Ответы на вопросы позиция QA Engineer.

Если у вас возникли вопросы по вопросам – пишите.

Если вы не нашли свой вопрос – пишите.

Если вы нашли ошибку -пишите.

Если у вас есть более подходящий вариант для ответа на какой-либо вопрос – пишите.

Куда писать?

<https://t.me/SommersetMoem>

README

Данный документ представляет собой набор вопросов и ответов, с которыми я сталкивался в процессе прохождения собеседований, также тут собраны полезные ссылки (статьи, видео и т.д.) Раздел ссылок находится после вопрос, просьба тем, кто использует кидайте свои полезности в данный раздел и соблюдайте форматирование, всех обнял.  
**Ars longa, vita brevis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Что такое тестирование? (В узком и широком смысле, на собесе можно выдавать оба по очереди) | | | | **В узком смысле тестирование** – это процесс сопоставления готового продукта с требованиями; **В широком смысле тестирование** – это деятельность, направленная на предоставление всем заинтересованным лицам исчерпывающих сведений о текущем качестве продукта и любых остаточных рисках, а также на сведение к минимуму дефектов, которые может обнаружить конечный пользователь, при заданных сроках и бюджете; |
| Что такое качество ПО? | | | | **Качество ПО** – это способность ПО отвечать запросу пользователя. |
| 3 цели тестирования? | | | | -Проверка программы на соответствия требованиям; -Дать оценку качества продукта; -Проинформировать команду о качестве; |
| Этапы жизненного цикла ПО? | | | | -Сбор **и анализ требований** (Requirement Gathering and Analysis): На этом этапе от клиента собирается вся необходимая информация для разработки продукта в соответствии с их ожиданиями. Любые неясности должны быть разрешены сразу на этом этапе. Бизнес-аналитик и менеджер проекта назначили встречу с заказчиком, чтобы собрать всю информацию, например, что заказчик хочет построить, кто будет конечным пользователем, какова цель продукта. Перед созданием продукта очень важно понимание или знание продукта. Например, клиент хочет иметь приложение, которое включает денежные транзакции. В этом случае требование должно быть четким, например, какие транзакции будут выполняться, как они будут проводиться, в какой валюте они будут проводиться и т. д. После того, как сбор требований завершен, проводится анализ для проверки возможности разработки продукта. После четкого понимания требования создается документ SRS (Спецификация требований к программному обеспечению). Этот документ должен быть полностью понят разработчикам, а также должен быть рассмотрен заказчиком для использования в будущем;  -**Дизайн** (Design): На этом этапе требования, собранные в документе SRS, используются в качестве входных данных, и создается архитектура программного обеспечения, которая используется для реализации разработки системы. Создаются два вида дизайн-документов:   * Высокоуровневый дизайн (HLD - High-Level Design):   + Краткое описание и название каждого модуля;   + Краткое описание функциональности каждого модуля;   + Отношения интерфейсов и зависимости между модулями;   + Таблицы базы данных, идентифицированные вместе с их ключевыми элементами;   + Полные архитектурные схемы с подробными сведениями о технологиях. * Низкоуровневый дизайн (LLD - Low-Level Design):   + Функциональная логика модулей;   + Таблицы базы данных, которые включают тип и размер;   + Полная детализация интерфейсов;   + Решение всех типов проблем с зависимостями;   + Список сообщений об ошибках;   + Полные входные и выходные значения для каждого модуля.   -**Разработка** (Implementation or Coding): Реализация / кодирование начинается, как только разработчик получает Design document. Дизайн программного обеспечения переведен в исходный код. На этом этапе реализуются все компоненты программного обеспечения;  -**Тестирование** (Testing): Тестирование начинается после завершения кодирования и выпуска модулей для тестирования. На этом этапе разработанное программное обеспечение тщательно тестируется, и все обнаруженные дефекты передаются разработчикам для их исправления. Повторное тестирование, регрессионное тестирование проводится до тех пор, пока программное обеспечение не будет соответствовать ожиданиям клиента. Тестировщики обращаются к документу SRS, чтобы убедиться, что программное обеспечение соответствует стандарту заказчика;  -**Развертывание** (Deployment): После тестирования продукта он развертывается в производственной среде или выполняется первое UAT (пользовательское приемочное тестирование), в зависимости от ожиданий клиента. В случае UAT создается копия производственной среды, и заказчик вместе с разработчиками выполняет тестирование. Если клиент остается доволен, то предоставляет согласие на [релиз](https://hackernoon.com/feel-the-release);  **-Поддержка** (Maintenance): Основное внимание на этом этапе SDLC уделяется обеспечению того, чтобы потребности продолжали удовлетворяться, и чтобы система продолжала работать в соответствии со спецификацией, упомянутой в первом этапе. После того, как система развернута и клиенты начинают использовать разработанную систему, следует 3 вида активностей:   * Исправление ошибок; * Обновление; * Улучшение.   - Утилизация: Со временем приложение теряет свою актуальность и перестает приносить прибыль, в таком случае происходит его утилизация и для клиентов оно становится недоступно. |
| На каком этапе ЖЦ ПО подключается QA? | | | | Тестировщик подключается на этапе **“Cбор и анализ требований”** т.к. чем раньше найдена ошибка, тем дешевле ее устранение (зачеркнуть на бумаге и изменить код приложения разные по стоимости человеко-ресурсов вещи). |
| Обязанности Тестировщика ПО? | | | | 1 вариант:  -Анализ требований; -Составление тестовой документации (артефактов); -Проведение тестирования на основании тестовой документации; -Предоставления отчета о проделанной работе;  2 вариант:  -верификация;  -валидация;  -еррор детекшен; |
| Семь Принципов тестирования ПО? | | | | 1. Исчерпывающее тестирование невозможно – это означает, что насколько бы тщательным тестирование не было, нельзя учесть все возможные сценарии и предвидеть все возможные ошибки; 2. Тестирование демонстрирует наличие дефектов, а не их отсутствие – это означает, что тестирование **может** **выявить**, что ошибки присутствуют,но**не может** **доказать**в полной мере,что дефектов нет; 3. Заблуждение об отсутствии ошибок – это означает, что мы можем сколько угодно находить ошибки, и даже, казалось бы, не обнаруживая их больше, но нет гарантии того, что ошибки найдены все и продукт полностью качественный и готовый; 4. Раннее тестирование сохраняет время и деньги – это означает, что исправить потенциальный баг на этапе требований гораздо дешевле, чем после того, как код уже написан; 5. Принцип скопления или кластеризация дефектов – это означает, что 20% нашего функционала содержат 80% ошибок. Это происходит из-за сложной логики в некоторых местах приложения; 6. Тестирование зависит от контекста – это означает, что к разным продуктам мы будем применять разные практики тестирования т.е. тестирование мобильного приложения отличается от тестирования веб приложения или тестирование интернет-магазина будет отличатся от тестирования кардиостимулятора; 7. Парадокс пестицида – это означает, что если к какому-либо функционалу применять постоянно повторяющийся набор тестов – то эти проверки в скором времени будут неэффективны в нахождении новых дефектов.   Поэтому тест-кейсы должны постоянно обновляться и видоизменяться. Важно пользоваться такими рекомендациями:   * добавлять новые тесты; * просматривать и изменять существующие; * применять разные виды и техники тестирования; * осуществлять тестирование новыми сотрудниками и др.    В целом посмотреть на продукт под другим углом.  Можно отметить здесь ещё тот факт, что в наибольшей степени парадокс пестицида может проявляться в регрессе и автотестах; |
| Что такое Верификация и Валидация? | | | | **Верификация** - это проверки, выполняемые в процессе разработки ПО для ответа на вопрос: **“правильно ли мы разрабатываем продукт?”**. Это в т.ч. включает проверку документации: requirements specification, design documents, database table design, и т.д. Верификация гарантирует, что ПО разрабатывается в соответствии со стандартами и процессами организации, полагаясь на статические методы тестирования (т.е. **без запуска ПО**, но, например, с unit/integration tests). **Верификация является превентивным** **подходом**.  **Валидация** - это процесс оценки конечного продукта, чтобы проверить, соответствует ли он потребностям бизнеса и ожиданиям клиентов, т.е. отвечает на вопрос: **“правильный ли мы разработали продукт?”**. Валидация является **динамическим тестированием, т.е. происходит с помощью выполнения кода** и прогона тестов на нём (UAT/CAT, usability, всё что угодно). |
| Уровни тестирования или “Пирамида тестирования”? | | | | **«Пирамида тестов»** - метафора, которая означает группировку динамических тестов программного обеспечения по разным уровням. Она также дает представление, **какое количество тестов должно быть в каждой из этих групп**. **Основной принцип разделения уровней** - тест должен быть на том же уровне, что и тестируемый объект. В тесте более высокого уровня вы не тестируете всю условную логику и пограничные случаи, которые уже покрыты тестами более низкого уровня.  Уровни тестирования:   * Unit/component/program/module testing - тестируется минимально-атомарный модуль программы, чаще всего это одна функция или метод. Таких тестов должно быть больше всего; * Integration testing - несколько модулей программы тестируются вместе; * System testing - вся программа тестируется полностью; * Acceptance testing - программа принимается заказчиком на соответствие заявленным требованиям либо тестировщики проходят end-to-end сценарии с точки зрения пользователя; |
| Виды тестирования после изменений? | | | | Существует несколько видов тестирования, связанного с изменениями: 1. Подтверждающее тестирование (Re-testing) 2. Регрессионное тестирование (Regression Testing) 3. Дымовое тестирование (Smoke Testing) 4. Санитарное тестирование (Sanity Testing) 5. Тестирование сборки (Build Verification Test)  Давайте разберем их более подробно. Подтверждающее тестирование (Re-testing) Подтверждающее тестирование направлено на проверку исправления бага. Суть его в том, что после исправление дефекта программное обеспечение может быть протестировано с использованием тестовых сценариев, которые завершились с ошибкой из-за найденного дефекта. То есть на новой версии программного обеспечения должны быть повторно выполнены шаги по воспроизведению сбоев, вызванных дефектом.  Целью подтверждающего тестирования является удостоверение в том, что найденный дефект был исправлен.  Предположим, что мы тестируем сайт. Положили товар в корзину, пробуем увеличить его количество, но ничего не выходит. Далее оформляем баг-репорт и отдаем разработчикам. Они его пофиксили и настает время для подтверждающего тестирование. Нам необходимо убедиться, что дефект пофикшен. Значит, как минимум, нам необходимо проверить, что баг не воспроизводится по тем шагам, которые указаны в баг-репорте. То есть продуем снова увеличить количество позиций товара в корзине и смотрим, увеличивается ли оно или снова нет. Регрессионное тестирование (Regression Testing) Код связан между собой и одно исправление может повлечь за собой новые проблемы. Если вернутся к примеру с корзиной, то окажется, что количество стало меняться, а вот цвет товара изменить теперь не получается.  Случилось это из-за того, что «цвет» и «количество» обращались к одному участку кода, который и был поправлен.  Получается, что изменение, внесенное в одну часть кода, будь то исправление или что-либо другое, может случайно повлиять на поведение других частей кода. Такие непреднамеренные побочные эффекты называются регрессиями. А, соответственно, регрессионное тестированиенаправлено на обнаружение таких непреднамеренных побочных эффектов.  Давайте представим это визуально.  Есть продукт. Он состоит из множества различных частей**.**  Продукт состоит из множества частейПродукт состоит из множества частей  В одной из частей был баг и разработчик его исправил. То есть были внесены изменения в одну из частей программы (на рисунке выделено зеленым).  Внесли изменение в одну из частейВнесли изменение в одну из частей  Данные изменения могли тем или иным образом отразиться и на работе других частей продукта. На рисунке выделено красным**.**  Изменения отразились на других частяхИзменения отразились на других частях  Либо может быть ситуация, когда в продукте появляется новый функционал. И его работа может повлиять на старый.  Новый функционал отразился на работе старогоНовый функционал отразился на работе старого  То есть нам нужно проверить работу старого функционала после исправления старого кода и/или написания нового. В этом и заключается регрессионное тестирование.  ***Получается, что при подтверждающим тестировании мы проверяем сам баг, а при регрессионным тестирование не вызвало ли исправление бага или написание нового кода каких-либо изменений в других местах.*** Дымовое тестирование (Smoke Testing) Дымовое тестирование — тестирование, проводимое на начальном этапе и, в первую очередь, направленное на проверку готовности разработанного продукта к проведению более расширенного тестирования.  Данный вид тестирования определяет общее состояние качества продукта.  Это короткий цикл тестов, подтверждающий (отрицающий) факт того, что приложение стартует и выполняет свои основные функции. Проверки практически всегда одинаковы и редко претерпевают изменениям. Поэтому целесообразно их автоматизировать.  Вывод о работоспособности основных функций делается на основании результатов поверхностного тестирования наиболее важных модулей приложения на предмет возможности выполнения требуемых задач и наличия быстро находимых критических и блокирующих дефектов. В случае отсутствия таковых дефектов дымовое тестирование объявляется пройденным, и приложение передается для проведения полного цикла тестирования, в противном случае, дымовое тестирование объявляется проваленным, и приложение уходит на доработку.  Данный тип тестирования позволяет на начальном этапе выявить основные быстро находимые критические дефекты. Санитарное или Санити тестирование (Sanity Testing) Относится к виду тестирования, которое используется с целью доказательства работоспособности конкретной функции или модуля согласно заявленным техническим требованиям.  Зачастую санитарное тестирование используют для проверки какой либо части программы или приложения в результате внесенных изменений на нее со стороны факторов окружающей среды. Выполнение его обычно происходит в ручном режиме.  ***Санитарное тестирование ориентировано на глубинное исследование определенной функции, а дымовое — на тестирование большого количества функционала за самые короткие сроки.*** Тестирование сборки (Build Verification Test) Направлено на определение соответствия выпущенной версии критериям качества для начала тестирования. По своим целям является аналогом дымового тестирования, направленного на приемку новой версии в дальнейшее тестирование или эксплуатацию.  Вглубь оно может проникать дальше, в зависимости от требований к качеству выпущенной версии. Разница различия между видами тестирования, связанными с изменениямиразличия между видами тестирования, связанными с изменениями |
| Виды тестирования по исполнителям тестирования? | | | | Классификация **по исполнителям тестирования**   * **Альфа-тестирование** — является ранней версией программного продукта, тестирование которой проводится внутри организации-разработчика; может быть вероятно частичное привлечение конечных пользователей. * **Бета-тестирование** — практически готовое ПО, выпускаемое для ограниченного количества пользователей, разрабатывается в первую очередь для тестирования конечными пользователями и получения отзывов клиентов о продукте для внесения соответствующих изменений. |
| Виды тестов по позитивности и формальность? | | | | **Позитивное тестирование** -проверка, что приложение работает в случае: -Если использовать его по назначению (не пытаться сломать);  **Негативное тестирование** – проверка, что приложение работает если: -Использовать приложение не так как задумывалось;  -Проверка на обработку исключений, а также проверяет, что вызываемая приложением функция не выполняется при срабатывании валидатора. |
| Виды тестирования по степени важности и уровню? | | | | По степени важности:  -Дымовое тестирование – проверка самой важной функциональности программного продукта.  -Тестирование критического пути – проверка функциональности, используемой типичными пользователями в повседневной деятельности.  -Расширенное тестирование – проверка всей заявленной функциональности.  По уровню:  -Модульное / юнит-тестирование – проверка корректной работы отдельных единиц ПО, модулей. Этот вид тестирования могут выполнять сами разработчики.  -Интеграционное тестирование – проверка взаимодействия между несколькими единицами ПО.  -Системное – проверка работы приложения целиком.  -Приёмочное – оценка соответствия заявленным требованиям к программному продукту. |
