Оглавление

[1. Основные парадигмы и стили программирования: императивное, процедурное, функциональное 3](#_Toc169196610)

[2. Экосистема языка Java. Решение проблемы кроссплатформенности с помощью JVM, client-server архитектуры, web-приложения 4](#_Toc169196611)

[3. Экосистема языка Java. Базовые фреймворки и библиотеки Java 5](#_Toc169196612)

[4. Развитие web-приложений на примере Java экосистемы: от сервлетов и JSP к web-сервисам 6](#_Toc169196613)

[5. Основные подходы и типы API для работы с web-сервисами: SOAP, REST, GraphQL, gRPC 7](#_Toc169196614)

[6. Cредства и инструменты для web-разработки на языке Java: IDEs, build tools, web servers, version control systems 8](#_Toc169196615)

[7. Средства и инструменты для разработки и тестирования API 9](#_Toc169196616)

[8. Система типов данных в языке Java, классы, объекты 10](#_Toc169196617)

[9. Основные коллекции в языке Java 11](#_Toc169196618)

[10. Работа со строками в языке Java: String, StringBuffer, StringBuilder 12](#_Toc169196619)

[11. ООП в Java: наследование, инкапсуляция, полиморфизм 13](#_Toc169196620)

[12. Интерфейсы и абстрактные классы в Java 14](#_Toc169196621)

[13. Методы в Java: статические методы, абстрактные методы, перегрузка методов (overriding, overloading) 15](#_Toc169196622)

[14. Обработка ошибок в Java: try-catch-finally 16](#_Toc169196623)

[15. Обработка ошибок в Java: logging, фреймворки для логирования 17](#_Toc169196624)

[16. Обработка ошибок в Java: exceptions (checked/unchecked) 19](#_Toc169196625)

[17. Типы автоматизированного тестирования кода: unit/integration testing 20](#_Toc169196626)

[18. Автоматизированное тестирование кода: mocks, stubs, e2e тестирование 21](#_Toc169196627)

[19. Multithreading и concurrency в Java: Thread, concurrent collections, synchronized блоки и методы 22](#_Toc169196628)

[20. Multithreading и concurrency в Java: cредства и подходы обеспечения thread safety в Java 23](#_Toc169196629)

[21. Асинхронные вычисления в web-приложениях средствами Java: потоки, futures 24](#_Toc169196630)

[22. Функциональное программирование средствами Java 8: lambda-функции, method references, Optional 25](#_Toc169196631)

[23. Функциональное программирование средствами Java 8: Stream API, intermediate и terminal функции 26](#_Toc169196632)

[24. Функциональное программирование средствами Java 8: функциональные интерфейсы 28](#_Toc169196633)

[25. Типы хранилищ данных: реляционные базы данных, NoSQL 29](#_Toc169196634)

[26. Основные принципы распределённых хранилищ данных: ACID/BASE, CAP теорема 30](#_Toc169196635)

[27. Основные средства и фреймворки для работы с БД в Java: JDBC, JPA/Hibernate, MyBatis 31](#_Toc169196636)

[28. Паттерны и принципы проектирования: SOLID, evergreen принципы 32](#_Toc169196637)

[29. Паттерны и принципы проектирования: MVC паттерн в java web-приложениях 33](#_Toc169196638)

[30. Паттерны и принципы проектирования: классификация Java паттернов, GoF паттерны 34](#_Toc169196639)

[31. Паттерны и принципы проектирования: общие принципы построения архитектуры, основные архитектурные стили 35](#_Toc169196640)

[32. Основы непрерывной разработки кода: Agile-методологии, CI/CD 36](#_Toc169196641)

[33. Основные подходы и средства улучшения качества кода: code review, testing, static analyzers 37](#_Toc169196642)

[34. Облачные решения: SaaS, PaaS, IaaS, AWS, GCP, Azure, serverless архитектура и cloud-native подход в построении приложений 38](#_Toc169196643)

1. Основные парадигмы и стили программирования: императивное, процедурное, функциональное

Парадигма языков программирования представляет собой совокупность правил, принципов и методик, используемых в процессе разработки ПО. От неё зависит стиль написания программы. При этом выбор языка программирования никак не зависит от парадигмы.

**Императивное программирование –** это подход, при котором программа описывается как последовательность команд, изменяющих состояние системы. Императивное программирование делится на подпарадигмы, такие как процедурное и структурное программирование. Примеры языков, поддерживающих императивное программирование: C++, Java, Python.

***Процедурное программирование*** **–** это вид императивного программирования, где основной концепцией является процедура (функция), представляющая собой последовательность действий.

Этот подход позволяет структурировать код в виде последовательности вызовов процедур, что облегчает чтение и понимание кода. Однако современное программирование часто переходит к более абстрактным парадигмам, таким как ООП.

***Объектно-ориентированное программирование (ООП)*** **–** этот стиль программирования организует код вокруг объектов и классов, которые объединяют данные и методы для работы с этими данными. ООП позволяет инкапсулировать детали реализации и управлять связями между различными частями программы.

**Декларативное программирование –** сосредоточено на описании того, что должно быть достигнуто, без указания конкретных шагов для достижения этой цели.

К декларативным парадигмам относятся функциональное, логическое программирование.

***Функциональное программирование*** **–** это декларативный стиль программирования, где программист описывает, что должно быть достигнуто, а не как достичь результата. В этом подходе акцент делается на использовании чистых функций, которые не изменяют свое состояние и не влияют на внешний мир. Функциональное программирование упрощает тестирование и отладку кода, так как функции могут быть протестированы независимо друг от друга.

2. Экосистема языка Java. Решение проблемы кроссплатформенности с помощью JVM, client-server архитектуры, web-приложения

Кроссплатформенность в контексте Java определяется как способность Java-приложений работать на различных операционных системах и аппаратных платформах без необходимости изменения исходного кода.

**JVM**

Экосистема языка Java обеспечивает решение проблемы кроссплатформенности через Java Virtual Machine (JVM), которая позволяет запускать Java-программы на любой платформе, поддерживающей JVM. Это означает, что одинаковый код может работать на различных операционных системах, таких как Windows, macOS, Linux, без необходимости его изменения или перекомпиляции.

***JVM абстрагирует аппаратное обеспечение***: JVM служит промежуточным слоем между Java-кодом и аппаратным обеспечением, позволяя программам написанным на Java работать на разных платформах без изменений.

***Байт-код***: Java-компилятор преобразует исходный код в байт-код, который затем исполняется JVM. Байт-код является платформо-независимым, что позволяет ему запускаться на любой машине с установленной JVM.

**Client-Server архитектуры**

Решение проблемы кроссплатформенности в экосистеме Java с помощью client-server архитектуры основывается на использовании стандартных протоколов и форматов данных, которые являются платформо-независимыми. Это позволяет клиентским и серверным приложениям, написанным на Java и работающим на разных операционных системах, эффективно взаимодействовать друг с другом.

***Использование байтовых потоков***

Важным аспектом кроссплатформенного взаимодействия является отправка и получение данных в виде байтовых потоков, а не символов. Это обеспечивает единообразие обработки данных на всех платформах, поскольку размер и представление байтов остаются неизменными.

***Протоколы***

Стандартные протоколы, такие как TCP/IP (Transmission Control Protocol *доставка данных между приложениями на разных хостах* /Internet Protocol *определяет адресацию и маршрутизацию пакетов данных в сети*), используются для обеспечения связи между клиентами и серверами. Эти протоколы определены независимо от платформы, что позволяет различным системам общаться друг с другом без проблем, связанных с различиями в архитектуре или операционных системах.

***Сериализация и десериализация данных***

Для обмена данными между клиентом и сервером необходимо использовать механизмы сериализации и десериализации, которые позволяют преобразовывать сложные структуры данных в байтовые потоки для передачи и обратно для восстановления исходных структур данных на приемной стороне. Это обеспечивает совместимость и возможность интерпретации данных на любой платформе.

***Примеры технологий***

Технологии, такие как RPC (Remote Procedure Call) и XDR (External Data Representation), были разработаны для решения проблем кроссплатформенного взаимодействия путем стандартизации вызовов удаленных процедур и представления данных соответственно. Современные варианты, такие как XML-RPC и JSON-RPC, используют текстовые форматы для обмена данными и вызова удаленных процедур, что также способствует кроссплатформенности.

Таким образом, благодаря использованию платформо-независимых протоколов, форматов данных и механизмов сериализации, Java обеспечивает высокую степень кроссплатформенности в client-server архитектурах, позволяя разработчикам создавать приложения, которые могут беспроблемно работать на различных операционных системах и устройствах.

**Web-приложения**

Для разработки кроссплатформенных веб-приложений на Java можно использовать различные технологии и фреймворки, такие как Servlets, JSP *(Java Server Pages представляет собой технологию серверного программирования, которая позволяет создавать динамические, платформо-независимые методы для разработки веб-приложений*), фреймворки такие как Spring, Hibernate и др. Позволяют создавать динамические веб-приложения, которые могут обрабатывать запросы от пользователей, взаимодействовать с базами данных и предоставлять содержимое, адаптированное под различные устройства и браузеры.

Для решения проблемы кроссплатформенности в экосистеме языка Java с помощью веб-приложений используются различные технологии и инструменты, которые обеспечивают разработку, развертывание и поддержку веб-приложений на различных платформах без необходимости изменения исходного кода.

**HTTP/Web/API серверы**

***Apache HTTP Server:*** Это открытый и бесплатный веб-сервер, который может обрабатывать HTTP запросы и предоставлять статический контент. Он часто используется в связке с другими серверами приложений, такими как Tomcat, для обработки динамического контента.

***Tomcat:*** Является сервером приложений, который предоставляет среду выполнения для сервлетов и JSP (JavaServer Pages). Sервлеты и JSP позволяют создавать динамические веб-страницы, которые могут генерировать HTML на лету в ответ на HTTP запросы.

***Jetty:*** Это еще один веб-сервер, который известен своей легковесностью и высокой производительностью. Он часто используется в современных Java-приложениях благодаря своему быстрому старту и низкому потреблению ресурсов.

**Фреймворки и библиотеки**

***Spring Framework:*** Это мощный фреймворк для разработки веб-приложений на Java, который предоставляет широкий набор инструментов для управления зависимостями, транзакциями, безопасности и многим другим. Он позволяет разработчикам сосредоточиться на бизнес-логике приложения, минимизируя количество ручной работы.

***Spring Boot***: Это расширение Spring Framework, которое упрощает создание автономных веб-приложений. Он предоставляет множество предварительно настроенных конфигураций и шаблонов, что позволяет быстро начать работу над новым проектом.

***Servlets***: Это компоненты Java EE, которые могут обрабатывать HTTP запросы и генерировать HTTP ответы. Они являются фундаментальной частью веб-разработки на Java и могут быть использованы для создания сложных веб-приложений.

//Java EE (Java Platform, Enterprise Edition) представляет собой расширенную версию Java SE (Standard Edition), предназначенную для разработки приложений уровня предприятия. Java EE включает в себя ряд спецификаций и API, направленных на упрощение процесса разработки и повышение производительности приложений, что делает ее незаменимым инструментом для профессионалов, стремящихся к получению сертификации Java.

Основные характеристики Java EE: Поддержка веб-приложений и сервлетов, разработка распределенных, транзакционных и портативных приложений, упрощение процесса разработки, стандартизированная модель разработки.//

***Hibernate*** играет важную роль в обеспечении кроссплатформенности приложений на Java, особенно в контексте работы с базами данных. Он позволяет разработчикам использовать один и тот же код для взаимодействия с различными реляционными базами данных, благодаря поддержке диалектов Hibernate для каждой из них. Это значит, что приложение, использующее Hibernate, может быть легко адаптировано для работы с различными базами данных, не требуя значительных изменений в коде.

Преимущества использования Hibernate для кроссплатформенности

Абстракция от базы данных: Разработчики могут сосредоточиться на бизнес-логике приложения, не беспокоясь о специфике конкретной базы данных. Hibernate автоматически генерирует SQL-запросы и управляет соединениями с базой данных, что упрощает разработку и поддержку приложений.

Масштабируемость и производительность: Hibernate оптимизирован для работы с большими объемами данных и может быть настроен для обеспечения высокой производительности приложений.

Тестирование и разработка: Hibernate упрощает тестирование и разработку, позволяя разработчикам использовать виртуальные базы данных, такие как H2, HSQL или Derby, которые полностью поддерживаются как диалекты Hibernate. Это позволяет тестировать приложение вне зависимости от конкретной базы данных, используемой в продакшене.

В заключение, Hibernate значительно упрощает разработку кроссплатформенных приложений на Java, предоставляя мощные инструменты для работы с реляционными базами данных и обеспечивая высокую степень переносимости кода между различными платформами и базами данных.

3. Экосистема языка Java. Базовые фреймворки и библиотеки Java

Фреймворки — это набор правил, шаблонов и инструментов, которые задают структуру и определяют правила для создания проекта, позволяя разработчикам писать только уникальный код, специфический для их продукта. Фреймворки освобождают от необходимости каждый раз заново писать общий код для взаимодействия с базой данных, аутентификации, поддержки сеансов и других стандартных функций.

Библиотеки позволяют разработчикам использовать уже готовые решения для решения различных задач, что значительно ускоряет процесс разработки и повышает качество программного обеспечения.

Если коротко, то фреймворки определяют архитектуру приложения и обеспечивают взаимодействие между его компонентами. В то время как библиотека — это набор функций и методов для решения конкретных задач. Она не определяет структуру приложения, фреймворк сам устанавливает общую структуру и порядок работы приложения.

**1. Spring Framework**

Spring — это фреймворк с открытым исходным кодом, который в основном используется для разработки корпоративных веб- и мобильных приложений. Он включает в себя различные компоненты, такие как Spring Core, Spring Web MVC, Spring AOP, Spring DAO, Spring context, Spring ORM и Spring Web Flow. Он также может быть использован с Kotlin и Groovy.

Spring популярен у многих компаний, например, Netflix и eBay используют Spring для разработки своих приложений и систем.

Основные преимущества Spring: чистота и доступность кода, легкость и простота в развертывании, совместимость с конфигурацией в стиле XML и аннотаций, поддержка внедрения зависимостей.

Единственным недостатком Spring является его сложность для новичков, не имеющих знаний в программировании на Java, так для работы с этим фреймворком необходимы хотя бы некоторые базовые знания этого языка программирования.

**2. Spring Boot**

Spring Boot — одно из расширений фреймворка Spring, используемое для быстрой разработки веб-приложений и микросервисов.

Spring Boot представляет собой автоматически настраиваемую среду разработки ПО, в которой Java-разработчики могут сразу же начать создавать программное обеспечение.

Помимо скорости данное расширение предлагает множество других преимуществ. Например, оно удаляет много повторяющихся настроек, предоставляет структурированный подход для более эффективной разработки ПО, имеет встроенный сервер для быстрой развертки приложений, а также поддерживает фреймворк JDBC. Подробнее о преимуществах и недостатках Spring Boot можно прочитать здесь.

**3. Spark**

Spark — это легковесный микрофреймворк, широко используемый для создания веб-приложений на Java.

Spark обеспечивает разработчикам простые в использовании API для работы с большими наборами данных, а также имеет набор библиотек более высокого уровня для поддержки SQL-запросов, потоковой обработки данных, машинного обучения, обработки графов и многого другого.

Spark крайне гибкий и не имеет строгих ограничений, благодаря чему разработчики программного обеспечения имеют большую свободу в принятии решений при написании кода.

**4. Hibernate**

Hibernate — Java-фреймворк, используемый в основном для работы с базами данных. Hibernate легко взаимодействует с любой базой данных и невероятно удобен при работе с несколькими базами одновременно.

Основные преимущества Hibernate: хорошая масштабируемость, простота модификации и конфигурирования, поддержка конвертации данных для нескольких баз данных.

Hibernate упрощает разработку приложений и способ их взаимодействия с базой данных. Он идеально подходит для проектов объектно-ориентированного программирования, требующих высокой производительности и переносимости одновременно.

**5. JUnit**

JUnit — это удобный фреймворк с открытым исходным кодом для модульного тестирования в Java. Он помогает инженерам ПО применять разработку, основанную на тестировании, и легко интегрируется с популярными средами разработки Java.

JUnit хорошо работает с популярными инструментами автоматизации и развертывания Java — Maven, Ant, Gradle и Jenkins.

Многие разработчики Java используют JUnit, потому что он позволяет им быстрее писать код и запускать тесты. При этом фреймворк гарантирует высокий уровень качества кода.

**6. Wicket**

Wicket — это полнофункциональный компонентный Java-фреймворк для веб-приложений, предоставляемый Apache Foundation. Фреймворк поддерживает Java, начиная с 11-й версии, и Servlet API 3.1 или новее.

Основываясь на последних релизах Java и Servlet API, фреймворк позволяет создавать выдающиеся веб-решения и обеспечивает надежную защиту данных с помощью Content Security Policy.

Wicket использует чистый Java и просто интегрируется с HTML, что позволяет эффективно разделять разметку и логический слой и проводить модульное тестирование внешнего кода.

Основные преимущества Wicket:

Реализация многократно используемых компонентов для последовательной и быстрой разработки веб-приложений

Использование моделей данных POJO, гарантирующих, что компоненты Wicket являются общими объектами Java

Объединение компонентов в многократно используемые пакеты для добавления пользовательских CSS и JavaScript

Простая интернационализация веб-приложения с поддержкой 25+ языков

Реализация Ajax для обновления приложений в режиме реального времени

Безопасная обработка URL и многое другое

**7. Dropwizard**

Dropwizard — это легкий и простой в использовании фреймворк для создания веб-приложений и RESTful веб-серверов.

Dropwizard работает как отличная альтернатива пакетам Spring и Spring WebMVC, когда речь идет о разработке REST API. В отличие от Spring, Dropwizard основан на библиотеке Jetty HTTP и хорошо реализует парадигму конфигурации по соглашению. Это позволяет быстро упаковывать приложения для простого развертывания в качестве микросервисов и эффективно загружать проекты разработки.

Основными преимуществами Dropwizard являются минималистичный фреймворк для быстрой разработки микросервисов, быстрое создание прототипов приложений, отличная поддержка внешних библиотек — Jetty, Guava, Jersey, Jackson и Metrics, с помощью Dropwizard разработчики программного обеспечения могут создавать простые приложения или небольшие веб-сервисы в течение нескольких часов.

Dropwizard достаточно просто освоить. Все, что нужно сделать, это установить его в IDE, например Eclipse, и воспользоваться базовым руководством с официального сайта.

**8. ATG**

ATG — это Java-фреймворк, разработанный компанией Oracle, который используется для построения решений электронной коммерции. Благодаря поддержке Oracle, ATG может работать на его сервере — Oracle Weblogic. Кроме того, ATG также совместим с IBM Websphere и JBoss, или может быть внедрен на местном сервере.

Платформа ATG имеет все необходимое для построения мощных и сложных B2B и B2C приложений:

Модуль персонализации (DPS) и модуль сценариев (DSS) для настройки контента

Динамическую прикладную инфраструктуру, которая предоставляет среду разработки на основе JavaBeans и JSP

Архитектуру данных в любом месте. Она адаптируется к любой базе данных благодаря использованию XML

Поддержку интеграции платежей и адаптацию к любому способу оплаты

Поддержку серверов приложений, включая JBOSS, WebLogic и WebSpher

Среди компаний, которые используют ATG для своих сайтов электронной коммерции, можно встретить Walmart, ASDA, Tesco, BestBuy, Macy’s, Nike и Sephora.

**9. Play**

Play — это производительный и продуктивный фреймворк для создания веб- и мобильных приложений с использованием Java и Scala. Фреймворк очень удобен для разработчиков, так как позволяет вносить изменения и обновлять код, используя браузер и текстовый редактор.

Play также предоставляет мощный набор инструментов например, инструментарий Akka, который обеспечивает минимальное потребление ресурсов и упрощает создание высоко масштабируемых приложений.

Основные преимущества Play: легкая, нестационарная и неблокирующая архитектура, использование RESTful API по умолчанию, поддержка Eclipse, Netbeans и Intellij IDEA IDE.

Среди компаний, которые используют Play в своих цифровых решениях, стоит отметить Coursera, Zalando, Keen, LinkedIn, Samsung и Walmart.

Play — отличный вариант для тех разработчиков, которые хотят перейти на Java с других языков программирования. Фреймворк не требует особой конфигурации, такой как настройка сервера WAMP/LAMP. Вместо этого разработчикам программного обеспечения нужно просто загрузить его, и он готов к работе.

Более того, фреймворк имеет все необходимое для того, чтобы сразу же начать создавать приложения: компилятор, веб-сервер, сервер базы данных в памяти, распределенное управление кэшем, интегрированный менеджер баз данных, множество библиотек, управление зависимостями и другое.

**10. Grails**

Grails — это фреймворк с открытым исходным кодом для виртуальной машины Java (JVM). Он построен на базе Spring Boot и работает на языке программирования Groovy. Grails совместим с JDK, контейнерами Java EE, Hibernate, Spring и Quartz.

Grails может быть использован для разработки всех видов приложений и популярен у таких компаний, как Oracle, IBM, Netflix, Cisco и Google.

Основные преимущества Grails:

Простота настройки и начала работы

Простая интеграция с уже написанным Java-кодом

Огромное количество плагинов

Grails — это хороший выбор для для стартапов и проектов, требующих больших временных затрат. Простота Grails также позволяет разрабатывать RAD в режиме реального времени.

**11. Vaadin**

Фреймворк позволяет программисту создавать настраиваемые веб-приложения, предоставляя ему оптимизированную и простую в использовании платформу для разработки. Vaadin также предоставляет доступ к DOM непосредственно с виртуальной машины Java. Важной особенностью фреймворка является тот факт, что взаимодействие, которое происходит между сервером и браузером, полностью автоматизировано.

