyan = cyan;
        this.magenta = magenta;
        this.yellow = yellow;
        this.black = black;
    }

    @Override
    public void convert() {
        int r = (int) (255 * (1 - cyan) * (1 - black));
        int g = (int) (255 * (1 - magenta) * (1 - black));
        int b = (int) (255 * (1 - yellow) * (1 - black));

        System.out.println("RGB: R=" + r + ", G=" + g + ", B=" + b);
    }
}
```

28. Разработать класс Neuron для реализации нейронной сети из двух нейронов и одного выхода. Сделать функцию прямого распространения с функцией активации в виде сигмоиды.

41. Условие задачи: «Ввести две строки (не менее 50 символов каждая) с клавиатуры. Необходимо вывести на экран две введенных ранее строки, подсчитать и вывести размер длины каждой строки, объединить данные строки в одну, сравнить данные строки и результат сравнения вывести на экран». По данному условию необходимо реализовать программу с интерактивным консольным меню, (т.е. вывод списка действий по цифрам. При этом при нажатии на цифру у нас должно выполняться определенное действие). При этом в программе данные пункты должны называться следующим образом:

1. Вывести все таблицы из MySQL.

2. Создать таблицу в MySQL.

3. Ввести две строки с клавиатуры, результат сохранить в MySQL с последующим выводом в консоль.

4. Подсчитать размер ранее введенных строк, результат сохранить в MySQL с последующим выводом в консоль.

5. Объединить две строки в единое целое, результат сохранить в MySQL с последующим выводом в консоль.

6. Сравнить две ранее введенные строки, результат сохранить в MySQL с последующим выводом в консоль.

7. Сохранить все данные (выше полученные результаты) из MySQL в Excel и вывести на экран.

```
package org.example;

import java.sql.*;
import java.util.Scanner;
import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;
import java.io.FileOutputStream;

public class Main {
   private static final String DB_URL =
"jdbc:postgresql://localhost:5432/postgres";
```

```
    private static final String DB_USER = "postgres";
    private static final String DB_PASSWORD = "postgres";
    private static final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        while (true) {
            System.out.println("\nВыберите действие:");
            System.out.println("1. Вывести все таблицы из PostgreSQL");
            System.out.println("2. Создать таблицу в PostgreSQL");
            System.out.println("3. Ввести две строки с клавиатуры,
результат сохранить в PostgreSQL с выводом");
            System.out.println("4. Подсчитать длины строк, результат
сохранить в PostgreSQL с выводом");
            System.out.println("5. Объединить строки, результат
сохранить в PostgreSQL с выводом");
            System.out.println("6. Сравнить строки, результат сохранить
в PostgreSQL с выводом");
            System.out.println("7. Сохранить данные из PostgreSQL в
Excel и вывести");
            System.out.println("0. Выход");

            int choice = scanner.nextInt();
            scanner.nextLine();

            switch (choice) {
                case 1 -> showTables();
                case 2 -> createTable();
                case 3 -> insertStrings();
                case 4 -> calculateStringLengths();
                case 5 -> concatenateStrings();
                case 6 -> compareStrings();
                case 7 -> saveToExcel();
                case 0 -> {
                    System.out.println("Выход из программы.");
                    System.exit(0);
                }
                default -> System.out.println("Неверный выбор.
Попробуйте снова.");
            }
        }
    }

    // 1. Вывести все таблицы из PostgreSQL
    private static void showTables() {
```

```
        try (Connection connection =
DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
             Statement statement = connection.createStatement();
             ResultSet resultSet = statement.executeQuery(
                     "SELECT table_name FROM information_schema.tables
WHERE table_schema = 'public'")) {