| Виды тестирования по цели и формальности? | | | | **По цели:**  Функциональное – проверка, что продукт соответствует требованиям заказчика;  Нефункциональное - объектами проверки являются такие нефункциональные параметры системы, как производительность, надежность и масштабируемость;  Нефункциональное тестирование включает:  -Тестирование производительности – работа ПО под определённой нагрузкой.  -Тестирование пользовательского интерфейса – удобство пользователя при взаимодействии с разными параметрами интерфейса (кнопки, цвета, выравнивание и т. д.).  -Тестирование UX – правильность логики использования программного продукта.  -Тестирование защищенности – определение безопасности ПО: защищено ли оно от атак хакеров, несанкционированного доступа к данным и т. д.  -Инсталляционное тестирование – оценка вероятности возникновения проблем при установке, удалении, а также обновлении ПО.  -Тестирование совместимости – тестирование работы программного продукта в определённом окружении.  -Тестирование надежности – работа программы при длительной средней ожидаемой нагрузке.  -Тестирование локализации –оценка правильности версии программного продукта (языковой и культурный аспекты).  **По формальности:**  -Тестирование по тестам – использование написанных заранее тест-кейсов.  -Исследовательское тестирование – одновременная разработка тестов и их использование.  -Свободное тестирование – проверка качества без разработки тестов и написания документации. Основывается на интуиции и опыте тестировщика. |
| Виды тестирования по степени автоматизации? | | | | По степени автоматизации:  -Мануальное (ручное) – без использования дополнительных программных средств, т. е. «вручную».  -Автоматизированное – с использованием программных средств (более детально в описании курса по автоматизации тестирования ПО). |
| Виды тестирования по знанию кода? | | | | **Классификация по знанию системы**   * Тестирование **белого ящика** (White Box) * — это юнит тесты, метод тестирования ПО, который предполагает полный доступ к коду проекта, т.е. внутренняя структура/устройство/реализация системы известны тестировщику (Тестируется код, а не бизнес-логика). * Тестирование **серого ящика** — метод тестирования ПО, который предполагает частичный доступ к коду проекта (комбинация White Box и Black Box методов). Тестирование отдельно бэкенда от фронтенда, например через постман т.е. мы не знаем код сервера, но и не тестируем полностью бизнес-требования т.к. не тестируем фронтенд. * Тестирование **чёрного ящика** (Black Box) — метод тестирования ПО, также известный как тестирование, основанное на [спецификации](https://habr.com/ru/post/587620/#spec) или тестирование поведения — техника тестирования, которая не предполагает доступа (полного или частичного) к системе, т.е. основывается на работе исключительно с внешним интерфейсом тестируемой системы. |
| Какие бывают виды интеграционного тестирования? | | | | Существует несколько видов интеграционного тестирования:  1) Big bang - все компоненты приложения тестируются сразу после того, как они были разработаны, без поэтапного интегрирования. Этот подход обычно используется в небольших проектах.  2) Последовательное тестирование - интеграция компонентов происходит постепенно, сначала тестируются отдельные модули, а затем их комбинации. Этот подход позволяет выявлять проблемы на ранних этапах разработки и облегчает их исправление.  3) Инкрементное тестирование - разработка приложения разбивается на отдельные этапы, после завершения каждого из которых происходит интеграция уже готовых компонентов. Этот подход позволяет обнаруживать проблемы на ранних этапах и быстрее реагировать на изменения в требованиях.  4) Тестирование по сценариям использования - этот подход предполагает тестирование приложения на соответствие ожиданиям пользователей. Компоненты интегрируются и тестируются в соответствии с различными сценариями использования.  5) Смешанное тестирование - используется комбинация различных подходов интеграционного тестирования в зависимости от характера и сложности проекта. |
| Из каких этапов состоит процесс тестирования? | | | | **Этапы**:  1)Инициация,  2)Выявление требований прямых и косвенных,  3)Генерация тестовых случаев,  4)Отбор показательных тестовых случаев,  5)Проведение проверок,  6)Фиксация результатов,  7)Анализ результатов,  8)Передача информации о соответствии проверенного продукта требованиям.  **Инициация** – событие, которое извещает команду тестирования о необходимости сессии тестирования, а также гарантирует выполнение требований к продукту для проведения тестирования.  **Выявление требований** – пожалуй, один из главных шагов в процессе тестирования. Неизвестны требования – нет тестирования. Необходимо собрать всю доступную информацию о предмете тестирования, вариантах использования и т. п. Первый источник – техническая документация и юзер-стори – это прямые требования. Качество же косвенных требований во многом зависят от добросовестности, ответственности, квалификации тестировщика и всей команды проекта.  **Генерация тестовых случаев** – выявление всех возможных случаев использования продукта, его характеристик и особенностей в процессе эксплуатации. Это значит: всех случаев, которые тестировщик может «придумать» на основе прямых и косвенных требований, известных ему. Этот этап требует высокой квалификации специалиста по тестированию.  **Отбор тестовых случаев** – отбор наиболее показательных, значимых и воспроизводимых тестовых случаев. От этого этапа зависит, насколько тестирование будет полезным, эффективным и анализируемым. Например, в «простом» примере с красной кнопкой понятно, что количество косвенных требований стремится к бесконечности, и проверять их все подряд – полный абсурд, но подобные кейсы должны быть сгенерированы хотя бы в голове проверяющего. А для того, чтобы они не вошли в проверки, необходимо выполнить соответствующий отбор и проверить только, действительно ли кнопка красная.  **Проведение проверок** – тут все понятно. Либо согласно документации, либо ad hoc (интуитивно, свободный поиск, без документации). В любом случае это проводится согласно списку отобранных проверок. Почему-то большинство именно этот пункт называет тестированием. И в голове обывателя, незнакомого с профессией, только один этот пункт и содержится J.  **Фиксация результатов** – создание внутренней и внешней тестовой документации в формализованном виде или в виде записей и т. п. На данном этапе отчет о тестирование даже если и создается, то не считается законченным.  **Анализ результатов** – вынесение решения о соответствии проверенного продукта требованиям. Формализация данного решения и его обоснование в виде отчета о тестировании. Сюда также входят процедуры по оценке покрытия требований проверками, тайм-шитинг и пр. Таким образом, проводится анализ не только результатов, но и самой сессии тестирования.  **Передача информации о соответствии продукта требованиям.** Формально: передача внешней тестовой документации заинтересованным в ней сторонам, зачастую инициатору сессии тестирования. В общем случае: помимо документации предоставляется информация о рисках, которые были выявлены в продукте, требованиях, процессах, передаются рекомендации по отработке этих рисков и т. п. Но это – уже QA J! |
| Какие есть модели разработки ПО? | | | | **Водопадная модель** (waterfall model) сейчас представляет скорее исторический интерес, т.к. в современных проектах практически неприменима, исключая авиастроение, военную или космическую отрасли, медицину и финансовый сектор. Она предполагает однократное выполнение каждой из фаз проекта, которые, в свою очередь, строго следуют друг за другом. Очень упрощенно можно сказать, что в рамках этой модели в любой момент времени команде «видна» лишь предыдущая и следующая фаза. В реальной же разработке ПО приходится «видеть весь проект целиком» и возвращаться к предыдущим фазам, чтобы исправить недоработки или что-то уточнить.  **V-образная модель** (V-model)  *V-модель (V-model): Модель, описывающая процессы жизненного цикла разработки программного обеспечения с момента составление спецификации требований до этапа сопровождения. V модель показывает интеграцию процессов тестирования в каждую фазу цикла разработки программного обеспечения. (ISTQB)*  V-образная модель (V-model) является логическим развитием водопадной. Можно заметить (рисунок 2.1.b), что в общем случае как водопадная, так и v-образная модели жизненного цикла ПО могут содержать один и тот же набор стадий, но принципиальное отличие заключается в том, как эта информация используется в процессе реализации проекта. Очень упрощенно можно сказать, что при использовании v-образной модели на каждой стадии «на спуске» нужно думать о том, что и как будет происходить на соответствующей стадии «на подъёме». Тестирование здесь появляется уже на самых ранних стадиях развития проекта, что позволяет минимизировать риски, а также обнаружить и устранить множество потенциальных проблем до того, как они станут проблемами реальными.  **Итерационная инкрементальная модель** (iterative model, incremental model)  *Инкрементная модель разработки (incremental development model): Модель жизненного цикла разработки, в которой проект разделен на серию приращений, каждое из которых добавляет часть функциональности в общих требованиях проекта. Требования приоритезированы и внедряются в порядке приоритетов. В некоторых (но не во всех) версиях этой модели жизненного цикла каждый подпроект следует «мини V-модели» со своими собственными фазами проектирования, кодирования и тестирования. (ISTQB)*  *Итеративная модель разработки (iterative development model): Модель жизненного цикла разработки, в которой проект разделен обычно на большое количество итераций. Итерация это полный цикл разработки, завершающийся выпуском (внутренним или внешним) рабочего продукта, являющегося частью конечного разрабатываемого продукта, который разрастается от итерации к итерации. (ISTQB)*  Итерационная инкрементальная модель является фундаментальной основой современного подхода к разработке ПО. Ключевой особенностью данной модели является разбиение проекта на относительно небольшие промежутки (итерации), каждый из которых в общем случае может включать в себя все классические стадии, присущие водопадной и v-образной моделям. Итогом итерации является приращение (инкремент) функциональности продукта, выраженное в промежуточном билде (build).  **Спиральная модель** (spiral model)  Спиральная модель представляет собой частный случай итерационной инкрементальной модели, в котором особое внимание уделяется управлению рисками, в особенности влияющими на организацию процесса разработки проекта и контрольные точки.  Обратите внимание на то, что здесь явно выделены четыре ключевые фазы:   * проработка целей, альтернатив и ограничений; * анализ рисков и прототипирование; * разработка (промежуточной версии) продукта; * планирование следующего цикла.   С точки зрения тестирования и управления качеством повышенное внимание рискам является ощутимым преимуществом при использовании спиральной модели для разработки концептуальных проектов, в которых требования естественным образом являются сложными и нестабильными (могут многократно меняться по ходу выполнения проекта).  **Гибкая модель (agile model)**  *Гибкая методология разработки программного обеспечения (agile software development): Группа методологий разработки программного обеспечения, основанных на итеративной поэтапной разработке, где требования и решения развиваются посредством сотрудничества между самоорганизующимися межфункциональными командами. (ISTQB)*  Гибкая модель представляет собой совокупность различных подходов к разработке ПО и базируется на т.н. «agile-манифесте». Положенные в основу гибкой модели подходы являются логическим развитием и продолжением всего того, что было за десятилетия создано и опробовано в водопадной, v-образной, итерационной инкрементальной, спиральной и иных моделях. Причём здесь впервые был достигнут ощутимый результат в снижении бюрократической составляющей и максимальной адаптации процесса разработки ПО к мгновенным изменениям рынка и требований заказчика.  **Модель Большого Взрыва** (**Big Bang Model**)  Big Bang Model не имеет определенного процесса. Деньги и усилия объединяются, поскольку вход и выход представляют собой разработанный продукт, который может совпадать, а может и не совпадать с тем, что нужно заказчику. Модель Большого Взрыва не требует особого планирования и составления графиков. Разработчик выполняет анализ требований и кодирование, а также разрабатывает продукт в соответствии с его пониманием. Эта модель используется только для небольших проектов. Нет команды тестирования и формального тестирования не проводится, и это может быть причиной провала проекта.  Преимущества модели большого взрыва: Это очень простая модель. Требуется меньше планирования и составления графиков. Разработчик может создавать собственное программное обеспечение.  Недостатки модели большого взрыва: Модели Большого взрыва нельзя использовать для крупных, текущих и сложных проектов. Высокий риск и неопределенность. |
| Идентификация, авторизация, аутентификация? | | | | **Идентификация** — процедура, в результате выполнения которой для субъекта идентификации выявляется его идентификатор, однозначно определяющий этого субъекта в информационной системе (система запрашивает логин, пользователь его указывает, система распознает его как существующий — это **идентификация)**  **Аутентификация** — процедура проверки подлинности, например проверка подлинности пользователя путем сравнения введенного им пароля с паролем, сохраненным в базе данных.  **Авторизация** — предоставление определенному лицу или группе лиц прав на выполнение определенных действий. |
| Какие есть варианты аутентификации? | | | | Аутентификация по паролю;  Аутентификация по сертификатам;  Аутентификация по одноразовым паролям;  Аутентификация по ключам доступа;  Аутентификация по токенам;  Звонок из службы поддержки;  Аутентификация по биометрии;  Гугл авторизация; |
| Что такое Agile и Scrum? | | | | **Agile** — это итеративный подход к управлению проектами и разработке программного обеспечения, который помогает командам быстрее и с меньшими проблемами поставлять ценность клиентам.  **Scrum** — это методология Agile, предназначенная для разработки продуктов в среде, подверженной изменениям.  **Основное отличие Scrum и Agile в том, что: Agile** — это общая философия, стиль управления проектами, а **Scrum** — один из нескольких методов, используемых для реализации этого стиля, agile-фреймворк.  **Вот что отличает Scrum от других методологий Agile:**   1. Работа разбита на спринты продолжительностью от одной до четырех недель. 2. Бэклог продукта содержит записи о том, какую работу необходимо выполнить. 3. Наличие Scrum-мастера и владельца продукта (product manager). 4. Члены команды проводят короткое «ежедневное собрание по обновлению Scrum». |
| Agile манифест? | | | | **Agile-манифест разработки:**   1. Люди и взаимодействия важнее процессов и инструментов; 2. Работающий продукт важнее исчерпывающей документации; 3. Сотрудничество с заказчиком важнее согласования условий контракта; 4. Готовность к изменениям важнее следования первоначальному плану; |
| Что такое тест-анализ? | | | | **Тест-анализ** – изучение требований и макетов |
| Что такое объект тестирования? | | | | **Объект тестирования** – части приложения, которые нужно тестировать |
| 3 этапа проведения тест-анализа? | | | | **3 этапа тест-анализа:** -Декомпозиция (Разбиение объектов тестирования до атомарного уровня);  -Визуализация (Представление требований в виде mindmap, блок-схем);  -Поиск серых зон (Несостыковки, противоречия, пропуски в требованиях); |