Данные могут быть связаны с использованием как MVC, так и MVP через платформу Vaadin. Он поставляется со встроенной поддержкой инфраструктуры Spring, а также поддерживает языки JVM, такие как Scala и Kotlin. Он также имеет поддержку WebSocket, которая обеспечивает автоматическую связь между сервером и клиентом. Как Java, так и HTML можно использовать для построения представлений во фреймворке Vaadin. Он имеет интуитивно понятный пользовательский интерфейс с поддержкой взаимодействий перетаскивания, что значительно упрощает весь процесс веб-разработки для программистов любого уровня подготовки.

Vaadin — это флагманский продукт, а также платформа, на которой разработчики могут выполнять реализацию веб-интерфейса HTML5 с использованием Java. Это платформа включает в себя веб-фреймворк Java и набор веб-компонентов, а также средства запуска приложений и инструменты.

**12. MapStruct**

Совсем недавно в поле моего зрения попала библиотека MapStruct, которая изменила мой мир. Я всегда не очень любил писать builder или свой метод для перекладывания значений из entity в ДТО. Для тех, кто так же, как и я, не любит писать этот шаблонный код, приходит на помощь MapStruct. Эта библиотека генерирует код, который будет автоматически мапить один класс в другой. Это невероятно ускоряет разработку и тестирование кода.

**13. Lombok**

Эта библиотека предназначена для генерации кода. Она способна генерировать код для достаточно популярных use-cases. Стоит лишь поставить необходимые аннотации.

Достаточно спорная библиотека. Кто-то её любит, кто-то ненавидит, но никого она не оставляет равнодушным.

**14. Apache POI**

На основе этой библиотеки можно создать внутренний фреймворк для удобной и гибкой генерации отчётов. Библиотека поддерживает различные форматы документов, например Word и Excel.

**15. Liquibase/ FlyWay**

Средства для миграции баз данных. Я слабо представляю современный проект, который не будет использовать автоматическое исполнение скриптов для баз данных. Каждая из этих библиотек обладает достоинствами и недостатками. Прежде чем остановить свой выбор на одной из них, я крайне рекомендую ознакомиться с документацией на официальном сайте и принять взвешенное решение. Но даже если спустя время вы поймёте, что решение было ошибочным, эти библиотеки предоставляют достаточно безболезненную миграцию друг на друга.

**16. Testcontainers**

Просто невероятная по своей мощности библиотека, которая позволяет запускать контейнеры в Docker для тестирования. У меня она нашла применение в интеграционных тестах. Есть возможность поднять настоящую базу данных с использованием миграций FlyWay/Liquibase и проверить полный флоу работы системы. Тест-контейнеры можно кастомизировать, а также писать свои.

Есть много других Java-фреймворков, которые не являются фреймворками полного стека или веб-разработки, но полезны в каждом приложении — например, log4j для ведения журнала, Junit и Selenium для тестирования и т.д. Об этом важно знать, хотя вы можете просто изучить их на ходу вместе с другими фреймворками, которые вы видели в приведенном выше списке.

**Apache Commons**: Это набор библиотек, предоставляющий удобные утилиты для работы с коллекциями, файловой системой, регулярными выражениями и многим другим. Apache Commons помогает сократить количество кода, необходимого для выполнения общих задач, и повышает читаемость кода.

**Guava**: Библиотека Google Guava предоставляет множество утилит и структур данных, таких как ImmutableList, ImmutableMap и ImmutableSet, а также функциональные интерфейсы, обобщенные коллекторы и другие полезные инструменты для разработки на Java.

**Jackson**: Jackson используется для сериализации и десериализации объектов Java в JSON и обратно. Это мощный инструмент для работы с данными в формате JSON, который часто используется в веб-разработке.

**JUnit**: Хотя JUnit является частью стандартной библиотеки Java для тестирования, она заслуживает упоминания как одна из самых важных библиотек для разработчиков Java. JUnit предоставляет аннотации и утверждения для написания модульных тестов в Java.

**Log4j**: Log4j является одной из наиболее популярных библиотек для логирования в Java. Она позволяет разработчикам легко интегрировать логирование в свои приложения, предоставляя гибкие возможности настройки уровней логирования и форматирования сообщений.

**\*Gradle** и **Maven**: Эти системы сборки являются стандартом де-факто в Java-экосистеме. Они автоматизируют процесс сборки проектов, управления зависимостями и выполнения тестов

4. Развитие web-приложений на примере Java экосистемы: от сервлетов и JSP к web-сервисам

На начальном этапе разработчики использовали сервлеты и JavaServer Pages (JSP) для создания веб-приложений.

Сервлеты представляют собой классы в Java, которые могут обрабатывать HTTP-запросы и генерировать HTTP-ответы.

JSP позволяет вставлять фрагменты Java-кода непосредственно в HTML-страницы, что упрощает создание динамического контента.

Пример использования сервлетов и JSP включает создание двух сервлетов: AddServlet и ListServlet, которые обрабатывают запросы на добавление и отображение списка пользователей соответственно. Эти сервлеты маппируются на определенные URL-адреса в файле конфигурации web.xml. Для отображения результатов используются JSP-страницы, такие как add.jsp и list.jsp.

Сервлеты — это Java классы, используемые для расширения возможностей серверов, хостящих приложения, доступ к которым осуществляется по модели запрос-ответ. Они играют роль контроллера в архитектуре MVC, обрабатывая бизнес-логику и управляя взаимодействием между моделью и представлением. Сервлеты способны обрабатывать все типы протокольных запросов, но чаще всего используются для работы с HTTP-запросами. Основной недостаток сервлетов заключается в том, что они требуют написания большого количества Java-кода, что может усложнить разработку интерфейса пользователя.

JSP (JavaServer Pages)

JSP представляет собой текстовые документы, содержащие статические данные (обычно HTML) и динамический контент, выраженный через элементы JSP. Это позволяет легко смешивать HTML и Java код, что значительно упрощает создание динамических веб-страниц. JSP автоматически компилируется в сервлеты перед выполнением, что обеспечивает высокую производительность. Однако, поскольку первым шагом в жизненном цикле JSP является его перевод в Java код и последующая компиляция, время ответа JSP может быть медленнее, чем у сервлетов. JSP идеально подходит для роли представления в архитектуре MVC, так как он позволяет легко разделять логику представления от бизнес-логики.

Веб-сервисы

С развитием технологий и стандартизации API, таких как RESTful веб-сервисы, разработка веб-приложений стала использовать более современный подход. Веб-сервисы предоставляют функциональность через HTTP, используя легковесные форматы данных, такие как JSON или XML. Это позволяет легко интегрировать веб-сервисы с различными клиентами, включая веб-браузеры и мобильные приложения.

В отличие от традиционных сервлетов и JSP, веб-сервисы обычно реализуются с использованием аннотаций и конвенций, что упрощает разработку и поддержку кода. Например, можно использовать аннотации @Path, @GET, @POST и другие из пакета javax.ws.rs для определения маршрутов и методов обработки запросов без необходимости явного маппинга в web.xml.

В целом, развитие веб-приложений в Java экосистеме демонстрирует эволюцию от сервлетов и JSP к более модульным, мощным и гибким решениям, таким как веб-сервисы, предлагающие лучшую масштабируемость, производительность и удобство интеграции.

5. Основные подходы и типы API для работы с web-сервисами: SOAP, REST, GraphQL, gRPC

***API*** (Application Programming Interface, программный интерфейс приложения) – это контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это».

1. ***SOAP*** (Simple Object Access Protocol):

SOAP — это протокол для обмена структурированной информацией в веб-сервисах с использованием XML. Он обладает высокой расширяемостью и позволяет осуществлять обмен данными через несколько различных транспортных протоколов, включая HTTP, SMTP (*Simple Mail Transfer Protocol*) и другие.

API SOAP известны своей надежностью и часто используются в энтерпрайз-разработке. SOAP предоставляет строгую спецификацию для обмена сообщениями, включая структуру сообщений, согласование версий, встроенную обработку ошибок, безопасность, использование с различными сетевыми протоколами. Однако зависимость SOAP от XML может приводить к большим объемам передаваемых данных, что делает его менее эффективным по сравнению с другими API в каких-то сценариях использования.

* SOAP API не хранит состояние и может быть протестирован на основе операций.
* Учитывая, что SOAP API используют XML, проверки обычно выполняются для xml ответов.
* Для тестирования часто используются такие инструменты, как SoapUI. Можно проводить контрактное тестирование, поскольку SOAP API четко определены в WSDL.

2. ***REST*** (Representational State Transfer):

REST API является наиболее распространенным типом API в клиент-серверной архитектуре и имеет ряд свойств — например, отсутствие хранения состояния и кэшируемость. REST API построен на стандартных протоколах HTTP и обычно возвращает данные в формате JSON, хотя может поддерживать и другие форматы.

REST API использует стандартные HTTP-методы для взаимодействия, включая GET, POST, PUT, DELETE и другие. Каждый метод соответствует определенному типу действий, которые могут быть выполнены над ресурсами API.

Простота и гибкость REST API сделали его лучшим выбором для многих разработчиков, но при работе с большими объемами данных он может быть неэффективными, так как для получения всех необходимых данных может требоваться несколько запросов.

* REST API не используют хранение состояния и могут быть протестированы индивидуально для каждого эндпоинта.
* Поскольку REST API обычно используют HTTP, в этих случаях обычно применяются такие инструменты, как Postman, Curl или REST-assured (на Java).
* Каждый эндпоинт тестируется с помощью различных HTTP-методов (GET, POST, PUT, DELETE и т.д.), и проверки делаются в отношении кода состояния, времени ответа и тела ответа.

3. ***GraphQL***:

Разработанный компанией Facebook, GraphQL представляет собой язык запросов к API и рабочую среду для выполнения этих запросов. В отличие от REST API, где вам нужно делать запросы к различным эндпоинтам, чтобы получить соответствующие данные, GraphQL позволяет сделать один единственный запрос, чтобы получить именно те данные, которые вам нужны, с помощью специально составленных запросов.

GraphQL позволяет клиентам указывать только необходимые поля и связи между ними. Пользователи определяют форму и размер ответа, что приводит к более эффективной загрузке данных и уменьшению избыточной выборки данных. Он отлично подходит для сложных систем и микросервисов, где данные распределены между различными сервисами.

* В отличие от REST и SOAP, GraphQL основан не на сервисах, а на схеме данных, что требует другого подхода в тестировании. По сути, вы проверяете правильность загрузки данных в соответствии с запросами и мутациями.
* Для тестирования можно использовать такие инструменты, как Postman, Apollo Client Developer Tools или даже браузерные IDE, например GraphiQL
* Также обычно тестируется избыток или недостаток данных (overfetching и underfetching), поскольку GraphQL позволяет клиентам точно указывать, какие данные им нужны.

4. ***gRPC*** (Google Remote Procedure Call):

gRPC — это высокопроизводительная платформа с открытым исходным кодом, разработанная компанией Google. Он использует протокол HTTP/2 для передачи данных и Protocol Buffers (protobuf), высокопроизводительный формат двоичных данных, в качестве языка определения интерфейса. Он использует принцип Remote Procedure Call (RPC) для вызова удаленных методов.

gRPC поддерживает четыре типа взаимодействия: унарный (стандартный запрос-ответ), потоковая передача данных с сервера, потоковая передача данных от клиента и двунаправленная потоковая передача. Эти возможности в сочетании с эффективностью делают gRPC идеальным решением для микросервисной архитектуры.

* Учитывая, что gRPC использует HTTP/2, необходим инструмент, поддерживающий этот протокол. Буферы протокола (protobuf) также требуют специальной обработки.
* Инструменты для тестирования включают grpccurl и ghz. Кроме того, юнит-тестирование можно выполнить с помощью различных библиотек, специфичных для данного языка.
* Поскольку gRPC поддерживает четыре типа связи (унарная, серверная потоковая, клиентская потоковая и двунаправленная потоковая), требуется тестирование каждого из этих типов.

**\*\*\*WebSocket API \*\*\*** (вдруг доп вопрос)

WebSocket API обеспечивает постоянный, полнодуплексный канал связи между клиентом и сервером. В отличие от REST и SOAP, которые придерживаются формата «запрос-ответ», WebSocket сохраняет соединение открытым, что позволяет передавать данные в режиме реального времени. Это делает WebSocket API идеальным для приложений, требующих функциональности в реальном времени, таких как чат-приложения, онлайн-игры и системы отслеживания в реальном времени.

Несмотря на свою мощь, WebSocket API могут быть более сложными в реализации и требуют больше ресурсов для поддержания открытых соединений.

**Краткое заключение**

Каждый из этих API имеет свои сильные стороны и сценарии использования, понимание которых может помочь в принятии обоснованных проектных решений. REST API являются популярным выбором благодаря своей простоте и широкому распространению. SOAP API хорошо работают в энтерпрайз-средах благодаря своим функциям безопасности. WebSocket API отлично подходят для приложений реального времени, gRPC API — для высокопроизводительных микросервисов, а API GraphQL обеспечивают гибкий поиск данных, что отлично подходит для сложных систем.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **SOAP API** | **REST API** | **GraphQL API** | **gRPC API** |
| **Формат данных** | XML | JSON, но может поддерживать несколько форматов | JSON | Protocol Buffers (бинарный формат) |
| **Протокол** | HTTP/HTTPS, SMTP, XMPP, и др. | HTTP/HTTPS | HTTP/HTTPS | HTTP/2 |
| **Тип связи** | Односторонняя связь  (запрос/ответ) | Односторонняя связь  (запрос/ответ) | Односторонняя связь  (запрос/ответ) | Двунаправленная/с сервера/от клиента потоковая передача, стандартный запрос-ответ |
| **Service Discovery** | Стандартизация с помощью ***WSDL*** | Не стандартизированы, часто используют ***OpenAPI (Swagger) для документации*** | Не стандартизировано, ***интроспекция для обнаружения схем*** | Не стандартизирован, но поддерживает ***Server Reflection*** |
| **Производительность** | Снижена из-за словесного формата XML | Варьируется, обычно ниже, чем бинарные протоколы из-за текстовой природы | Варьируется, эффективная загрузка данных может повысить производительность | Высокий (бинарный протокол, преимущества HTTP/2) |
| **Сценарий использования** | Распределенные энтерпрайз- среды, унаследованные системы | Общее назначение, широкое применение | Когда требуется гибкая и эффективная загрузка данных | Микросервисы, системы, критичные к производительности |
| **Сложность** | Высокая | От низкой до умеренной | Умеренная | От умеренной до высокой |

6. Cредства и инструменты для web-разработки на языке Java: IDEs, build tools, web servers, version control systems

**IDEs (Integrated Development Environments): предоставляют всю необходимую функциональность для разработки Java-приложений в одной удобной среде**

*IntelliJ IDEA*:

* Одна из самых мощных и функциональных Java IDE, разрабатываемая компанией JetBrains.
* Предлагает высокоинтеллектуальные возможности для написания, анализа и рефакторинга кода.
* Имеет отличную интеграцию с различными Java фреймворками и средами для веб-разработки, такими как Spring, JSF, Struts, GWT.
* Поддерживает множество расширений и плагинов, что значительно расширяет ее функциональность.
* Обладает эффективными инструментами для отладки, профилирования и тестирования приложений.
* Отличается высокой производительностью и предлагает удобный и настраиваемый интерфейс.
* Доступна как в бесплатной Community версии, так и в платной Ultimate версии с дополнительными возможностями.

*Eclipse*:

* Классическая Java IDE с обширным набором возможностей.
* Имеет большую экосистему плагинов, включая поддержку фреймворков таких как Spring, Apache Struts, JSF.
* Предоставляет мощные инструменты для написания, отладки и развертывания веб-приложений.
* Обладает хорошей интеграцией с системами контроля версий, серверами приложений и другими инструментами.
* Может быть более сложной в освоении, чем IntelliJ IDEA, но предлагает больше настраиваемости.
* Доступна бесплатно и является одним из наиболее широко используемых инструментов среди Java-разработчиков.

*NetBeans*:

* Бесплатная Java IDE с хорошей поддержкой Java EE и веб-приложений.
* Имеет встроенные средства для работы с технологиями, такими как JSF, Servlets, JSP.
* Обеспечивает интеграцию с серверами приложений, такими как GlassFish и Apache Tomcat.
* Предлагает удобный визуальный дизайнер для создания графических интерфейсов.
* Может быть менее производительной, чем IntelliJ IDEA или Eclipse, особенно для крупных проектов.
* Рекомендуется для начинающих Java-разработчиков или для простых веб-приложений.

*Spring Tool Suite (STS)* – специализированная среда для приложений на основе Spring, предлагающая всестороннюю поддержку Spring

* Это специализированная IDE, основанная на платформе Eclipse, разработанная для создания приложений на базе Spring Framework.
* Имеет глубокую интеграцию с экосистемой Spring и предоставляет богатый набор инструментов для работы с этим популярным фреймворком.
* Предлагает такие возможности, как:
  + Автоматическое создание Spring-компонентов и конфигурации
  + Графический редактор для Spring Bean и контекста
  + Инструменты для разработки и развертывания Spring Boot приложений
  + Средства для работы с Spring Data, Spring Security, Spring MVC и другими модулями
* Обеспечивает эффективную поддержку жизненного цикла разработки Spring-приложений от создания до развертывания.
* Дает возможность быстро прототипировать и тестировать Spring-приложения.
* Является бесплатным и поставляется в виде дистрибутива на базе Eclipse.
* Рекомендуется для Java-разработчиков, активно использующих Spring Framework в своих проектах.

**Build Tools: инструменты для автоматизации сборки проектов, которые облегчают управление зависимостями и создание исполняемых файлов**

***Maven*** – чаще всего используется на большинстве проектов

На смену Ant пришел Maven, который не такой гибкий, но значительно сокращает объем рутинной работы. Основные особенности:

* Конфигурация Maven — это один файл pom.xml;
* Из коробки поддерживаются различные типы сборки: JAR, WAR, EAR;
* Введена стандартная структура каталогов для проекта. Это сделано для того, чтобы по умолчанию вам не нужно было объяснять системе сборки, где лежат исходные файлы, ресурсы и куда их нужно переместить после компиляции.

Также у каждого проекта может быть только один файл JAR файл, тогда как при сборке Ant модификации были безграничны: вы вообще можете добавить файл в какую угодно папку, собрать 5 классов в 3 JAR-а и так далее. Все это, безусловно, ухудшало поддерживаемость кода

* Цикл сборки разбит на phases (фазы). Каждая фаза включает в себя определенный стандартный сценарий, который называется goal (цель). Упрощенно, вы указываете до какой фазы нужно выполнить проект (например, только скомпилировать), и выполняется набор сценариев связанных указанной и предыдущими фазами build lifecycle. Дополнительные сценарии реализуются через плагины к Maven;

Жизненный цикл maven-проекта — это список фаз, определяющий порядок действий при его построении. Он содержит три независимых порядка выполнения: clean, default и site.

Для расширения и модификации стандартных ЖЦ используются плагины.

Плагины, хоть и предоставляют программистам дополнительные возможности, но Maven не дает большой свободы модификаций.

* Поддерживается модульная архитектура проекта. Можно объявлять зависимости между модулями;
* Поддерживаются профили. Это возможность, которая позволяется выполнять сборку проекта для различных окружений (машин) по-разному. Например, в профиле для разработки приложения определяются настройки и плагины, которых нет в профиле для продакшена, и наоборот. Т.е. можно, например, иметь разные настройки для подключения к базе данных для рабочей машины и сервера, где будет разворачиваться приложения. Или добавить профиль для развертывания среды окружения, который содержит только плагины и скрипты, которые подготовят ваше рабочее место при переезде с места на место. Название профиля передается как опция (ключ -P) к сборке проекта;
* Стандартизированные названия билдов

Каждый билд имеет атрибуты groupId, artifactId и version. Первые два уникальны и используются для определения в репозитории. Обычно groupId - доменное имя организации, а artifactId - название текущего проекта. Поэтому эта пара и является определяющей: нет двух компаний с одинаковым доменом, как и не существует двух одинаковых проектах в одной компании.

* Для централизованного хранения библиотек введен центральный репозиторий. Все популярные библиотеки публикуются и периодически обновляются в центральном репозитории. Каждая библиотека уникально идентифицируется по параметрам: groupId, artifactId, version. Вам не нужно скачивать зависимости и хранить их где-то вместе с проектом, они объявляются декларативно в теге dependency и скачиваются автоматически при сборке проекта.
* Maven выполняет автоматическое разрешение зависимостей (так называются библиотеки, от которых зависит ваше приложение). Таким образом, например, если библиотека A зависит от библиотеки C, и библиотека B зависит от библиотеки C, но эти C – разных версий, Maven автоматически включит в проект только последнюю версию C (иногда это минус, но все настаивается). Также просто, например, проверить не появились ли новые версии для библиотек, входящих в состав проекта:

mvn versions:display-dependency-updates

Поскольку зависимости выкачиваются автоматически, очень удобно делить проект между участниками команды. Предположите, как-бы вы управляли большим числом зависимостей вручную;

Maven поддерживает архетипы. Это предопределенная структура проекта (шаблон) для быстрого начала разработки приложения. При создании проекта вы можете указать архетип, и вот уже есть некая начальная заготовка. Естественно, можно создавать собственные.

Для Maven есть огромное число полезных плагинов. Основной недостаток, которым ему обычно пеняют, вытекает из его преимуществ. Декларативный стиль описания задач не позволяет так же просто “подшаманить” в определенных случаях, как это делает Ant. Но и это обычно решается через через различные плагины. Например, задачи Ant можно запускать из Maven через antrun-plugin.

Нужно отметить, что когда вы работаете с Maven проектом в IDE, настройки извлекаются именно из Maven. Например, через compiler-plugin проекту указывается версия JDK:

***Gradle*** - современный и гибкий build-инструмент, набирающий популярность среди Java-разработчиков. Использует Groovy DSL для описания сборочных скриптов и поддерживает многомодульные проекты. Чаще для андроид.

* Более современная система сборки, использующая язык Groovy для описания конфигурации.
* Предлагает более гибкий и расширяемый подход по сравнению с Maven.
* Обладает высокой производительностью за счет инкрементальной сборки.
* Предоставляет мощные возможности для управления зависимостями и плагинами.
* Хорошо интегрируется с IDE и позволяет легко автоматизировать различные задачи сборки.
* Часто используется в проектах, где требуется больше гибкости и кастомизации по сравнению с Maven.

