1. Реализация асинхронного выполнения в Джава.

В java есть многопоточное программирование, в котором задачи выполняются разными thread’ми, потоками, но при этом, при выполнении какой-то задачи в synchronized блоке, монитор (или семафор или мьютекс (двоичный семафор)) закрывается, и выполнение этой части кода становится недоступным для других потоков, поэтому появляется простой программы, а асинхронное программирование позволяет избежать данной проблемы. Благодаря асинхронному программированию компьютер выполняет задачи, не дожидаясь завершения каждой из них, прежде чем начинать следующую. Объяснение с примерами можно посмотреть здесь: <https://clck.ru/3BdP4j>

1. Паттерн “Одиночка” (Singleton) и его использование в Джава программах.

Данный паттерн используется по умолчанию в Spring Framework, когда мы создаём бины с помощью специализированных аннотаций, или специальных методов в конфигурации, или при помощи xml конфигурации, и заключается он в том, что создать объект конкретного класса можно только один, для этого часто используется приватный конструктор. Чтобы создать объект класса, который использует данный паттерн, нужно вызвать определённый СТАТИЧЕСКИЙ метод. Простой пример можно привести с регулярными выражениями, а именно, чтобы создать объект класса Pattern, необходимо вызвать статический метод compile, который вернёт новый объект класса Pattern, ведь конструктор данного класса приватный.

Кстати говоря, противоположный паттерну синглтон можно назвать паттерн Prototype, когда каждый раз создаётся новый объект.

Пример реализации:

|  |
| --- |
| public class Singleton {  // Статический экземпляр класса  private static Singleton instance;  // Конструктор класса  private Singleton() {}  // Метод для получения экземпляра класса  public static synchronized Singleton getInstance() {  if (instance == null) {  instance = new Singleton();  }  return instance;  }  } |

1. Использование статических методов для создания экземпляра объекта вместо конструкторов.

Было описано выше, обычно статические методы используются для того, чтобы реализовывать паттерн синглтон (одиночка).

|  |
| --- |
| public class MyClass {  private static MyClass instance;  private MyClass() {  // Конструктор класса  }  public static MyClass getInstance() {  if (instance == null) {  instance = new MyClass();  }  return instance;  }  public void doSomething() {  // Логика действия  }  }  public class Main {  public static void main(String[] args) {  // Получение экземпляра класса через статический метод  MyClass myClassInstance = MyClass.getInstance();  // Вызов метода экземпляра  myClassInstance.doSomething();  }  } |

1. Паттерн “Строитель” и его использование большом количестве параметров конструктора.

Данный паттерн относится к порождающим паттернам проектирования и призван упросить “строительство” сложного объекта, путём вызова цепочки методов, чем-то напоминает Stream API.

Пример использования:

- порядок создания сложного объекта не должен зависеть от того, из каких частей состоит объект и того, как эти объекты взаимосвязаны;

- процесс конструирования объекта должен предоставлять различные представления объекта, который мы конструируем.

Также можно отметить, что в библиотеке Lombok есть все необходимые аннотации для упрощения разработки, в том числе, и аннотация @Builder.

1. Понятие dependency injection (внедрение зависимости). Преимущества использования внедрения зависимостей при написании программ на языке Джава. Преимущества этого подхода перед паттерном Одиночка.

Внедрение зависимостей одно из ключевых понятий, очень и очень часто используется при работе со Spring Framework, позволяет внедрять объекты какого-либо класса в другой объект, и для этого могут использоваться:

* Сеттеры.
* Конструкторы.
* Непосредственное внедрение через рефлексию для какого-либо поля.
* Задавать определённые factory методы, и внедрять через них.

При неправильно подобранной архитектуре, могут появляться так называемые циклические зависимости, когда для создания объекта класса Class1 необходим экземпляр класса Class2, а также для создания объекта класса Class2 необходим экземпляр класса Class1. При таком случае лучшем решением будет изменить архитектуру проектирования.

В сравнении с паттерном Одиночка, внедрение зависимостей предоставляет более гибкий и масштабируемый подход к управлению зависимостями в системе, ключевые особенности перед паттерном singleton.