| Какие есть виды тестовой документации? | | | | **Какие есть виды тестовой документации?**  Тестовая документация бывает двух видов: **внешняя и внутренняя.**  **Внешняя документация:**   * Замечание – короткая записка, комментарий о небольшой неточности в реализации продукта. * Баг-репорт – описание выявленного случая несоответствия производимого продукта требованиям, к нему выдвигаемым – ошибки или ее проявления. Он обязательно должен содержать следующие элементы:    + Идею тестового случая, вызвавшего ошибку.   + Описание исходного состояния системы для выполнения кейса.   + Шаги, необходимые для того, чтобы выявить ошибку или ее проявление.   + Ожидаемый результат, т. е. то, что должно было произойти в соответствии с требованиями.   + Фактический результат, т. е. то, что произошло на самом деле.   + Входные данные, которые использовались во время воспроизведения кейса.   + Прочую информацию, без которой повторить кейс не получится.   + Критичность и/или приоритет.   + Экранный снимок (скрин).   + Версию, сборку, ресурс и другие данные об окружении. * Запрос на изменение (улучшение) – описание неявных/некритичных косвенных требований, которые не были учтены при планировании/реализации продукта, но несоблюдение, которых может вызвать неприятие у конечного потребителя. И пути/рекомендации по модификации продукта для соответствия им. * Отчет о тестировании (тест репорт) – документ, предоставляющий сведения о соответствии/ несоответствии продукта требованиям. Может так же содержать описание некоторых подробностей проведенной сессии тестирования, например, затраченное время, использованные виды тестирования, перечень проверенных случаев и т. п. В идеальном варианте фраза вида «Тест пройден. Ошибка не воспроизводится/Функционал работает корректно/Соответствует требованиям» означает, что продукт или его часть полностью соответствует требованиям прямым и косвенным (в производстве ПО).   **Внутренняя документация:**   * Тест-план (план тестирования) – формализованное и укрупненное описание одной сессии тестирования по одному или нескольким направлениям проверок. Т.е. перечень направлений проверок, которые должны быть проведены в рамках сессии тестирования (и, сообразных этим направлениям, требований). Также может содержать в себе необходимую информацию об окружении, методике, прочих условиях важных для показательности данной сессии тестирования. Под направлением проверок также может пониматься более детализированная тестовая документация (в виде ссылки на нее): чек листы, тестовые комплекты, тестовые сценарии, на которую необходимо опираться при проведении сессии тестирования. Основная цель документа – описать границы сессии тестирования, стабилизировать показательность данной сессии. * Тестовый сценарий – последовательность действий над продуктом, которые связаны единым ограниченным бизнес-процессом использования, и сообразных им  проверок корректности поведения продукта в ходе этих действий. Может содержать информацию об исходном состоянии продукта для запуска сценария, входных данных и прочие сведения, имеющие определяющее значение для успешного и показательного проведения проверок по сценарию. Особенностью является линейность действий и проверок, т.е. зависимость последующих действий и проверок от успешности предыдущих. Цель документа – стабилизация покрытия аспектов продукта, необходимых для выполнения функциональной задачи, показательными необходимыми и достаточными проверками. Фактически при успешном прохождении всего тестового сценария мы можем сделать заключение о том, что продукт может выполнять ту или иную возложенную на него функцию. * Тестовый комплект – некоторый набор формализованных тестовых случаев, объединенных между собой по общему логическому признаку. * Чек-лист (лист проверок) – перечень формализованных тестовых случаев в виде удобном для проведения проверок. Тестовые случаи в чек-листе не должны быть зависимыми друг от друга. Обязательно должен содержать в себе информацию о: идеях проверок, наборах входных данных, ожидаемых результатах, булевую отметку о прохождении/непрохождении тестового случая, булевую отметку о совпадении/несовпадении фактического и ожидаемого результата по каждой проверке. Может так же содержать шаги для проведения проверки, данные об особенностях окружения и прочую информацию необходимую для проведения проверок. Цель – обеспечить стабильность покрытия требований проверками необходимыми и достаточными для заключения о соответствии им продукта. Особенностью является то, что чек-листы компонуются теми тестовыми случаями, которые показательны для определенного требования. * Тестовый случай (тест-кейс) – формализованное описание одной показательной проверки на соответствие требованиям прямым или косвенным. Обязательно должен содержать следующую информацию:    + Идея проверки.   + Описание проверяемого требования или проверяемой части требования.   + Используемое для проверки тестовое окружение.   + Исходное состояние продукта перед началом проверки.   + Шаги для приведения продукта в состояние, подлежащее проверке.   + Входные данные для использования при воспроизведении шагов.   + Ожидаемый результат.   + Прочую информацию, необходимую для проведения проверки. |
| Какие бывают требования? | | | | Требования бывают прямыми, косвенными, функциональные нефункциональные. |
| Какие требования входят в функциональные? | | | | Функциональные требования - описывают какие функции должен выполнять продукт. |
| Какие требования входят в не функциональные? | | | | Нефункциональные требования - требования к окружению, поддерживаемости, надежности и прочим характеристикам продукта. |
| Примеры прямых требований? | | | | Прямые требования - формализованными в технической документации, спеках, юзер-стори и прочих формальных артефактах. |
| Примеры косвенных требований? | | | | Косвенные требования - проистекающими из прямых, либо являющимися негласным стандартом для данной продукции или основывающиеся на опыте и здравом смысле использования данного продукта или продуктах, подобных ему |
| 8 характеристик требований? | | | | Требования должны быть:   1. Полными; (Все ли описано?) 2. Однозначными; («Отчет должен загружаться быстро» → что значит «быстро»?) 3. Непротиворечивость; (Требования не должны противоречить сами себе) 4. Необходимость; (Кратко, но емко) 5. Осуществимость; (А можно ли реализовать то, что тут написано? Насколько это будет сложно и дорого?) 6. Тестируемость; (Можно ли протестировать этот функционал?) |
| Источники требований? | | | | **Источники требований:**  -заказчик  -требования  -фокус группа  -анкетирования  -нормы  -мозговой штурм  -наблюдение  -лучшие практики  -конкуренты |
| Классы эквивалентности? Диапазон, набор значений? | | | | **Классы эквивалентности** - одно или несколько значений ввода, к которым программное обеспечение применяет одинаковую логику;  **Диапазон** – интервал чисел с границами;  **Набор значений** – множество значений, каждое из которых прописано в требованиях отдельно (У них нет границ и граничных значений, например: буквы, нужно взять одну из диапазона, а все остальные можно не проверять)  **Техника анализа классов эквивалентности классический пример:**  Есть поле ввода с диапазоном допустимых значений от 1 до 100.  Сами понимаете, что на 95 тестов на допустимые значения и на несметное количество тестов на недопустимые значения уйдет очень много времени. И здесь нам помогут классы эквивалентности.  Исходя из того, что с одной стороны, все допустимые значения могут влиять на поле ввода одинаково, следовательно все числа от 1 до 100 можно смело считать эквивалентными. С другой стороны, все недопустимые значения должны одинаково влиять на поле ввода (в идеале не должно быть возможности ввода этих значений в поле). Таким образом, есть уже несколько классов эквивалентности:  допустимые значения (от 1 до 100); недопустимые значения:  1. от — ∞ до 0; 2. от 101 до + ∞; 3. специальные символы (# @ + — / \_ : ; “ ‘ и т.д.); 4. буквы. Используя классы эквивалентности можно протестировать поле ввода минимум из 4 тестов. |
| Граничные значения? | | | | **Граничные значения** – значения диапазона входных данных, при которых меняется поведение приложения. Это соседние значения диапазона, но относящиеся к разным эквивалентным классам.  **Проверка:**  **Есть два подхода:**  1)(граница — 1), граница, (граница + 1) – рекомендации из книги Ли Копленд;  2)Берем только значения на самой границе – рекомендации экзамена ISTQB;  **Техника анализа граничных значений:** Это техника проверки поведения продукта на крайних (граничных) значениях входных данных. Граничное тестирование также может включать тесты, проверяющие поведение системы на входных данных, выходящих за допустимый диапазон значений. При этом система должна определённым (заранее оговоренным) способом обрабатывать такие ситуации. Например, с помощью исключительной ситуации или сообщения об ошибке.  **На каждой границе диапазона следует проверить по три значения:**  граничное значение; значение перед границей; значение после границы. Цель этой техники — найти ошибки, связанные с граничными значениями.  **Алгоритм использования техники граничных значений:**  1) Выделить классы эквивалентности; Как и в предыдущей технике, этот шаг является очень важным и от того, насколько правильным будет разбиение на классы эквивалентности, зависит эффективность тестов граничных значений.  2) Определить граничные значения этих классов; нужно понять, к какому классу будет относиться каждая граница; нужно провести тесты по проверке значения до границы, на границе и сразу после границы. Количество тестов для проверки граничных значений будет равен количеству границ, умноженному на 3. Рекомендуется проверять значения вплотную к границе. К примеру, есть диапазон целых чисел, граница находится в числе 100. Таким образом, будем проводить тесты с числом 99 (до границы), 100 (сама граница), 101 (после границы).  Техника анализа граничных значений. Пример использования на существующем проекте:  Проект — Btrack.com  Протестировать поля ввода — Dev-Est/Qa-est  Условие — в поля ввода можно внести только целые числа от 3 до 10 000.  Определяемся с существующими границами — так как в условии все значения от 3 до 10 000 приведут к одному и тому же результату, то границы две: нижняя и верхняя.  Первое граничное значение — 3  Второе граничное значение — 10 000  Добавляем к ним, стоящие рядом значения ( опять же, если бы, числа были дробными, то пришлось бы, для начала, определится с количеством знаков после запятой. Получаем ГЗ для данного поля:  4 варианта ответа, все они будут верные:  Вариант 1: 0, 1, 3, 4, 9 999, 10 000, 10 001.  Вариант 2:  2, 3, 4, 9999, 10000, 10001.  Вариант 3:  2, 3, 10 000, 10 001.  Вариант 4:  2, 3, 10 001.  Все эти варианты будут правильными т.к. все зависит от того, как мы выделим классы эквивалентности и сколько времени у нас займет проверка дополнительного тестового случая. |
| Техника тест-дизайна “Доменное тестирование” | | | | [Доменное тестирование — это методика разработки тестов, относящаяся к методу черного ящика, использующаяся для определения действенных и эффективных тестовых сценариев в случаях, когда входные или выходные данные имеют определенный диапазон или множество значений1](https://sedtest-school.ru/test-design/chto-takoe-domennoe-testirovanie/). [Доменное тестирование включает в себя две основные техники: эквивалентное разбиение и анализ граничных значений2](https://crashtest.by/domain-testing/).  Пример доменного тестирования:  Предположим, что мы хотим протестировать поле ввода для возраста пользователя на сайте. Мы знаем, что возможный диапазон значений — от 18 до 99 лет. Мы можем разделить этот диапазон на несколько классов эквивалентности: позитивный (от 18 до 99), негативный (меньше 18 или больше 99) и невалидный (не число или пустое значение). Для каждого класса мы выбираем одно или несколько представительных значений для проверки. Например:   * Позитивный класс: 18, 50, 99 * Негативный класс: -1, 0, 17, 100, 150 * Невалидный класс: “abc”, “18.5”, “”, null   Затем мы фокусируемся на граничных значениях каждого класса и проверяем их поведение. Например:   * Позитивный класс: 18 (минимальное), 99 (максимальное) * Негативный класс: 17 (минимальное -1), 100 (максимальное +1)   Таким образом, мы можем покрыть все возможные случаи с минимальным количеством тестов. |
| Техника тест-дизайна “Попарное тестирование”? | | | | **Попарное тестирование:** Суть этого метода, также известного как pairwise testing, в том, что каждое значение каждого проверяемого параметра должно быть протестировано на взаимодействие с каждым значением всех остальных параметров. После составления такой матрицы мы убираем тесты, которые дублируют друг друга, оставляя максимальное покрытие при минимальном необходимом наборе сценариев.   Попарное тестирование позволяет обнаружить максимум ошибок без избыточных проверок.   Pairwise testing: пример:   Для Parwise достаточно, чтобы каждое значение всех параметров хотя бы единожды сочеталось с другими значениями остальных параметров. Таким образом, матрицу можно значительно сократить. Например:  № Браузер Операционная система Язык 1 Opera Windows RU 2 Google Chrome Linux RU 3 Opera Linux EN 4 Google Chrome Windows EN   При составлении матрицы принятия решений для двух браузеров, двух ОС и двух языков было бы нужно 8 сценариев. При попарном тестировании достаточно четырех.   Все это можно просчитать и вручную, но не обязательно – гораздо удобнее автоматизировать процесс. Для этого существует программа попарного независимого комбинированного тестирования – Pairwise Independent Combinatorial Testing (PICT). Для проведения тестирования специалист создает текстовый файл с перечислением и их возможных значений, а затем запускает PICT через cmd – командную строку. Скомбинированные тесты отображаются в виде таблицы в самой консоли. Так же результаты по желанию можно выгрузить в файл Excel. |
| Техника тест-дизайна “Таблица принятия решений”? | | | | **Таблица принятия решений** Другое название метода – матрица принятия решений. Эта техника подходит для более сложных систем, например – двухфакторной аутентификации. Предположим, чтобы войти в систему, пользователю нужно ввести сначала логин и пароль, а затем еще подтвердить свою личность присланным в смс кодом.   Какие возможны сценарии: 1. Правильный логин и правильный пароль. 2. Правильный логин, неправильный пароль. 3. Неправильный логин, правильный пароль. 4. Неправильный логин, неправильный пароль.   Первый из этих сценариев сопровождается либо правильным, либо неправильным вводом смс-кода, итого у нас получается 5 тестов. При этом только один из сценариев приведет к положительному результату (пользователь успешно авторизуется), а остальные закончатся неудачей.   Однако, может быть так, что система выдает разные сообщения в зависимости от того, на каком этапе была допущена ошибка, скажем: invalid login, invalid password. Соответственно, групп потребуется больше, а таблица станет обширнее.   Этот метод хорош тем, что он показывает сразу все возможные сценарии в форме, понятной даже неспециалисту. |
| Техника тест-дизайна “Предугадывание ошибок”? | | | | **Предугадывание ошибок** Используя свои знания о системе, QA-специалист может «предугадать», при каких входных условиях есть риск ошибок. Для этого важно иметь опыт, хорошо знать продукт и уметь выстроить коммуникации с коллегами.   Например, в спецификации указано, что поле должно принимать код из четырех цифр. В числе возможных тестов: Что произойдет, если не ввести код? Что произойдет, если не ввести спецсимволы? Что произойдет, если ввести не цифры, а другие символы? Что произойдет, если ввести не четыре цифры, а другое количество? Преимущества:  1. Эта проверка эффективна в качестве дополнения к другим техникам. 2. Выявляет тестовые случаи, которые “никогда не должны случиться”.  Недостатки: 1. Техника в значительной степени основана на интуиции. 2. Необходим опыт в тестировании подобных систем. 3. Малое покрытие тестами. Итоги |
| Что такое тест-кейс? | | | | **Тест-кейс** — это профессиональная документация тестировщика, последовательность действий, направленная на проверку какого-либо функционала, описывающая как прийти к фактическому результату.  **Атрибуты тест-кейса:**   * **Уникальный идентификатор тест-кейса** — необходим для удобной организации хранения и навигации по нашим тест-наборам. * **Название**— основная тема, или идея тест-кейса. Кратное описание его сути. * **Предусловия** — описание условий, которые не имеют прямого отношения к проверяемому функционалу, но должны быть выполнены. Например, оставить комментарий на вашем портале может только зарегистрированный пользователь. Значит для тест-кейса «Создание комментария» будет необходимо выполнение предусловия «пользователь зарегистрирован», и «пользователь авторизован» * **Шаги** — описание последовательности действий, которая должна привести нас к ожидаемому результату * **Ожидаемый результат** — результат: что мы ожидаем увидеть после выполнения шагов. |
| Что такое чек-лист? | | | | **Чек-лист** — это список проверок, которые помогают тестировщику протестировать приложение или отдельные функции.  Сила тест-кейса в том, что в нем все расписано очень детально, и с помощью тест-кейсов тестировать сможет даже человек, который ни разу не видел тестируемое им приложение. Но создание и поддержка кейсов требует времени, сил и является рутиной. Помимо прочего, очевидно, тест-кейс часто подразумевает только один конкретный тест, когда в чек-листе подразумевается целый перечень разных проверок.  Сила чек-листа в том, что он простой. Там нет глубокой детализации, это просто памятка. К тому же, он довольно наглядный с точки зрения отчетности. Минус в том, что другому человеку может быть сложно вникнуть в суть проверок без деталей и шагов. Чек-листы стали популярнее с приходом гибких моделей разработки, когда писать детальные кейсы может не быть времени и смысла, т.к. всё меняется слишком быстро, к тому же команда может быть небольшой и расписывать кейсы просто не для кого. |
| Критерии выбора тестовой документации? | | | | **Стабильность функциональности:**  Если нестабильна, описываем чек-листами, если стабильна – тест-кейсами;  **Сложность бизнес-логики**:  При сложной бизнес-логике лучше пользоваться тест-кейсами;  **Размер проекта:**  На Большом проекте при использовании чек-листа есть вероятность пропустить дефекты;  **Стабильность команды:**  Если команда Нестабильна, на сложном проекте нужно будет потратить больше времени на вхождение в проект => без тест-кейсов будет сложно;  **Бюджет:**  Если бюджет позволяет, то можно выбрать тест-кейсы чтобы не пропустить дефекты;  **Время разработки проекта:**  На Краткосрочный проект нет смысла писать дорогую в поддержании и написании детализированную тестовую документацию, можно взять более простой вариант – чек-лист;  **Желание заказчика**:  Мы можем посоветовать заказчику что-либо, но решение принимает он; |