Оба, и Ant, и Maven используют XML для конфигурирования сборки. Gradle - предметно-ориентированный язык на основе Groovy. И это его основное отличие от Maven. В остальном Gradle руководствуется теми же принципами. Здесь за выполнение всей работы также ответственны плагины и нет широкой свободы действий.

Gradle распространен в мобильной Android- разработке, а Maven используется в системах управления предприятиями. Почему так? Вопрос остается открытым. Возможно, так просто сложилось. Но Maven все-таки гораздо популярнее на рынке.

***Ant*** - более старый, но все еще активно используемый инструмент для сборки Java-проектов. Использует XML для описания процесса сборки

Это аналог make-файла, а по сути набор скриптов (которые называются tasks). Ant – это пример императивного стиля описания сборки проекта. Вы описываете некое действие, например, скомпилировать файлы в директории проекта, потом, скопировать их рабочую директорию… И вот из таких маленьких кирпичиков, поэтапно собираете весь сценарий сборки проекта. А точнее, несколько сценариев для разных целей. Написав единожды хороший универсальный сценарий, можно копировать его в последующие проекты.

Удобно в Ant то, что вы имеете полный и наглядный контроль над сборкой проекта, а неудобно что вы описываете огромное количество очевидных задач, и по-прежнему управляете зависимостями проекта вручную (существует возможность подключить Ivy для управления зависимостями).

Если вы хотите именно пошагового понимания, что делает IDE за ширмой - соберите проект с помощью Ant. Но правда жизни состоит в том, что большие проекты используют большое число библиотек, причем библиотека A, от которой зависит ваш проект, может в свою очередь зависеть еще от десяти других, и из этих десяти, половина будет пересекаться с другими зависимостями, от которых зависят ваши библиотеки, включенные в ваш проект

**Web Servers:**

*Apache Tomcat* - один из самых популярных open-source Java servlet контейнеров и веб-серверов. Tomcat реализует спецификации Java Servlet и JavaServer Pages (JSP) и используется для развертывания и запуска Java веб-приложений.

* Один из наиболее распространенных и широко используемых веб-контейнеров для Java-приложений.
* Реализует спецификации Java Servlet и Java Server Pages (JSP).
* Является легким, открытым и бесплатным решением.
* Обеспечивает стабильную и надежную работу веб-приложений.
* Хорошо интегрируется с другими инструментами Java-экосистемы.
* Поддерживается большим сообществом и имеет обширную документацию.
* Часто используется в качестве основного контейнера в средних и небольших Java-проектах.

Описание: Apache Tomcat - это контейнер сервлетов и HTTP-сервер, разработанный Apache Software Foundation. Он поддерживает спецификации сервлетов, JSP, EL и WebSocket, что позволяет создавать динамические веб-сайты на Java. Однако Tomcat не является полноценным сервером приложений, так как не поддерживает полностью спецификации Java EE, включая EJB.

Преимущества: Tomcat популярен среди разработчиков за свою простоту использования, легкость управления, быстрый старт и минимальные накладные расходы. Имеет меньший объем памяти по сравнению с GlassFish.

*Jetty* - легковесный и высокопроизводительный Java веб-сервер и servlet-контейнер. Jetty отличается своей небольшой footprint и скоростью работы, что делает его популярным выбором для микросервисной архитектуры и облачных приложений.

* Еще один популярный и легкий веб-контейнер для Java-приложений.
* Отличается высокой производительностью и масштабируемостью.
* Предоставляет гибкий и встраиваемый подход, позволяя интегрировать его в другие приложения.
* Хорошо подходит для разработки и развертывания микросервисов на основе Java.
* Часто используется в средах с ограниченными ресурсами, таких как IoT или мобильные устройства.
* Активно развивается и поддерживается сообществом.

Описание: Jetty - это высокопроизводительный, масштабируемый сервер приложений и контейнер сервлетов, написанный на Java. Он поддерживает спецификации Java Servlet, JSP и WebSocket. Jetty может быть использован как встроенный сервер или как отдельный сервер приложений.

Преимущества: Jetty известен своей гибкостью и легковесностью. Он может быть легко настроен и интегрирован в различные среды и приложения. Jetty также поддерживает широкий спектр технологий и стандартов, что делает его универсальным выбором для многих проектов.

*GlassFish* - open-source Java EE-совместимый веб-сервер, разработанный Oracle. GlassFish предоставляет полную реализацию спецификаций Java EE и может использоваться для развертывания и запуска Java корпоративных приложений.

* Полнофункциональный Java EE-совместимый приложенческий сервер, разработанный Sun Microsystems (ныне Oracle).
* Предоставляет расширенную функциональность для построения Enterprise-уровневых Java-приложений.
* Включает реализацию всех основных Java EE-спецификаций, таких как Servlet, JSP, EJB, JMS и др.
* Обеспечивает высокую надежность, безопасность и управляемость для корпоративных Java-приложений.
* Часто используется в крупных организациях и Enterprise-проектах, требующих комплексной функциональности.
* Поддерживается Oracle и имеет обширную экосистему расширений и инструментов.

Описание: GlassFish - это полнофункциональный сервер приложений Java EE, разработанный Oracle. Он является референсной реализацией стандарта Java EE, что означает, что GlassFish всегда будет поддерживать последние возможности Java EE первым. GlassFish поддерживает все основные технологии Java EE, включая Servlets, EJB, JPA, JSF и JMS.

Преимущества: Благодаря тому, что GlassFish является референсной реализацией Java EE, он обеспечивает полную совместимость со стандартами Java EE и поддерживает последние инновации в этой области. Однако, GlassFish считается более тяжелым в плане ресурсов и сложнее в управлении по сравнению с Tomcat и Jetty

Выбор веб-контейнера зависит от масштаба и требований проекта. Tomcat является наиболее популярным и широко используемым вариантом, особенно для небольших и средних Java-приложений. Jetty подходит для высокопроизводительных и встраиваемых решений, а GlassFish - для Enterprise-уровневых Java EE-приложений, требующих расширенной функциональности.

**Version Control Systems: позволяют отслеживать изменения в коде и совместно работать над проектами**

*Git*

Описание:

* Git - это распределенная система контроля версий, которая позволяет каждому разработчику иметь полную копию истории изменений проекта на своем компьютере. Это обеспечивает высокую скорость работы и эффективность даже в больших командах.

Принцип работы:

* В Git изменения кода группируются в коммиты, которые затем могут быть переданы другим разработчикам или объединены с изменениями других. Git поддерживает множество веток, что упрощает параллельную разработку и тестирование различных функций.

Преимущества:

* Гибкая модель ветвления, которая облегчает работу с различными версиями кода и упрощает процесс слияния изменений.
* Широкое сообщество и большое количество обучающих материалов, что облегчает изучение и применение Git.
* Поддержка большинством современных IDE и инструментов разработки.
* Эффективная работа с большими проектами благодаря оптимизированным алгоритмам хранения данных.

*Subversion (SVN)*

Описание:

* Subversion - это централизованная система контроля версий, где весь код хранится в одном центральном репозитории. Это обеспечивает строгий контроль над изменениями и доступом к коду.

Принцип работы:

* SVN использует модель "checkout-edit-commit", где разработчики сначала загружают последнюю версию файлов, вносят изменения, а затем отправляют эти изменения обратно в репозиторий. SVN поддерживает версионирование файлов и директорий, а также предоставляет механизмы для разрешения конфликтов.

Преимущества:

* Простота использования и понятный интерфейс, особенно для тех, кто уже знаком с другими централизованными системами контроля версий.
* Хорошая поддержка в большинстве IDE и возможность интеграции с различными инструментами разработки.
* Эффективное управление правами доступа и ревизиями, что важно для корпоративных проектов и команд.

*Mercurial*

Описание:

* Mercurial - это еще одна распределенная система контроля версий, написанная на Python. Она предлагает схожий набор функций с Git, но с акцентом на простоту использования и понятность документации.

Принцип работы:

* Как и Git, Mercurial позволяет разработчикам работать с локальными копиями репозитория, делать коммиты и обмениваться изменениями. Mercurial поддерживает ветвление и слияние, а также предоставляет инструменты для разрешения конфликтов.

Преимущества:

* Простота синтаксиса и документации, что делает Mercurial более доступным для новичков и тех, кто переходит с других систем контроля версий, таких как Subversion.
* Хорошая поддержка различных операционных систем, включая Windows, благодаря своей основе на Python.
* Наличие нескольких моделей ветвления, которые могут быть полезны в разных ситуациях разработки.

Mercurial, созданная как альтернатива Git, является распределенной системой управления версиями. Несмотря на меньшую популярность по сравнению с Git, Mercurial часто используется среди разработчиков на Python благодаря своей простоте и подробной документации. Основное отличие Mercurial заключается в ее структуре, основанной на Python, что делает систему особенно привлекательной для программистов, работающих с этим языком. Несмотря на некоторые ограничения, такие как сложности с объединением родительских веток из-за архитектуры плагинов, Mercurial остается удобным и достаточно функциональным инструментом для управления версиями.

7. Средства и инструменты для разработки и тестирования API

***Средства разработки API***:

*Spring Boot*: Ведущий фреймворк для быстрой разработки микросервисов и REST API на Java. Предоставляет встроенную поддержку для создания, развертывания и управления API.

* Описание:Ведущий фреймворк для быстрой разработки микросервисов и RESTful API на Java. Предоставляет встроенную поддержку для создания, развертывания и управления API.
* Принцип работы: Spring Boot автоматически настраивает необходимые компоненты и конфигурацию приложения на основе разумных по умолчанию. Разработчики могут сосредоточиться на бизнес-логике, а не на технической инфраструктуре.
* Преимущества: Быстрая разработка и развертывание API, богатый набор встроенных возможностей, простая конфигурация, хорошая интеграция с другими компонентами Spring.
* Недостатки: Большой размер и сложность фреймворка, может быть избыточным для простых проектов.

*JAX-RS (Jersey, RESTEasy, Apache CXF):* Стандартные Java API для REST веб-сервисов. Позволяют быстро создавать RESTful API с аннотациями и встроенной поддержкой маршрутизации, сериализации и других функций.

* Описание: Стандартные Java API для построения RESTful веб-сервисов. Предоставляют аннотации и инструменты для быстрого создания API.
* Принцип работы: JAX-RS реализует спецификацию Java API for RESTful Web Services, обеспечивая простую и последовательную работу с HTTP-запросами и ответами.
* Преимущества: Стандартизированный подход, широкая поддержка в экосистеме Java, гибкость и расширяемость.
* Недостатки: Может требовать больше ручной конфигурации, чем Spring Boot.

*Spark Java:* Легкий и гибкий фреймворк для разработки API, который позволяет быстро создавать и развертывать REST-сервисы.

* Описание: Легкий и гибкий фреймворк для быстрой разработки RESTful API на Java.
* Принцип работы: Spark Java предоставляет DSL для определения маршрутов, обработки запросов и ответов. Отличается простотой и минималистичностью.
* Преимущества: Небольшой размер, высокая производительность, простота использования и настройки.
* Недостатки: Ограниченная функциональность по сравнению с крупными фреймворками, меньшая поддержка сообщества.

*Vert.x:* Реактивный фреймворк для создания масштабируемых и высокопроизводительных сетевых приложений, включая API.

* Описание: Реактивный фреймворк для создания высокопроизводительных сетевых приложений, включая RESTful API.
* Принцип работы: Vert.x использует реактивный, неблокирующий подход к обработке запросов, обеспечивая эффективное масштабирование.
* Преимущества: Высокая производительность, масштабируемость, поддержка реактивного программирования, интеграция с другими технологиями.
* Недостатки: Крутая кривая обучения, может быть сложнее в использовании, чем другие фреймворки.

***Средства тестирования API:***

*JUnit:* Основной фреймворк модульного тестирования в Java. Позволяет писать и запускать автоматические тесты для API.

* Описание: Основной фреймворк модульного тестирования в Java, позволяющий писать и запускать автоматические тесты.
* Принцип работы: JUnit предоставляет аннотации и API для определения тестовых методов, утверждений и жизненного цикла тестов.
* Преимущества: Простота использования, широкая поддержка в IDE и инструментах, большое сообщество.
* Недостатки: Ограничен только модульным тестированием, может требовать дополнительных библиотек для более сложных тестовых сценариев.

*RestAssured:* DSL для тестирования REST-сервисов, упрощающий написание и чтение тестов API.

* Описание: DSL (Domain-Specific Language) для тестирования REST-сервисов, упрощающий написание и чтение тестов API.
* Принцип работы: RestAssured предоставляет fluent API для выполнения HTTP-запросов, проверки ответов и валидации данных.
* Преимущества: Лаконичный и читабельный синтаксис, интеграция с популярными assert-библиотеками, удобство в использовании.
* Недостатки: Ограничен только тестированием REST API, может быть избыточным для простых тестовых сценариев.

*Postman*: Популярный GUI-инструмент для интерактивного тестирования, отладки и документирования API.

* Описание: Инструмент для проектирования, тестирования и документирования API.
* Принцип работы: Postman предоставляет интуитивный интерфейс для создания и отправки HTTP-запросов, а также анализа ответов.
* Преимущества: Простота использования, расширяемость с помощью скриптов, интеграция с CI/CD.
* Недостатки: Ограничения бесплатной версии, может быть недостаточно функциональным для сложных сценариев.

*SoapUI*: Комплексный инструмент для функционального, нагрузочного и других видов тестирования API, как REST, так и SOAP.

* Описание: Инструмент для тестирования веб-сервисов и API.
* Принцип работы: SoapUI позволяет создавать тестовые кейсы для SOAP и REST API, проверять запросы и ответы.
* Преимущества: Удобный графический интерфейс, широкий набор функций, интеграция с другими инструментами.
* Недостатки: Может быть громоздким для простых тестовых сценариев, требует изучения.

*Cucumber*: Фреймворк для поведенческого тестирования (BDD), позволяющий описывать тесты API на естественном языке.

* Описание: Фреймворк для разработки на основе поведения (BDD - Behavior-Driven Development).
* Принцип работы: Cucumber позволяет описывать тестовые сценарии на естественном языке, которые затем автоматизируются.
* Преимущества: Вовлечение бизнес-пользователей в процесс тестирования, улучшенная коммуникация между командами, повышение качества.
* Недостатки: Может увеличить время разработки, требует обучения команды.

*Mockito*: Мощный мок-фреймворк для заглушек зависимостей при модульном тестировании API.

* Описание: Фреймворк для создания and проверки mock-объектов в модульных тестах.
* Принцип работы: Mockito позволяет создавать, настраивать и проверять поведение mock-объектов, которые имитируют реальные зависимости в тестируемом коде.
* Преимущества: Простой и интуитивно понятный API, широкая поддержка в сообществе, хорошая интеграция с JUnit.
* Недостатки: Может быть сложнее в использовании для более сложных сценариев mock-объектов.

8. Система типов данных в языке Java, классы, объекты

Система типов данных:

Java - строго типизированный язык программирования, что означает, что каждая переменная должна быть объявлена с определенным типом данных.

Когда код на Java компилируется, машина проверяет соответствие типов операндов во всех методах, конструкторах и других операторах. Если в программе есть хотя бы одна недопустимая операция, компилятор не превратит её в байт-код. Поэтому контроль типов данных помогает уменьшить количество ошибок при написании программы.

В Java типы данных делят на две большие группы: ***примитивные*** и ***ссылочные***.

**Примитивные** типы данных в Java:

* целые числа (byte, short, int, long);

Тип ***byte***. Эти переменные используют, чтобы работать с потоком данных, который получили из файла или по сети.

Тип ***short***. По сравнению с byte у него увеличенный, но всё же ограниченный диапазон значений. Применяют short редко — например, когда нужно экономить память.

Тип ***int***. В языке Java int — самый популярный тип целочисленных данных. При вычислениях в виртуальной машине остальные целочисленные типы (byte, short) занимают столько же памяти, сколько int.

Тип ***long*** применяют, когда нужно работать с большими целочисленными значениями.

* числа с плавающей точкой (float, double);

Тип данных ***double*** используют для работы с десятичными числами.

Тип ***float*** используют как экономичный вариант хранения больших массивов данных с плавающей точкой. Когда переменной присваивают тип float, язык Java воспринимает её как тип данных double. Чтобы этого не происходило, нужно добавлять в конце переменной символ **f** или **F.** Не стоит использовать float, когда в вычислениях нужна точность больше пяти знаков после запятой.

* логический (boolean);

Тип данных ***boolean*** используют чтобы работать с логическими значениями. У такой переменной может быть только два значения: false (ложь) и true (истина). Это не аналог 1 или 0.

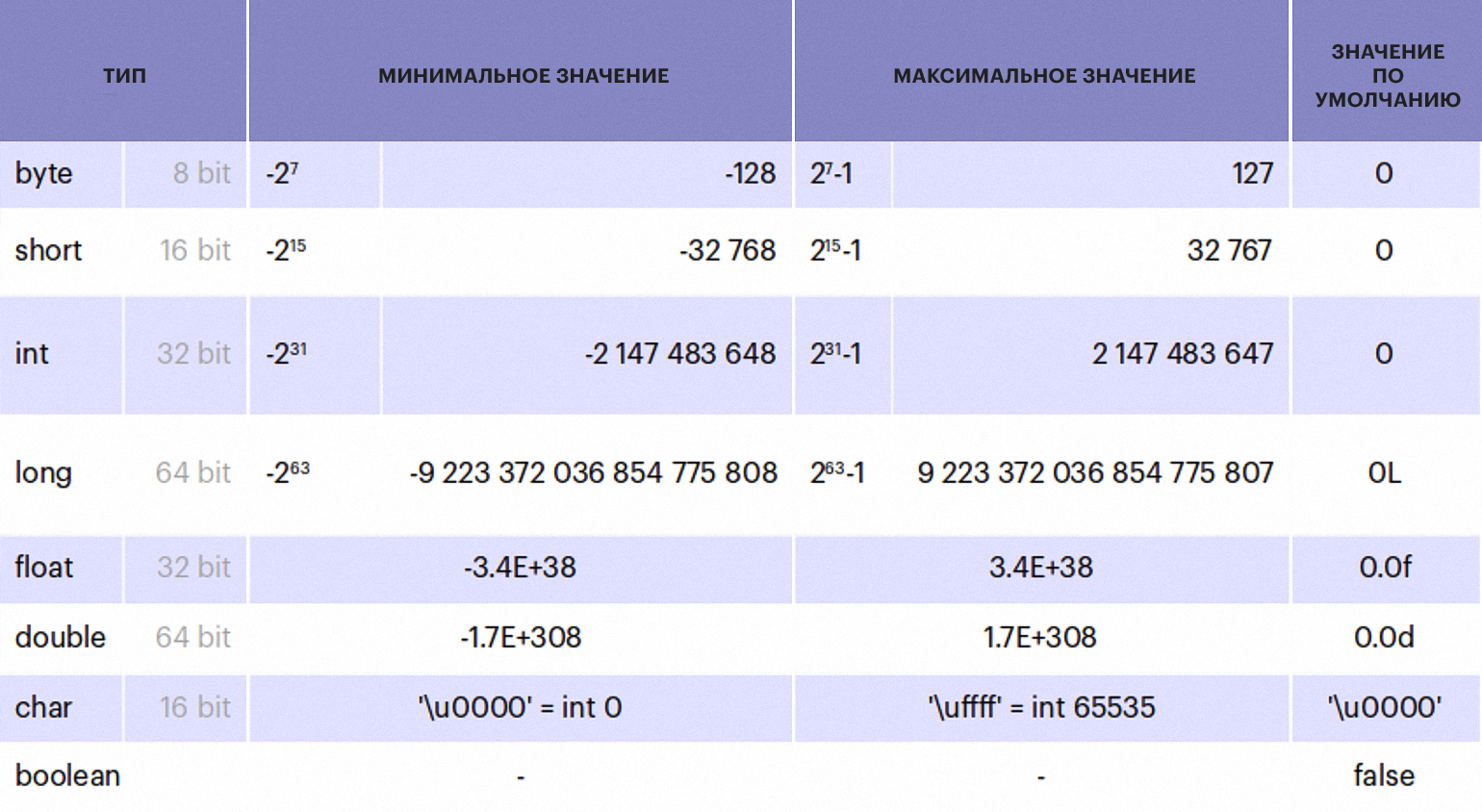
* символьный (char).

Тип данных ***char*** используют, чтобы хранить в переменных любые 16-разрядные символы Unicode. Но их нужно записывать строго в одинарные кавычки ' ', и только по одному. 'ж' не равно "ж".

**Ссылочные** типы данных ещё называют ссылками. К ним относятся все классы, интерфейсы, массивы, а также тип данных String.

Строчные переменные можно склеивать оператором +, который используют для конкатенации.

|  |  |
| --- | --- |
| **Примитивные переменные** | **Ссылочные переменные** |
| Хранят **значение** | Хранят **адрес объекта в памяти**, на который ссылаются (отсюда и название).  Используются **для доступа** к объектам (его нельзя получить, если на объект нет ссылки) |
| Создаются присваиванием значения | Создаются через **конструкторы** классов (**присваивание** только **создаёт вторую ссылку** на существующий объект) |
| Имеют **строго заданный диапазон** допустимых значений | По умолчанию их значение — null |
| **В аргументы** методов попадают **копии** значения переменной (это передача по значению) | В методы **передаётся** **значение ссылки** — операция выполняется над оригинальным объектом, на который ссылается переменная |
|  | Могут использоваться для ссылки на любой объект объявленного или совместимого типа |



**Boxing и unboxing** — как превратить примитив в объект

Иногда с примитивами приходится работать как с объектами — например, передавать им значение по ссылке или создавать список из чисел (а списки работают только с объектами).

Поэтому у каждого примитива есть соответствующий ему ссылочный тип — его называют классом-обёрткой. В таких классах хранятся методы для преобразования типов данных, а также другие константы и методы, которые применяются при работе с примитивами.