* Улучшение модульности. DI помогает разделить ответственность между различными частями системы, улучшая ее модульность и упрощая понимание и поддержку кода.
* Повышение читаемости и поддерживаемости кода. Код, использующий внедрение зависимостей, становится более читаемым, поскольку явно указывает на зависимости между компонентами.
* Отказ от жестко заданной структуры. DI представляет собой форму инверсии управления, которая позволяет отделить создание объектов от их использования, что снижает связность кода и увеличивает его гибкость.
* Легче управлять жизненным циклом объектов. Использование контейнеров внедрения зависимостей позволяет более тонко управлять жизненным циклом объектов, например, управлять временем их создания и уничтожения с помощью scope.
* Минимизация "переноса" зависимостей. Внедрение зависимостей через контейнеры позволяет сократить количество прямых зависимостей между компонентами, упрощая архитектуру системы.

Приведу пример из нашего текущего разрабатываемого проекта:

|  |
| --- |
| @Service public class TelegramBot extends TelegramLongPollingBot {  // Поля класса TelegramBot  private final BotConfig botConfig;  private final WeatherMapper weatherMapper;  private BotLanguage botLanguage;  private BotCommand botCommand;   @Autowired // Внедрение зависимостей на основе типов параметров в конструкторе  public TelegramBot(BotConfig botConfig, WeatherMapper weatherMapper) throws TelegramApiException  {  super(botConfig.getBotToken());  this.botConfig = botConfig;  this.weatherMapper = weatherMapper;  botLanguage = BotLanguage.*ENGLISH*;  execute(new SetMyCommands(BotModel.*commandListForBotMenu*(), new BotCommandScopeDefault(), null));  } |

1. Преимущество использования try-с-ресурсами по сравнению с использованием try-finally. Интерфейс интерфейса AutoCloseable.

Данное нововведение try-с-ресурсами появилось недавно, а именно в JDK 7, и призвано упросить разработку программ, ведь данный try-с-ресурсами позволяет автоматически закрывать какой-либо ресурс, тем самым избежать утечки памяти или прочих проблем. Выглядит конструкция следующим образом:

**try (*описание\_ресурса)* {**

**// использование ресурса**

**}**

Где ресурсом может быть любой поток данных, например файловый, или сам файл, или же класс Scanner. Область применимости таких инструкций try ограничена ресурсами, которые реализуют интерфейс AutoCloseable, определенный в пакете java.lang и содержащий в себе тот самый метод close(). Для конструкций try-с-ресурсами также можно использовать блоки catch и finally.

К слову говоря, блок finally может быть не всегда выполнен, например он не выполниться, если JVM умрёт.

1. Паттерн Декоратор (Decorator) Преимущества его использовании (например композиция по сравнению с наследованием).

Данный паттерн позволяет добавлять новую функциональность каким-либо объектам, благодаря обёрткам над их методами.

Для примера определим интерфейс DataSource:

|  |
| --- |
| public interface DataSource {  void writeData(String data);  String readData();  } |

Теперь создадим абстрактный декоратор, который будет использовать этот интерфейс:

|  |
| --- |
| public abstract class DataSourceDecorator implements DataSource {  protected DataSource dataSource;  public DataSourceDecorator(DataSource dataSource) {  this.dataSource = dataSource;  }  @Override  public void writeData(String data) {  dataSource.writeData(data);  }  @Override  public String readData() {  return dataSource.readData();  }  } |

Далее, создадим конкретный декоратор, который добавит новое поведение к исходному объекту:

|  |
| --- |
| public class EncryptionDecorator extends DataSourceDecorator {  public EncryptionDecorator(DataSource dataSource) {  super(dataSource);  }  @Override  public void writeData(String data) {  String encryptedData = encrypt(data);  super.writeData(encryptedData);  }  private String encrypt(String data) {  // Реализация шифрования  return data + " (encrypted)";  }  } |

В этом примере EncryptionDecorator оборачивает объект DataSource и добавляет новую функциональность — шифрование данных перед записью. Сигнатура методов остается неизменной, что позволяет использовать декоратор вместе с другими объектами типа DataSource.

1. Преимущества использования списков перед массивами.