            System.out.println("Список таблиц:");
            while (resultSet.next()) {
                System.out.println(resultSet.getString(1));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // 2. Создать таблицу в PostgreSQL
    private static void createTable() {
        String createTableSQL = """
                CREATE TABLE IF NOT EXISTS strings_data (
                    id SERIAL PRIMARY KEY,
                    string1 TEXT,
                    string2 TEXT,
                    length1 INT,
                    length2 INT,
                    concatenated_string TEXT,
                    comparison_result TEXT
                )
                """;

        try (Connection connection =
DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
             Statement statement = connection.createStatement()) {

            statement.execute(createTableSQL);
            System.out.println("Таблица создана или уже существует.");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // 3. Ввести две строки с клавиатуры, результат сохранить в
PostgreSQL с выводом
    private static void insertStrings() {
```

```
        System.out.print("Введите первую строку (не менее 50 символов):
");
        String string1 = scanner.nextLine();
        System.out.print("Введите вторую строку (не менее 50 символов):
");
        String string2 = scanner.nextLine();

        try (Connection connection =
DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
             PreparedStatement statement = connection.prepareStatement(
                     "INSERT INTO strings_data (string1, string2)
VALUES (?, ?)")) {

            statement.setString(1, string1);
            statement.setString(2, string2);
            statement.executeUpdate();

            System.out.println("Строки успешно сохранены в
PostgreSQL.");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // 4. Подсчитать длины строк, результат сохранить в PostgreSQL с
выводом
    private static void calculateStringLengths() {
        try (Connection connection =
DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
             Statement statement = connection.createStatement();
             ResultSet resultSet = statement.executeQuery("SELECT *
FROM strings_data ORDER BY id DESC LIMIT 1")) {

            if (resultSet.next()) {
                String string1 = resultSet.getString("string1");
                String string2 = resultSet.getString("string2");
                int length1 = string1.length();
                int length2 = string2.length();

                System.out.println("Длина первой строки: " + length1);
                System.out.println("Длина второй строки: " + length2);

                PreparedStatement updateStatement =
connection.prepareStatement(
```

```
                        "UPDATE strings_data SET length1 = ?, length2 =
? WHERE id = ?");
                updateStatement.setInt(1, length1);
                updateStatement.setInt(2, length2);
                updateStatement.setInt(3, resultSet.getInt("id"));
                updateStatement.executeUpdate();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // 5. Объединить строки, результат сохранить в PostgreSQL с выводом
    private static void concatenateStrings() {
        try (Connection connection =
DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
             Statement statement = connection.createStatement();
             ResultSet resultSet = statement.executeQuery("SELECT *
FROM strings_data ORDER BY id DESC LIMIT 1")) {

            if (resultSet.next()) {
                String concatenated = resultSet.getString("string1") +
resultSet.getString("string2");
                System.out.println("Объединенные строки: " +
concatenated);

                PreparedStatement updateStatement =
connection.prepareStatement(
                        "UPDATE strings_data SET concatenated_string =
? WHERE id = ?");
                updateStatement.setString(1, concatenated);
                updateStatement.setInt(2, resultSet.getInt("id"));
                updateStatement.executeUpdate();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // 6. Сравнить строки, результат сохранить в PostgreSQL с выводом
    private static void compareStrings() {
        try (Connection connection =
DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
             Statement statement = connection.createStatement();
```

```
            ResultSet resultSet = statement.executeQuery("SELECT *
FROM strings_data ORDER BY id DESC LIMIT 1")) {

            if (resultSet.next()) {
                String string1 = resultSet.getString("string1");
                String string2 = resultSet.getString("string2");
                String comparisonResult = string1.equals(string2) ?
"Равны" : "Не равны";

                System.out.println("Результат сравнения строк: " +
comparisonResult);