(byte Byte) (short Short) (int Integer) (long Long) (char Character) (float Float) (double Double) (Boolean Boolean)

Ссылочные типы данных (обёртки) пишут с прописной буквы, потому что это полноценные классы. А в Java названия всех классов должны начинаться с большой буквы — язык чувствителен к регистру.

**Классы и объекты:**

***Класс*** — это шаблонная конструкция, которая позволяет описать в программе объект, его свойства (атрибуты или поля класса) и поведение (методы класса)

***Объект*** — это экземпляр класса, который имеет свои собственные значения полей и может вызывать методы класса.

Для создания объекта используется оператор new, например: MyClass obj = new MyClass();.

С помощью специального слова ***static*** мы можем создать статические поля и методы. Эти поля и методы описывают уже *не объект класса, а сам класс*. То есть они вызываются по имени класса, а их значение — общее для всех объектов данного класса.

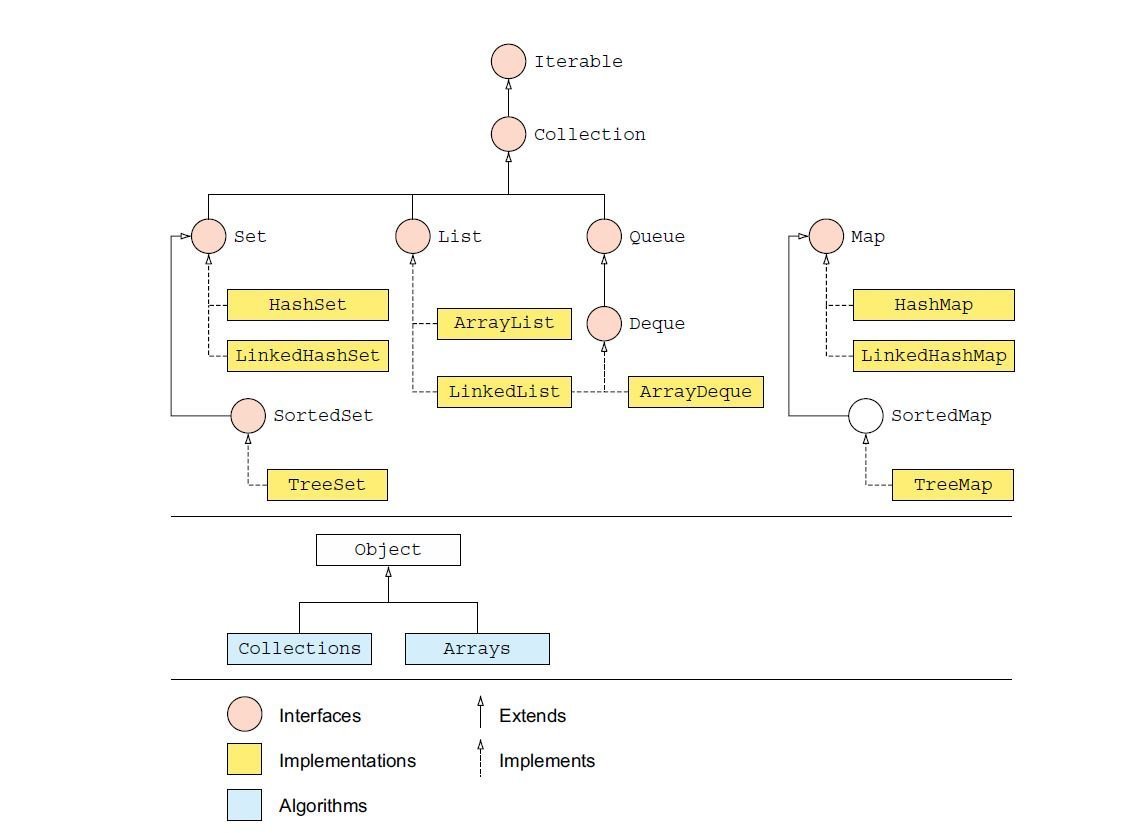
Ключевое слово ***this*** представляет ссылку на текущий экземпляр класса.

* Классы создаются в файлах с расширением .java. Главный класс должен называться так же, как имя файла. Например, в файле Main.java — класс Main.
* Разные классы в одном проекте могут иметь одинаковые имена, для этого им надо располагаться в разных пакетах (package).
* Нельзя создать метод или объявить переменную за пределами классов.
* Каждый раз мы начинаем программу со статического метода main, который имеет возвращаемый тип int, а в параметрах у него указана переменная args типа String[] (массив строк).
* Статические поля нужно использовать правильно. Например, можно с помощью статического поля сделать счётчик количества созданных объектов — он описывает именно класс, в котором расположен, а не сами объекты данного класса. Но нет смысла делать статическим поле, описывающее объект, если не можете придумать, как использовать информацию из этого поля в другом классе.

9. Основные коллекции в языке Java

Коллекции — это наборы однородных элементов.

Коллекции предоставляются через **Java Collections Framework (JCF)**, который обеспечивает единообразный интерфейс для работы с различными типами коллекций. JCF включает в себя интерфейсы, реализации этих интерфейсов и утилитарные классы для выполнения операций над коллекциями, таких как сортировка, поиск и преобразование.



**List**: Упорядоченный список, в котором у каждого элемента есть индекс. Дубликаты значений допускаются.

* ArrayList: Реализация списка, основанная на массиве.
* LinkedList: Реализация списка, основанная на связном списке.
* Vector: Похож на ArrayList, но является потокобезопасной.
* Stack: Стек, реализованный с помощью Vector.

**Set**: Неупорядоченная коллекция уникальных элементов (без дубликатов). Уникальность элементов проверяется с помощью метода equals().

* HashSet: Реализация множества, основанная на хеш-таблице.
* LinkedHashSet: Реализация множества, сохраняющая порядок добавления элементов.
* TreeSet: Реализация множества, сохраняющая элементы в отсортированном порядке.

**Queue**: Очередь. Структура данных, используемая для хранения элементов в определенном порядке, обычно FIFO (First-In-First-Out).

* ArrayDeque: Реализация очереди, основанная на массиве.
* PriorityQueue: Очередь с приоритетами, где элементы сортируются по приоритету.

**Map**: Состоит из пар «ключ-значение». Ключи уникальны, а значения могут повторяться. Порядок элементов не гарантирован. Map позволяет искать объекты (значения) по ключу

* HashMap: Реализация карты, основанная на хеш-таблице.
* LinkedHashMap: Реализация карты, сохраняющая порядок добавления элементов.
* TreeMap: Реализация карты, сохраняющая элементы в отсортированном порядке.
* Hashtable: Похож на HashMap, но является потокобезопасной.

Не путайте интерфейс **Collection** и фреймворк **Collections**. Map не наследуется от интерфейса Collection, но входит в состав фреймворка Collections.

**Реализации List**

Класс ***ArrayList*** подойдёт в большинстве случаев, если вы уже определились, что вам нужен именно список (а не Map, например).

Строится на базе обычного массива. Если при создании не указать размерность, то под значения выделяется 10 ячеек. При попытке добавить элемент, для которого места уже нет, массив автоматически расширяется — программисту об этом специально заботиться не нужно.

Класс ***LinkedList*** реализует одновременно List и Deque. Это список, в котором у каждого элемента есть ссылка на предыдущий и следующий элементы.

Если добавлять и удалять элементы с произвольными индексами в списке нужно чаще, чем итерироваться по нему, то лучше LinkedList. В остальных случаях — ArrayList.

**Реализации Queue**

Про ***LinkedList***, мы рассказали выше.

Класс ***ArrayDeque*** — это реализация двунаправленной очереди в виде массива с переменным числом элементов. Новые значения можно добавлять в начало или конец списка, и удалять оттуда же. Причём эти операции выполняются быстрее, чем при использовании LinkedList.

Класс ***PriorityQueue*** — упорядоченная очередь. По умолчанию элементы добавляются в естественном порядке: числа по возрастанию, строки по алфавиту и так далее, либо алгоритм сравнения задаёт разработчик.

Этот класс может быть полезен, например, для нахождения n минимальных чисел в большом неупорядоченном списке.

Такая реализация выгоднее по скорости и объёму памяти, чем подход с сортировкой первоначального списка.

**Реализации Set**

Класс ***HashSet*** использует для хранения данных в хеш-таблице. Это значит, что при манипуляциях с элементами используется хеш-функция — **hashCode()** в Java.

Прежде чем добавить новый элемент в множество, вычисляется его **hashCode()** — чтобы определить бакет, куда он может быть помещён. Если бакет пуст, элемент будет добавлен. Иначе уже добавленные элементы с таким же значением хеша сравниваются с кандидатом при помощи метода **equals()**. Если дубликат не найден, новый элемент становится частью множества. Он попадёт в тот же бакет.

Хеш-таблица — структура данных, в которой все элементы помещаются в бакеты (buckets), соответствующие результату вычисления хеш-функции.

Алгоритм вычисления — хеш-функция, а результат вычисления — хеш-код.

Тогда ключ от номера 356 попадёт в коробку 5 (3 + 5 + 6 = 14; 1 + 4 = 5), а ключ от номера

О классе ***TreeSet*** вспоминают в тех случаях, когда множество должно быть упорядочено. Каким образом упорядочивать — определяет разработчик при создании нового TreeSet. По умолчанию элементы располагаются в естественном порядке.

**Реализации Map**

Класс ***HashMap*** хранит данные в виде хеш-таблицы, как и HashSet. Более того, **HashSet внутри использует HashMap**. При этом ключом выступает сам элемент.

Класс ***TreeMap***.Элементы здесь упорядочены (в естественном или заданном при создании порядке) в каждый момент времени. При этом вставка и удаление более затратны, чем в случае с HashMap.

Класс ***LinkedHashMap*** расширяет возможности HashMap тем, что позволяет итерироваться по элементам в порядке их добавления. Как и в LinkedList, здесь каждая пара-значение содержит ссылку на предыдущий и последующий элементы.

* В каких коллекциях допускаются null-элементы?

Почти во всех, но нельзя добавлять null-значения в упорядоченные структуры, которые при добавлении нового элемента используют сравнение. Поэтому null-значения запрещены в TreeMap и TreeSet. Ещё они недопустимы в ArrayDeque, так как методы этого класса (например, poll() — удаление элемента из начала очереди) используют null как признак пустоты коллекции.

10. Работа со строками в языке Java: String, StringBuffer, StringBuilder

Реализация строк на Java представлена тремя основными классами: String, StringBuffer, StringBuilder.

**String**

Строка — объект, что представляет последовательность символов. Для создания и манипулирования строками Java платформа предоставляет **общедоступный final (не может иметь подклассов)** класс java.lang.String. Данный **класс является неизменяемым** (**immutable**) — созданный объект класса String не может быть изменен. Можно подумать, что методы имеют право изменять этот объект, но это неверно. Методы могут только создавать и возвращать новые строки, в которых хранится результат операции. Неизменяемость строк предоставляет ряд возможностей:

* использование строк в многопоточных средах (String является потокобезопасным)
* использование String Pool (это коллекция ссылок на String объекты, используется для оптимизации памяти)
* использование строк в качестве ключей в HashMap (ключ рекомендуется делать неизменяемым)

**Создание**

* Используя строковые литералы (“blablabla”):

Строковый литерал — последовательность символов заключенных в двойные кавычки. Важно понимать, что всегда, когда вы используете строковый литерал компилятор создает объект со значением этого литерала:

* С помощью конструкторов:

String habr = "habrahabr";

String second = new String(habr);

Конструкторы могут формировать объект строки с помощью ***массива символов***. Происходит копирование массива, для этого используются статические методы copyOf и copyOfRange (копирование всего массива и его части (если указаны 2-й и 3-й параметр конструктора) соответственно) класса Arrays, которые в свою очередь используют платформо-зависимую реализацию System.arraycopy.

Можно также создать объект строки с помощью ***массива байтов***. Дополнительно можно передать параметр класса Charset, что будет отвечать за кодировку. Происходит декодирование массива с помощью указанной кодировки (если не указано — используется Charset.defaultCharset(), который зависит от кодировки операционной системы) и, далее, полученный массив символов копируется в значение объекта.

Ну и наконец-то ***конструкторы использующие объекты StringBuffer и StringBuilder***, их значения (getValue()) и длину (length()) для создания объекта строки.

Конструкторов класса String пятнадцать (два из которых помечены как deprecated).

**Длина**

Важной частью каждой строки есть ее длина. Узнать ее можно обратившись к объекту String с помощью метода доступа (accessor method) length(), который возвращает количество символов в строке.

**Конкатенация**

Конкатенация — операция объединения строк, что возвращает новую строку, что есть результат объединения второй строки с окончанием первой. Операция для объекта String может быть выполнена двумя способами:

* Метод concat

String javaHub = "habrhabr".concat(".ru").concat("/hub").concat("/java");

Важно понимать, что метод concat не изменяет строку, а лишь создает новую как результат слияния текущей и переданной в качестве параметра. Да, метод возвращает новый объект String, поэтому возможны такие длинные «цепочки».

* Перегруженные операторы "+" и "+="

String habr = "habra" + "habr";

habr += ".ru"; // "habrahabr.ru"

Это одни с немногих перегруженных операторов в Java — язык не позволяет перегружать операции для объектов пользовательских классов. Оператор "+" не использует метод concat, тут используется следующий механизм:

// все просто и красиво

String habrahabr = habra + habr;

// а на самом деле

String habrahabr = new StringBuilder()).append(habra).append(habr).toString(); // может быть использован StringBuffer

Используйте метод concat, если слияние нужно провести только один раз, для остальных случаев рекомендовано использовать или оператор "+" или StringBuffer / StringBuilder. Также стоит отметить, что получить NPE (NullPointerException), если один с операндов равен null, невозможно с помощью оператора "+" или "+=", чего не скажешь о методе concat.

**Форматирование**

Класс String предоставляет возможность создания форматированных строк. За это отвечает статический метод ***format***, например:

String formatString = "We are printing double variable (%f), string ('%s') and integer variable (%d).";

System.out.println(String.format(formatString, 2.3, "habr", 10));

**Методы**

Очень дофига

**Преобразование**

* Число в строку

int integerVariable = 10;

String first = integerVariable + ""; // конкатенация с пустой строкой

String second = String.valueOf(integerVariable); // вызов статического метода valueOf класса String

String third = Integer.toString(integerVariable); // вызов метода toString класса-обертки

* Строку в число

String string = "10";

int first = Integer.parseInt(string); получаем примитивный тип (primitive type) используя метод parseXхх нужного класса-обертки, где Xxx - имя примитива с заглавной буквы (например parseInt)

int second = Integer.valueOf(string); // получаем объект wrapper класса и автоматически распаковываем

**StringBuffer**

Строки являются неизменными, поэтому частая их модификация приводит к созданию новых объектов, что в свою очередь расходует память. Для решения этой проблемы был создан класс java.lang.StringBuffer, который позволяет более эффективно работать над модификацией строки. Класс является **mutable**, то есть **изменяемым** — используйте его, если Вы хотите изменять содержимое строки. StringBuffer может быть использован в многопоточных средах, так как все необходимые методы являются синхронизированными.

***Создание***

Существует четыре способа создания объекта класса StringBuffer. Каждый объект имеет свою вместимость (capacity), что отвечает за длину внутреннего буфера. Если длина строки, что хранится в внутреннем буфере, не превышает размер этого буфера (capacity), то нет необходимости выделять новый массив буфера. Если же буфер переполняется — он автоматически становится больше.

StringBuffer firstBuffer = new StringBuffer(); // capacity = 16

StringBuffer secondBuffer = new StringBuffer("habrahabr"); // capacity = str.length() + 16

StringBuffer thirdBuffer = new StringBuffer(secondBuffer); // параметр - любой класс, что реализует CharSequence

StringBuffer fourthBuffer = new StringBuffer(50); // передаем capacity

***Модификация***

В большинстве случаев мы используем StringBuffer для многократного выполнения операций добавления (append), вставки (insert) и удаления (delete) подстрок. Тут все очень просто, например:

String domain = ".ru";

StringBuffer buffer = new StringBuffer("habrahabr");

buffer.append(domain); // "habrahabr.ru"

buffer.delete(buffer.length() - domain.length(), buffer.length()); // "habrahabr"

buffer.insert(buffer.length(), domain); // "habrahabr.ru"

Все остальные методы для работы с StringBuffer можно посмотреть в документации.

**StringBuilder**

StringBuilder — класс, что представляет изменяемую последовательность символов. Класс был введен в Java 5 и имеет полностью идентичный API с StringBuffer. Единственное отличие — StringBuilder ***не синхронизирован***. Это означает, что его использование в многопоточных средах есть нежелательным.

Следовательно, если вы работаете с многопоточностью, Вам идеально подходит StringBuffer, иначе используйте StringBuilder, который работает намного быстрее в большинстве реализаций.

11. ООП в Java: наследование, инкапсуляция, полиморфизм

Объектно-ориентированное программирование (ООП) в Java — это парадигма разработки, которая организует программу вокруг объектов, представляющих собой экземпляры классов. Классы в Java служат шаблонами для создания объектов, объединяя данные (переменные) и методы (функции), которые оперируют этими данными. Основные принципы ООП в Java включают инкапсуляцию, наследование и полиморфизм.

**Инкапсуляция** – сокрытие внутренних данных компонента и деталей его реализации от других компонентов приложения и предоставление набора методов для взаимодействия с ним (API).

* Она способствует переиспользованию компонентов: компоненты взаимодействуют друг с другом только посредством их API и безразличны к изменениям внутренней структуры, они могут использоваться в более широком контексте.
* Инкапсуляция ускоряет процесс разработки: слабо связанные друг с другом компоненты (то есть компоненты, чей код как можно меньше обращается или использует код других компонентов) могут разрабатываться, тестироваться и дополняться независимо.
* Правильно инкапсулированные компоненты более легки для понимания и процесса отладки, что упрощает поддержку приложения.

В языке Java инкапсуляция реализована с помощью системы классов, которые позволяют собрать информацию об объекте в одном месте; пакетов, которые группируют классы по какому-либо критерию, и модификаторов доступа, которыми можно пометить весь класс или его поле или метод.

* ***public*** – полный доступ к сущности (полю или методу класса) *из любого пакета*;
* ***protected*** – доступ к сущности только *для классов своего пакета и наследников класса*;
* ***неявный модификатор по умолчанию*** (при отсутствии трёх явных) – доступ к сущности только *для классов своего пакета*;
* ***private*** – доступ только *внутри класса, в котором объявлена сущность*.

Используются методы getterы и setterы.

**Наследование** – позволяет создавать новые классы на основе существующих, наследуя их свойства и методы.

В терминах Java такой общий класс называется ***суперклассом*** (superclass), или ***базовым классом*** (base class), или ***классом-родителем*** (parent class).

Класс, его наследующий, - ***подклассом*** (subclass), или ***дочерним классом*** (child class), или ***классом-потомком*** (derived class).

Любой класс может иметь не более одного класса-родителя. А все классы, суперкласс у которых явно не указан, по умолчанию наследуются от класса Object. ключевое слово extends.

**Полиморфизм** позволяет объектам принимать множество форм. В Java это проявляется в возможности переопределения методов в подклассах и перегрузки методов с разным набором параметров.

Вызов переопределённого метода через переменную класса-родителя получает поведение, которое будет соответствовать реальному классу-потомку, на который ссылается эта переменная

\*\*\***Абстракция** как 4 принцип: фокусировка разработчика на конкретных свойствах объекта зависит от тех задач, которые призван решать объект. Следствием такого подхода является то, что, если в императивных языках разработчику необходимо думать в терминах компьютерной логики, в объектно-ориентированных языках разработчик думает в терминах проблемной области, в которой он разрабатывает приложения.

Абстракция (от лат. abstractio — выделение, отвлечение или отделение) — теоретический прием исследования, позволяющий отвлечься от некоторых несущественных в определенном отношении свойств изучаемых явлений и выделить свойства существенные и определяющие.

12. Интерфейсы и абстрактные классы в Java

**Интерфейс** — это контракт, который реализуется в некотором классе. У интерфейса не может быть состояния, поэтому в нем нельзя использовать изменяемые поля экземпляра. В интерфейсе могут быть только неизменяемые final-поля.

* Интерфейсы очень полезны для уменьшения связанности (coupling) кода и реализации полиморфизма.
* Если вы заметили, что ряд ваших классов содержит одинаковые методы, но с разными реализациями, то стоит использовать интерфейс.
* Методы интерфейса неявно абстрактны и обязаны быть реализованы в классе, реализующем этот интерфейс.
* Интерфейс может содержать только неизменяемые переменные. Они неявно final и static

После появления в Java 8 методов по умолчанию, некоторые разработчики решили, что интерфейсы стали абстрактными классами. Однако это не так, поскольку у интерфейсов не может быть состояния.

У методов по умолчанию может быть реализация, а у абстрактных методов — нет. Методы по умолчанию — результат появления лямбда-выражений и Stream API, но использовать их нужно с осторожностью.

В качестве примера default-метода из JDK можно привести метод forEach() из интерфейса Iterable. Вместо копирования кода этого метода во все реализации Iterable, мы можем переиспользовать метод forEach.

Важно отметить, что у default-метода должна быть реализация и default-метод не может быть статическим.

**Абстрактный класс** похож на обычный, но отличается тем, что может содержать абстрактные методы — методы без реализации, и нельзя создать экземпляр абстрактного класса.

У абстрактных классов может быть состояние в виде изменяемых полей экземпляра.

Аналогично интерфейсам в абстрактных классах могут быть абстрактные методы. Абстрактный метод — это метод без тела (без реализации). Но в отличие от интерфейсов, абстрактные методы в абстрактных классах должны быть явно объявлены как абстрактные *abstract*.