| Что такое тест-набор? | | | | **Тест Сьют (тестовый набор)** - это набор [тест кейсов](http://software-testing.org/testing/chto-takoe-test-keys-test-case-iz-kakih-poley-sostoit-tipichnyy-test-keys.html), которые объединены тем что относятся к одному тестируемому модулю, функциональности, приоритету или одному типу тестирования. Каждый тест сьют состоит из более чем одного тест кейса и зачастую выполняется всей «пачкой» в процессе тестирования. |
| Что такое баг? | | | | **Баг** — некорректная работа программы, вызванная ошибкой в программном коде или дизайне продукта. |
| Какие атрибуты есть в баг-репорте? | | | | |  |  | | --- | --- | | Заголовок (Summary) | Короткое описание проблемы, явно указывающее на причину и тип ошибочной ситуации. | | Проект (Project) | Название тестируемого проекта | | Компонент приложения (Component) | Название части или функции тестируемого продукта | | Номер версии (Version) | Версия, на которой была найдена ошибка | | Критичность  (Severity) | Наиболее распространена пятиуровневая система критичности:  S1 Блокирующий (Blocker)  S2 Критический (Critical)  S3 Значительный (Major)  S4 Незначительный (Minor)  S5 Тривиальный (Trivial) | | Приоритет (Priority) | Приоритет дефекта:  P1 Высокий (High)  P2 Средний (Medium)  P3 Низкий (Low) | | Статус (Status) | Статус бага. Зависит от используемой процедуры и жизненного цикла бага. Например:   * Новый * Открыт * Закрыт | | Автор (Author) | Создатель баг репорта | | Назначен на (Assigned To) | Имя сотрудника, назначенного на решение проблемы | | Описание (Description) | Информация об окружении, на котором был найден баг: операционная система, сервис пак, имя и версия браузера, версия ПО чипа, версия библиотеки  и т.д. | | Шаги, по которым можно легко воспроизвести ситуацию, приведшую к ошибке. | | Полученный результат | | Ожидаемый результат | | Прикрепленный файл (Attachment) | Файл с логами, скриншот или любой другой документ, который может помочь прояснить причину ошибки или указать на способ решения проблемы | |
| Какие существуют степени серьезности бага? | | | | Наиболее распространена пятиуровневая система критичности:  S1 Блокирующий (Blocker)  S2 Критический (Critical)  S3 Значительный (Major)  S4 Незначительный (Minor)  S5 Тривиальный (Trivial) |
| Какие есть виды приоритета? | | | | Приоритет дефекта:  P1 Высокий (High)  P2 Средний (Medium)  P3 Низкий (Low) |
| Пример высокого приоритета и низкой серьезности и низкого приоритета и высокой серьезности? | | | | **Высокий приоритет, но низкая серьезность:**  У компании Google на главной странице ошибка в слове “Gogle”. Приоритет будет “Высокий”, но степень серьезности низкая. Приоритет высокий т.к. у компании с мировым именем ошибка в названии – это большие риски и кринж:D  **Низкий приоритет, но высокая серьезность:**  У нас есть организация, в которой каждый год 31 декабря с помощью определенного функционала мы делаем годовой отчет. В июне мы узнаем, что этот функционал сломался. Высокая серьезность будет т.к. функционал полностью не работает, а низкий приоритет т.к. мы этим функционалом пользуемся раз в год и если мы нашли ошибку в июне, то по сути нам он понадобится только через пол года. |
| Жизненный цикл дефекта? | | | |  |
| Что такое тест-стратегия? | | | | **Стратегия тестирования** — это статический документ высокого уровня, обычно разрабатываемый менеджером проекта. Это документ, который отражает подход к тестированию продукта и достижению целей, и дает четкое представление о том, что команда тестирования будет делать для всего проекта. Обычно он выводится из Спецификации бизнес-требований (BRS). Как только стратегия тестирования готова, группа тестирования начинает писать подробный план тестирования и продолжает дальнейшие этапы тестирования.  **Содержание стратегии:**   * **Обзор и объем** (Scope and overview): объем работ по тестированию (что тестировать и зачем тестировать) и обзор тестируемого продукта; * **Подход к тестированию** (Test Approach):   + Уровни тестирования (Test levels);   + Виды тестирования (Test Types);   + Роли и обязанности (Roles and responsibilities);   + Требования к окружениям (Environment requirements); * **Инструменты тестирования** (Testing tools): инструменты, необходимые для проведения тестов (TMS, багтрекинговая система, стек автоматизации); * **Отраслевые стандарты**, которым необходимо следовать (Industry standards to follow): В этом разделе описывается отраслевой стандарт для производства высококачественной системы, которая соответствует ожиданиям клиентов или превосходит их. Обычно менеджер проекта определяет модели и процедуры тестирования, которым необходимо следовать для достижения целей проекта; * **Результаты тестирования** (Test deliverables): документация, которую необходимо создать до, во время и по окончании тестирования; * **Метрики тестирования** (Testing metrics): метрики, которые следует использовать в проекте для анализа статуса проекта; * **Матрица отслеживания требований** (RTM); * **Риски и способы их снижения** (Risk and mitigation): все риски тестирования и план по их снижению; * **Инструмент отчетности** (Reporting tool): как будут отслеживаться дефекты и проблемы; * **Результаты тестов** (Test Summary): виды сводных отчетов о тестах, которые будут создаваться, с указанием периодичности. Сводные отчеты о тестах будут генерироваться ежедневно, еженедельно или ежемесячно, в зависимости от критичности проекта. |
| Что такое тест-план? | | | | **“План тестирования (test plan):** Документ, описывающий цели, подходы, ресурсы и график запланированных тестовых активностей. Он определяет объекты тестирования, свойства для тестирования, задания, ответственных за задания, степень независимости каждого тестировщика, тестовое окружение, метод проектирования тестов, определяет используемые критерии входа и критерии выхода и причины их выбора, а также любые риски, требующие планирования на случай чрезвычайных обстоятельств.”  **Что входит?**   Перечень планируемых тестовых активностей ([Test Activities](https://theqalead.com/wp-content/uploads/2021/06/Test-activities-infographic-1024x579.png));   Тестовая логистика ([Test Logistics](https://theqalead.com/wp-content/uploads/2021/06/Test-logistics-infographic-1024x579.png));   Необходимые ресурсы ([Resources](https://theqalead.com/wp-content/uploads/2021/06/Resources-infographic-1024x579.png));   Необходимые коммуникации ([Your Support Network](https://theqalead.com/wp-content/uploads/2021/06/Typical-requirements-infographic-1024x579.png));   Оценки трудозатрат (Estimates);   Зависимости и риски (Dependencies, Risks and Assumptions);   Порядок обсуждений и отчетности в процессе работы (Communication, Commitment and Progress Reporting);  **ПО тест-планов**:   * **Мастер Тест-План** ([Master Test Plan](https://tryqa.com/what-are-master-test-plans-level-test-plan-examples-when-to-use/)): *“Главный план тестирования (master test plan, project test plan): План тестирования, обычно охватывающий несколько уровней тестирования.” (ISTQB)*. Это может быть как единственный базовый план, так и главный в иерархии нескольких планов, самый статичный и высокоуровневый. Нужен когда:   + продукт имеет множество релизов или итераций, между которыми сохраняется общая информация, которую нет смысла повторять;   + различные тестовые команды работают над одним продуктом, выполняя различные задачи, которые необходимо объединить в рамках одного документа; * **Детальный Тест-план** (Phase Test plan): *“Уровневый план тестирования (level test plan): План тестирования, обычно относящийся к одному уровню тестирования.” (ISTQB).* Детальный план составляется на каждый релиз/итерацию или для каждой команды в рамках проекта и является динамическим, т.е. может претерпевать изменения по необходимости. Его основная цель - кратко и доходчиво отразить задачи тестирования. Детальных планов может быть несколько для отдельных модулей ПО или команд тестирования. Кроме того, могут быть созданы планы для отдельных уровней тестирования (Level Test Plan) или видов тестирования. В Agile проектах могут быть планы итерационного тестирования ([iteration testing plans](https://tryqa.com/what-is-release-and-iteration-planning-in-agile-methodology/)) для каждой итерации; * **План приемочных испытаний** (Acceptance Test Plan, ПСИ): план приемочного тестирования отличают от обычного плана тестирования факторы, которые приводят к принятию бизнес-решения. План приемочного тестирования - это один из жизненно важных документов, который содержит руководство по выполнению приемочного тестирования для конкретного проекта. Пишется на основе бизнес-требований (Business Requirements). Ревью этого плана обычно выполняется by Managers/Business Analysts/Customers. |
| Что такое отчет о тестировании? | | | | **Отчет о тестировании** — документ, включающий в себя результаты работ по тестированию и содержащий информацию, достаточную для соотнесения текущей ситуации с тест-планом и принятия необходимых управленческих решений.  **Пункты отчета:** 1. Состав команды; 2. Сроки выполнения, за которые составляется отчет; 3. Описание процессов тестирования; 4. Изменения тестовой модели, дополнение ТК; 5. Процент пройденных ТК; 6. Критичные и блокирующие проблемы и принятые меры по их устранению; 7. Результаты регресса (плюс акцент на сохранившихся проблемах); 8. План на следующую итерацию\ неделю\ месяц; |
| Виды мобильных приложений? | | | | Виды МП:  -Веб-приложения – сайт адаптированный под мобильное устройство;  -Нативные приложения – приложение которые нужно устанавливать;  -Гибридные – комбинация двух предыдущих, в основе нативное приложение, а веб наполняют HTML, CSS, Js; |
| Что такое матрица устройств? | | | | **Матрица устройств** – набор устройств, на котором будем тестировать. Она помогает сократить набор устройств и покрыть большинство комбинаций. |
| Какие есть эмуляторы для Android и iOS? | | | | **Android – Android Studio;**  **iOS – Xcode;** |
| Что такое клиент-серверная архитектура? | | | | **Клиент - сервер** - вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Фактически клиент и сервер — это программное обеспечение. Обычно эти программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине. Программы-серверы ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде:   * данных (например, загрузка файлов посредством HTTP, FTP, BitTorrent, потоковое мультимедиа или работа с базами данных); * сервисных функций (например, работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями или просмотр web-страниц во всемирной паутине). |
| Схема КСА и принцип работы? | | | | **Характеристики архитектуры «клиент-сервер»**   * Асимметричность протоколов. Между клиентами и сервером существуют отношения «один ко многим». Инициатором диалога с сервером обычно является клиент. * Инкапсуляция услуг. После получения запроса на услугу от клиента, сервер решает, как должна быть выполнена данная услуга. Модификация («апгрейд») сервера может производиться без влияния на работу клиентов, поскольку это не влияет на опубликованный интерфейс взаимодействия между ними. Иными словами, максимум, что может при этом почувствовать пользователь - незначительная задержка отклика сервера в течение небольшого времени апгрейда. * Целостность. Программы и общие данные для сервера управляются централизованно, что снижает стоимость обслуживания и защищает целостность данных. В то же время, данные клиентов остаются персонифицированными и независимыми. * Местная прозрачность. Сервер — это программный процесс, который может исполняться на той же машине, что и клиент, либо на другой машине, подключенной по сети. Программное обеспечение «клиент-сервер» обычно скрывает местоположение сервера от клиентов, перенаправляя запрос на услуги через сеть. * Обмен на основе сообщений. Клиенты и сервер являются нежёстко связанными («loosely-coupled») процессами, которые обмениваются сообщениями: запросами на услуги и ответами на них. * Модульный дизайн, способный к расширению. Модульный дизайн программной платформы «клиент-сервер» придаёт ей устойчивость к отказам, то есть, отказ в каком-то модуле не вызывает отказа всего приложения. В такой системе, один или больше серверов могут отказать без остановки всей системы в целом, до тех пор, пока услуги отказавшего сервера могут быть предоставлены с резервного сервера. Другое преимущество модульности в том, что приложение «клиент-сервер» может автоматически реагировать на повышение или понижение нагрузки на систему, путем добавления или отключения услуг или серверов. * Независимость от платформы. Идеальное приложение «клиент-сервер» не зависит от платформ оборудования или операционной системы. Клиенты и серверы могут развертываться на различных аппаратных платформах и разных операционных системах. * Масштабируемость. Системы «клиент-сервер» могут масштабироваться как горизонтально (по числу серверов и клиентов), так и вертикально (по производительности и спектру услуг). * Разделение функционала. Система «клиент-сервер» - это соотношение между процессами, работающими на одной или на разных машинах. Сервер - это процесс предоставления услуг. Клиент - это потребитель услуг. * Общее использование ресурсов. Один сервер может предоставлять услуги множеству клиентов одновременно, и регулировать их доступ к совместно используемым ресурсам. |
| Уровни КСА какие есть? | | | | **Одноуровневая архитектура (1-Tier)**  Одноуровневая архитектура «клиент-сервер» (1-Tier) - такая, где все прикладные программы рассредоточены по рабочим станциям, которые обращаются к общему серверу баз данных или к общему файловому серверу. Никаких прикладных программ сервер при этом не исполняет, только предоставляет данные.  В целом, такая архитектура очень надежна, однако, ей сложно управлять, поскольку в каждой рабочей станции данные будут присутствовать в разных вариантах. Поэтому возникает проблема их синхронизации на отдельных машинах. В общем, как можно видеть из рисунка, в этой архитектуре просматривается еще один уровень - базы данных, что дает повод во многих случаях называть её двухуровневой.    **Двухуровневая архитектура (2-Tier)**  К двухуровневой архитектуре «клиент-сервер» следует относить такую, в которой прикладные программы сосредоточены на сервере приложений (Application Server), например, сервере 1С или сервере CRM, а в рабочих станциях находятся программы-клиенты, которые предоставляют для пользователей интерфейс для работы с приложениями на общем сервере.  Такая архитектура представляется наиболее логичной для архитектуры «клиент-сервер». В ней, однако, можно выделить два варианта. Когда общие данные хранятся на сервере, а логика их обработки и бизнес-данные хранятся на клиентской машине, то такая архитектура носит название “fat client thin server” (толстый клиент, тонкий сервер). Когда не только данные, но и логика их обработки и бизнес-данные хранятся на сервере, то это называется “thin client fat server” (тонкий клиент, толстый сервер). Такая архитектура послужила прообразом облачных вычислений (Cloud Computing).    **Трехуровневая архитектура (3-Tier)**  В трехуровневой архитектуре сервер баз данных, файловый сервер и другие представляют собой отдельный уровень, результаты работы которого использует сервер приложений. Логика данных и бизнес-логика находятся в сервере приложений. Все обращения клиентов к базе данных происходят через промежуточное программное обеспечение (middleware), которое находится на сервере приложений. Вследствие этого, повышается гибкость работы и производительность. |
| Что такое толстый и тонкий клиент? | | | | **Толстый клиент** – выполняет все вычисления за счет компьютерных мощностей клиента;  **Тонкий клиент** – выполняет все расчет на стороне сервера; |
| Какие бывают виды ВЕБ? | | | | Статический сайт – это тот, который возвращает тот же жесткий кодированный контент с сервера всякий раз, когда запрашивается конкретный ресурс. Сервер будет обрабатывать только GET запросы т.к. сервер никаких данных не сохраняет и на сайте нет возможности сохранить никакие данные.  **Динамический сайт** – это тот который может генерировать и возвращать контент на основе конкретного URL, запроса и данных. На сайте есть авторизация, какой-либо функционал, связанный с передачей и обработкой данных на сервере. |