Списки представлены в Java Collections в виде класса LinkedList и представляет собой двухсвязанный список со ссылками на хвост и голову. А массив представлен в виде ArrayList, и является динамическим. Также в Java есть потокобезопасный класс Vector, но он является устаревшим. Преимущество зависит от того, где что используется.

LinkedList поддерживает быстрое удаление элементов из последовательности, а также изменение порядка элементов, но при поиске элементов, время выполнения будет линейной. В свою очередь ArrayList позволяет быстро находить элементы по индексу, но при удалении элементов, конкретно в ArrayList элемент удаляется, но память не освобождается, для этого нужно явным образом вызывать метод trimToSize().

1. Основные системы сборки Gradle и Maven. Использование Gradle и основная терминология. Управление зависимостями в Gradle.

Данные системы сборки проектов на java и Kotlin являются одними из самых популярных, начнём с gradle:

* Gradle позволяет использовать как Groovy, так и Kotlin для описания сборочных скриптов, что обеспечивает большую гибкость и удобство в написании и поддержке сборочных файлов.
* Автоматическое распределение задач по нескольким потокам для ускорения процесса сборки.
* Возможность выполнять только те части сборки, которые были изменены, что значительно ускоряет процесс разработки.

Для системы Maven:

* Maven использует XML для описания проекта и его зависимости, что делает процесс сборки более предсказуемым и стандартизированным.
* Maven определяет стандартный жизненный цикл проекта, который включает фазы компиляции, тестирования, пакетирования и развертывания.
* Maven имеет встроенные репозитории для хранения библиотек и плагинов, что упрощает управление зависимостями.
* Процесс сборки Maven не зависит от операционной системы, что обеспечивает консистентность сборки на различных платформах.

Обе данные системы сборки проводят тестирование проекта, а также стягивают все изменения из центрального Maven Repository репозитория. Также Maven и Gradle позволяют собирать документацию в отдельной директории.

1. Анатомия jar. Сканирование пакетов.

**JAR-файл** — это Java-архив (**J**ava **AR**chive). Это простой архивный файл, сжатый (иногда с нулевой компрессией) по алгоритму **zip**, т.е. с использованием алгоритма сжатия Deflate.

JAR файл был создан для удобства распространения программ, написанных на **Java**. Так как обычная программа содержит сотни, тысячи, а иногда и миллионы файлов. Файл может содержать:

* файл манифеста META-INF/MANIFEST.MF. Манифест - это текстовый файл формата “ключ : значение”, он содержит описание **jar-файла**.
* **java-файлы** (исходный код)
* **class-файлы**
* файлы, необходимые для работы программы: картинки, файлы с настройками и прочее (ресурсы)
* электронные подписи, которые позволяют защитить программу от модификации

1. Реализация REST API с помощью Spring Framework.

Сначала, что такое REST? Под данным словом подразумевается паттерн проектирования веб-приложений и в целом приложений. REST описывает то, как посредством протокола НТТР должен взаимодействовать клиент с сервером. С помощью него можно реализовывать микросервисную архитектуру приложения. Приведём пример: У нас есть приложение для аренды автомобилей. В нём есть условно два модуля: сервис по выбору автомобилей от клиента, и сервис по оплате. Так вот, чтобы корректно работало всё приложение, эти два сервиса: “Выбор авто и сервис оплаты”, необходимо, чтобы они общались между собой, при выборе клиентом определённой авто, необходимо отсылать данные об этом автомобиле сервису оплаты аренды. Стоимость оплаты, какой клиент берёт в аренду автомобиль, дата аренды и т.д. По такому принципу работает REST.

В Spring Framework существуют специальные аннотации, по типу @ResponceBody, @RestController и т.д. для реализации паттерна REST.

1. Понятие Инверсии управления.

Инверсия управления (Inversion of Control, IoC) — это принцип проектирования, согласно которому подчиненные модули не должны напрямую управлять своими зависимостями. Вместо этого, управление зависит от более высокоуровневого модуля. Этот подход позволяет упростить тестирование и поддержку кода, поскольку зависимости могут быть легко заменены или изменены без необходимости переписывания больших блоков кода.

В контексте Java, IoC часто ассоциируется с фреймворками, такими как Spring, которые автоматизируют процесс внедрения зависимостей. Однако, принцип IoC может быть реализован и без использования данных фреймворков.