Рекомендуется использовать абстрактный класс, когда вам нужно изменяемое состояние.

|  |  |
| --- | --- |
| **Интерфейсы** | **Абстрактные классы** |
| Могут содержать только final static поля. Интерфейс никогда не может изменять свое состояние | Могут быть любые поля, в том числе статические, изменяемые и неизменяемые |
| Класс может реализовывать несколько интерфейсов | Класс может расширять только один абстрактный класс |
| Может быть реализован с помощью ключевого слова implements.  Может расширять другой интерфейс с помощью extends | Может быть только расширен с помощью extends |
| Можно использовать только static final поля. Параметры и локальные переменные в методах | Могут быть изменяемые поля экземпляра. Параметры и локальные переменные в методах |
| В лямбда-выражениях могут использоваться только функциональные интерфейсы | Абстрактные классы с одним абстрактным методом не могут использоваться в лямбда-выражениях |
| Не может быть конструктора | Может содержать конструктор |
| Могут быть абстрактные методы.  Могут быть default и static методы  (c Java 8)  Могут быть private методы с реализацией (с Java 9) | Могут быть любые методы |

13. Методы в Java: статические методы, абстрактные методы, перегрузка методов (overriding, overloading)

**Статические методы**

* Статические методы принадлежат классу, а не отдельному объекту.
* Они могут вызываться без создания экземпляра класса, используя синтаксис ClassName.methodName().
* Статические методы имеют доступ только к статическим членам класса (переменным и методам). Они не имеют доступа к нестатическим полям и методам, так как для этого требуется экземпляр класса.
* Статические методы часто используются для реализации вспомогательных функций, которые не зависят от состояния объекта.

**Абстрактные методы**

* Абстрактные методы объявляются в абстрактных классах и не имеют реализации.
* Они определяют контракт, который должен быть реализован в наследующих классах.
* Класс, содержащий хотя бы один абстрактный метод, также должен быть объявлен как абстрактный.
* Абстрактные классы не могут быть напрямую инстанцированы, вместо этого должен быть создан экземпляр конкретного наследующего класса.
* Наследующие классы должны реализовать все абстрактные методы родительского класса или также быть объявлены как абстрактные.
* Абстрактные методы используются для определения общего поведения, которое должно быть реализовано в подклассах.

**Переопределение (Overriding)**

* Переопределение методов позволяет дочернему классу предоставить свою реализацию метода, унаследованного от родительского класса.
* Сигнатура переопределяемого метода в дочернем классе должна точно совпадать с сигнатурой метода в родительском классе.
* Доступность переопределяемого метода в дочернем классе должна быть такой же или более доступной, чем в родительском классе.
* Переопределение методов позволяет реализовать полиморфизм, когда один и тот же метод ведет себя по-разному в зависимости от типа объекта.

@Override

**Перегрузка методов (Overloading)**

* Перегрузка методов позволяет определять несколько методов в одном классе с одинаковым именем, но с разными сигнатурами.
* Сигнатура метода определяется количеством, типами и порядком параметров.
* При вызове метода компилятор использует сигнатуру для определения, какая реализация должна быть вызвана.
* Перегрузка методов позволяет создавать более гибкие и интуитивно понятные интерфейсы.

14. Обработка ошибок в Java: try-catch-finally

**Error** — это критические условия, в которых работа программы должна быть завершена.

**Exception**. Эти исключительные ситуации возникают, если разработчик допустил невыполнимую операцию, не предусмотрел особые случаи в бизнес-логике программы (или сообщает о них с помощью исключений).

* Невыполнимая операция

Java не знает, что делать дальше. Частая ситуация — обращение к несуществующему элементу массива.

* Особый случай в бизнес-логике программы

Пользователь вводит дату начала некоторого периода и дату его окончания. Вторая дата не может быть раньше первой.

* **try** – определяет блок кода, в котором может произойти исключение;
* **catch** – определяет блок кода, в котором происходит обработка исключения;
* **finally** – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.

Эти ключевые слова используются для создания в программном коде специальных обрабатывающих конструкций: try{}catch, try{}catch{}finally, try{}finally{}.

**throw** – используется для возбуждения исключения;

**throws** – используется в сигнатуре методов для предупреждения, о том, что метод может выбросить исключение.

15. Обработка ошибок в Java: logging, фреймворки для логирования

**Логирование** - это процесс записи информации о действиях и событиях, происходящих в приложении, в специальные файлы или другие устройства хранения данных. Логи помогают разработчикам отслеживать работу приложения, выявлять проблемы и анализировать его поведение. Логирование – процесс формирования логов.

**Лог (log)** - это хронологическая запись наиболее значимой информации о работе системы.

Уровни логирования:

* TRACE: Самый низкий уровень, используется для отслеживания деталей выполнения программы.
* DEBUG: Уровень для отладочной информации, которая полезна для разработчиков.
* INFO: Уровень для информационных сообщений о ходе работы программы.
* WARN: Уровень для предупреждений о потенциальных проблемах, которые не являются критическими.
* ERROR: Уровень для сообщений об ошибках, которые могут привести к неправильной работе приложения.
* FATAL: Самый высокий уровень, используется для сообщений об ошибках, которые приводят к критическим сбоям и остановке работы приложения.

Фреймворки для логирования:

* Log4j — это надежный, быстрый и гибкий фреймворк для логирования на Java, позволяет настраивать поведение логирования во время выполнения через внешние файлы конфигурации. Поддерживает множество уровней логирования и направляет информацию логирования в разные цели: базы данных, файлы, консоль и др.
* Logback является прямым потомком Log4j и считается его продолжением. Он предлагает простоту использования и легкую конфигурацию, делая его хорошим выбором для проектов, которым требуется надежное решение без необходимости сложной настройки. Logback поддерживает большинство функций Log4j и предлагает дополнительные возможности, например более эффективное управление памятью.
* java.util.logging (JUL) является встроенным API для логирования в Java. Он предоставляет базовые возможности для записи сообщений в журналы, что полезно для отслеживания ошибок, аудита и мониторинга приложений. JUL позволяет настраивать уровни логирования, обработчики (handlers) для перенаправления логов в различные места (например, файлы, консоль), а также форматтеры (formatters) для управления форматом логовых сообщений.
* SLF4J выступает в роли абстрактной фасады для различных фреймворков логирования, таких как Logback или Log4j 2. Его главное преимущество заключается в гибкости: разработчики не привязаны к конкретному фреймворку логирования и могут динамически переключаться между ними без изменения своего кода. SLF4J предоставляет единый API для логирования, что упрощает интеграцию различных фреймворков логирования в одном проекте

Стек ELK — это популярное открытое решение для сбора, анализа и визуализации журналов и данных. Он состоит из трех основных компонентов:

**Elasticsearch**: Elasticsearch - это распределенный, RESTful поисковый и аналитический движок. Он используется для хранения и индексации данных, а также для предоставления мощных возможностей поиска и запросов.

**Logstash**: Logstash - это конвейер обработки данных, который принимает данные из различных источников, преобразует их и затем отправляет в Elasticsearch для хранения.

**Kibana**: Kibana - это инструмент визуализации и исследования данных, который используется для создания панелей мониторинга, визуализаций и отчетов на основе данных, хранящихся в Elasticsearch.

Преимущества использования стека ELK включают:

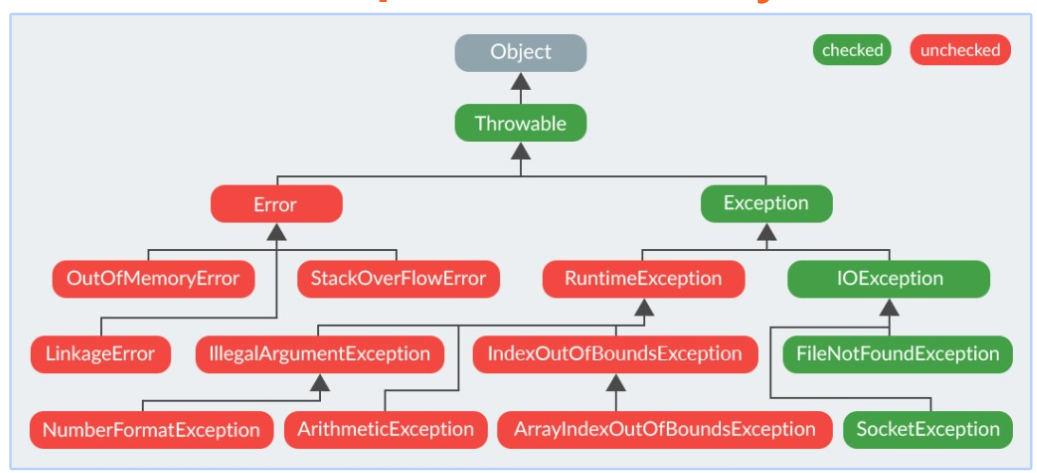
* Централизованное управление журналами: Стек ELK предоставляет централизованную платформу для сбора, хранения и анализа журналов из различных источников, упрощая процесс устранения неполадок и мониторинга приложений.
* Анализ данных в реальном времени: позволяет быстро анализировать и визуализировать ваши данные.
* Гибкость и масштабируемость: Стек ELK является highly scalable и может справляться с большими объемами данных.
* Настраиваемые панели мониторинга и визуализации: Kibana предоставляет удобный интерфейс для создания пользовательских панелей мониторинга и визуализаций, упрощая процесс получения информации из ваших данных.

Стек ELK широко используется в различных сценариях, таких как:

* Журналирование приложений: Сбор и анализ журналов из приложений, служб и инфраструктуры.
* Безопасность и соответствие требованиям: Мониторинг и анализ событий, связанных с безопасностью, таких как активность пользователей и системные предупреждения.
* Мониторинг и оповещение: Отслеживание и визуализация метрик производительности, а также настройка оповещений для критических событий.
* Бизнес-аналитика: Анализ и визуализация данных из различных источников для получения информации и поддержки принятия решений.

Чтобы начать использовать стек ELK, вам необходимо установить и настроить каждый из трех компонентов. Это можно сделать вручную или с помощью предварительно настроенного решения, такого как **Elastic Cloud** **или развертывание на основе Docker**. После настройки вы можете начать ingesting данные, создавать визуализации и строить пользовательские панели мониторинга, чтобы удовлетворить ваши конкретные потребности.

16. Обработка ошибок в Java: exceptions (checked/unchecked)



Все исключения в Java делятся на 2 вида — проверяемые (**checked**) и непроверяемые (**unchecked**).

**Проверяемые исключения** — это ошибки, которые Java-компилятор может предсказать во время компиляции кода. Это чаще не ошибки програмной логики, а ошибки ввода-вывода, ошибки SQL и другие.

Код, который может вызвать проверяемое исключение, **должен либо обработать** его с помощью блока try-catch, **либо передать его вверх по стеку вызовов** с помощью ключевого слова throws.

**Непроверяемые исключения** — это ошибки, которые Java-компилятор не может предсказать во время компиляции. Поэтому они обычно связаны с логическими ошибками в коде.

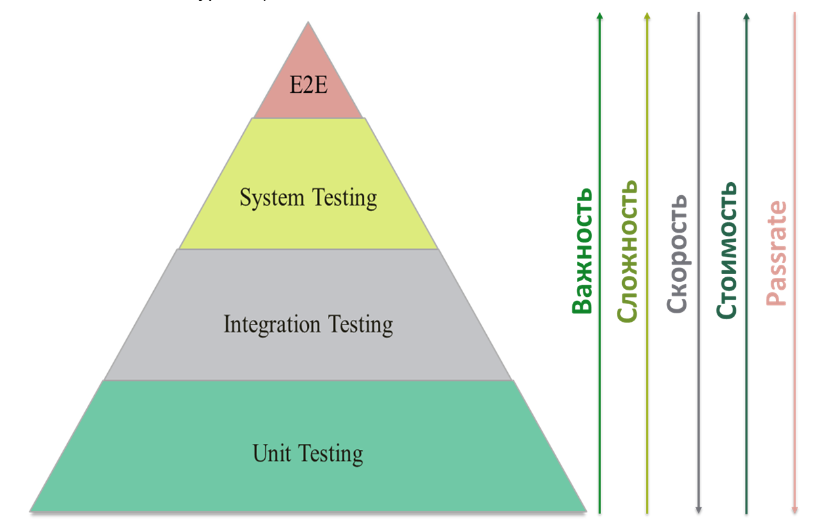
Код, который может вызвать непроверяемое исключение, **не обязан его обрабатывать** или объявлять с помощью throws.

**@ControllerAdvice** - Аннотация, которая позволяет объявить класс в качестве ***глобального обработчика исключений*** для всех контроллеров в приложении. Это означает, что методы в этом классе будут вызываться при возникновении исключений в любом из контроллеров. Обычно это используется для централизованной обработки исключений и управления ошибками в приложении.

**@ExceptionHandler** – Аннотация, которая используется внутри классов @ControllerAdvice или контроллеров ***для определения методов, которые обрабатывают конкретные типы исключений***. Когда исключение указанного типа происходит в контроллере, метод, помеченный @ExceptionHandler, будет вызван для его обработки.

**@Valid** - Аннотация используется в Spring Boot ***для включения автоматической валидации объектов***. Когда объект помечен аннотацией @Valid перед его параметром в методе контроллера, Spring будет автоматически проводить валидацию этого объекта перед его использованием в методе.

17. Типы автоматизированного тестирования кода: unit/integration testing



**Модульное** (Unit testing) — тестирование одного модуля в изоляции.

**Интеграционное** (Integration Testing) — тестирование группы взаимодействующих модулей.

**Системное** (System Testing) — тестирование системы в целом. Цель — убедиться, что система работает как предполагалось.

**Приемочное** (E2E) - Проверяет систему на соответствие требованиям пользователя. Цель — убедиться, что система готова к использованию.

|  |  |
| --- | --- |
| Модульное тестирование проверяет только функциональность самих модулей и может не выявить ошибки интеграции или другие общесистемные проблемы. | Интеграционное тестирование может обнаружить ошибки, когда модули интегрируются для построения общей системы. |
| Все начинается со спецификации модуля. | Все начинается со спецификации интерфейса |
| Он обращает внимание на поведение отдельных модулей. | Он уделяет внимание интеграции между модулями. |
| Модульный тест не проверяет, правильно ли ваш код работает с внешними зависимостями. | Интеграционные тесты проверяют, правильно ли ваш код работает с внешними зависимостями. |
| Обычно это выполняется разработчиком | Обычно его выполняет команда тестировщиков. |
| Найти ошибки легко | Найти ошибки сложно |
| Сопровождение модульного теста обходится дешево | Поддержка интеграционного теста стоит дорого |

18. Автоматизированное тестирование кода: mocks, stubs, e2e тестирование

**Mocks (Моки):**

* Mocks - это объекты, которые имитируют поведение реальных объектов в системе, используются в тестах для изоляции единицы тестирования, предоставляя контролируемые ответы на вызовы.
* Mocks позволяют контролировать входные данные, выходные данные и поведение зависимых компонентов, чтобы сфокусироваться на тестировании целевого функционала.
* Использование mocks помогает избежать побочных эффектов, связанных с взаимодействием с реальными, внешними компонентами (база данных, веб-сервис).
* Mocks также позволяют симулировать сценарии ошибок и исключительных ситуаций.

**Stubs (Заглушки):**

* Stubs - это упрощенные реализации внешних компонентов, которые предоставляют заранее определенные ответы для тестируемого кода.
* Stubs используются для замены реального поведения зависимых компонентов, чтобы изолировать тестируемый код.
* В отличие от mocks, stubs не проверяют взаимодействие с ними (были ли сделаны определённые вызовы), а просто предоставляют ожидаемые данные.
* Stubs помогают ускорить выполнение тестов, так как они не требуют запуска реальных внешних компонентов.

**End-to-End (E2E) тестирование**:

* E2E тестирование симулирует пользовательские сценарии, используя реальные компоненты приложения.
* Цель E2E тестов - проверить, что все компоненты системы работают вместе правильно, как единое целое.
* E2E тесты запускают приложение в реальной среде (или максимально приближенной к ней) и взаимодействуют с ним, как обычный пользователь.
* E2E тесты помогают обнаруживать ошибки, связанные с интеграцией компонентов, и гарантировать, что приложение работает как ожидается в реальных условиях.
* Однако E2E тесты могут быть более медленными и сложными в поддержке, чем unit-тесты с использованием mocks и stubs.

**Selenium** - популярная платформа для автоматизации E2E-тестирования веб-приложений. Это инструмент с открытым исходным кодом, который позволяет программировать автоматизированные тесты для веб-браузеров. Основные возможности Selenium:

**Selenium WebDriver:**

* Selenium WebDriver - это API, который предоставляет единый интерфейс для управления различными веб-браузерами.
* WebDriver позволяет программно взаимодействовать с элементами страницы, симулировать пользовательские действия, получать данные и проверять ожидаемое поведение.
* WebDriver совместим с популярными языками программирования, такими как Java, Python, C#, Ruby и другие.

**Selenium Grid:**

* Selenium Grid - это распределенная архитектура, которая позволяет запускать тесты на нескольких машинах одновременно.
* Это помогает параллельно выполнять тесты на разных браузерах, операционных системах и устройствах.
* Selenium Grid значительно ускоряет выполнение E2E-тестов, особенно для крупных проектов.

**Запись и воспроизведение:**

* Selenium предоставляет инструменты для записи пользовательских действий и их последующего воспроизведения в виде автоматизированных тестов.
* Это упрощает создание базовых тестов без необходимости писать код вручную.

**Интеграция с CI/CD:**

* Selenium тесты можно интегрировать в процессы непрерывной интеграции и доставки (CI/CD).
* Это позволяет автоматически запускать E2E-тесты при каждом изменении кода, обеспечивая быструю обратную связь.

19. Multithreading и concurrency в Java: Thread, concurrent collections, synchronized блоки и методы

**Многопоточность** в Java - это возможность выполнять несколько потоков одновременно в рамках одной программы, что увеличивает производительность и позволяет создавать более отзывчивые приложения.

**Поток** представляет собой последовательность инструкций, которая может выполняться параллельно или конкурентно с другими потоками.

Java предоставляет мощные средства для работы с многопоточностью, включая встроенную поддержку потоков и синхронизацию, а также богатую библиотеку классов для управления параллельным выполнением.

**Методы многопоточности в Java**

* **start():** Инициирует выполнение потока.
* **run():** Запускает действие для потока.
* **isAlive():** Проверяет, жив ли поток.
* **sleep():** Приостанавливает поток на заданное время.
* **yield()**: Передает управление другому потоку.
* **suspend():** Немедленно приостанавливает выполнение потока.
* **resume():** Возобновляет выполнение приостановленного потока.
* **interrupt():** Прерывает выполнение потока.
* **destroy():** Уничтожает группу потоков.
* **stop():** Останавливает выполнение потока.

**Преимущества многопоточности**

* Потоки независимы, что позволяет продолжать выполнение других потоков, даже если один из них завершился из-за ошибки.
* Многопоточность экономит время, позволяя выполнять множество операций одновременно.
* Потоки разделяют адресное пространство, что делает переключение между ними быстрее, чем между процессами.

**Жизненный цикл потока**

Жизненный цикл потока в Java включает несколько состояний, через которые проходит поток в процессе его выполнения. Вот подробное описание каждого из этих состояний:

* **НОВЫЙ** (New): В этом состоянии поток только создан, но еще не начал свое выполнение. Обычно поток создается внутри родительского процесса или другого потока. При создании поток получает ресурсы, необходимые для своего выполнения, такие как стек вызовов и регистры. В этом состоянии код потока еще не выполняется. Поток переходит из состояния "НОВЫЙ" в состояние "ГОТОВЫЙ к выполнению" (Runnable) после вызова метода `start()`.
* **ГОТОВЫЙ к выполнению** (Runnable): Поток, который готов к выполнению, переходит в состояние "ГОТОВЫЙ к выполнению". В этом состоянии поток может быть либо выполняется, либо ожидает своей очереди для выполнения. Это зависит от планировщика потоков, который определяет, когда поток получит время для выполнения [1].
* **ВЫПОЛНЕНИЕ** (Running): Когда поток получает процессорное время, он переходит в состояние выполнения. В этом состоянии поток фактически выполняет свой код и инструкции. Он использует процессорные ресурсы для выполнения своих задач.
* **БЛОКИРОВАННЫЙ** (Blocked): Поток может быть переведен в состояние блокировки, когда ему требуется ждать какого-либо события или завершения определенной операции. Например, поток может быть заблокирован в ожидании завершения операции ввода-вывода или получения данных из сети. В состоянии блокировки поток не использует процессорное время и освобождает его для выполнения других потоков.
* **ОЖИДАЮЩИЙ** (Waiting): Поток переходит в состояние "ОЖИДАЮЩИЙ", когда он вызывает метод `wait()` или `join()`. Поток переходит из состояния "ОЖИДАЮЩИЙ" в состояние "ГОТОВЫЙ к выполнению" (Runnable), когда другой поток вызывает метод `notify()` или `notifyAll()`, или когда поток, на который он ожидает, завершает свое выполнение.
* **ОЖИДАЮЩИЙ с таймаутом** (Timed Waiting): Поток переходит в состояние "ОЖИДАЮЩИЙ с таймаутом", когда он вызывает метод с параметром таймаута, например, `sleep()`. Поток остается в этом состоянии до истечения таймаута или до получения уведомления.
* **ЗАВЕРШЕННОЕ** (Terminated): Поток переходит в состояние "ЗАВЕРШЕННОЕ", когда он завершает свое выполнение по одной из следующих причин:
  + - Поток завершается нормально, когда весь код потока был выполнен программой.
  + - Поток завершается из-за неожиданного ошибочного события, например, ошибки сегментации или необработанного исключения.

В этом состоянии ресурсы, связанные с потоком, освобождаются, и поток больше не существует. Результаты выполнения потока могут быть доступны другим потокам или процессам.

**Создание потока**

Создание потока в Java может быть выполнено двумя основными способами: **расширением класса Thread** или **реализацией интерфейса Runnable**.

**Расширение класса Thread**

При расширении класса `Thread` вы создаете ***подкласс `Thread***`, ***переопределяя его метод `run()`***. Этот метод содержит код, который должен быть выполнен в новом потоке.