                PreparedStatement updateStatement =
connection.prepareStatement(
                        "UPDATE strings_data SET comparison_result = ?
WHERE id = ?");
                updateStatement.setString(1, comparisonResult);
                updateStatement.setInt(2, resultSet.getInt("id"));
                updateStatement.executeUpdate();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // 7. Сохранить данные из PostgreSQL в Excel
    private static void saveToExcel() {
        String excelFilePath = "strings_data.xlsx";

        try (Connection connection =
DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
             Statement statement = connection.createStatement();
             ResultSet resultSet = statement.executeQuery("SELECT *
FROM strings_data");
             Workbook workbook = new XSSFWorkbook();
             FileOutputStream outputStream = new
FileOutputStream(excelFilePath)) {

            Sheet sheet = workbook.createSheet("Данные строк");
            Row header = sheet.createRow(0);

            header.createCell(0).setCellValue("ID");
            header.createCell(1).setCellValue("Строка 1");
            header.createCell(2).setCellValue("Строка 2");
            header.createCell(3).setCellValue("Длина 1");
```

```
            header.createCell(4).setCellValue("Длина 2");
            header.createCell(5).setCellValue("Объединенная строка");
            header.createCell(6).setCellValue("Результат сравнения");

            int rowIndex = 1;
            while (resultSet.next()) {
                Row row = sheet.createRow(rowIndex++);
                row.createCell(0).setCellValue(resultSet.getInt("id"));

row.createCell(1).setCellValue(resultSet.getString("string1"));

row.createCell(2).setCellValue(resultSet.getString("string2"));

row.createCell(3).setCellValue(resultSet.getInt("length1"));

row.createCell(4).setCellValue(resultSet.getInt("length2"));

row.createCell(5).setCellValue(resultSet.getString("concatenated_strin
g"));

row.createCell(6).setCellValue(resultSet.getString("comparison_result"
));
            }

            workbook.write(outputStream);
            System.out.println("Данные успешно сохранены в " +
excelFilePath);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

POM.XML

```
<dependencies>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>42.7.5</version>
    </dependency>
    <dependency>
      <groupId>org.apache.poi</groupId>
      <artifactId>poi-ooxml</artifactId>
```

```
        <version>5.3.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.23.1</version>
    </dependency>
</dependencies>
```

42. Написать на основе Spring Boot клиент-серверное приложение MyUser, в котором можно управлять данными пользователей из базы данных через веб-интерфейс: имя, фамилия, возраст, номер группы. База данных может быть любой – MySQL, PostgreSQL и т.д. При этом должна быть доступна возможность добавления/удаления/редактирования пользователей.
Зависимости в spring проекте: Spring Web, Spring Data JPA, PostgreSQL Driver.

1. Создать бд в postgresql `myuserdb`
2. application.properties

```
spring.application.name=user
spring.datasource.url=jdbc:postgresql://localhost:5432/myuserdb
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

3. Код

```
@Entity
@Table(name = "my_user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private int age;
    private String groupNumber;

    // Getters and Setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
```

```
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
    }

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

    public String getGroupNumber() {
        return groupNumber;
    }

    public void setGroupNumber(String groupNumber) {
        this.groupNumber = groupNumber;
    }
}
```

```
@SpringBootApplication
public class UserApplication {

    public static void main(String[] args) {
       SpringApplication.run(UserApplication.class, args);
    }

}
```

```
@RestController
```

```java
@RequestMapping("/api/users")
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping
    public List<User> getAllUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public Optional<User> getUserById(@PathVariable Long id) {
        return userService.getUserById(id);
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.saveUser(user);
    }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User
userDetails) {
        User user = userService.getUserById(id).orElseThrow();
        user.setFirstName(userDetails.getFirstName());
        user.setLastName(userDetails.getLastName());
        user.setAge(userDetails.getAge());
        user.setGroupNumber(userDetails.getGroupNumber());
        return userService.saveUser(user);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.deleteUser(id);
    }
}
```

```java
public interface UserRepository extends JpaRepository<User, Long> {
}
```

```java
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
```

```
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }

    public User saveUser(User user) {
        return userRepository.save(user);
    }