| Что такое микросервисы? | | | | **Микросервисная архитектура или просто микросервисы** - это особый метод разработки программных систем, который пытается сосредоточиться на создании однофункциональных модулей с четко определенными интерфейсами и операциями. Эта тенденция стала популярной в последние годы, поскольку предприятия стремятся стать более гибкими и перейти к DevOps и непрерывному тестированию. Минусы микросервисного подхода При кажущейся простоте и логичности деление большого монолитного приложения на самостоятельные сервисы — сложная техническая задача. Сам микросервисный подход не лишён недостатков, а его плюсы могут оборачиваться минусами:   * **Сложнее осуществлять мониторинг**. Монолит один, и отслеживать, как он работает проще. Микросервисов сотни, а иногда тысячи, и уследить за каждым физически невозможно. Поэтому приходится уделять много внимания системам управления и мониторинга. * **Каждый микросервис может использовать тот язык программирования и те технологии, что удобны команде, которая его разрабатывает**. С одной стороны, это плюс — какие-то фреймворки и языки лучше заточены под конкретные вещи. Если мы делаем сервис обработки изображений или машинного обучения, можем не заострять внимание на технологиях монолита, а выбрать решение под задачу. С другой стороны, это минус — нужно как-то конфигурировать все микросервисы между собой и поддерживать «зоопарк» технологий. * **Снижение доверия**. Когда у нас сотни узлов, могут возникать проблемы с аутентификацией и авторизацией, поскольку есть вероятность подсоединения мошенников. * **Сложности развёртывания**. Чтобы требования по отказоустойчивости выполнялись, микросервисы нужно развёртывать на отдельных серверах. И здесь не работает подход: «Берём приложение, ставим и запускаем». Нужны системы оркестрации и деплоймента.  Плюсы микросервисного подхода Один из основных плюсов микросервисов — повышение показателей доступности и отказоустойчивости. Когда монолит падает, он полностью перестаёт работать. В приложении с микросервисной архитектурой перестаёт работать только какая-то часть. Например, в интернет-магазине может сломаться корзина, но клиенты без проблем продолжат пользоваться каталогом, добавлять товары в избранное и т.д.  Ещё один плюс — повышение надёжности. Предположим, начинающий разработчик залил непроверенные изменения в прод. В случае с монолитом всё сразу упадёт, а с микросервисом сломается только участок системы, остальное будет работать.  Также важно, что в приложении с микросервисной архитектурой локализуются сложность и риски отказов, а производительность системы масштабируется по горизонтальному пути. Компоненты (узлы) автономны, а значит легче поддаются тестированию. |
| Архитектура веб-сервиса бывает? | | | | 1. **SOA (Service Based Architecture)** - модульный подход к разработке программного обеспечения, основанный на использовании распределённых, слабо связанных заменяемых компонентов, оснащённых стандартизированными интерфейсами для взаимодействия по стандартизированным протоколам.    * Программные комплексы, разработанные в соответствии с сервис-ориентированной архитектурой, обычно реализуются как набор веб-служб, взаимодействующих по протоколу SOAP, но существуют и другие реализации, например, на основе REST. 2. **ROA (REST-Oriented Architecture)** - архитектурный стиль приложения и подход к разработке для создания ПО в виде ресурсов с RESTful интерфейсами. Эти ресурсы являются программными компонентами, которые могут быть пере использованы для различных целей. 3. **MOM (Message-Oriented Model)** сосредоточена на тех аспектах архитектуры, которые относятся к сообщениям и их обработке. 4. **SOM (Service-Oriented Model)** сосредоточена на тех аспектах архитектуры, которые относятся к сервису и действиям.    * Главная цель SOM - устанавливать отношения между агентом, сервисом, который он реализует, и запросами.    * SOM построен на основе MOM, но сосредоточен больше на действия, чем на сообщения. 5. **ROM (Resource-Oriented Model)** сосредоточена на тех аспектах архитектуры, которые относятся к ресурсам, и сервис модель которых связана с манипулированием ресурсами. 6. **PM (Policy Model)** сосредоточена на тех аспектах архитектуры, которые относятся к политике, расширениям, защите и качеству сервиса. 7. **MM (Management Model)** сосредоточена на тех аспектах архитектуры, которые относятся к регулированию веб сервисов. |
| Что такое протокол? | | | | **Протокол** - набор правил, задающих форматы сообщений и процедуры, которые позволяют компьютерам и прикладным программам обмениваться информацией. |
| Две модели сетей? | | | | Существует две модели сети:  OSI и TCP/IP; |
| 7 уровней модели OSI? | | | | **Физический уровень** занимается реальной передачей необработанных битов по каналу связи. При разработке сети необходимо убедиться, что когда одна сторона передает единицу, то принимающая сторона получает также единицу, а не ноль. Принципиальными вопросами здесь являются следующие: какое напряжение должно использоваться для отображения единицы, а какое для нуля; сколько микросекунд длится бит; может ли передача производиться одновременно в двух направлениях; как устанавливается начальная связь и как она прекращается, когда обе стороны закончили свои задачи; из какого количества проводов должен состоять кабель и какова функция каждого провода. Вопросы разработки в основном связаны с механическими, электрическими и процедурными интерфейсами, а также с физическим носителем, лежащим ниже физического уровня.  Основная задача **Канальный уровень** - быть способным передавать «сырые» данные физического уровня по надежной линии связи, свободной от необнаруженных ошибок, и маскировать реальные ошибки, так что сетевой уровень их не видит. Эта задача выполняется при помощи разбиения входных данных на кадры, обычный размер которых колеблется от нескольких сот до нескольких тысяч байт. Кадры данных передаются последовательно с обработкой кадров подтверждения, отсылаемых обратно получателем. Еще одна проблема, возникающая на уровне передачи данных (а также и на большей части более высоких уровней), - как не допустить ситуации, когда быстрый передатчик заваливает приемник данными. Может быть предусмотрен некий механизм регуляции, который информировал бы передатчик о наличии свободного места в буфере приемника на текущий момент. В широковещательных сетях существует еще одна проблема уровня передачи данных: как управлять доступом к совместно используемому каналу. Эта проблема разрешается введением специального дополнительного подуровня уровня передачи данных - подуровня доступа к носителю.  **Сетевой уровень** занимается управлением операциями подсети. Важнейшим моментом здесь является определение маршрутов пересылки пакетов от источника к пункту назначения. Маршруты могут быть жестко заданы в виде таблиц и редко меняться либо, что бывает чаще, автоматически изменяться, чтобы избегать отказавших компонентов. Кроме того, они могут задаваться в начале каждого соединения, например, терминальной сессии, такого как подключения к удаленной машине. Наконец, они могут быть в высокой степени динамическими, то есть вычисляемыми заново для каждого пакета с учетом текущей загруженности сети. Если в подсети одновременно присутствует слишком большое количество пакетов, то они могут закрыть дорогу друг другу, образуя заторы в узких местах. Недопущение подобной закупорки также является задачей сетевого уровня в соединении с более высокими уровнями, которые адаптируют загрузку. В более общем смысле, сетевой уровень занимается предоставлением определенного уровня сервиса (это касается задержек, времени передачи, вопросов синхронизации). При путешествии пакета из одной сети в другую также может возникнуть ряд проблем. Так, способ адресации, применяемый в одной сети, может отличаться от принятого в другой. Сеть может вообще отказаться принимать пакеты из-за того, что они слишком большого размера. Также могут различаться протоколы и т. д. Именно сетевой уровень должен разрешать все эти проблемы, позволяя объединять разнородные сети. В широковещательных сетях проблема маршрутизации очень проста, поэтому в них сетевой уровень очень примитивный или вообще отсутствует.  Основная функция **транспортного уровня** - принять данные от сеансового уровня, разбить их при необходимости на небольшие части, передать их сетевому уровню и гарантировать, что эти части в правильном виде прибудут по назначению. Кроме того, все это должно быть сделано эффективно и таким образом, чтобы изолировать более высокие уровни от каких-либо изменений в аппаратной технологии с течением времени. Транспортный уровень также определяет тип сервиса, предоставляемого сеансовому уровню и, в конечном счете, пользователям сети. Наиболее популярной разновидностью транспортного соединения является защищенный от ошибок канал между двумя узлами, поставляющий сообщения или байты в том порядке, в каком они были отправлены. Однако транспортный уровень может предоставлять и другие типы сервисов, например пересылку отдельных сообщений без гарантии соблюдения порядка их доставки или одновременную отправку сообщения различным адресатам по принципу широковещания. Тип сервиса определяется при установке соединения. (Строго говоря, полностью защищенный от ошибок канал создать совершенно невозможно. Говорят, лишь о таком канале, уровень ошибок в котором достаточно мал, чтобы им можно было пренебречь на практике.) Транспортный уровень является настоящим сквозным уровнем, то есть доставляющим сообщения от источника адресату. Другими словами, программа на машине-источнике поддерживает связь с подобной программой на другой машине при помощи заголовков сообщений и управляющих сообщений. На более низких уровнях для поддержки этого соединения устанавливаются соединения между всеми соседними машинами, через которые проходит маршрут сообщений.  **Сеансовый уровень** позволяет пользователям различных компьютеров устанавливать сеансы связи друг с другом. При этом предоставляются различные типы сервисов, среди которых управление диалогом (отслеживание очередности передачи данных), управление маркерами (предотвращение одновременного выполнения критичной операции несколькими системами) и синхронизация (установка служебных меток внутри длинных сообщений, позволяющих продолжить передачу с того места, на котором она оборвалась, даже после сбоя и восстановления).  В отличие от более низких уровней, задача которых - достоверная передача битов и байтов, **уровень представления** занимается по большей части синтаксисом и семантикой передаваемой информации. Чтобы было возможно общение компьютеров с различными внутренними представлениями данных, необходимо преобразовывать форматы данных друг в друга, передавая их по сети в неком стандартизированном виде. Уровень представления занимается этими преобразованиями, предоставляя возможность определения и изменения структур данных более высокого уровня (например, записей баз данных).  **Прикладной уровень** содержит набор популярных протоколов, необходимых пользователям. Одним из наиболее распространенных является протокол передачи гипертекста HTTP (HyperText Transfer Protocol), который составляет основу технологии Всемирной паутины. Когда браузер запрашивает веб-страницу, он передает ее имя (адрес) и рассчитывает на то, что сервер, на котором расположена страница, будет использовать HTTP. Сервер в ответ отсылает страницу. Другие прикладные протоколы используются для передачи файлов, электронной почты, сетевых рассылок. |
| 4 уровня TCP/IP? | | | | **TCP IP** – сетевая модель, в основе которой лежит стек протоколов, описывающих порядок передачи цифровых данных между устройствами в сети Интернет. Протоколы были разработаны еще в 70-х годах прошлого века для обеспечения обмена информацией между разными сетями. Сетевая модель TCP IP представляет собой набор правил, регламентирующих порядок передачи данных между устройствами: электронная почта, мультимедийные файлы, удаленный доступ к рабочей станции.    **Канальный уровень**  На аппаратном уровне (Link Layer) определены правила взаимодействия сетевого оборудования между собой. Для передачи той или иной информации между хостами она должна быть поделена на пакеты и передана по нужному каналу связи.  На канальном уровне сетевой модели TCP IP определены физические свойства среды обмена информацией:   * максимальное расстояние, на которое передаются пакеты; * частота сигнала; * время задержки ответа.   Наиболее часто на канальном уровне используется протокол Ethernet.  **Межсетевой уровень**  Мировая паутина состоит из множества локальных подсетей, которые объединяются между собой посредством протокола TCP IP. Для организации взаимодействия между ними и корректного предоставления информации необходимо обеспечить возможность соединяться с другими локальными сетями. В основе такой маршрутизации лежит обращение к IP с использованием маски подсети. Если передать данные нужно в пределах одной локальной сети, пакеты отправляются напрямую по IP, в этом случае использование маски не требуется.  Назначение маски подсети – помочь маршрутизатору определить, какому хосту и как передавать данные. Пакет данных может путешествовать через несколько маршрутизаторов, пока не достигнет получателя. IP может быть представлен в двух форматах: v4 и v6, которые не совместимы между собой.  v4 имеет формат из четырех блоков чисел от 0 до 255, которые разделяются точками. До 1998 года использовался только этот формат, но с ростом количества устройств в Интернете возникла необходимость большего количества уникальных адресов. v6 использует 128-битные адреса, состоящие из восьми блоков, разделяемых двоеточием, при записи адреса допускаются сокращения по определенным правилам.  Протокол lP предназначен для идентификации адресата, но он не гарантирует целостность данных. lP инкапсулирует в себе другие протоколы такие как ICMP (межсетевой протокол управляющих сообщений) и IGMP (межсетевой протокол группового управления). Первый служит для передачи сообщений об ошибках при попытке связи между разными хостами. Второй объединяет сетевые устройства в группы для передачи информации только тем компьютерам, которые ее запросили, например, в онлайн-играх или воспроизведении потокового видео.  **Транспортный уровень**  Transport Layer берет на себя функцию контроля доставки пакетов. На этом уровне работают протоколы TCP и UDP. Первый устанавливает соединение между двумя хостами и гарантирует предоставление информации в полном объеме. Если во время передачи часть информации была утеряна, протокол запрашивает ее повторно, таким образом у адресата есть полный пакет данных, собранный в нужном порядке.  Протокол UDP не устанавливает соединение между хостами, а передает автономные датаграммы. В процессе передачи часть из них может быть утеряна, проверка целостности информации не производится. UDP используется в случаях, когда требуется снизить нагрузку на сеть, а потеря какой-то доли информации не является критичной для адресата, например, при воспроизведении потокового видео.  **Прикладной уровень**  Applicatopn Layer объединяет три уровня сетевой модели OSI: сеансовый, уровень представления и прикладной. На прикладном уровне происходит поддержание сеанса связи между хостами, преобразование передаваемых данных, работа с конечным пользователем и сетью. Здесь же используются стандарты API интерфейса, которые передают команды для выполнения определенных задач.  На прикладном уровне используются производные протоколы, предназначенные для выполнения тех или иных действий. HTTPS открывает сайты в Интернете, электронная почта отправляется с использованием протокола SMTP, для динамического назначения адресов в сети применяется набор правил, определенных протоколом DHCP. |
| Чем отличается TCP и UDP? | | | | Ключевым различием между TCP и UDP является скорость и повторная отправка утерянных пакетов.  TCP медленный, но убеждается, что отправленная информация дошла до конечного получателя, если нет, то он отправляет ее повторно. Этот протокол транспортного уровня используются в почте например, т.к. там очень важно чтобы письмо дошло до адресата.  UDP быстрый, но ему без разницы дошла ли отправленная информация до конечного пользователя, он используется там, где важна скорость и не критична потеря информации. Например: видеосвязь и видеоигры, понятно почему видеосвязь всегда такая плохая? |
| Что такое QUIC? | | | | **QUIC** — новый транспортный протокол, работающий поверх UDP.  Он сочетает в себе скорость UDP (т.к. использует вместо тройного рукопожатия одинарное) и повторную отправку потерянных пакетов, как в TCP. |
| Из чего состоит http запрос? | | | | 1. Стартовая строка содержит метод, путь до ресурса, версию протокола. 2. Заголовки. 3. Blank line – пустая строка(ее обязательно ставят между заголовками и боди запроса). 4. Тело сообщения. |