Преимущества:

* Можно использовать методы класса `Thread`, такие как `sleep()`, `join()`, `interrupt()`, и другие.
* Можно получить доступ к методам и полям класса `Thread` напрямую.

Недостатки:

* Можно наследовать только от одного класса, поэтому этот подход не подходит, если ваш класс уже наследуется от другого класса.

**Реализация интерфейса Runnable**

При реализации интерфейса Runnable вы ***создаете класс, который реализует этот интерфейс, предоставляя реализацию метода run().*** Затем вы ***передаете экземпляр*** этого класса ***в конструктор класса Thread***.

Преимущества:

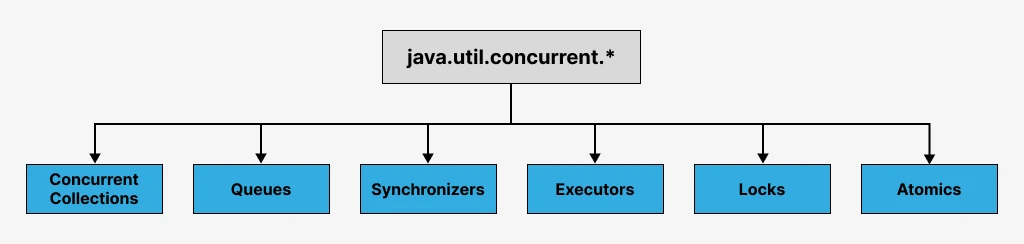
* Позволяет реализовать несколько интерфейсов, что делает этот подход более гибким.
* Позволяет использовать анонимные классы и лямбда-выражения для создания потоков.

Недостатки:

* Нельзя использовать методы класса `Thread` напрямую, если они не переопределены в классе `Runnable`.

Если вам **нужно использовать методы класса `Thread`** или если **ваш класс уже наследуется от другого класса**, **расширение класса `Thread`** может быть предпочтительным.

Если же вам нужна **большая гибкость** или вы хотите **реализовать несколько интерфейсов**, **реализация интерфейса `Runnable**` может быть лучшим выбором.



**Concurrent Collections** — это специализированные коллекции, предназначенные для использования в многопоточных приложениях. Они обеспечивают потокобезопасный доступ к данным, позволяя нескольким потокам одновременно безопасно взаимодействовать с коллекцией.

**ConcurrentHashMap:**

* Потокобезопасная реализация HashMap, которая использует блокировки на уровне сегментов для повышения производительности.
* Позволяет нескольким потокам одновременно читать и изменять данные в карте.
* Она использует тонкую блокировку для минимизации конфликтов и максимизации пропускной способности. Множество потоков может одновременно читать из карты, в то время как операции записи блокируют только части карты, изменяемые в данный момент

**ConcurrentLinkedQueue:**

* Потокобезопасная реализация LinkedList, поддерживающая ожидание при добавлении/удалении элементов.
* Подходит для использования в случаях, когда необходимо асинхронно обрабатывать задачи в многопоточной среде.

**ConcurrentSkipListSet** и **ConcurrentSkipListMap**:

* Очередь, поддерживающая блокирующие операции, что означает, что вставки и удаления ждут, пока не станет доступным место или элементы соответственно. Это особенно полезно в сценариях продюсера-потребителя, где продюсеры могут блокироваться, если очередь полна, а потребители могут блокироваться, если очередь пуста
* Позволяют нескольким потокам одновременно безопасно модифицировать отсортированные наборы и отображения.

**BlockingQueue**:

* Интерфейс, предоставляющий блокирующие методы для добавления, извлечения и просмотра элементов.
* Реализации, такие как ArrayBlockingQueue и LinkedBlockingQueue, используются для организации потокобезопасных очередей.

**CopyOnWriteArrayList** и **CopyOnWriteArraySet**:

* Потокобезопасные реализации ArrayList и HashSet, которые создают новую копию коллекции при модификации.
* Подходят для случаев, когда чтение из коллекции происходит чаще, чем запись.

**Синхронизированный метод** — это метод, объявленный с ключевым словом **synchronized**. Это означает, что только один поток может выполнять этот метод за раз для конкретного экземпляра класса. Все остальные вызовы этого метода будут ожидать, пока первый поток завершит выполнение метода.

**Синхронизированный блок** позволяет синхронизировать только часть метода, а не весь метод целиком. Это достигается путем использования конструкции synchronized внутри метода, указывая объект, на котором будет происходить синхронизация.

Пример синхронизированного блока в экземплярном методе:

public void add(int value){

**synchronized(this){**

**this.count += value;**

**}**

}

Ключевые моменты по использованию synchronized:

* Синхронизация на уровне методов является более простым и распространенным подходом, но менее гибким.
* Синхронизация на уровне блоков кода предоставляет больше гибкости, так как позволяет определять, какие именно части кода нуждаются в синхронизации.
* Объект-монитор, используемый для синхронизации, должен быть доступен всем потокам, которые должны иметь взаимоисключающий доступ к общему ресурсу.
* Излишнее использование synchronized может привести к снижению производительности из-за контекстных переключений и блокировок.
* synchronized гарантирует атомарность операций внутри синхронизированного блока/метода.

20. Multithreading и concurrency в Java: cредства и подходы обеспечения thread safety в Java

**Synchronized**:

* Ключевое слово synchronized позволяет синхронизировать доступ к методам или блокам кода.
* Обеспечивает взаимоисключающий доступ к критическим секциям.
* Может привести к блокированию и снижению производительности при высокой конкуренции между потоками.

**Volatile**:

* Ключевое слово volatile гарантирует, что изменения переменной будут видны всем потокам.
* Используется для предотвращения оптимизации компилятора, которая может привести к потере актуальности данных, переменная должна быть прочитана напрямую из основной памяти, а не из кэша потока.
* Обеспечивает thread safety только для простых операций чтения/записи, не подходит для сложных вычислений.

**Atomic Primitives**:

* Классы ***AtomicInteger***, ***AtomicLong***, ***AtomicReference*** и др. предоставляют атомарные операции для работы с примитивными типами и ссылками.

**Атомарная операция** (atomic operation) - это операция, которая выполняется неделимым образом, то есть либо полностью выполняется, либо не выполняется совсем

* Используют низкоуровневые синхронизирующие примитивы процессора для обеспечения thread safety.
* Более эффективны, чем synchronized, особенно при высокой конкуренции между потоками.

**Concurrent Collections**:

* Специализированные потокобезопасные реализации коллекций, такие как ConcurrentHashMap, ConcurrentLinkedQueue и др.
* Обеспечивают thread safety за счет внутренней синхронизации с минимальным блокированием.

**Locks**:

* Интерфейс Lock и его реализации (ReentrantLock, ReadWriteLock) предоставляют более гибкий контроль над синхронизацией, чем synchronized.
* Позволяют избежать проблем с неравномерным распределением нагрузки, которые могут возникать при использовании synchronized.

**Condition Variables**:

* Класс Condition используется совместно с Lock для организации ожидания и сигнализации между потоками.
* Позволяет реализовывать более сложные паттерны синхронизации, чем доступные с wait() и notify().

**ThreadLocal**:

* Позволяет создавать переменные, которые имеют отдельное значение для каждого потока.
* Предотвращает необходимость явной синхронизации при работе с общими переменными.

21. Асинхронные вычисления в web-приложениях средствами Java: потоки, futures

**Потоки**

* Создание и запуск потока: Для создания нового потока необходимо реализовать интерфейс Runnable или унаследовать класс Thread. Затем поток запускается вызовом метода start().
* Управление жизненным циклом: Методы join(), sleep(), interrupt() и др. позволяют управлять состоянием и взаимодействием между потоками.
* Синхронизация: Использование synchronized блоков и методов для управления доступом к общим ресурсам и предотвращения race conditions.

**Интерфейс Future**

**Future** - это интерфейс, который представляет результат асинхронной операции. Когда вы запускаете асинхронную задачу, она возвращает Future, который вы можете использовать для управления ее выполнением. Он предоставляет следующие основные методы:

* ***Блокирующее ожидание*** результата: Метод get() блокирует текущий поток, пока асинхронная операция не завершится, и возвращает ее результат. Есть также версия get(long timeout, TimeUnit unit), которая позволяет установить таймаут.
* ***Проверка состояния***: Методы isDone() и isCancelled() позволяют проверить, завершилась ли асинхронная операция и была ли она отменена соответственно.
* ***Отмена выполнения***: Метод cancel(boolean mayInterruptIfRunning) позволяет отменить выполнение асинхронной операции. Параметр mayInterruptIfRunning указывает, следует ли прерывать выполнение, если оно уже началось.

**CompletableFuture** - это реализация интерфейса Future, которая предоставляет дополнительные возможности для управления асинхронными вычислениями:

* ***Создание асинхронных операций***: Методы supplyAsync() и runAsync() позволяют запускать асинхронные вычисления, возвращающие значение или выполняющие побочные эффекты соответственно.
* ***Обработка результатов***: Методы thenApply(), thenAccept() и thenCompose() позволяют применять функции к результатам асинхронных операций, выполнять действия с ними и связывать асинхронные операции.
* ***Обработка ошибок***: Методы exceptionally() и handle() позволяют обрабатывать исключения, возникающие в асинхронных операциях.
* ***Комбинирование операций***: Методы allOf() и anyOf() позволяют ожидать завершения всех или любой из переданных асинхронных операций соответственно.

Использование CompletableFuture позволяет создавать более структурированный и чистый код для организации асинхронных вычислений, упрощая управление жизненным циклом задач, комбинирование операций и обработку ошибок по сравнению с низкоуровневым управлением потоками.

22. Функциональное программирование средствами Java 8: lambda-функции, method references, Optional

**Lambda-функции**

**Лямбда-выражения в Java** — это анонимные функции, которые могут быть переданы в качестве аргументов другим функциям, используемым в качестве выражений. Они позволяют писать более компактный и выразительный код, особенно при работе с Stream API.

Лямбда-выражения состоят из следующих частей:

* Список параметров: Определяет параметры, которые принимает лямбда-выражение.
* Стрелка `->`: Разделяет параметры от тела лямбда-выражения.
* Тело выражения: Содержит код, который будет выполнен при вызове лямбда-выражения.

Лямбда-выражения и операции Stream API позволяют писать более краткий и функциональный код, особенно при работе с коллекциями и обработке данных. Они обеспечивают удобный способ работы с последовательностями элементов, а лямбда-выражения позволяют передавать функциональность в виде аргументов, что делает код более гибким и модульным.

**Optional**

**Optional** - это контейнерный класс, который представляет значение, которое может быть или не быть. Это помогает избежать NullPointerException и делает код более читаемым.

***Основные особенности и применение Optional***:

* Представление отсутствующих значений: Optional используется для представления значения, которое может отсутствовать. Это позволяет избежать использования null-значений, которые часто приводят к ошибкам NullPointerException.
* Функциональные методы обработки: Optional предоставляет ряд методов для функциональной обработки данных, таких как map(), flatMap(), filter(), orElse(), orElseGet() и др.
* Безопасная навигация: Optional позволяет выполнять безопасную навигацию по объектам, избегая NullPointerException.
* Потоки и коллекции: Optional можно использовать в потоках и коллекциях, позволяя применять функциональные методы обработки к данным, которые могут отсутствовать.
* Обработка исключений: Optional можно использовать для моделирования ситуаций, когда метод может вернуть значение или выбросить исключение.

Использование Optional способствует написанию более читаемого, безопасного и функционального кода в Java. Он помогает избежать проблем с null-значениями и позволяет применять функциональные методы обработки данных.

**method references**

Метод-ссылка позволяет передавать ссылку на существующий метод вместо определения анонимной функции с помощью lambda-выражения.

***Существует несколько видов метод-ссылок***:

* Ссылка на статический метод:

Function<String, Integer> parse = **Integer::parseInt**;

* Ссылка на метод экземпляра:

Comparator<String> comp = **String::compareTo**;

* Ссылка на конструктор:

Supplier<ArrayList<String>> supplier = **ArrayList::new**;

* Ссылка на метод экземпляра конкретного объекта:

String name = "John Doe";

Predicate<String> startsWith = **name::startsWith**;

***Преимущества использования метод-ссылок***:

* Краткость и читаемость кода: Метод-ссылки позволяют сократить объем кода по сравнению с использованием lambda-выражений.
* Повторное использование кода: Вместо определения анонимной функции, можно использовать уже существующий метод.
* Улучшение производительности: Метод-ссылки могут быть оптимизированы компилятором лучше, чем lambda-выражения.

Метод-ссылки широко используются в сочетании с функциональными интерфейсами и потоками для создания более лаконичного и выразительного кода.

23. Функциональное программирование средствами Java 8: Stream API, intermediate и terminal функции

Stream API в Java предоставляет набор операций для работы с последовательностями элементов (стримами). Stream API позволяет выполнять различные операции над стримами, такие как преобразования, фильтрация, сортировка, агрегация и другие.

Все методы Stream API можно разделить на две группы: промежуточные (intermediate) и терминальные (terminal). **Промежуточные** операции во***звращают трансформированный поток***. А **терминальные** операции ***возвращают конкретный результат***.

**Промежуточные операции**: Эти операции возвращают Stream и могут быть объединены в цепочку. Они не выполняют никакой работы до тех пор, пока не будет вызвана терминальная операция. Примеры:

* **filter**: Фильтрует элементы потока, оставляя только те, которые удовлетворяют условию.
* **map**: Преобразует элементы потока, применяя функцию к каждому элементу (новый поток после действий).
* **distinct**: Удаляет дублирующиеся элементы из стрима.
* **sorted**: Сортирует элементы стрима в заданном порядке (по умолчанию – по возрастанию).
* **limit**: Ограничивает количество элементов в стриме до заданного числа.
* **skip**: Пропускает указанное количество элементов в стриме.
* **peek**: Применяет заданное действие к каждому элементу стрима без изменения самого стрима.

**Терминальные операции**: Эти операции возвращают результат или побочный эффект и завершают поток. После выполнения терминальной операции поток не может быть использован. Примеры:

* **forEach**: Применяет действие к каждому элементу потока.
* **toArray**: Собирает элементы потока в массив.
* **reduce**: Сокращает поток до одного значения, применяя накопительную операцию.
* **count**: Возвращает количество элементов в стриме.
* **anyMatch**: Проверяет, удовлетворяет ли хотя бы один элемент стрима заданному условию.
* **allMatch**: Проверяет, удовлетворяют ли все элементы стрима заданному условию.
* **noneMatch**: Проверяет, не удовлетворяет ли ни один элемент стрима заданному условию.
* **findFirst**: Возвращает первый элемент стрима.
* **findAny**: Возвращает любой элемент стрима.
* **min**: Возвращает минимальный элемент стрима в заданном порядке.
* **max**: Возвращает максимальный элемент стрима в заданном порядке.

24. Функциональное программирование средствами Java 8: функциональные интерфейсы

**Функциональный интерфейс** — это интерфейс, который содержит ровно один абстрактный метод, то есть описание метода без тела. Статические методы и методы по умолчанию при этом не в счёт, их в функциональном интерфейсе может быть сколько угодно.

Когда параметром метода является функциональный интерфейс, при вызове этого метода одним из аргументов должен быть блок кода.

Передаваемый блок кода должен удовлетворять следующему условию: его сигнатура должна совпадать с сигнатурой единственного абстрактного метода функционального интерфейса.

@FunctionalInterface

public interface ToIntBiFunction<T, U> {

/\*\*\* Applies this function to the given arguments.

\* @param t the first function argument

\* @param u the second function argument

\* @return the function result \*/

int applyAsInt(T t, U u);}

**Важно**. В Java есть несколько готовых функциональных интерфейсов с разным числом и типами входных-выходных параметров. (Как раз из таких ToIntBiFunction выше.) А если мы создаём новый функциональный интерфейс, то важно не забыть аннотацию **@FunctionalInterface**. Увидев её, компилятор проверит, что интерфейс и правда является функциональным.

**Базовые функциональные интерфейсы Java 8**

* Predicate

Predicate — функциональный интерфейс для проверки соблюдения некоторого условия. Если условие соблюдается, возвращает *true*, иначе — *false*.

* Consumer

Consumer (с англ. — “потребитель”) — функциональный интерфейс, который принимает в качестве входного аргумента объект типа T, совершает некоторые действия, но при этом ничего не возвращает.

* Supplier

Supplier (с англ. — поставщик) — функциональный интерфейс, который не принимает никаких аргументов, но возвращает некоторый объект типа T.

* Function

Function — этот функциональный интерфейс принимает аргумент T и приводит его к объекту типа R, который и возвращается как результат.

* UnaryOperator

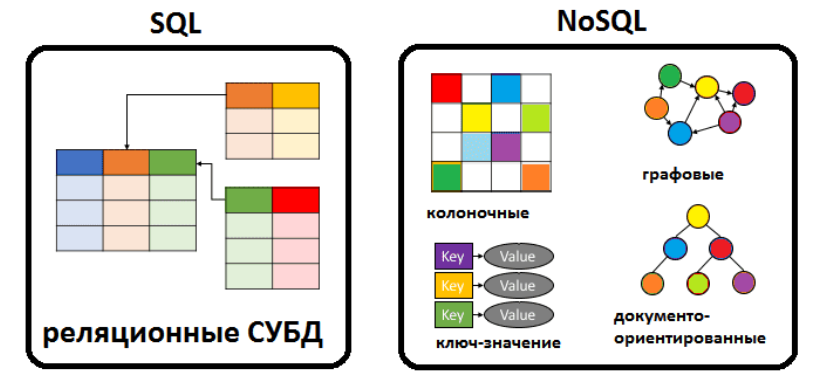
UnaryOperator — функциональный интерфейс, принимает в качестве параметра объект типа T, выполняет над ним некоторые операции и возвращает результат операций в виде объекта того же типа T.

Методы Stream работают именно с данными функциональными интерфейсами.

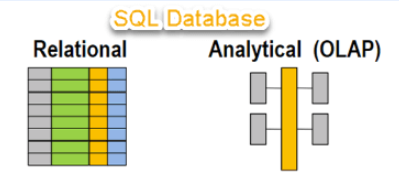
Типичный пример **метода в Stream c аргументом Function** — метод **map**, который принимает элементы одного типа, что-то с ними делает и передает дальше, но это уже могут быть элементы другого типа.

25. Типы хранилищ данных: реляционные базы данных, NoSQL

**База данных** - это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе. Управляются с помощью СУБД (MySQL, PostgreSQL…)



**SQL** (Structured Query Language) — это язык структурированных запросов, используемый для управления и манипулирования реляционными базами данных. SQL-базы данных применяются там, где необходимо хранить и управлять данными структурированной природы.



* Используют реляционную модель данных, где информация хранится в таблицах, связанных между собой. В таблице имеется набор столбцов, каждый из которых соответствует определённому типу данных. Например, целочисленные данные, строки, даты, время, нулевые значения, булевы значения и т. д. Это означает, что набор данных может быть разбит на отдельные поля для каждого вида информации.
* В SQL-базах данных используется язык запросов SQL, который позволяет пользователям создавать таблицы, добавлять, изменять и удалять данные, а также запрашивать информацию из базы данных.
* **Primary key** — особенное поле в SQL-таблице, которое позволяет однозначно идентифицировать каждую запись в ней.
* **Foreign key** - столбец (или несколько столбцов), ссылающийся на первичный ключ другой таблицы.
* **Виды связей**
  + Один к одному
  + Один ко многим
  + Многие ко многим

У первых двух могут быть обязательная связь (дочерняя таблица существует только с родительской) и необязательная.

**NoSQL** (Not Only SQL) — это широкий термин, который относится к нереляционным моделям баз данных, которые используют различные структуры для хранения данных: документы, ключ-значение, столбцовые и графовые БД. NoSQL-базы данных применяются, когда необходимо хранить данные неструктурированной природы, например, большие объёмы текстовых данных, изображения и видео.

* Данные в NoSQL-базах хранятся в документах, коллекциях или графах.

**Документ** — это структурированный контейнер для хранения данных в формате пар ключ-значение, где пары могут иметь разные типы данных.

**Коллекция** — это группа документов, связанных между собой.

**Граф** — это набор вершин и связей между ними.

* NoSQL-базы данных используют специальные языки запросов, которые позволяют пользователям запрашивать и манипулировать данными, такие как: MongoDB Query Language для MongoDB, Cassandra Query Language для Apache Cassandra и т. д.

26. Основные принципы распределённых хранилищ данных: ACID/BASE, CAP теорема

Основные принципы распределённых хранилищ данных включают в себя концепции ACID/BASE и CAP теоремы, которые помогают понять различные аспекты проектирования и выбора систем хранения данных.

**ACID** (Atomicity, Consistency, Isolation, Durability это классический подход, используемый реляционными СУБД, обеспечивающий надёжность и целостность данных.

* Атомарность (Atomicity) - операции либо выполняются полностью, либо не выполняются вовсе.
* Согласованность (Consistency) - данные всегда находятся в валидном состоянии.
* Изолированность (Isolation) - параллельные транзакции не влияют друг на друга.
* Долговечность (Durability) - изменения сохраняются даже при сбоях.

**BASE** (Basically Available, Soft state, Eventually consistent): это альтернативный подход, используемый в NoSQL-хранилищах, ориентированный на доступность и масштабируемость в ущерб строгой согласованности.

* **Базовая доступность** (Basically Available) - система доступна в основном, но может временно не отвечать.
* **Мягкое состояние** (Soft state) - состояние системы может меняться со временем, даже без внешнего вмешательства.
* **Конечная согласованность** (Eventual consistency) - система в конечном итоге придёт в согласованное состояние.

**CAP Теорема**

Согласно CAP теореме, распределённая система может иметь не более двух из трёх свойств:

* Согласованность (Consistency) - все узлы видят одни и те же данные в одно и то же время.
* Доступность (Availability) - система отвечает на все запросы.
* Устойчивость к разбиению (Partition tolerance) - система продолжает работать при сбоях сети.

Это означает, что выбор между этими свойствами зависит от конкретных требований к системе:

**CA** (Consistent, Available): Все узлы в системе видят одно и то же состояние данных, обеспечивая высокую согласованность. Однако такая система может столкнуться с проблемами доступности в случае разделения сети.