**Пример реализации:**

Допустим, у нас есть интерфейс DataSource и две его реализации: PostgresDataSource и MongoDBDataSource. Мы хотим выбирать источник данных во время выполнения программы, а не компиляции.

|  |
| --- |
| interface DataSource {  void connect();  }  class PostgresDataSource implements DataSource {  @Override  public void connect() {  System.out.println("Connecting to Postgres...");  }  }  class MongoDBDataSource implements DataSource {  @Override  public void connect() {  System.out.println("Connecting to MongoDB...");  }  } |

Чтобы реализовать IoC, мы можем создать класс ConnectionFactory, который будет отвечать за создание экземпляров DataSource:

|  |
| --- |
| class ConnectionFactory {  private String type; // Тип источника данных  public ConnectionFactory(String type) {  this.type = type;  }  public DataSource createConnection() {  switch (type) {  case "postgres":  return new PostgresDataSource();  case "mongo":  return new MongoDBDataSource();  default:  throw new IllegalArgumentException("Unknown datasource type");  }  }  } |

Теперь, вместо того чтобы напрямую создавать экземпляры DataSource в нашем коде, мы делегируем эту ответственность классу ConnectionFactory. Это позволяет нам легко менять тип источника данных, не изменяя остальной части кода.

|  |
| --- |
| public class Main {  public static void main(String[] args) {  ConnectionFactory factory = new ConnectionFactory("postgres");  DataSource dataSource = factory.createConnection();  dataSource.connect();  }  } |

1. Spring Boot и его использование.

Spring Boot предлагает довольно большой объём преимуществ по сравнению с традиционным Spring Framework, делая разработку приложений быстрее и проще. Вот некоторые из основных особенностей Spring Boot:

1. **Автоматическая конфигурация**: Одной из самых значительных особенностей Spring Boot является его способность автоматически настраивать приложение на основе зависимостей, указанных в файле pom.xml или build.gradle. Это значительно сокращает количество необходимого кода и упрощает развертывание приложения.
2. **Стандартные серверы**: Spring Boot включает встроенные серверы, такие как Tomcat и Jetty, что позволяет запускать приложения без необходимости устанавливать и настраивать отдельные серверы.
3. **Уменьшение количества кода**: Spring Boot снижает объем необходимого кода за счет автоматической генерации конфигураций и уменьшения шаблонного кода, что делает разработку быстрее и эффективнее.
4. **Поддержка баз данных в памяти** : Spring Boot предоставляет поддержку баз данных в памяти, таких как H2, что упрощает разработку и тестирование приложений.
5. **Проекты "Starter"**: Spring Boot предлагает проекты "Starter", которые содержат все необходимые зависимости для работы с определенными технологиями (например, Spring MVC, Spring Security). Это упрощает начало работы с новыми технологиями и уменьшает вероятность ошибок в конфигурации.
6. **DevTools** : Spring Boot DevTools предоставляет удобные инструменты для разработки, такие как автоматическое перезагрузка приложения при изменении исходного кода и возможность быстро переключаться между различными версиями приложения.
7. **Простота развертывания**: Благодаря встроенным серверам и автоматической конфигурации, приложения на Spring Boot легче развернуть и масштабировать, что особенно важно для микросервисной архитектуры.
8. **Оптимизирован для микросервисов**: Spring Boot идеально подходит для разработки микросервисов благодаря своей легкости, скорости разработки и гибкости конфигурации.

По простому, Spring Boot упрощает конфигурацию приложения, например создание соединения с базой данных. Также имеет встроенный сервер для запуска веб-приложений или целых микросервисов и представляет более высокий уровень абстракции по сравнению со Spring Framework.

1. Работа с базами данных в джава приложениях. Встроенные СУБД для Джава приложений.

Работа с встроенными системами управления базами данных (СУБД) в Java позволяет интегрировать базу данных непосредственно в приложение, обеспечивая удобство использования и повышенную производительность за счет отсутствия необходимости взаимодействия с внешним сервером базы данных. Встроенные СУБД обычно используются в небольших проектах или для разработки прототипов из-за их легкости установки и использования.