| Отличие TSL и SSL? | | | | TLS (Transport Layer Security) и SSL (Secure Sockets Layer) — это криптографические протоколы, которые обеспечивают безопасную передачу данных через интернет.  Они имеют несколько различий:  -История и развитие: SSL был первоначально разработан компанией Netscape в 1990-х годах и использовался для обеспечения безопасности передачи данных в Интернете. Затем стандарт был модернизирован и стал известен как TLS. TLS представляет собой новую версию протокола SSL.  -Совместимость: TLS является более современной версией протокола и обеспечивает более сильную безопасность, чем SSL. Многие старые версии SSL считаются небезопасными и не рекомендуются к использованию.  -Версии: SSL имеет несколько версий, таких как SSL 2.0, SSL 3.0. TLS также имеет несколько версий, начиная с TLS 1.0 и до TLS 1.3 (на момент ответа). Каждая последующая версия TLS обновляла протокол, исправляла уязвимости и повышала безопасность.  -Шифрование и алгоритмы: TLS и SSL используют различные алгоритмы шифрования для обеспечения конфиденциальности данных. Некоторые старые алгоритмы, такие как MD5 и SHA-1, которые использовались в SSL, считаются устаревшими и небезопасными. В последних версиях TLS используются более современные алгоритмы шифрования, такие как AES (Advanced Encryption Standard) и SHA-256 (Secure Hash Algorithm 256-bit).  -Установка соединения: Хотя оба протокола работают похожим образом и обеспечивают безопасное соединение, TLS обычно считается более безопасным и предпочтительным в современных приложениях и системах. Многие современные браузеры и серверы предпочитают использовать TLS для обеспечения безопасной передачи данных.  В целом, TLS является обновленной и более безопасной версией протокола, разработанной на основе SSL. Он предлагает более современные алгоритмы шифрования и лучшую безопасность в сравнении с устаревшими версиями SSL. |
| Из чего состоит http-ответ и какие есть статус-коды? | | | | 1)Строка состояния – версия http протокола, статус код, статус сообщение.  2)Заголовки.  3)Blank line – пустая строка (ее обязательно ставят между заголовками и боди запроса).  4)Тело ответа. |
| Какие бывают виды заголовков? | | | | **Общие заголовки** – содержат параметры, как для запросов, так и для ответов.  Например:  -connection – параметр состояния соединения между клиентом и сервером.  -upgrade-insecure-request – клиент предпочитает зашифрованный ответ.  **Заголовки запроса** – дополнительная инфа о клиенте.  Например:  -host – доменное имя устройства.  -user agent – название браузера, его версия.  Заголовки сущности – задают характеристики контента в теле.  Например:  -content type – тип передаваемых данных. |
| Методы формирования http запроса? (9 шт.) | | | | Основными и чаще всего используемыми методами являются GET, POST, PUT, DELETE которые эквивалентны базовым функциям при работе с БД или любыми хранимыми вычислительными сущностями - [CRUD](https://ru.wikipedia.org/wiki/CRUD) (create, read, update, delete).   * **OPTIONS**: используется для определения возможностей веб-сервера или параметров соединения для конкретного ресурса. В ответ серверу следует включить заголовок Allow со списком поддерживаемых методов. Также в заголовке ответа может включаться информация о поддерживаемых расширениях. Предполагается, что запрос клиента может содержать тело сообщения для указания интересующих его сведений. Формат тела и порядок работы с ним в настоящий момент не определен; сервер пока должен его игнорировать. Аналогичная ситуация и с телом в ответе сервера. Для того, чтобы узнать возможности всего сервера, клиент должен указать в URI звёздочку - «\*». Запросы «OPTIONS \* HTTP/1.1» могут также применяться для проверки работоспособности сервера (аналогично «пингованию») и тестирования на предмет поддержки сервером протокола HTTP версии 1.1. Результат выполнения этого метода не кэшируется; * **GET**: используется для запроса содержимого указанного ресурса. С помощью метода GET можно также начать какой-либо процесс. В этом случае в тело ответного сообщения следует включить информацию о ходе выполнения процесса. Клиент может передавать параметры выполнения запроса в URI целевого ресурса после символа «?»: GET /path/resource?param1=value1&param2=value2 HTTP/1.1. Согласно стандарту HTTP, запросы типа GET считаются идемпотентными. Кроме обычного метода GET, различают ещё   + Условный GET - содержит заголовки If-Modified-Since, If-Match, If-Range и подобные;   + Частичный GET - содержит в запросе Range.   Порядок выполнения подобных запросов определен стандартами отдельно;   * **HEAD**: Аналогичен методу GET, за исключением того, что в ответе сервера отсутствует тело. Запрос HEAD обычно применяется для извлечения метаданных, проверки наличия ресурса (валидация URL) и чтобы узнать, не изменился ли он с момента последнего обращения. Заголовки ответа могут кэшироваться. При несовпадении метаданных ресурса с соответствующей информацией в кэше - копия ресурса помечается как устаревшая; * **POST**: применяется для передачи пользовательских данных заданному ресурсу. Например, в блогах посетители обычно могут вводить свои комментарии к записям в HTML-форму, после чего они передаются серверу методом POST, и он помещает их на страницу. При этом передаваемые данные (в примере с блогами - текст комментария) включаются в тело запроса. Аналогично с помощью метода POST обычно загружаются файлы на сервер. В отличие от метода GET, метод POST не считается идемпотентным, то есть многократное повторение одних и тех же запросов POST может возвращать разные результаты (например, после каждой отправки комментария будет появляться очередная копия этого комментария). При результате выполнения 200 (Ok) в тело ответа следует включить сообщение об итоге выполнения запроса. Если был создан ресурс, то серверу следует вернуть ответ 201 (Created) с указанием URI нового ресурса в заголовке Location. Сообщение ответа сервера на выполнение метода POST не кэшируется. Стоит отметить, что [не всегда данные могут быть лишь в теле](https://stackoverflow.com/questions/611906/http-post-with-url-query-parameters-good-idea-or-not); * **PUT**: применяется для загрузки содержимого запроса на указанный в запросе URI. Если по заданному URI не существует ресурса, то сервер создаёт его и возвращает статус 201 (Created). Если же ресурс был изменен, то сервер возвращает 200 (Ok) или 204 (No Content). Сервер не должен игнорировать некорректные заголовки Content-\*, передаваемые клиентом вместе с сообщением. Если какой-то из этих заголовков не может быть распознан или недопустим при текущих условиях, то необходимо вернуть код ошибки 501 (Not Implemented). Фундаментальное различие методов POST и PUT заключается в понимании предназначений URI ресурсов. Метод POST предполагает, что по указанному URI будет производиться обработка передаваемого клиентом содержимого. Используя PUT, клиент предполагает, что загружаемое содержимое соответствует находящемуся по данному URI ресурсу. Сообщения ответов сервера на метод PUT не кэшируются; * **PATCH**: аналогично PUT, но применяется только к фрагменту ресурса; * **DELETE**: удаляет указанный ресурс; * **TRACE**: возвращает полученный запрос так, что клиент может увидеть, какую информацию промежуточные серверы добавляют или изменяют в запросе; * **CONNECT**: преобразует соединение запроса в прозрачный TCP/IP-туннель, обычно чтобы содействовать установлению защищенного SSL-соединения через нешифрованный прокси. |
| Что такое URL, URN, URI? | | | | Как в интернетах ваших:   * URI - имя и адрес ресурса в сети, включает в себя URL и URN; * URL - адрес ресурса в сети, определяет местонахождение и способ обращения к нему; * URN - имя ресурса в сети, определяет только название ресурса, но не говорит как к нему подключиться.   Пример:   * URI - <https://wiki.merionet.ru/images/vse-chto-vam-nuzhno-znat-pro-devops/1.png> * URL - <https://wiki.merionet.ru> * URN - images/vse-chto-vam-nuzhno-znat-pro-devops/1.png |
| URI может состоять из 8 частей? | | | | **Схема** (scheme) — протокол, по которому передаются данные. Например: HTTP и HTTPS.  **Логин: пароль** (username: password) указывают серверу, какой пользователь к нему обратился. К логину пользователя привязаны права доступа.  **Символ @** (at коммерческое, «собака») отделяет логин: пароль и имя хоста: порт (hostname: port). Если логин: пароль не применяют, «собаку» — тоже.  **Имя хоста: порт** (hostname:port) — доменное имя или IP-адрес сервера, к которому обращается пользователь.  **Путь (path)** — месторасположение ресурса. Похоже на путь к файлу или папке в файловой системе.  Например: /home/index.html или /search/.  /home/index.html — указание на главную страницу веб-приложения.  /search/ — указание на папку для поиска.  Слэш ("/") — разделитель пути до ресурса.  **Параметры запроса** (query) — дополнительные параметры вида ключ=значение. Перечисляются после знака ? и разделяются через &.  **Якорь** — это «закладка» на определенном месте веб-страницы. При клике на связанной с ним ссылке браузер автоматически прокручивает страницу до обозначенного якорем места. |
| Что такое IP? | | | | Каждое устройство в сети имеет свой уникальный IP-адрес (3 уровень модели OSI). Он нужен для того, чтобы устройства сети понимали куда необходимо направить запрос и ответ.  **IP-адрес** — это уникальный идентификатор сервера, на котором находится нужная информация. |
| Что такое белый и серый IP? | | | | локальные ip-адреса — это серые ip-адреса, а глобальные — это белые.  Локальные нам выдает наш роутер (с этим айпи нельзя выйти в глобальную сеть), а глобальные нам выдает наш провайдер и он используется для выхода в интернет. |
| Что такое маска-подсети? | | | | **Маска подсети** — это не IP-адрес, а инструкция, которая говорит о том, как следует читать IP-адрес.  **Маска подсети нужна для того, чтобы разделить один большой адрес на несколько маленьких.** |
| Что такое мак-адрес? | | | | **MAC-адрес** является уникальным цифровым идентификатором устройства, которое имеет выход в сеть. Его присваивают сетевой карте в ходе изготовления, поэтому это адрес карты, а не всего компьютера. |
| Сколько таких адресов в системе? | | | | Кол-во мак адресов равно кол-ву сетевых карт, которые установлены в компьютер. |
| Что такое DNS? | | | | **DNS** — это технология, которая позволяет браузеру вроде Firefox, Chrome или Edge найти запрошенный пользователем сайт по его имени.  **Как работает DNS?**  Принцип работы DNS похож на поиск и вызов контактов из телефонной книги смартфона. Ищем имя, нажимаем «позвонить», и телефон соединяет нас с нужным абонентом. Понятно, что смартфон в ходе звонка не использует само имя человека, вызов возможен только по номеру телефона. Если вы внесете имя без номера телефона, позвонить человеку не сможете.  Так и с сайтом. Каждому имени сайта соответствует набор цифр формата ХХХ.ХХХ.ХХХ.ХХХ. Этот набор называется IP-адресом, примером реального IP-адреса является 192.168.0.154 или 203.113.89.134. Когда пользователь вводит в адресной строке браузера имя сайта, например google.com, компьютер запрашивает IP-адрес этого сайта на специальном DNS-сервере и после получения ip адреса компьютера на котором находится сервер гугла, он обращается к нему и после ответа сервера гугла открывает сам сайт. |
| Что такое Кэш? | | | | **Кэш** — это данные, которые компьютер уже получил и использовал один раз, а потом сохранил на будущее. Смысл кэша в том, чтобы в следующий раз взять данные не с далёкого и медленного сервера, а из собственного быстрого кэша. То же самое, что закупиться продуктами на неделю и потом ходить не в магазин, а в холодильник. |
| Что такое куки и зачем они нужны? | | | | Куки – данные, которые хранятся на устройстве пользователя. Эти данные приходят с сервера и сохраняются на устройстве пользователя, а когда пользователь использует приложение они отправляются на сервер (Например: данные для аутентификации).  Для поддержки авторизованного (не анонимного) доступа в HTTP используются cookies; причем такой способ авторизации позволяет сохранить сессию даже после перезагрузки клиента и сервера. Смысл в том, что после каждого ответа сервера сессия закрывается т.к. http не хранит промежуточные данные между запросами и если бы данные для авторизации не хранились в куки нам бы пришлось после каждого запроса повторно авторизироваться. |
| Что хранится в куках 3 пункта? | | | | В куках хранятся:  -Данные для аутентификации.  -Данные о пользователе напр. Язык, валюта, что он смотрел?  -Данные для сбора статистики. |
| Что такое local storage? | | | | **Local storage** – хранилище данных, которое встроено в браузер. Оно отличается от кук тем, что данные из local storage не передаются на сервер, их использует клиентская часть приложения.  Например:  Ты заполнил форму бронирования отеля и при повторном заполнении данных поля предзаполнятся т.к. подтянуться данные из local storage. |
| Виды рендеринга страницы? | | | | **Рендиринг бывает:**   **SSR**: рендеринг на стороне сервера - рендеринг клиентского или универсального приложения в HTML на сервере;   CSR: рендеринг на стороне клиента - рендеринг приложения в браузере, обычно с использованием DOM; |
| Разделы в devtools? | | | | **Chrome DevTools** — это набор инструментов, встроенных в браузер Google Chrome, для создания и отладки сайтов. С их помощью можно просматривать исходный код сайта, отлаживать работу frontend: HTML, CSS и JavaScript. Также DevTools позволяет проверять сетевой трафик, быстродействие сайта и многое другое.  **Какие вкладки есть в DevTools:**  **Elements.** Здесь отображается весь HTML- и CSS-код открытой страницы. На данной вкладке можно просмотреть и внести исправления в файлы [CSS](https://blog.skillfactory.ru/glossary/css/) и [JavaScript](https://blog.skillfactory.ru/glossary/javascript/), изменить [элементы DOM](https://blog.skillfactory.ru/glossary/dom/)(программного интерфейса (API) для HTML- и XML-документов). Отредактировать HTML-элементы на странице, открытой в браузере, можно, кликнув по нужному элементу правой кнопкой мыши и выбрав пункт Edit as HTML. Изменения можно наблюдать в режиме реального времени. Манипуляции отображаются только в браузере и не видны другим пользователям. Для того чтобы применить исправленное, необходимо поработать с соответствующими файлами на веб-сервере.  **Console.** Консоль позволяет смотреть вывод JavaScript, а также исполнять свой код для тестирования и отладки страницы. Если на открытой странице не подгрузились какие-либо данные, например стили, шрифты или картинки, здесь отобразятся соответствующие ошибки с подробным описанием. Также в консоль можно ввести команду на языке JavaScript, и она выполнится. Вкладка Console отображает все ошибки при загрузке страницы  **Sources.** Вкладка отображает загруженные файлы из всех источников, к которым обращался сайт. В большей степени она используется при отладке кода, позволяет увидеть все файлы и просмотреть их содержимое. Sources можно использовать в качестве полноценного редактора кода, получив доступ к локальным файлам через Workspaces.  **Network.** На вкладке отображаются сетевые запросы, который делает сайт. Как правило, ее используют при оптимизации скорости загрузки страницы, а также для мониторинга выполняемых запросов. Запросы к данным представлены в виде таблицы. Сверху расположены инструменты: очистка таблицы, включение и отключение записи запросов и другие. Под таблицей можно увидеть количество запросов, общее время загрузки всех данных, время загрузки DOM и ресурсов, участвующих в отображении текущей страницы.  **Performances.** Вкладка отображает нагрузку, которую создает сайт на компьютер пользователя. Здесь можно увидеть показатели FPS, загрузки CPU и сетевые запросы, необходимые данные и инструменты для повышения производительности страницы. На панели есть таймлайн использования сети, выполнения JavaScript и загрузки памяти. После первого построения таймлайнов можно найти данные о всем жизненном цикле страницы и выполнении кода. Также можно посмотреть время исполнения отдельных частей кода и выбрать конкретный период на шкале, чтобы увидеть, какие процессы происходили в этот интервал. Все это позволяет проанализировать каждое событие, которое происходило в момент загрузки или во время взаимодействия с пользователем.  **Memory.** Здесь расположено несколько инструментов, которые помогают отслеживать, какую нагрузку на систему оказывает выполнение кода:  -Heap Snapshot. С помощью него можно посмотреть, как распределяется память между объектами JavaScript и связанными с ними элементами DOM.  -Allocation instrumentation on timeline. Этот инструмент используется для устранения утечек памяти. Он показывает, как распределяется память между переменными в коде.  -Allocation sampling. Профайлер записывает, как распределяется память на отдельные функции JavaScript.  **Application.** Панель, где можно быстро очистить хранилище и кэш, а также управлять базами данных.  **Security.** Отвечает за надежность ресурса. Здесь можно получить информацию о данных протокола и сертификата безопасности, если они есть. Также, если источник небезопасный, узнать, какие именно запросы не защищены. Поэтому этот инструмент, как правило, используется для решения проблем со смешанным контентом и другими подобными задачами.  **Lighthouse.**На этой вкладке можно проверить производительность сайта.  -Performance. Позволяет узнать скорость загрузки сайта. Итоговый показатель зависит от времени загрузки интерактивных элементов, шрифтов и прочего контента, а также от времени блокировки и отрисовки стилей.  -Progressive Web App. Позволяет проверить, регистрирует ли сайт Service Workers, возможна ли работа сайта офлайн, а также возвращает ошибку 200.  Best Practices. Помогает проверить безопасность сайта и узнать, применяются ли современные стандарты веб-разработки. На показатель влияет использование устаревших API, HTTPS, корректность кодировки и многое другое.  -Accessibility. Позволяет узнать, насколько удобен сайт, как воспринимается контент и можно ли управлять интерфейсом и передвигаться по сайту без мыши.  SEO. Позволяет понять, насколько соблюдаются рекомендации Google по оптимизации сайта. На показатель влияют использование метатегов, наличие alt у изображений, адаптивная верстка и пр. |