**CP** (Consistent, Partition-tolerant): Система обеспечивает согласованность данных, но может временно терять доступность в случае разделения сети. Это типичный пример для систем, использующих ACID свойства, например, баз данных на основе кластеров.

**AP** (Available, Partition-tolerant): Система стремится к высокой доступности, даже если это приводит к тому, что данные могут быть несогласованными. Большинство NoSQL баз данных принципиально не гарантируют строгой согласованности данных, предпочитая доступность и масштабируемость .

Реляционные СУБД (ACID) обычно выбирают CA, жертвуя P.

NoSQL хранилища (BASE) обычно выбирают AP, жертвуя C.

27. Основные средства и фреймворки для работы с БД в Java: JDBC, JPA/Hibernate, MyBatis

В Java существуют несколько основных средств и фреймворков для работы с базами данных:

**JDBC (Java Database Connectivity):**

* JDBC является стандартным API для подключения к базам данных из Java-приложений. Он позволяет выполнять SQL-запросы напрямую к базе данных
* Предоставляет универсальный интерфейс для выполнения SQL-запросов и обработки результатов.

Преимущества: Поддерживается всеми базами данных, не требует дополнительных фреймворков. Хорошо подходит для простых запросов и когда необходимо полный контроль над SQL.

Недостатки: Код может быстро запутаться при сложных запросах и обработке результатов. Требуется написание большого количества кода для повторяющихся задач.

**JPA (Java Persistence API) / Hibernate**:

* JPA определяет стандарт для доступа к реляционным базам данных
* Hibernate - наиболее популярная реализация спецификации JPA.
* Предоставляет объектно-реляционное отображение (ORM), позволяющее работать с БД на уровне Java-объектов.
* Упрощает написание кода для подключения, выполнения запросов и обработки результатов.

Преимущества: Предоставляет абстракцию над SQL, упрощает работу с базами данных, особенно при работе с сложными отношениями между сущностями. Поддерживает кэширование, транзакции и другие важные функции.

Недостатки: Может быть избыточным для простых запросов. Сложности могут возникнуть при работе с сложными запросами и связями между таблицами

**MyBatis**:

* MyBatis - фреймворк, облегчающий работу с реляционными БД, альтернатива JPA/Hibernate.
* Предоставляет более гибкий подход к работе с данными, чем JPA, за счёт использования SQL-запросов.
* Позволяет писать SQL-запросы непосредственно в Java-коде или в отдельных XML-файлах.
* Упрощает связывание результатов SQL-запросов с Java-объектами.

Преимущества: Поддерживает динамическое создание SQL-запросов, что делает его удобным для сложных запросов и условий. Не требует сложной настройки и хорошо подходит для проектов, где требуется гибкость в SQL.

Недостатки: Не предоставляет такой же уровень абстракции и автоматизации, как Hibernate. Требуется больше ручной работы для написания и поддержки SQL-запросов

28. Паттерны и принципы проектирования: SOLID, evergreen принципы

**SOLID** — это аббревиатура пяти основных принципов проектирования в объектно‑ориентированном программировании:\*\*

* **Single responsibility** — принцип единственной ответственности

*Каждый класс должен отвечать только за одну операцию*.

Если класс отвечает за несколько операций сразу, вероятность возникновения багов возрастает – внося изменения, касающиеся одной из операций вы, сами того не подозревая, можете затронуть и другие.

Назначение. Принцип служит для разделения типов поведения, благодаря которому ошибки, вызванные модификациями в одном поведении, не распространялись на прочие, не связанные с ним типы.

* **Open-closed** — принцип открытости / закрытости

*Классы должны  быть  открыты для расширения, но закрыты для модификации.*

Когда вы меняете текущее поведение класса, эти изменения сказываются на всех системах, работающих с данным классом. Если хотите, чтобы класс выполнял больше операций, то идеальный вариант – не заменять старые на новые, а добавлять новые к уже существующим.

Назначение. Принцип служит для того, чтобы делать поведение класса более разнообразным, не вмешиваясь в текущие операции, которые он выполняет. Благодаря этому вы избегаете ошибок в тех фрагментах кода, где задействован этот класс.

* **Liskov substitution** — принцип подстановки Барбары Лисков

*Наследующий класс должен дополнять, а не замещать поведение базового класса.*

В случаях, когда класс-потомок не способен выполнять те же действия, что и класс-родитель, возникает риск появления ошибок.

Если у вас имеется класс и вы создаете на его базе другой класс, исходный класс становится родителем, а новый – его потомком. Класс-потомок должен производить такие же операции, как и класс-родитель. Это называется наследственностью.

Необходимо, чтобы класс-потомок был способен обрабатывать те же запросы, что и родитель, и выдавать тот же результат. Или же результат может отличаться, но при этом относиться к тому же типу. На картинке это показано так: класс-родитель подаёт кофе (в любых видах), значит, для класса-потомка приемлемо подавать капучино (разновидность кофе), но неприемлемо подавать воду.

Если класс-потомок не удовлетворяет этим требованиям, значит, он слишком сильно отличается от родителя и нарушает принцип.

Назначение. Принцип служит для того, чтобы обеспечить постоянство: класс-родитель и класс-потомок могут использоваться одинаковым образом без нарушения работы программы.

* **Interface segregation** — принцип разделения интерфейса

*Не следует ставить клиент в зависимость от методов, которые он не использует.*

Когда классу приходится производить действия, не несущие никакой реальной пользы, это выливается в пустую трату ресурса, а в случае, если класс выполнять эти действия не способен, ведёт к возникновению багов.

Класс должен производить только те операции, которые необходимы для осуществления его функций. Все другие действия следует либо удалить совсем, либо переместить, если есть вероятность, что они понадобятся другому классу в будущем.

Назначение. Принцип служит для того, чтобы раздробить единый набор действий на ряд наборов поменьше – таким образом, каждый класс делает то, что от него действительно требуется, и ничего больше.

* **Dependency inversion** — принцип инверсии зависимостей

*Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.*

*Зависимость основывается на Абстракциях. Нет зависимости на что-то конкретное*

Согласно данному принципу, класс не должен соединяться с инструментом, который применяет для выполнения операции. Вместо этого он должен быть соединён с интерфейсом, который поможет установить связь между инструментом и классом.

Кроме того, принцип гласит, что ни интерфейс, ни класс, не обязаны вникать в специфику работы инструмента. Напротив, это инструмент должен подходить под требования интерфейса.

Назначение. Этот принцип служит для того, чтобы устранить зависимость классов верхнего уровня от классов нижнего уровня за счёт введения интерфейсов.

**Аббревиатура SOLID** была предложена **Робертом Мартином** (также известным как Дядя Боб), автором нескольких книг, широко известных в сообществе разработчиков.

Следование принципам позволяет строить (что главное) **масштабируемые** и **сопровождаемые** программные продукты.

Код, который написан с соблюдением принципов SOLID проще понимать, поддерживать, расширять, изменять его функциональность.

**Evergreen Принципы:**

**CQS ( Command-Query Separation)-** Принцип разделения команд и запросов)

* Этот принцип гласит, что методы должны быть либо командами, выполняющими действие, либо запросами, возвращающими данные, но не тем и другим одновременно.

**DRY (Don't Repeat Yourself)** - не повторяй себя, избегай дублирования кода.

* Этот принцип поощряет избегать дублирования кода или логики.

**KISS** (Keep It Simple, Stupid) - держи код простым и понятным.

**YAGNI (You Ain't Gonna Need It)** - не реализуй то, что тебе в данный момент не нужно.

* Этот принцип предполагает, что вы должны реализовывать только те функциональные возможности, которые вам действительно нужны в данный момент, а не пытаться предвидеть будущие требования.

**Premature Optimization** - Преждевременная оптимизация

* Этот принцип предупреждает против оптимизации кода до того, как это действительно необходимо. Чрезмерное внимание к производительности на ранних этапах может привести к запутанному и трудному в обслуживании коду. Оптимизируйте только после профилирования, выявившего реальные узкие места.

**Cohesion** – связность

* Высокая связность означает, что элементы внутри модуля или класса сильно связаны друг с другом и совместно работают для достижения единственной, четко определенной ответственности.

**Coupling** – зацепление

* Низкое зацепление означает, что модули или классы имеют минимальные взаимозависимости.

**Code to Interfaces** - Программирование на интерфейсах

* Этот принцип предполагает программировать на абстрактных интерфейсах, а не на конкретных реализациях. Это делает код более адаптируемым к изменениям.

**Composition over Inheritance** - отдавай предпочтение композиции классов вместо наследования.

* Отдавайте предпочтение композиции объектов из более мелких объектов, а не созданию глубоких иерархий наследования. Композиция, как правило, более гибкая и проще для понимания.

**Separation of Concerns** - разделяй программные сущности по их обязанностям и ответственности.

**Law of Demeter** - сущность должна взаимодействовать только со "своими" объектами.

**SLAP** (Single Level of Abstraction Principle) - Принцип единственного уровня абстракции.

* Этот принцип заявляет, что внутри модуля или класса код должен быть написан на одном, последовательном уровне абстракции. Смешивание высокоуровневых и низкоуровневых концепций затрудняет понимание кода.

29. Паттерны и принципы проектирования: MVC паттерн в java web-приложениях

**MVC** (Model-View-Controller) – это паттерн проектирования, который разделяет приложение на три основных компонента: модель (Model), представление (View) и контроллер (Controller). Этот подход позволяет разделить логику работы с данными, пользовательский интерфейс и управление потоком данных между ними, что упрощает разработку и поддержку кода.

* **Модель** (Model) отвечает за работу с данными, их сохранение, обработку и извлечение. Модель не содержит никакой логики пользовательского интерфейса или обработки запросов. Она полностью независима от представления и контроллера.
* **Представление** (View) отвечает за отображение данных пользователю. View получает данные от Модели и отображает их в пользовательском интерфейсе. View не содержит бизнес-логики, а лишь отвечает за визуализацию. за отображение данных пользователю. Это может быть HTML-страница, которая показывает данные модели в удобочитаемом виде.
* **Контроллер** (Controller) служит связующим звеном между ***моделью и представлением***. Он принимает ввод пользователя, обрабатывает его и обновляет модель и представление соответственно.

Использование MVC в Java веб-приложениях обеспечивает несколько преимуществ:

* Модульность: код разбивается на логические части, что упрощает его поддержку и развитие.
* Переиспользование кода: компоненты могут быть использованы в различных частях приложения или даже в других проектах.
* Улучшенная производительность: благодаря разделению ответственностей, можно оптимизировать работу каждого компонента независимо.
* Повышение тестируемости, так как компоненты можно тестировать по отдельности.

Этот паттерн широко используется в разработке веб-приложений на Java, включая такие фреймворки, как Spring MVC, Struts или JSF, которые предоставляют дополнительные инструменты и абстракции для работы с MVC архитектурой.

30. Паттерны и принципы проектирования: классификация Java паттернов, GoF паттерны

Java паттерны, известные как **Gang of Four** (GoF) паттерны, классифицируются на три основные категории: ***порождающие*** (Creational), ***структурные*** (Structural) и ***поведенческие*** (Behavioral).

**Порождающие паттерны** (Creational)

Эти паттерны предоставляют механизмы создания объектов, абстрагируя процесс инициализации и сборки объектов.

* Singleton
* Factory Method
* Abstract Factory
* Builder
* Prototype

**Структурные паттерны** (Structural)

Эти паттерны описывают, как объекты и классы могут быть скомпонованы для формирования более крупных структур. Эти паттерны помогают обеспечить эффективное взаимодействие между различными частями системы.

* Adapter
* Composite
* Proxy
* Flyweight
* Facade
* Bridge
* Decorator

**Поведенческие паттерны** (Behavioral)

Эти паттерны описывают, как классы и объекты взаимодействуют и распределяют обязанности между объектами. Они описывают, как объекты взаимодействуют друг с другом, а также процессы управления и передачи информации.

* Template Method
* Mediator
* Chain of Responsibility
* Observer
* Strategy
* Command
* State
* Visitor
* Interpreter
* Iterator
* Memento

Эти паттерны проектирования представляют собой проверенные временем решения для распространенных проблем проектирования. Они помогают создавать более гибкие, модульные и расширяемые Java-приложения путем разделения ответственности, инкапсуляции изменений и повышения повторного использования кода.

Эти паттерны были впервые собраны и описаны в книге "Design Patterns: Elements of Reusable Object-Oriented Software", написанной Эрихом Гаммой, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом, которые стали известны как "Банда Четырех" (Gang of Four)

Название "Gang of Four" (Банда Четырех) происходит от количества авторов книги "Design Patterns: Elements of Reusable Object-Oriented Software", которая была опубликована в 1994 году. Эта книга стала основополагающим трудом по паттернам проектирования и была написана четырьмя авторами: Эрихом Гаммой, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом. Из-за того, что их было четверо, они получили прозвище "Gang of Four", что в переводе означает "Банда Четырех"

31. Паттерны и принципы проектирования: общие принципы построения архитектуры, основные архитектурные стили

Общие принципы построения архитектуры:

* **Разделение ответственности** (Separation of Concerns): Архитектура должна разделять различные части приложения по их назначению и ответственности, чтобы упростить разработку, тестирование и обслуживание.
* **Модульность** (Modularity): Архитектура должна быть построена из независимых, слабосвязанных модулей, которые могут быть легко заменены, расширены или протестированы по отдельности.
* **Абстракция** (Abstraction): Архитектура должна использовать абстрактные интерфейсы для связи между компонентами, скрывая детали реализации.
* **Инверсия управления** (Inversion of Control): Архитектура должна использовать принцип инверсии управления, где высокоуровневые модули не зависят от низкоуровневых деталей, а, наоборот, низкоуровневые модули зависят от высокоуровневых абстракций.
* **Слабая связность** (Loose Coupling): Компоненты архитектуры должны быть как можно менее связаны друг с другом, чтобы изменения в одном компоненте не затрагивали другие.
* **Высокая связность** (High Cohesion): Компоненты архитектуры должны иметь высокую внутреннюю связанность, выполняя четко определенные обязанности.

Архитектурный шаблон — это уже придуманный способ решения какой-то задачи по проектированию программного обеспечения.

Архитектурные шаблоны, в отличие от паттернов, задействуют на более высоком уровне абстракции — при планировании взаимодействия пользователя приложения с сервером, данными и другими компонентами проекта.

В огромном полотне проектирования программного обеспечения стили это широкие мазки кисти, а паттерны — это замысловатые детали, которые придают жизнь шедевру.

Основные архитектурные стили в программировании и разработке ПО включают:

**1. Многослойный архитектурный стиль.**

Многослойный архитектурный стиль является одним из наиболее распространённых архитектурных паттернов. Он часто используется для традиционных веб-приложений и корпоративных приложений:

* Принципы: Этот архитектурный стиль разделяет функциональность на отдельные слои. Типичный пример это 3-х уровневая архитектура: слой представления, бизнес-логики и слой хранения данных.
* Преимущества: Легко понимать, тестировать и поддерживать; каждый слой может разрабатываться и обновляться независимо.
* Недостатки: Это может привести к накладным расходам на производительность; внесение изменений, затрагивающих несколько слоёв, может быть сложным.
* Применение: Веб-приложения, корпоративные приложения.
* Антипаттерны: Циклические зависимости, пропуск слоёв.

**2. Компонентно-ориентированный архитектурный стиль**

Этот стиль подчеркивает разделение функциональности на различные слои в рамках программной системы:

* Принципы: Этот стиль архитектуры организует систему в виде слабосвязанных, повторно используемых компонентов.
* Преимущества: Высокий уровень повторного использования, гибкости и поддерживаемости.
* Недостатки: Сложность управления компонентами и их взаимодействиями.
* Применение: Веб-приложения, десктоп-приложения, распределённые системы.
* Антипаттерны: Слишком большие компоненты, избыточные компоненты.

**3. Сервисно-ориентированный архитектурный стиль**

Этот стиль разрабатывает программное обеспечение в виде набора сервисов, взаимодействующих друг с другом. Каждый сервис является автономным и представляет определённую бизнес-активность с определённым результатом.

* Принципы: Архитектура SOA проектирует приложения в виде набора сервисов, которые обмениваются данными через сеть.
* Преимущества: Гибкость, масштабируемость, повторное использование и слабая связанность.
* Недостатки: Повышенная сложность, зависимость от сети и возможные проблемы с производительностью.
* Применение: Корпоративные системы, веб-сервисы, микросервисы.
* Антипаттерны: Игнорирование бизнес-потребностей, применение SOA там, где это не требуется.

**4. Стиль архитектуры распределённой системы**

Этот стиль относится к системе, в которой компоненты, размещённые на сетевых компьютерах, взаимодействуют и координируют свои действия, передавая сообщения. Компоненты взаимодействуют друг с другом для достижения общей цели.

* Принципы: Эта архитектура предполагает, что несколько систем работают вместе через сеть, чтобы представлять собой единую систему для конечного пользователя.
* Преимущества: Масштабируемость, устойчивость к сбоям и общий доступ к ресурсам.
* Недостатки: Увеличенная сложность, зависимость от сети и проблемы, связанные с согласованностью данных.
* Применение: Распределённые базы данных, облачные вычисления, телекоммуникационные сети.
* Антипаттерны: Не учитываются сбои в сети, игнорируются проблемы с согласованностью данных.

**5. Стиль предметно-ориентированной архитектуры**

Этот стиль сосредотачивается на основной предметной области и логике предметной области, основываясь на бизнес-модели предметной области. Он подчеркивает сотрудничество между техническими и предметными экспертами для итеративного совершенствования модели, которая точно и эффективно решает бизнес-проблемы.

* Принципы: Ориентирован на основную предметную область и логику предметной области и основывает дизайн на модели предметной области.
* Преимущества: Повышение понимания сложных бизнес-доменов и стимулирование общения между техническими и бизнес-командами.
* Недостатки: Может быть излишним для простых доменов и требует глубокого понимания.
* Применение: Сложные бизнес-системы, корпоративное ПО.
* Антипаттерны: Игнорирование единого языка, не включение предметных экспертов.

**6. Событийно-ориентированный архитектурный стиль**

Событийно-ориентированная архитектура (Event-driven architecture), представляет собой архитектуру и модель программного обеспечения для проектирования приложений. В событийной системе захват, коммуникация, обработка и сохранение событий являются основной структурой решения.

* Принципы: Этот стиль архитектуры основан на событиях, таких как действия пользователей, сигналы датчиков или сообщения от других программ.
* Преимущества: Высокая масштабируемость, слабая связанность, стимулирование потока информации в реальном времени или практически в реальном времени.
* Недостатки: Увеличенная сложность из-за асинхронного программирования, может быть сложно поддерживать и отлаживать.
* Применение: Приложения с графическим интерфейсом, системы реального времени для анализа данных, сложные системы обработки событий.
* Антипаттерны: Игнорирование порядка событий, отсутствие устойчивости событий.

**7. Архитектурный стиль разделения ответственности**

Принцип разделения ответственности (Separation of concerns) — это принцип проектирования, позволяющий разделить компьютерную программу на отдельные разделы, каждый из которых обрабатывает отдельную задачу.

* Принципы: Различные области функциональности обрабатываются отдельными, независимыми разделами системы.
* Преимущества: Улучшает понимаемость кода, уменьшает сложность, способствует модульности и параллельной разработке.
* Недостатки: Это может увеличить сложность из-за управления интерфейсами и потребовать больше коммуникации между модулями.
* Применение: Практически все типы программных систем.
* Антипаттерны: Смешивание функциональности, отсутствие четких границ модулей.

**8. Архитектурный стиль «Переводчик»**

Паттерн интерпретатора (Interpreter Pattern) — это паттерн проектирования, который определяет, как оценивать предложения в языке. Основная идея заключается в создании класса для каждого символа (терминального или нетерминального) в специализированном компьютерном языке.

* Принципы: Инструкции программы выполняются непосредственно, без предварительной конвертации в машинный код.
* Преимущества: Проще в отладке и тестировании, большая гибкость.
* Недостатки: Медленнее, чем скомпилированные языки, требует больше ресурсов.
* Применение: Скриптовые языки, некоторые языки программирования высокого уровня.
* Антипаттерны: Использование интерпретаторов в случаях, где производительность критически важна.

**9. Архитектурный стиль параллелизма**

Параллелизм (Concurrency) — это свойство системы, в которой несколько независимых задач выполняются одновременно.

* Принципы: Различные части программы выполняются независимо, потенциально одновременно.
* Преимущества: Может значительно улучшить производительность, особенно на многопроцессорных системах.
* Недостатки: Проектирование и устранение проблем с состоянием гонки (race conditions) и взаимной блокировкой (deadlocks) может быть сложным.
* Применение: Системы реального времени, вычислительные системы с высокой производительностью, веб-серверы.
* Антипаттерны: Пренебрежение возможными проблемами конкурентности и неправильная синхронизация общих ресурсов.

**10. Архитектура ориентированная на данные**

Этот стиль сфокусирован на том, как организуются и преобразовываются данные. Он часто используется в системах, которые обрабатывают большие объемы данных, выполняют сложные вычисления или требуют высокой масштабируемости.

* Принципы: База данных находится в центре архитектуры и все взаимодействия происходят через базу данных.
* Преимущества: Может обеспечить последовательность, целостность и надёжность данных.
* Недостатки: Может создавать узкие места в работе с данными и потенциальные проблемы с масштабируемостью.
* Применение: Множество корпоративных приложений, систем управления взаимоотношениями с клиентами (CRM) и систем управления предприятием (ERP).
* Антипаттерны: Пренебрежение возможностью узких мест в работе с данными, не учитывающая масштабируемости данных.
* **Монолитная архитектура** - традиционный подход, при котором все компоненты приложения тесно связаны и работают в рамках одного процесса. Это обеспечивает удобство разработки, тестирования и развертывания. Однако с увеличением сложности приложения, монолит может стать трудно управляемым, трудозатратным в обслуживании и модификации.
* **Микросервисная архитектура** - приложение разбивается на набор небольших, независимых сервисов, каждый из которых выполняет определенную функцию. Такой подход предоставляет гибкость при разработке, улучшает масштабируемость и устойчивость системы. Однако он также вводит сложность управления, согласования и мониторинга различных сервисов.