Некоторые популярные встроенные СУБД для Java:

* **H2**: Лёгкая встраиваемая СУБД, поддерживающая стандарт SQL и JDBC. Подходит для разработки и тестирования.
* **HyperSQL (HSQLDB)**: Также известная как HSQL, эта СУБД поддерживает большинство стандартных SQL-функций и может работать как в режиме клиента-сервера, так и встроенном. Используется в проектах, таких как OpenOffice и LibreOffice.
* **Apache Derby**: Бесплатная, открытая встраиваемая СУБД, разработанная Sun Microsystems (теперь Oracle). Поддерживает стандарты JDBC и SQL.
* **Berkley DB**: Не реляционная база данных, предлагающая высокую производительность и масштабируемость. Подходит для хранения пар ключ-значение.
* **Java DB (Derby)**: Встроенная СУБД от Oracle, предназначенная для разработки и тестирования Java-приложений.

Для начала работы с базами данных необходимо прописать зависимость в pom.xml или build.gradle. Приведём пример с H2:

|  |
| --- |
| <dependency>  <groupId>com.h2database</groupId>  <artifactId>h2</artifactId>  <version>1.4.199</version>  <scope>test</scope>  </dependency> |

1. Реляционные СУБД для работы с джава приложениями. Пакет java.sql и его классы.

В качестве реляционных баз данных можно использовать различные СУБД, PostgreSQL, MySQL, SQLite, Oracle, и т.д.

Для работы с базами данных в IDE необходимо настроить подключение к базе данных и предоставить ей пароль, юзернейм, и драйвер, который можно найти в вышеупомянутом Maven Repository. Для каждой СУБД драйвер будет разный.

Пример подключения к базе данных СУБД PostgreSQL:

|  |
| --- |
| <dependency>  <groupId>org.postgresql</groupId>  <artifactId>postgresql</artifactId>  <version>42.5.0</version>  </dependency> |

Далее нужно в коде успешно подключиться к базе данных:

|  |
| --- |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  public class PostgreSQLJDBCExample {  public static void main(String[] args) {  String url = "jdbc:postgresql://localhost/test";  Properties props = new Properties();  props.setProperty("user","username");  props.setProperty("password","password");  try (Connection con = DriverManager.getConnection(url, props)) {  if (con != null) {  System.out.println("Connected to the database!");  }  } catch (SQLException ex) {  System.out.println("An error ' occurred while connecting to the database");  ex.printStackTrace();  }  }  } |

1. Роль интерфейса JDBC для работы с джава приложениями.

Интерфейс JDBC (Java Database Connectivity) играет критическую роль в работе с базами данных в Java-приложениях. JDBC предоставляет набор API, который позволяет Java-приложениям взаимодействовать с различными реляционными базами данных. Вот основные аспекты, на которых JDBC сосредоточена:

1. Абстракция над базами данных.

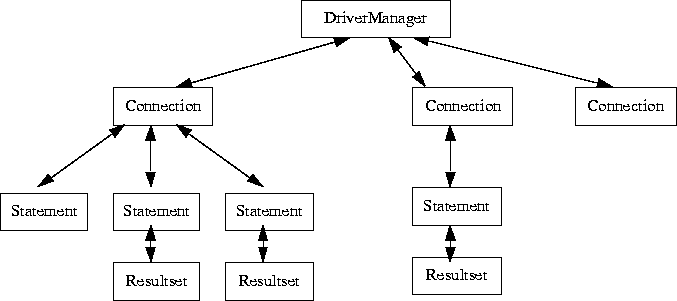
2. Управление транзакциями.

3. Поддержка SQL.

4. Управление ресурсами.

С помощью JDBC можно оптимально работать с базами данных различных СУБД, важно помнить только о безопасности, например при использовании так называемого Statement не допускать SQL инъекций и подобного.

1. Основные компоненты JDBC API.