    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

4. В папке resources/static

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>MyUser App</title>
    <script
src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
    <style>
        .user-item {
            margin-bottom: 10px;
        }
        .user-item button {
            margin-left: 10px;
        }
    </style>
</head>
<body>
<h1>User Management</h1>
<div>
    <h2>Add User</h2>
    <form id="add-user-form">
        <input type="text" id="firstName" placeholder="First Name"
required><br>
        <input type="text" id="lastName" placeholder="Last Name"
required><br>
```

```
        <input type="number" id="age" placeholder="Age" required><br>
        <input type="text" id="groupNumber" placeholder="Group Number"
required><br>
        <button type="submit">Add User</button>
    </form>
</div>
<div id="user-list"></div>

<script>
    function loadUsers() {
        axios.get('/api/users')
            .then(response => {
                const users = response.data;
                const userList = document.getElementById('user-list');
                userList.innerHTML = '';
                users.forEach(user => {
                    const userDiv = document.createElement('div');
                    userDiv.className = 'user-item';
                    userDiv.innerHTML = `
                        <p>${user.firstName} ${user.lastName} -
${user.age} - ${user.groupNumber}</p>
                        <button
onclick="editUser(${user.id})">Edit</button>
                        <button
onclick="deleteUser(${user.id})">Delete</button>
                    `;
                    userList.appendChild(userDiv);
                });
            })
            .catch(error => {
                console.error('There was an error fetching the users!',
error);
            });
    }

    function addUser(event) {
        event.preventDefault();
        const firstName = document.getElementById('firstName').value;
        const lastName = document.getElementById('lastName').value;
        const age = document.getElementById('age').value;
        const groupNumber =
document.getElementById('groupNumber').value;

        axios.post('/api/users', {
            firstName: firstName,
```

```
            lastName: lastName,
            age: age,
            groupNumber: groupNumber
        })
        .then(response => {
            loadUsers();
            document.getElementById('add-user-form').reset();
        })
        .catch(error => {
            console.error('There was an error adding the user!',
error);
        });
    }

    function editUser(id) {
        const firstName = prompt('Enter new first name:');
        const lastName = prompt('Enter new last name:');
        const age = prompt('Enter new age:');
        const groupNumber = prompt('Enter new group number:');

        axios.put(`/api/users/${id}`, {
            firstName: firstName,
            lastName: lastName,
            age: age,
            groupNumber: groupNumber
        })
        .then(response => {
            loadUsers();
        })
        .catch(error => {
            console.error('There was an error editing the user!',
error);
        });
    }

    function deleteUser(id) {
        axios.delete(`/api/users/${id}`)
            .then(response => {
                loadUsers();
            })
            .catch(error => {
                console.error('There was an error deleting the user!',
error);
            });
    }
```

```
   document.getElementById('add-user-form').addEventListener('submit',
addUser);

   loadUsers();
</script>
</body>
</html>
```

5. Запускать по адресу **http://localhost:8080**

43. Написать на основе Spring Boot Security форму для авторизации и регистрации пользователя. При этом после авторизации пользователя должно быть перенаправление на главную страницу. Главная страница должна содержать запись «Hello World!». При этом до авторизации главная страница не должна быть доступна для пользователя.

```
//CustomUserDetailsService

package org.example.task_43;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;



@Service
public class CustomUserDetailsService implements UserDetailsService {
    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        return userRepository.findByUsername(username)
                .orElseThrow(() -> new UsernameNotFoundException("User not
found: " + username));
    }
}
```

```
```
// Task43Application

package org.example.task_43;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Task43Application {

    public static void main(String[] args) {
        SpringApplication.run(Task43Application.class, args);
    }

}

```
```
//UserController

package org.example.task_43;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;

@Controller
public class UserController {
    @Autowired
    private UserService userService;

    @GetMapping("/login")
    public String login() {
        return "login";
    }
```
```

```
    @GetMapping("/register")
    public String registerForm(Model model) {
        model.addAttribute("user", new Users());
        return "register";
    }