Выбор между монолитной и микросервисной архитектурой в значительной степени зависит от конкретных требований проекта, его сложности, требований к масштабированию и ресурсов команды. Важно понимать, что ни один из подходов не является «серебряной пулей». Определение наилучшего подхода требует анализа требований, ресурсов и будущей стратегии разработки.

**Клиент/сервер**

Стиль, при котором вся система программы разделяется на два приложения, где клиент может выполнить запрос к серверу.

**Компонентная архитектура**

Весь дизайн полностью раскладывается на логические компоненты и функциональные. Их можно использовать повторно, а интерфейсы связи тщательно прорабатываются.

**Проблемно-ориентированный дизайн**

Это стиль, позволяющий проработать бизнес-процессы, и ориентирован на создание моделей деловой активности приложения.

**Многослойная архитектура**

Каждая функциональная область разделяется на слои, что позволяет в отдельности работать с каждым из них.

**Шина сообщений**

Стиль, позволяющий отправлять сообщения по нескольким каналам, что дает возможность взаимодействия модулей, при этом конкретные сведения модулям не предоставляются.

**3-уровневая архитектура**

Функциональность разделяется на определенные сегменты, стиль схож по своей структуре с многослойной архитектурой. Разница только в том, что сегменты физически находятся на разных компьютерах.

**Объектно-ориентированный стиль**

Каждый объект является самостоятельным и может многократно использоваться, он содержит набор данных и поведения, а ответственность системы распределяется между этими объектами.

**Сервисно-ориентированный стиль**

В нем используются отдельные сервисы, обеспечивающие функциональность программы. Они обмениваются между собой сообщениями и образуют единую среду, при этом являются независимыми друг от друга.

**Современные тенденции в архитектуре ПО**

Сейчас существует множество тенденций в области архитектуры ПО и постоянно появляются новые, отвечающие современным вызовам и потребностям рынка. Мы рассмотрим те из них, которые считаем самыми главными.

**Микросервисная архитектура**

Это подход к разработке программного обеспечения, при котором приложение состоит из набора небольших, слабо связанных сервисов. Каждый такой сервис отвечает за отдельную бизнес-функцию и может разрабатываться и развиваться независимо. Микросервисная архитектура позволяет обеспечить высокую модульность, гибкость, масштабируемость, отказоустойчивость, быстроту разработки и доставки программного обеспечения. Микросервисная архитектура также способствует распределению работы между разными командами, которые могут использовать разные технологии и методологии.

**Serverless архитектура**

Это модель облачных вычислений, при которой разработчики не нуждаются в управлении серверами, а должны только писать и запускать код, выполняемый на платформе провайдера облачных услуг. Такая платформа динамически управляет выделением ресурсов, масштабированием, биллингом и другими аспектами выполнения кода. Serverless Architecture позволяет сосредоточиться на бизнес-логике, снизить затраты на инфраструктуру, повысить производительность и гибкость программного обеспечения. Serverless архитектура также называется «Функция как услуга» (Function as a Service, FaaS), поскольку единицей кода является функция, вызываемая при определенном событии или запросе.

**Контейнеризация**

Это технология, позволяющая запускать приложения изолированно от основной операционной системы. Приложение упаковывается в специальную оболочку-контейнер, внутри которой имеется среда, необходимая для работы. Контейнеры могут быть легко переносимыми, масштабируемыми и контролируемыми. Контейнеризация обеспечивает высокую эффективность, быстроту, консистентность и безопасность программного обеспечения. Она также способствует стандартизации, интеграции, распределению и автоматизации процессов разработки и эксплуатации программного обеспечения.

Рассматривается 21 архитектурный стиль.

* **Data-flow styles**
  + PF (Pipe and Filter)
  + UPF (Uniform Pipe and Filter)
* **Replication styles**
  + RR (Replicated Repository)
  + $ (Cache)
* **Hierarchical styles**
  + CS (Client-Server)
  + LS (Layered System)
  + LCS (Layered-Client-Server)
  + CSS (Client-Stateless-Server)
  + C$SS (Client-Cache-Stateless-Server)
  + LC$SS (Layered-Client-Cache-Stateless-Server)
  + RS (Remote Session)
  + RDA (Remote Data Access)
* **Mobile Code Styles**
  + VM (Virtual Machine)
  + REV (Remote Evaluation)
  + COD (Code on Demand)
  + LCODC$SS (Layered-Code-on-Demand-Client-Cache-Stateless-Server)
  + MA (Mobile Agent)
* **Peer-to-Peer Styles**
  + EBI (Event-based Integration)
  + C2 (Component- and message-based)
  + DO (Distributed Objects)
  + BDO (Brokered Distributed Objects)

Архитектурные стили используются, чтобы получить следующие преимущества.

* **Переиспользование архитектуры**. Создание архитектуры может быть сложной задачей, особенно когда у вас есть только требования и «чистый лист». Выбор стиля сужает пространство решений и направляет архитектурную мысль. При этом для новых задач можно применять хорошо известные и изученные решения, обладающие известными достоинствами и недостатками применительно к вашей ситуации.
* **Переиспользование кода**. Часто у архитектурных стилей бывают неизменяемые части, которые можно один раз реализовать, а затем переиспользовать в каждой системе. Трёхзвенная архитектура, например, реализуется многими библиотеками для разработки веб-приложений, для событийно-ориентированных стилей есть хороший middleware (например, уже упоминавшийся ROS), дря распределённых стилей — технологии наподобие gRPC или WCF, и т.д. Это существенно сокращает затраты на разработку.
* **Упрощение общения и понимания системы**. Как и в случае с паттернами, достаточно просто назвать стиль по имени и не надо объяснять, как что устроено — опытныетразработчики вас сразу поймут.
* **Упрощение интеграции приложений**, на тактическом уровне за счёт переиспользования стандартов и middleware, типичных для стиля, на стратегическом — за счёт того, что понятно, куда и как встраиваться, не нарушив архитектурные ограничения каждого из приложений.
* **Применение специфичных** для стиля **методов анализа**. Поскольку стили накладывают ограничения на структуру систем, иногда эти ограничения достаточно жёсткие, чтобы про систему можно было что-то доказать или что-то посчитать. Хороший пример в этом плане — стиль Pipes and Filters, к которому применимы алгоритмы анализа графов, которые можно использовать для анализа пропускной способности системы, её надёжности и т.п.
* **Специфичные** для стиля **методы визуализации** — тот же Pipes and Filters удобно рисовать с помощью визуальных языков, ориентированных на данные, таких как Data Flow Diagram. Или становится возможным использование предметноориентированных языков — например, на том же Pipes and Filters построено программирование в LabVIEW и Matlab/Simulink. Pipes and Filters тут не исключение, например, Microsoft Robotics Developer Studio имела по сути визуальный DSL для создания программ в распределённом веб-сервисном стиле.

8 главных стилей:

* Многослойный.
* Компонентно-ориентированный.
* Сервисно-ориентированный.
* Распределенная система.
* Предметно-ориентированный.
* «Переводчик».
* Параллелизм.
* Ориентированный на данные.

**Многослойный** — самый распространенный. Типичный пример — многоуровневый стиль из 3 слоев: слой представления, бизнес-логика и хранение данных. Плюсы — легкий в понимании, тестировании и поддержке. Минус — появление накладных расходов на производительность.

**Компонентно-ориентированный** разделяет функциональность на слои в рамках одной системы, организованной как повторно используемые и слабосвязанные компоненты. Преимущества — гибкость, поддерживаемость, повторное использование. Недостаток — компонентами сложно управлять.

**Сервисно-ориентированный** для разработки ПО как набора взаимодействующих сервисов через сеть. Плюсы — слабая связанность, гибкость и возможность повторного использования. Минусы — сложность, сетевая зависимость.

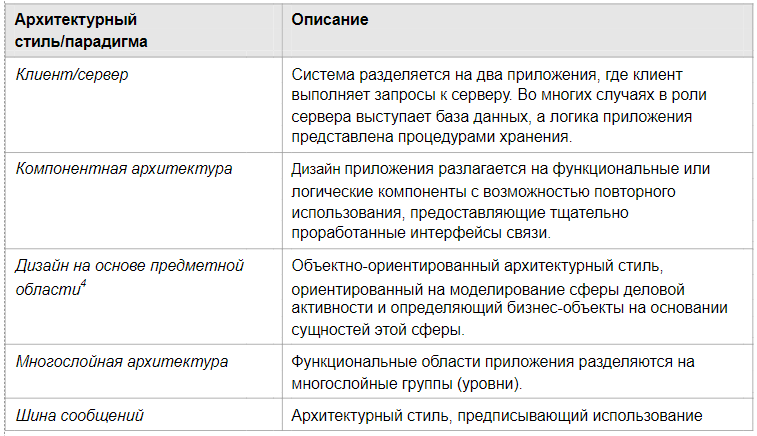
**Распределенная система**. Компоненты на сетевых компьютерах передают сообщения для взаимодействия и координирования действий, чтобы достичь цели. Преимущества — общий доступ, устойчивость к ошибкам и сбоям, возможность масштабирования. Недостатки — сложность, сетевая зависимость.

**Предметно-ориентированный** — создание программных абстракций (моделей предметных областей). Отличается ограниченными связями, целостностью и взаимосвязью. Минус — требует глубокого понимания и опыта.

«**Переводчик**» определяет способы оценки предложений. Для каждого символа создается терминальный или нетерминальный стиль. Отличается большой гибкостью и простой отладкой. Недостаток — требует больше ресурсов.

В стиле **параллелизма** несколько независимых задач выполняются в одно и то же время. Параллелизм улучшает производительность, что особенно актуально для многопроцессорных систем. Минус — сложная взаимная блокировка.

**Архитектура, ориентированная на данные**. Стиль фокусируется на организации и преобразовании данных. Подходит для больших объемов данных, решения сложных задач. Обеспечивает правильную последовательность данных, целостность и надежность. Среди недостатков — возможные проблемы с масштабируемостью.

32. Основы непрерывной разработки кода: Agile-методологии, CI/CD

**Agile-методологии** - это семейство гибких подходов к разработке программного обеспечения, которые фокусируются на итеративной и инкрементальной доставке ценности.

На современном конкурентном рынке способность быстро адаптироваться к отзывам клиентов и меняющимся требованиям имеет решающее значение. Итеративный подход Agile позволяет командам разработчиков часто пересматривать свои приоритеты на основе отзывов в режиме реального времени, что приводит к повышению удовлетворенности клиентов и лучшему соответствию продукта рынку.

Некоторые ключевые принципы Agile:

* Удовлетворение потребностей заказчика путем ранней и непрерывной доставки ценного ПО
* Приветствие изменений требований, даже на поздних стадиях разработки
* Частая доставка рабочего ПО (от нескольких недель до нескольких месяцев)
* Постоянное внимание к техническому совершенству и хорошему дизайну
* Самоорганизующиеся и мотивированные команды
* Регулярная рефлексия над процессами и приспособление к меняющимся обстоятельствам

**Основные ценности и принципы гибкой разработки**:

* Люди и взаимодействие важнее процессов и инструментов
* Рабочее программное обеспечение важнее всеобъемлющей документации
* Сотрудничество с клиентами в процессе согласования контрактов
* Реагирование на изменения, а не следование плану

**Популярные Agile-фреймворки**: Scrum, Kanban, Extreme Programming (XP).

**Scrum**: Scrum делит процесс разработки на итерации фиксированной продолжительности, называемые спринтами, которые обычно длятся 2-4 недели. Кульминацией каждого спринта является потенциальный выпуск нового продукта. Scrum-команды состоят из таких ролей, как владелец продукта, Scrum-мастер и команда разработчиков, каждая из которых несет определенную ответственность за обеспечение бесперебойного выполнения и постоянного совершенствования.

**Канбан**: Канбан фокусируется на визуализации рабочего процесса и ограничении незавершенного производства (WIP) для повышения эффективности. В нем используется доска Канбан, на которой задачи отображаются в столбцах, представляющих различные этапы процесса разработки. Команды переводят задачи с одного этапа на другой, насколько позволяют возможности, обеспечивая непрерывную реализацию без повторений фиксированной продолжительности.

**CI/CD (Continuous Integration/** **Continuous Delivery)**

**CI/CD** - это практики автоматизации сборки, тестирования и развертывания программного обеспечения. Цель - сделать процесс доставки быстрым, надежным и частым.

**Continuous Integration** (непрерывная интеграция) - автоматизированная сборка и тестирование кода при каждом коммите.

Это практика разработки, при которой разработчики часто интегрируют изменения своего кода в общий репозиторий, часто по нескольку раз в день. Затем каждая интеграция автоматически проверяется путем создания приложения и запуска набора автоматических тестов для максимально быстрого обнаружения ошибок интеграции.

Основная цель CI – выявлять и устранять проблемы интеграции на ранней стадии процесса разработки, снижая сложность и риск, связанные с объединением крупных изменений в конце цикла разработки.

***Ключевые практики в CI***

* Частые фиксации: Такая практика гарантирует, что изменения будут постепенными и их будет легче интегрировать и тестировать.
* Автоматизированные сборки: Каждая фиксация запускает автоматизированный процесс сборки, компилируя код и обеспечивая его плавную интеграцию с существующей кодовой базой. Автоматизированные сборки помогают выявлять ошибки интеграции на ранней стадии.
* Немедленная обратная связь: Автоматизированные платформы тестирования запускают набор тестов после каждой сборки. Такая немедленная обратная связь позволяет разработчикам оперативно выявлять и устранять проблемы, поддерживая стабильность кодовой базы.

**Continuous Delivery** (непрерывная поставка) - автоматизированное развертывание протестированного кода в production.

Это расширение Continuous Integration (CI), которое автоматизирует процесс выпуска, обеспечивая быстрое, безопасное и устойчивое внедрение изменений в код в рабочей среде. CD стремится сделать развертывание предсказуемым и надежным, чтобы программное обеспечение можно было уверенно выпускать в любое время.

Непрерывная поставка включает в себя развертывание каждого изменения кода в промежуточной среде и обеспечение его готовности к выпуску посредством тщательного автоматизированного тестирования и валидации. Основная цель CD – дать возможность командам разработчиков быстро и безопасно предоставлять пользователям новые функции, улучшения и исправления ошибок. Такая практика значительно сокращает время между написанием кода и его передачей в рабочую среду.

* Непрерывная поставка гарантирует, что программное обеспечение может быть выпущено в любое время, но фактическое развертывание в рабочей среде – это решение, принимаемое вручную.
* Непрерывное развертывание является еще одним шагом вперед благодаря автоматическому внедрению каждого изменения, которое проходит все этапы конвейера непосредственно в рабочей среде без вмешательства человека.

***Ключевые практики в области CD***

* Автоматизированное тестирование: Комплексные автоматизированные тесты, включая модульные, интеграционные, системные и приемочные тесты, имеют решающее значение для проверки правильности функционирования программного кода.
* Конвейеры развертывания: Конвейер развертывания автоматизирует процесс переноса изменений в коде на различные этапы тестирования и развертывания, обеспечивая согласованность и надежность выпусков.
* Автоматизация выпуска: Автоматизированные сценарии и инструменты управляют процессом развертывания, снижая риск человеческой ошибки и делая процесс выпуска повторяемым и надежным.

Принципы CI/CD:

* Каждый коммит запускает полный цикл сборки, тестирования и развертывания
* Использование автоматизированных скриптов и инструментов (Jenkins, Travis CI, GitLab CI, Azure DevOps, etc.)
* Раннее и частое развертывание изменений в production

Таким образом, Agile-методологии и CI/CD вместе создают основу для непрерывной разработки и доставки программного обеспечения. Это позволяет быстро реагировать на изменения, сокращает время вывода на рынок и повышает качество продукта.

***Лучшие практики для успешного использования CI/CD в Agile***

* Поддерживайте чистоту кодовой базы: регулярно проводите рефакторинг кода и поддерживайте стандарты кодирования, чтобы обеспечить чистоту и управляемость кодовой базы.
* Регулярно обновляйте и оптимизируйте конвейеры: Постоянно просматривайте и улучшайте свои конвейеры CI/CD для повышения эффективности.
* Обеспечьте всестороннее тестирование: внедрите и поддерживайте надежный набор автоматизированных тестов для проверки каждого аспекта приложения.
* Развивайте культуру сотрудничества: поощряйте взаимодействие между группами разработки, тестирования и эксплуатации, чтобы обеспечить бесперебойную и эффективную работу CI/CD.
* Мониторинг и измерение: Используйте инструменты мониторинга и аналитики для отслеживания производительности вашего конвейера CI/CD и определения областей для улучшения.

33. Основные подходы и средства улучшения качества кода: code review, testing, static analyzers

**Code review** — это процесс проверки исходного кода другими разработчиками или специализированными инструментами для обнаружения ошибок, несоответствий стандартам кодирования и потенциальных уязвимостей безопасности. Этот процесс способствует повышению качества кода, обмену знаниями между членами команды и улучшению архитектуры приложения.

***Ключевые принципы эффективного code review:***

* Регулярность - code review должен быть рутинным процессом
* Конструктивность - фокус на улучшении кода, а не на критике личности
* Вовлеченность - все участники должны активно участвовать и давать обратную связь
* Автоматизация - использование инструментов (GitHub, GitLab, Gerrit) для облегчения процесса

**Testing** — это важный аспект обеспечения качества кода, который включает в себя различные методы, такие как модульное тестирование, интеграционное тестирование и системное тестирование. В Java одним из популярных фреймворков для модульного тестирования является JUnit, который позволяет создавать и выполнять тестовые случаи для отдельных классов или методов.

***Для эффективного тестирования важно:***

* Написание тестов одновременно с разработкой (test-driven development)
* Использование фреймворков для автоматизации тестирования (JUnit, TestNG, Selenium, Cypress и др.)
* Интеграция тестирования в CI/CD pipeline (конвейер)

**Статические анализаторы кода**

Статические анализаторы кода автоматически сканируют исходный код на наличие ошибок, уязвимостей и нарушений стандартов кодирования без его выполнения. Эти инструменты могут значительно упростить процесс обнаружения проблем на ранних стадиях разработки.

***Инструменты для статического анализа кода в Java***:

* **Checkstyle** и PMD: Эти инструменты помогают поддерживать стандарты кодирования и обнаруживают проблемы со структурой и стилем кода.
* **SpotBugs** (ранее FindBugs): Идентифицирует шаблоны ошибок в коде и поддерживает интеграцию с инструментами сборки.
* **SonarQube**: Обеспечивает комплексную проверку качества кода и безопасности, поддерживая множество языков и фреймворков.
* **Coverity**: Выявляет уязвимости и проблемы в коде, предлагая продвинутую детектирование дефектов.

Использование этих подходов и инструментов позволяет разработчикам Java создавать более чистый, надежный и безопасный код, сокращая время на исправление ошибок и улучшая общую производительность приложений

**Рефакторинг** — это переработка исходного кода программы, чтобы он стал более простым и понятным. Рефакторинг не меняет поведение программы, не исправляет ошибки и не добавляет новую функциональность. Он делает код более понятным и удобочитаемым.

34. Облачные решения: SaaS, PaaS, IaaS, AWS, GCP, Azure, serverless архитектура и cloud-native подход в построении приложений

**SaaS, PaaS, IaaS**

Облачные вычисления предоставляют различные модели обслуживания:

* **Software as a Service** (SaaS) - готовое программное обеспечение, доступное через интернет.

***Примеры***: Office 365, Salesforce, Dropbox.

* **Platform as a Service** (PaaS) – предлагает облачную платформу для разработки, тестирования и развертывания приложений, освобождая разработчиков от необходимости управлять инфраструктурой

***Примеры***: AWS Elastic Beanstalk, Heroku, Google App Engine.

* **Infrastructure as a Service** (IaaS) - аренда виртуальной инфраструктуры: серверов, хранилища, сетей. Позволяет клиентам контролировать операционные системы и приложения, но не физическое оборудование.

***Примеры***: AWS EC2, Microsoft Azure VMs, Google Compute Engine.

**Основные облачные платформы**

* **AWS** **(Amazon Web Services)** - ведущая облачная платформа, предлагающая широкий спектр сервисов IaaS, PaaS и SaaS., включая EC2 для виртуальных машин, S3 для хранения данных и Lambda для serverless вычислений.
* **Google Cloud Platform (GCP)** - растущая облачная платформа с сильными предложениями в области данных, аналитики и машинного обучения.
* **Microsoft Azure** - облачная платформа с сильными интеграциями с Microsoft-экосистемой, включая Office 365, Active Directory и другие продукты. Активно используется в корпоративном секторе.

**Serverless и cloud-native архитектуры**

**Serverless** - это архитектурный подход, при котором приложения полностью основаны на управляемых облачных сервисах, таких как AWS Lambda, Azure Functions, Google Cloud Functions. Позволяет разработчикам сосредоточиться на написании бизнес-логики, избавляясь от необходимости управления серверной инфраструктурой. Приложения и сервисы строятся с использованием сервисов, предоставляемых облачным провайдером, таких как базы данных, идентификационные сервисы, API-шлюзы и очереди событий. Однако этот подход может привести к зависимости от конкретного облачного провайдера.

**Cloud-native** - это подход к построению приложений, специально разработанных для работы в облаке. Он включает использование контейнеров (Docker), оркестрации (Kubernetes), микросервисов, CI/CD и других облачных практик.

Преимущества cloud-native подхода:

* Масштабируемость и отказоустойчивость
* Гибкость и быстрое внедрение изменений
* Снижение затрат на инфраструктуру
* Оптимизация использования ресурсов

В целом, облачные решения предоставляют большую гибкость, масштабируемость и эффективность по сравнению с традиционными on-premise системами. Выбор правильной модели обслуживания и архитектуры играет ключевую роль в успешной реализации облачных проектов.