* **DriverManager**: Управляет набором драйверов баз данных. Он отвечает за загрузку драйверов, которые могут открывать соединения с конкретными базами данных. Приложение вызывает метод getConnection() у DriverManager, чтобы установить соединение с базой данных.
* **Driver**: Интерфейс, который должен быть реализован каждым драйвером баз данных. Драйвер отвечает за создание соединений с базой данных и выполнение SQL-запросов.
* **Connection**: Интерфейс, который представляет соединение с базой данных. Используется для создания объектов Statement, PreparedStatement и CallableStatement, а также для управления транзакциями.
* **Statement**: Интерфейс, который используется для выполнения статических SQL-запросов и получения результатов. Объекты Statement создаются с использованием объекта Connection.
* **Подготовленное утверждение**: Подинтерфейс Statement, который используется для выполнения предварительно компилированных SQL-запросов с параметрами. Это повышает производительность и безопасность (предотвращение SQL-инъекций).
* **CallableStatement**: Подинтерфейс PreparedStatement, который используется для выполнения хранимых процедур в базе данных.
* **ResultSet**: Интерфейс, который представляет набор результатов SQL-запроса. Используется для перебора и получения данных из результата запроса.
* **Исключение SQLException**: Класс, представляющий ошибки, возникающие при работе с базой данных. Он обеспечивает информацию об ошибке, включая сообщения об ошибках, коды ошибок и состояние SQL.

1. JDBC URL и его использование.

JDBC URL (Uniform Resource Locator) служит для подключения к базе данных с использованием JDBC. Он представляет собой строку, которая содержит информацию, необходимую для установки соединения с базой данных, включая тип базы данных, хост, порт, имя базы данных и другие параметры подключения.

Использование JDBC URL для подключения к базе данных:

|  |
| --- |
| Connection conn = DriverManager.getConnection(  "jdbc:sqlserver://localhost:1433;databaseName=AdventureWorks;integratedSecurity=true;",  "username",  "password"  ); |

Важно отметить, что JDBC URL должен быть правильно сформирован в соответствии с требованиями выбранной СУБД.

1. Работа JDBC драйвера.
2. **Загрузка драйвера**: В начале работы с базой данных необходимо загрузить соответствующий JDBC драйвер. Это делается с помощью метода Class.forName(), передавая ему имя класса драйвера. Например, для MySQL это будет "com.mysql.jdbc.Driver".
3. **Создание соединения**: После загрузки драйвера следует установить соединение с базой данных. Для этого используется метод DriverManager.getConnection(), которому передается URL подключения к базе данных, логин и пароль от пользователя базы данных.
4. **Выполнение запросов**: Соединение предоставляет доступ к объекту Statement или его подклассам (PreparedStatement, CallableStatement), которые используются для отправки SQL-запросов на сервер базы данных. Результаты запросов можно получить через объект ResultSet.
5. **Закрытие ресурсов**: После завершения работы с базой данных важно корректно закрыть все используемые ресурсы (соединения, операторы, наборы результатов) с помощью методов close().

|  |
| --- |
| package javaapplication1;  import java.sql.\*;  public class Main {  public static void main(String[] args) throws SQLException {  // Загрузка драйвера  // Class.forName("com.mysql.jdbc.Driver");  Connection conn = null;  try {  // Установление соединения  conn = DriverManager.getConnection(  "jdbc:mysql://localhost:3306/db\_name", "user", "password");  if (conn == null) {  System.out.println("Нет соединения с БД!");  System.exit(0);  }  // Создание оператора  Statement stmt = conn.createStatement();  // Выполнение запроса и обработка результата  ResultSet rs = stmt.executeQuery("SELECT \* FROM users");  while (rs.next()) {  System.out.println(rs.getRow() + ". " + rs.getString("firstname")  + "\t" + rs.getString("lastname"));  }  // Закрытие оператора  stmt.close();  } catch (SQLException e) {  e.printStackTrace();  } finally {  if (conn!= null) {  conn.close(); // Закрытие соединения  }  }  }  } |

1. В чем заключается роль DI в Spring. Использования ServiceLoader.

Роль внедрения зависимостей играет очень важную роль, в Spring Framework она позволяет правильно выстраивать иерархию всего приложения для корректной работы. Важно помнить про циклические зависимости и NullPointerException при внедрении или обращении к внедрённым полям.

1. Интерфейс Connection. Пулы соединений.