    @PostMapping("/register")
    public String register(Users user) {
        userService.register(user);
        return "redirect:/login";
    }

    @GetMapping("/")
    public String home() {
        return "home";
    }
}
```
```
// UserRepository

package org.example.task_43;

import org.springframework.data.jpa.repository.JpaRepository;
import java.util.Optional;

public interface UserRepository extends JpaRepository<Users, Long> {
    Optional<Users> findByUsername(String username);
}

```
```
// Users

package org.example.task_43;

import jakarta.persistence.*;
import lombok.*;
import org.springframework.security.core.GrantedAuthority;
```

```
import org.springframework.security.core.userdetails.UserDetails;

import java.util.Collection;
import java.util.List;

@Data
@Entity
public class Users implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String password;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of();
    }
}

```
```
//UserService

package org.example.task_43;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

@Service
public class UserService {
    @Autowired
    private UserRepository repo;
    @Autowired
    private PasswordEncoder passwordEncoder;
```

```
    public void register(Users user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        repo.save(user);
    }
}

```
```
// WebSecurityConfig

package org.example.task_43;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builders.Aut
henticationManagerBuilder;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWeb
Security;
import
org.springframework.security.config.annotation.web.configurers.AbstractHtt
pConfigurer;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.beans.factory.annotation.Autowired;


@Configuration
@EnableWebSecurity
public class WebSecurityConfig {
    @Autowired
    private UserDetailsService userDetailsService;
```

```
    @Bean
    public static PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {
        http
                .csrf(AbstractHttpConfigurer::disable) // Отключение CSRF
для простоты (не рекомендуется для production)
                .authorizeHttpRequests(authorize -> authorize
                        .requestMatchers("/register",
"/login").permitAll() // Доступ к этим URL без авторизации
                        .anyRequest().authenticated() // Остальные запросы
требуют авторизации
                )
                .formLogin(form -> form
                        .loginPage("/login")
                        .defaultSuccessUrl("/", true)
                        .permitAll()
                )
                .logout(logout -> logout
                        .logoutUrl("/logout") // URL для выхода
                        .logoutSuccessUrl("/login") // Перенаправление
после выхода
                        .permitAll()
                );
        return http.build();
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
        auth
                .userDetailsService(userDetailsService)
                .passwordEncoder(passwordEncoder());
    }
}
```

```
```
HOME

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Home</title>
</head>
<body>
<h1>Hello World!</h1>
<a th:href="@{/logout}">Logout</a>
</body>
</html>
```

```
```
LOGIN

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Login</title>
</head>
<body>
<form th:action="@{/login}" method="post">
    <div>
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>
    </div>
    <div>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" required>
    </div>
    <button type="submit">Login</button>
</form>
<p>Don't have an account? <a th:href="@{/register}">Register here</a></p>
</body>
</html>
```

```
```
REGISTER

<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Register</title>
</head>
<body>
<form th:action="@{/register}" method="post">

        <label for="username">Username:</label>
        <input type="text" id="username" name="username">

        <label for="password">Password:</label>
        <input type="password" id="password" name="password">

    <button type="submit">Register</button>
</form>
</body>
</html>


```
```
// application.properties

spring.application.name=task_43

spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=admin
spring.datasource.password=1234

spring.jpa.properties.hibernate.dialect =
org.hibernate.dialect.PostgreSQLDialect

spring.jpa.hibernate.ddl-auto=update
```
```

```
#spring.security.user.name=root
#spring.security.user.password=root
#spring.security.user.roles=manager


spring.cache.type=none
spring.thymeleaf.cache=false

spring.web.resources.add-mappings=true
server.port=5000

```

44. Разработать MVC-приложение арифметический калькулятор на основе Spring Boot. Применить шаблонизатор Thymeleaf. Все результаты вычисления должны сохраняться и выводиться из MySQL.