| Что такое HTML? | | | | **HTML** (от английского HyperText Markup Language) — это язык гипертекстовой разметки текста. Он нужен, чтобы размещать на веб-странице элементы: текст, картинки, таблицы и видео. |
| Два подхода к верстке? | | | | **Адаптивный** – поле при уменьшении экрана в один момент “ломается ” и меняется под размер устройства.  **Респонсивный** – поле медленно уменьшается вместе с разрешением. |
| Что такое боекпойнт в адаптивной верстке? | | | | **Брекпойнт** – точка, при которой ломается дизайн и происходит переход под новое разрешение. |
| Что такое адаптивность? | | | | **Адаптивность** – это способность приложения адаптироваться под экраны с разным разрешением. |
| Что такое CSS? | | | | **CSS (Cascade Style Sheets)** — это формальный язык описания внешнего вида страницы; каскадные таблицы стилей. |
| 2 типа валидации на веб форме? | | | | Валидация происходит на стороне фронта и на стороне бэка. Валидация должна присутствовать и на бэка и на фронте, если ее нет, то это баг. |
| Какие существуют UI-стандарты? | | | | Существует множество UI-стандартов, некоторые из которых являются общепризнанными и широко используются в индустрии разработки ПО. Ниже перечислены некоторые из них:  1) Material Design - стандарт дизайна, разработанный компанией Google. Включает в себя рекомендации по использованию цветовой палитры, шрифтов, иконок и анимаций. 2) Apple Human Interface Guidelines - руководство от Apple, в котором описываются стандарты дизайна для приложений под iOS, macOS и watchOS. 3) Microsoft Design Language - стандарт дизайна, используемый в операционных системах и приложениях от Microsoft. Он включает в себя рекомендации по использованию типографики, цвета, анимации и иконок. 4) Bootstrap - фреймворк, который содержит рекомендации по использованию готовых компонентов интерфейса, таких как кнопки, формы, таблицы и другие элементы. 5 ) Materialize - фреймворк, основанный на Material Design, который содержит готовые компоненты интерфейса, такие как кнопки, навигационные панели, формы и др. Кроме этого, многие компании имеют свои собственные UI-стандарты и руководства для разработки интерфейсов своих продуктов. |
| Что такое SOAP? | | | | **SOAP** — это протокол, по которому веб-сервисы взаимодействуют друг с другом или с клиентами. Название происходит от сокращения Simple Object Access Protocol («простой протокол доступа к объектам»). SOAP API — это веб-сервис, использующий протокол SOAP для обмена сообщениями между серверами и клиентами. При этом сообщения должны быть написаны на языке [XML](https://blog.skillfactory.ru/glossary/xml/) в соответствии со строгими стандартами, иначе сервер вернет ошибку.  Особенности SOAP:  - Он безопасный.  - SOAP более применим в сложных архитектурах, где взаимодействие с объектами выходит за рамки теории CRUD. |
| Что такое XML и какие есть правила написания? | | **XML** — расширяемый язык разметки. Используется для хранения и передачи данных. Так что увидеть его можно не только в API, но и в коде.  **Правила написания:**  - Все XML элементы должны иметь закрывающий тег;  - Теги XML регистрозависимы;  - XML элементы должны соблюдать корректную вложенность;  - У XML документа должен быть корневой элемент;  - XML пролог;  - Значения XML атрибутов должны заключаться в кавычки; | | |
| Что такое REST? | [**REST**](https://ru.wikipedia.org/wiki/REST)**(REpresentational State Transfer)** архитектурный стиль взаимодействия компонентов распределенного приложения в сети. Другими словами, REST — это набор правил того, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределенной гипермедиа-системы. В определенных случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) это приводит к повышению производительности и упрощению архитектуры. В широком смысле компоненты в REST взаимодействуют наподобие взаимодействия клиентов и серверов во Всемирной паутине. REST является альтернативой RPC.  Автор идеи и термина [Рой Филдинг](https://en.wikipedia.org/wiki/Roy_Fielding) 2000г.  REST на сегодняшний день практически вытеснил все остальные подходы, в том числе дизайн основанный на [SOAP](https://ru.wikipedia.org/wiki/SOAP) и [WSDL](https://ru.wikipedia.org/wiki/WSDL).  **Что нам дает REST подход?**  -Масштабируемости взаимодействия компонентов системы (приложения)  -Общность интерфейсов  -Независимое внедрение компонентов  -Промежуточные компоненты, снижающие задержку, усиливающие безопасность  **Преимущества REST:**  -Отсутствие дополнительных внутренних прослоек, что означает передачу данных в том же виде, что и сами данные. Т.е. данные не оборачиваются в [XML](https://ru.wikipedia.org/wiki/XML), как это делает [SOAP](https://ru.wikipedia.org/wiki/SOAP) и [XML-RPC](https://ru.wikipedia.org/wiki/XML-RPC), не используется [AMF](https://ru.wikipedia.org/wiki/Action_Message_Format), как это делает Flash и т.д. Просто отдаются сами данные.  -Каждая единица информации (ресурс) однозначно определяется URL — это значит, что URL по сути является первичным ключом для единицы данных. Причем совершенно не имеет значения, в каком формате находятся данные по адресу — это может быть и HTML, и jpeg, и документ Microsoft Word.  -Как происходит управление информацией ресурса — это целиком и полностью основывается на протоколе передачи данных. Наиболее распространенный протокол конечно же [HTTP](https://ru.wikipedia.org/wiki/HTTP). Для HTTP действие над данными задается с помощью методов: GET (получить), PUT (добавить, заменить), POST (добавить, изменить, удалить), DELETE (удалить). Таким образом, действия [CRUD](https://ru.wikipedia.org/wiki/CRUD) (Create-Read-Update-Delete) могут выполняться как со всеми 4-мя методами, так и только с помощью GET и POST. | | | |
| Шесть правил REST? | **Чтобы распределенная система считалась сконструированной по REST архитектуре (Restful), необходимо, чтобы она удовлетворяла следующим критериям:**  -Client-Server. Система должна быть разделена на клиентов и на серверов. Разделение интерфейсов означает, что, например, клиенты не связаны с хранением данных, которое остается внутри каждого сервера, так что мобильность кода клиента улучшается. Серверы не связаны с интерфейсом пользователя или состоянием, так что серверы могут быть проще и масштабируемы. Серверы и клиенты могут быть заменяемы и разрабатываться независимо, пока интерфейс не изменяется.  -Stateless. Сервер не должен хранить какой-либо информации о клиентах. В запросе должна храниться вся необходимая информация для обработки запроса и, если необходимо, идентификации клиента.  -Cache․ Каждый ответ должен быть отмечен является ли он кэшируемым или нет, для предотвращения повторного использования клиентами устаревших или некорректных данных в ответ на дальнейшие запросы.  -Uniform Interface. Единый интерфейс определяет интерфейс между клиентами и серверами. Это упрощает и отделяет архитектуру, которая позволяет каждой части развиваться самостоятельно.  **Четыре принципа единого интерфейса:**  1)Identification of resources (основан на ресурсах). В REST ресурсом является все то, чему можно дать имя. Например, пользователь, изображение, предмет (майка, голодная собака, текущая погода) и т.д. Каждый ресурс в REST должен быть идентифицирован посредством стабильного идентификатора, который не меняется при изменении состояния ресурса. Идентификатором в REST является URI.  2)Manipulation of resources through representations. (Манипуляции над ресурсами через представления). Представление в REST используется для выполнения действий над ресурсами. Представление ресурса представляет собой текущее или желаемое состояние ресурса. Например, если ресурсом является пользователь, то представлением может являться XML или HTML описание этого пользователя.  3)Self-descriptive messages (само-документируемые сообщения). Под само-описательностью имеется ввиду, что запрос и ответ должны хранить в себе всю необходимую информацию для их обработки. Не должны быть дополнительные сообщения или кэши для обработки одного запроса. Другими словами, отсутствие состояния, сохраняемого между запросами к ресурсам. Это очень важно для масштабирования системы.  4)HATEOAS (hypermedia as the engine of application state). Статус ресурса передается через содержимое body, параметры строки запроса, заголовки запросов и запрашиваемый URI (имя ресурса). Это называется гипермедиа (или гиперссылки с гипертекстом). HATEOAS также означает, что, в случае необходимости ссылки могут содержатся в теле ответа (или заголовках) для поддержки URI , извлечения самого объекта или запрошенных объектов.  -Layered System. В REST допускается разделить систему на иерархию слоев, но с условием, что каждый компонент может видеть компоненты только непосредственно следующего слоя. Например, если вы вызывайте службу PayPal а он в свою очередь вызывает службу Visa, вы о вызове службы Visa ничего не должны знать.  -Code-On-Demand (опционально). В REST позволяется загрузка и выполнение кода или программы на стороне клиента.  Серверы могут временно расширять или кастомизировать функционал клиента, передавая ему логику, которую он может исполнять. Например, это могут быть скомпилированные Java-апплеты или клиентские скрипты на Javascript.  Важно! Сама архитектура REST не привязана к конкретным технологиям и протоколам, но в реалиях современного Веб, построение RESTful API почти всегда подразумевает использование HTTP и каких-либо распространенных форматов представления ресурсов, например JSON, или, менее популярного сегодня, XML. | | | |
| Отличия REST и SOAP? | **Отличия REST и SOAP:**  **Коротко:**  SOAP более применим в сложных архитектурах, где взаимодействие с объектами выходит за рамки теории CRUD, а вот те приложения, которые не покидают рамки данной теории, вполне приемлемо может оказаться именно REST из-за своей простоты и прозрачности. Кроме того, REST более производительный, так как не требует затрат на разбор сложных ХМЛ команд. SOAP в свою очередь более надежный и безопасный.  **Развернуто:**  SOAP и REST нельзя сравнивать напрямую, поскольку первый — это протокол (или, по крайней мере, пытается им быть), а второй - архитектурный стиль.  Основное различие между SOAP и REST заключается в степени связи между реализациями клиента и сервера. Клиент SOAP работает как пользовательское настольное приложение, тесно связанное с сервером. Между клиентом и сервером существует жесткое соглашение, и ожидается, что все сломается, если какая-либо из сторон что-то изменит. Вам нужно постоянное обновление после любого изменения, но легче определить, выполняется ли контракт.  SOAP более применим в сложных архитектурах, где взаимодействие с объектами выходит за рамки теории CRUD, а вот в тех приложениях, которые не покидают рамки данной теории, вполне применимым может оказаться именно REST ввиду своей простоты и прозрачности. Действительно, если любым объектам вашего сервиса не нужны более сложные взаимоотношения, кроме: «Создать», «Прочитать», «Изменить», «Удалить» (как правило - в 99% случаев этого достаточно), возможно, именно REST станет правильным выбором. Кроме того, REST по сравнению с SOAP, может оказаться и более производительным, так как не требует затрат на разбор сложных XML команд на сервере (выполняются обычные HTTP запросы - PUT, GET, POST, DELETE). Хотя SOAP, в свою очередь, более надежен и безопасен. | | | |
| Почему json лучше? | **Преимущества JSON:**  -Меньше слов — больше дела XML требует открытия и закрытия тегов, а JSON использует пары имя / значение, четко обозначенные «{«и»}» для объектов, «[«и»]» для массивов, «,» (запятую) для разделения пары и «:»(двоеточие) для отделения имени от значения.  -Размер имеет значение При одинаковом объеме информации JSON почти всегда значительно меньше, что приводит к более быстрой передаче и обработке.  -Близость к javascript JSON является подмножеством JavaScript, поэтому код для его анализа и упаковки вполне естественно вписывается в код JavaScript. | | | |
| В чем отличия GET и POST? | **Различия методов GET и POST**  Основное состоит в способе передачи данных веб-формы обрабатывающему скрипту, а именно:   * Метод GET отправляет скрипту всю собранную информацию формы как часть URL: http://www.komtet.ru/script.php?login=admin&name=komtet * Метод POST передает данные таким образом, что пользователь сайта уже не видит передаваемые скрипту данные: http://www.komtet.ru/script.php   Кроме того:   * Количество информации, передаваемой методом GET через URL строку ограничено 2048 символами (минус служебная информация браузера); * Страницу, сгенерированную методом GET, можно добавить в закладки и поделиться ссылкой; * Sensitive data в таком открытом виде очевидно плохо влияют на безопасность; * Метод POST в отличие от метода GET позволяет передавать запросу файлы; * При использовании метода GET существует риск того, что поисковый робот может выполнить тот или иной открытый запрос. | | | |
| В чем отличия PUT и POST? | 1) Метод PUT должен быть идемпотентным, то есть несколько одинаковых PUT на один endpoint не меняют состояния API. POST не обязан быть идемпотентным.  2) POST это отправка новых данных на сервер. PUT вносит изменения в уже имеющуюся на сервере информацию.  Сам вопрос супернеоднозначный, лучше отвечать про идемпотентность или говорить, что зависит от реализации на конкретном проекте. | | | |
| Что такое безопасный метод? | Это означает, что они предназначены только для получения информации и не должны изменять состояние сервера. Они не должны иметь побочных эффектов, за исключением безобидных эффектов, таких как: логирование, кеширование, показ баннерной рекламы или увеличение веб-счетчика. | | | |
| Какие методы безопасные? | **Методы GET, HEAD, OPTIONS и TRACE.** | | | |
| Что такое идемпотентность, какие методы идемпотентны? | С точки зрения RESTful-сервиса, операция (или вызов сервиса) **идемпотентна** тогда, когда клиенты могут делать один и тот же вызов неоднократно при одном и том же результате на сервере.  Другими словами, создание большого количества идентичных запросов имеет такой же эффект, как и один запрос.  Заметьте, что в то время, как идемпотентные операции производят один и тот же результат на сервере, ответ сам по себе может не быть тем же самым (например, состояние ресурса может измениться между запросами).  Методы GET, PUT и DELETE по определению идемпотентны. Тем не менее, есть один нюанс с методом DELETE. Проблема в том, что успешный DELETE-запрос возвращает статус 200 (OK) или 204 (No Content), но для последующих запросов будет все время возвращать 404 (Not Found), Состояние на сервере после каждого вызова DELETE то же самое, но ответы разные. | | | |
| Что такое логи? | **Логи (лог-файлы)** — это файлы, содержащие системную информацию работы сервера или компьютера, в которые заносятся определенные действия пользователя или программы. | | | |
| Какие бывают логи? | **Типы логов:**  -Серверные — обращения к серверу и ошибки, которые возникают во время обращений;  -Системные — все системные события;  -Логи авторизации и аутентификации — процессы входа в систему и выхода из нее, проблемы с доступом и другие;  -Логи приложений, которые находятся в этой системе;  -Логи баз данных — обращения к БД. | | | |
| Уровни логирования? | Уровни логирования  В идеале логи пишутся во время работы всех IT-систем, однако если писать все подряд и «складывать в кучу», полезная информация превратится в хаос.    Чтобы упростить поиск и чтение логов, их делят на уровни. Основных четыре:  Debug — запись масштабных переходов состояний, например, обращение к базе данных, старт/пауза сервиса, успешная обработка записи и пр.  Warning — нештатная ситуация, потенциальная проблема, может быть странный формат запроса или некорректный параметр вызова.  Error — типичная ошибка.  Fatal — тотальный сбой работоспособности, когда нет доступа к базе данных или сети, сервису не хватает места на жестком диске.  Дополнительно файл логирования может расширяться записями еще двух уровней:  Trace — пошаговые записи процесса. Полезен, когда сложно локализовать ошибку.  Info — общая информация о работе службы или сервиса. | | | |
| Виды интерфейсов? | **Виды:**  -GUI – графический, аудио, текстовый.  -API – взаимодействие между двумя сервиса.  -CLI – командная строка. | | | |
| Типы API? | **Типы API:**  -RPC (Remote Procedure Call) – удаленный вызов процедур,  -SOAP (Simple Object Access Protocol) – простой протокол доступа к объектам,  -REST (Representational State Transfer) – передача состояния представления. | | | |
| Как работает API? | API (Application programming interface) — это контракт, который предоставляет программа. «Ко мне можно обращаться так и так, я обязуюсь делать то и это».  Если переводить на русский, это было бы слово «договор». Договор между двумя сторонами, как договор на покупку машины:  -мои обязанности — внести такую-то сумму,  -обязанность продавца — дать машину.  Перевести можно, да. Но никто так не делает ¯\\_(ツ)\_/¯  По сути, api это мост между двумя разными сервисами по которому они соединяются и обмениваются информацией. | | | |