Интерфейс Connection в JDBC представляет собой основной интерфейс для взаимодействия с базой данных. Он предоставляет методы для выполнения операций CRUD (Create, Read, Update, Delete) и других операций с базой данных. Однако создание нового физического соединения с базой данных может быть ресурсоемким и замедлять работу приложения, особенно в средах с большим количеством пользователей или высокими требованиями к производительности.

Пулы соединений JDBC:

Для решения этой проблемы используется концепция пула соединений (connection pool). Пул соединений — это группа повторно используемых соединений с базой данных. Когда приложение запрашивает соединение, оно получает одно из уже существующих соединений из пула вместо создания нового. После использования соединения, оно возвращается обратно в пул, готовое к дальнейшему использованию. Это значительно улучшает производительность за счет сокращения времени на установку новых соединений.

1. Принципы SOLID и их использование на Джава.

Данные принципы включают 5 основных пунктов, которые позволят программистам писать хороший и удобномаштабируемый код для программ, вот основные принципы:

1. ***Single Responsibility Principle*** (Принцип единственной ответственности):

Класс (или функция/метод) должен быть ответственен лишь за что-то одно. Если класс отвечает за решение нескольких задач, его подсистемы, реализующие решение этих задач, оказываются связанными друг с другом. Изменения в одной такой подсистеме ведут к изменениям в другой.

1. ***Open-Closed Principle*** (Принцип открытости-закрытости):

Программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации. В этом контексте открытость для расширения — это возможность добавить для класса, модуля или функции новое поведение, если необходимость в этом возникнет, а закрытость для изменений — это запрет на изменение исходного кода  программных сущностей.

1. ***Liskov Substitution Principle*** (Принцип подстановки Барбары Лисков):

Необходимо, чтобы подклассы могли бы служить заменой для своих суперклассов. Цель этого принципа заключаются в том, чтобы классы-наследники могли бы использоваться вместо родительских классов, от которых они образованы, не нарушая работу программы.

1. ***Interface Segregation Principle*** (Принцип разделения интерфейса):

Создавайте узкоспециализированные интерфейсы, предназначенные для конкретного клиента. Клиенты не должны зависеть от интерфейсов, которые они не используют.

1. ***Dependency Inversion Principle*** (Принцип инверсии зависимостей):

Объектом зависимости должна быть абстракция, а не что-то конкретное.

**Паттерны проектирования:**

* 1. Опишите паттерн абстрактная фабрика, приведите пример реализации.

|  |
| --- |
| * // Абстрактные продукты * interface UsbDevice { * void connectUsb(); * } * interface HdmDevice { * void connectHdm(); * } * // Конкретные продукты * class UsbMouse implements UsbDevice { * @Override * public void connectUsb() { * System.out.println("USB Mouse connected."); * } * } * class HdmMouse implements HdmDevice { * @Override * public void connectHdm() { * System.out.println("HDM Mouse connected."); * } * } * // Абстрактная фабрика * abstract class DeviceFactory { * abstract UsbDevice createUsbDevice(); * abstract HdmDevice createHdmDevice(); * } * // Конкретные фабрики * class UsbDeviceFactory extends DeviceFactory { * @Override * UsbDevice createUsbDevice() { * return new UsbMouse(); * } * @Override * HdmDevice createHdmDevice() { * throw new UnsupportedOperationException("Not supported."); * } * } * class HdmDeviceFactory extends DeviceFactory { * @Override * UsbDevice createUsbDevice() { * throw new UnsupportedOperationException("Not supported."); * } * @Override * HdmDevice createHdmDevice() { * return new HdmMouse(); * } * } * // Клиентский код * public class Client { * public static void main(String[] args) { * DeviceFactory usbFactory = new UsbDeviceFactory(); * DeviceFactory hdmFactory = new HdmDeviceFactory(); * UsbDevice mouse = usbFactory.createUsbDevice(); * mouse.connectUsb(); * HdmDevice keyboard = hdmFactory.createHdmDevice(); * keyboard.connectHdm(); * } * } |

В этом примере DeviceFactory является абстрактной фабрикой, которая определяет интерфейс для создания устройств USB и HDMI. UsbDeviceFactory и HdmDeviceFactory являются конкретными фабриками, каждая из которых реализует свой набор продуктов. Клиентский код работает с абстрактными фабриками и продуктами, не заботясь о конкретных классах продуктов.

1. Опишите паттерн адаптер, его отличие от декоратора, приведите пример реализации.

Паттерн Адаптер (Adapter) также относится к категории структурных паттернов проектирования. Он позволяет объектам с несовместимыми интерфейсами работать вместе. Адаптер преобразует интерфейс одного класса в интерфейс другого, который клиенты ожидают. Таким образом, адаптер позволяет классам работать вместе, даже если они были разработаны для работы с другими интерфейсами.

|  |
| --- |
| // Класс, который требуется адаптировать  class OldMediaPlayer {  void playVHS() {  System.out.println("Playing VHS...");  }  }  // Интерфейс, который должен быть адаптирован  interface MediaPlayer {  void playCD();  }  // Адаптер  class CDPlayerAdapter implements MediaPlayer {  private OldMediaPlayer oldPlayer;  public CDPlayerAdapter(OldMediaPlayer oldPlayer) {  this.oldPlayer = oldPlayer;  }  @Override  public void playCD() {  oldPlayer.playVHS(); // Преобразование вызова playCD() в playVHS()  }  }  // Клиентский код  public class Client {  public static void main(String[] args) {  OldMediaPlayer oldPlayer = new OldMediaPlayer();  MediaPlayer adapter = new CDPlayerAdapter(oldPlayer);  adapter.playCD(); // Используем адаптер для работы с OldMediaPlayer  }  } |

1. Опишите паттерн декоратор, его отличие от адаптера, приведите пример реализации.

Паттерн и пример реализации были описаны выше.

**Отличие декоратора от адаптера:**

* **Декоратор** добавляет новые функциональные возможности к объекту, сохраняя его оригинальный интерфейс. Он оборачивает объект вокруг себя, предоставляя новый интерфейс, который расширяет поведение оригинального объекта.
* **Адаптер**, с другой стороны, предназначен для интеграции двух несовместимых компонентов, преобразуя интерфейс одного компонента в интерфейс другого.

1. Опишите паттерн фасад, приведите пример реализации.

Паттерн Фасад (Facade) относится к категории структурных паттернов проектирования. Он предоставляет упрощённый интерфейс к сложной системе классов, библиотеке или фреймворку. Фасад делает систему более простой для понимания и использования, скрывая сложности внутреннего устройства и предоставляя единый интерфейс для взаимодействия с ней.

Фасад определяет интерфейс, который упрощает использование сложной системы, скрывая детали реализации и предоставляя клиентам высокоуровневый интерфейс. Это позволяет клиентам взаимодействовать с системой через единственный класс-фасад, не заботясь о внутреннем устройстве системы.

|  |
| --- |
| // Интерфейс системы  interface ImageProcessingSystem {  void readImage(String imagePath);  void applyFilter(String filterName);  void saveImage(String outputPath);  }  // Реализация системы  class ComplexImageProcessor implements ImageProcessingSystem {  @Override  public void readImage(String imagePath) {  System.out.println("Reading image from " + imagePath);  }  @Override  public void applyFilter(String filterName) {  System.out.println("Applying filter: " + filterName);  }  @Override  public void saveImage(String outputPath) {  System.out.println("Saving image to " + outputPath);  }  }  // Фасад  class SimpleImageProcessor {  private ImageProcessingSystem system;  public SimpleImageProcessor(ImageProcessingSystem system) {  this.system = system;  }  public void processImage(String imagePath, String filterName, String outputPath) {  system.readImage(imagePath);  system.applyFilter(filterName);  system.saveImage(outputPath);  }  }  // Клиентский код  public class Client {  public static void main(String[] args) {  ImageProcessingSystem complexSystem = new ComplexImageProcessor();  SimpleImageProcessor simpleProcessor = new SimpleImageProcessor(complexSystem);  simpleProcessor.processImage("path/to/image.jpg", "grayscale", "path/to/output.jpg");  }  } |

В этом примере ComplexImageProcessor представляет собой сложную систему, которую мы хотим упростить для клиентов. SimpleImageProcessor является фасадом, который предоставляет простой интерфейс для работы с этой системой. Клиентский код теперь может легко работать с изображениями, используя simpleProcessor, не заботясь о внутренних деталях реализации системы.