| Способ вызова API? | Три основных метода вызова API:  -XMLHttpReques;  -Fetch;  -Axios***.*** | | | |
| Что такое микросервисная архитектура? | **Микросервисы** – принцип организации сиcтемы на основе микросервисов и их взаимодействия друг с другом.  **Характеристики микросервиса:**  1)Он небольшой.  2)Он независимый.  3)Он строится вокруг бизнес-потребности и использует ограниченный контекст (Bounded Context).  4)Он взаимодействует с другими микросервисами по сети на основе паттерна Smart endpoints and dumb pipes.  5)Его распределенная суть обязывает использовать подход Design for failure.  6)Централизация ограничена сверху на минимуме.  7)Процессы его разработки и поддержки требуют автоматизации.  8)Его развитие итерационное.  **Плюсы:**  1)Отказоустойчивость т.е. если падает микросервис все приложение продолжает работать.  2)Независимость компонентов и независимость команд разработки таким образом каждая команда отвечает за определенную бизнес-логику.  3)Возможность использовать разные ЯП и технологии.  4)Масштабируемость т.е. для изменения вам не придется переделывать всю систему полностью.  5)Микросервисы можно перепрофилировать для других задач после начального запуска, что обеспечит повторное использование.  **Минусы:**  1)Сложнее осуществлять мониторинг т.е. следить за 1 монолитом проще чем за 1000 микросервисов.  2)Сложно поддерживать зоопарк технологий это вытекает из плюса выше.  3) Снижение доверия. Когда у нас сотни узлов, могут возникать проблемы с аутентификацией и авторизацией, поскольку есть вероятность подсоединения мошенников.  4)Сложности развёртывания. Чтобы требования по отказоустойчивости выполнялись, микросервисы нужно развёртывать на отдельных серверах. И здесь не работает подход: «Берём приложение, ставим и запускаем». Нужны системы оркестрации и деплоймента. | | | |
| Монолитная архитектура? | **Монолит** — это традиционная модель программного обеспечения, которая представляет собой единый модуль, работающий автономно и независимо от других приложений.  **Плюсы:**  1)Простое развертывание. Использование одного исполняемого файла или каталога упрощает развертывание.  2)Разработка. Приложение легче разрабатывать, когда оно создано с использованием одной базы кода.  2)Производительность. В централизованной базе кода и репозитории один интерфейс API часто может выполнять ту функцию, которую при работе с микросервисами выполняют многочисленные API.  3)Упрощенное тестирование. Монолитное приложение представляет собой единый централизованный модуль, поэтому сквозное тестирование можно проводить быстрее, чем при использовании распределенного приложения.  4)Удобная отладка. Весь код находится в одном месте, благодаря чему становится легче выполнять запросы и находить проблемы.  **Минусы:**  1)Снижение скорости разработки. Большое монолитное приложение усложняет и замедляет разработку.  2)Масштабируемость. Невозможно масштабировать отдельные компоненты.  3)Надежность. Ошибка в одном модуле может повлиять на доступность всего приложения.  4)Препятствия для внедрения технологий. Любые изменения в инфраструктуре или языке разработки влияют на приложение целиком, что зачастую приводит к увеличению стоимости и временных затрат.  5)Недостаточная гибкость. Возможности монолитных приложений ограничены используемыми технологиями.  6)Развертывание. При внесении небольшого изменения потребуется повторное развертывание всего монолитного приложения. | | | |
| Что такое БД? | **Базы данных** – это хранилище информации, которые управляются системами управления базами данных. | | | |
| Виды БД: | * 1. **Иерархическая БД** – это БД, в которых организованы в древовидную структуру (напр. Файловая система ПК).   2. **Сетевые** БД – это БД связи, в которых представлены графами и узлами (напр. Система бронирования авиабилетов).   3. **Объектно-ориентированные БД** – это БД, в которых данные представлены в виде объектов с атрибутами и методами (напр. Система управления контентом).   4. **Реляционные БД** – это БД , данные в которых организованны в таблицы со строками и столбцами.   **NoSQL БД** – это БД которые не используют табличную модель хранения, а используют например json или xml форматы. | | | |
| Реляционная БД состоит из? | * + 1. **Таблицы** – это совокупность строк и столбцов.     2. **Столбцы таблицы** – это ее поля.     3. **Строки таблицы** – это ее записи.     4. **Ячейка** – место пересечения строки и столбца.     5. **Первичный ключ** – это уникальный идентификатор записи в таблице.   **Первичный составной ключ** - если первичный ключ состоит из двух и более атрибутов, его называют составным ключом. Так, номер паспорта и серия паспорта не могут быть первичными ключами по отдельности, так как могут оказаться одинаковыми у двух и более людей. Но не бывает двух личных документов одного типа с одинаковыми серией и номером. | | | |
| Что такое СУБД? | **СУБД** — комплекс программ, позволяющих создать базу данных (БД) и манипулировать данными (вставлять, обновлять, удалять и выбирать). | | | |
| Оператор SELECT? | **SELECT** – указывает какой столбец нужно выбрать, используем ‘ \* ’, если нужно выбрать все столбцы. | | | |
| Оператор FROM? | **FROM** – указывает на таблицу с которой будем работать. | | | |
| Условие WHERE? | **WHERE** – проверяет, соответствует ли каждая строка таблицы условию и выбирает подходящую. | | | |
| Оператор BETWEEN? | **BETWEEN** – указывает, что нужно выбрать между двумя значениями, которые указываются через AND. | | | |
| Функция COUNT()? | **COUNT** – возвращает количество строк найденных в таблице. | | | |
| Функция DISTINCT? | | | **DISTINCT** – возвращает уникальные строки в таблице. | |
| Функция SUM? | **SUM** – возвращает сумму столбца. | | | |
| Функция AVG? | **AVG** – возвращает среднее значение столбца. | | | |
| Функция MIN? | **MIN** – возвращает минимальное значение из столбца. | | | |
| Функция MAX? | **MAX** – возвращает максимальное значение из столбца. | | | |
| Конструкция CATS? | **CAST** – преобразует типы данных столбцов. | | | |
| Функция GROUP\_BY? | **GROUP BY** – группирует данные в указанном поле. | | | |
| Функция ORDER BY? | **ORDER BY** – сортирует данные в порядке возрастания или убывания | | | |
| JOIN? | **Что такое SQL Join?**  SQL Join – одна из наиболее часто используемых команд в SQL-синтаксисе. Она используется для поиска информации в базах данных по заранее определенным критериям. В частности, Join отвечает за объединение нескольких групп данных в единый поток информации. И это действительно необходимо, потому что в 100% случаев контент в реляционных базах данных с поддержкой SQL-синтаксиса делится на множество таблиц, фильтровать данные, в которых можно с помощью специальных команд и запросом информации из общего пула таблиц. SQL Join помогает настроить фильтр поиска в базе данных, опираясь на взаимосвязи между различными элементами БД и их отличительные черты (теги, ID, наименования и т.п.).  **SQL Inner Join**  Этот режим объединения результатов поиска в базах данных SQL включается автоматически. Если вы не укажете намеренно тип Join, то сработает именно Inner Join. С помощью него можно указать сразу два критерия (две таблицы) и по ним отсеять контент.  Достаточно прописать SQL-запрос в духе:  SELECT \*  FROM table-1  JOIN table-2 ON table-1.parameter=table-2.parameter  WHERE table-1.parameter IS ‘myData’  Фактически мы пытаемся выудить данные из первой таблицы и объединить их с данными из второй таблицы, при этом фильтруя только те записи, в которых совпадает значение параметра. В первой таблице оно приравнивается к myData.  На практике это может использоваться на сайте с музыкальными инструментами, например. Можно запрашивать гитары конкретного бренда, при этом еще и выбирая дополнительное условие в духе количества струн.  SELECT \*  FROM SevenStringGuitars  JOIN Ibanez ON SevenStringGuitar.brandId=Ibanez.brandId  Таким SQL-запросом мы можем отфильтровать все инструменты бренда Ibanez в категории «Гитары» с 7 струнами.  **Left**  Левое объединение подразумевает как раз вышеописанный сценарий. Когда мы берем одну таблицу, подключаем вторую и при этом показываем не только точные совпадения, но еще и весь список строк, полученных из левой таблицы, для которых не нашлось пары в правой таблице.  На практике это может выглядеть так:  SELECT \*  FROM table1  LEFT JOIN table2 ON table1.parameter=table2.parameter  Теперь мы объединяем первую и вторую таблицу, доставая информацию как о совпадениях по заданным параметрам, так и по контенту без пары в левой таблице. При желании, надстраивая подобный фильтр, можно вовсе исключить целую категорию строк:  SELECT \*  FROM table1  LEFT JOIN table2 ON table1.parameter=table2.parameter  WHERE table2.parameter IS NULL  На живом примере фильтрация такого рода может выглядеть так:  SELECT \*  FROM Russian  LEFT JOIN Rap ON Russian.genreId=Rap.genreId  Представим, что мы запустили продвинутый поиск на сайте с музыкальными альбомами. Мы хотим послушать что-то на русском языке. Причем готовы даже оценить качество отечественного рэпа. При этом в целом мы рэп не любим и не хотим, чтобы он попадался на каких-то других языках.  **Right**  Понятно, что правое объединение будет работать в обратную сторону и покажет элементы из правой таблицы, для которых не нашлось пары в левой.  Получится следующий SQL-запрос:  SELECT \*  FROM table1  RIGHT JOIN table2 ON table1.parameter=table2.parameter  Если взять пример из предыдущей главы, то в реальности можно обернуть ситуацию в противоположную сторону. Искать только рэп-музыку, исключив все русское, кроме хип-хопа. Получится что-то в духе:  SELECT \*  FROM Russian  RIGHT JOIN Rap ON Russian.genreId=Rap.genreId | | | |
| Что такое дамп? | **Дамп базы** – это файл с набор SQL команд, который позволяет создать БД и таблицы и восстановить информацию. Это используется, например, чтобы воссоздать БД, как на проде только на тестовом стенде, чтобы протестировать на настоящих данных. | | | |
| Как работает браузер? | **Как работает браузер?**   1. **Пользователь открывает свой браузер** и вводит адрес нужного сайта. 2. **Браузер ищет сервер.** Сервер — это программа, без которой не работал бы ни один сайт в интернете. Браузер ищет сервер по IP-адресу, который уникален для каждого сайта. Сначала он далеко не идет, а ищет его в кэше роутера, операционной системе или же в истории подключений, которая хранит информацию об IP-адреса сервера, если его уже посещали ранее. Если браузер там его не находит, он смотрит в DNS (Domain Name System). Она представляет собой что-то вроде телефонной книги, которая хранит информацию о том, какое доменное имя соответствует адресу. 3. **Браузер пытается установить соединение с сервером.** Теперь, когда браузер нашел нужный IP-адрес, он устанавливает с ним соединение с помощью специального протокола TCP/IP, который отвечает за передачу данных в интернете. Для установки соединения используется процесс “рукопожатие” (с англ. Handshaking). 4. При установлении защищенного соединения с использованием протокола HTTPS происходит следующий процесс:   -Браузер отправляет запрос на сервер, и сервер отвечает, предоставляя свой открытый ключ и цифровой сертификат. Цифровой сертификат содержит информацию о сервере и его открытом ключе, который был подписан соответствующим доверенным центром сертификации.  -Браузер проверяет подлинность сертификата сервера с помощью публичных ключей, которые уже имеются у него от доверенных центров сертификации. Если сертификат действителен и может быть проверен, то браузер продолжает процесс.  -Затем браузер генерирует случайный сеансовый ключ, который будет использоваться для шифрования данных, передаваемых во время текущего сеанса связи.  Браузер шифрует сеансовый ключ с помощью открытого ключа сервера, который был получен из сертификата сервера. Зашифрованный сеансовый ключ отправляется обратно на сервер.  -Сервер получает зашифрованный сеансовый ключ и использует свой закрытый ключ для его расшифровки. Теперь сервер и браузер обладают одним и тем же сеансовым ключом.  Дальнейшая коммуникация между браузером и сервером происходит с использованием этого сеансового ключа для шифрования и расшифровки данных, обеспечивая конфиденциальность и целостность информации.   1. **Браузер отправляет HTTP запрос на сервер.** Таким образом он запрашивает информацию для того, чтобы отобразить страницу. Эта коммуникация осуществляется с помощью GET-запроса и POST-запроса. 2. **Сервер обрабатывает запрос и отправляет ответ браузеру.** Запрос обрабатывается следующими веб-серверами: Apache, nginx, lighttpd. После этого сервер отправляет браузеру ответ с данными о файлах cookie, способах кэширования ну и, конечно же, контентом для отображения страницы. 3. **Браузер обрабатывает ответ и отображает запрашиваемый контент.** Это называется рендерингом. Пока он происходит, браузер и сервер обмениваются данными. По завершении, пользователь видит загруженную страницу. | | | |
| Как шифруются сообщения между браузером и сервером? | Это происходит благодаря алгоритму Диффи-Хеллмана — это криптографический протокол, который используется для безопасного обмена секретными ключами между двумя устройствами.  В контексте работы браузера, Диффи-Хеллман используется для защиты соединения между клиентом (браузером) и сервером.  Когда вы заходите на защищенный сайт (по протоколу HTTPS), ваш браузер и сервер обмениваются ключами с помощью алгоритма Диффи-Хеллмана. Эти ключи затем используются для шифрования и расшифрования передаваемой между браузером и сервером информации.  Пример применения алгоритма Диффи-Хеллмана можно представить себе так:  -Браузер и сервер обмениваются открытыми ключами (это числа, которые все знают).  -Браузер и сервер генерируют случайные числа (они не обязательно секретны).  -Браузер шифрует свое случайное число с помощью открытого ключа сервера, и отправляет результат серверу.  -Сервер расшифровывает это число с помощью своего секретного ключа, и делает то же самое со своим случайным числом.  -Браузер и сервер обмениваются результатами. Они могут использовать эти числа для создания общего секретного ключа, который будет использоваться для шифрования и расшифрования передаваемой информации.  Таким образом, благодаря алгоритму Диффи-Хеллмана, браузер и сервер могут создать общий секретный ключ, который будет использоваться для защиты информации, передаваемой между ними. Этот процесс защищает соединение между браузером и сервером от посторонних глаз, делая его более безопасным.  В алгоритме Диффи-Хеллмана информация шифруется открытыми ключами и расшифровывается закрытыми. | | | |
| Ка тестировать поле “Email” | Можно(позитивные):  — локальная часть начинается с цифры (4test@9m.com)  — локальная часть начинается с буквы (test@9m.com)  — локальная часть заканчивается буквой (test@9m.com)  — локальная часть заканчивается цифрой (test4@9m.com)  — локальная часть содержит нижнее подчеркивание (te\_st@9m.com)  (но не начинается и не заканчивается им)  — локальная часть содержит точку (te.st@9m.com)  (но не начинается и не заканчивается ей)  — локальная часть содержит дефис (te-st@9m.com)  (но не начинается и не заканчивается им)  — локальная часть из 4 символов (test@9m.com)  — локальная часть из 32 символов (testtesttesttesttesttesttesttest@9m.com)  — домен из двух букв (или буквы + цифры) (test@9m.ru)  — поддомен из двух символов(букв и\или цифр) (test@9m.com)  — поддомен из буквы, дефиса и буквы(test@m-t.com)  — поддомен + поддомен + домен(test@mini.9m.com)  Нельзя(должно вызвать предупреждение, негативные):  — имеет спецзнаки в локальной части (#%^$@9m.com)  — содержит пробелы в локальной части (te st@9m.com)  — локальная часть начинается с точки (.test@9m.com)  — локальная часть начинается с дефиса (-test@9m.com)  — локальная часть начинается нижним подчеркиванием (\_test@9m.com)  — локальная часть заканчивается точкой (test.@9m.com)  — локальная часть заканчивается дефисом (test-@9m.com)  — локальная часть заканчивает нижним подчеркиванием (test\_@9m.com)  — локальная часть содержит точки подряд (te..st@9m.com)  — локальная часть содержит нижние подчеркивания подряд (te\_\_st@9m.com)  — локальная часть содержит дефисы подряд (te—st@9m.com)  — локальная часть содержит сочетание разрешенных спецсимволов (te.-\_.st@9m.com)  — локальная часть из 3 символов (tes@9m.com)  — локальная часть из 33 символов (testtesttesttesttesttesttesttesto@9m.com)  — пустая локальная часть (@9m.com)  — пустая доменная часть (test@)  — пустые локальная и доменная части (@)  — два знака @@ подряд (test@@9m.com)  — два знака @@ через символы (te@st@9m.com)  — не содержит знак @ (test9m.com)  — доменная часть не содержит точку (test@9mcom)  — поддомен начинающийся с точки (test@.9m.com)  — поддомен начинающийся с дефиса (test@-9m.com)  — поддомен начинающийся с нижнего подчеркивания (test@\_9m.com)  — домен начинающийся с дефиса (test@9m.-com)  — домен начинающийся с нижнего подчеркивания (test@9m.\_com)  — поддомен заканчивающийся дефисом (test@9m-.com)  — поддомен заканчивающийся нижним подчеркиванием (test@9m\_.com)  (помни что точка служит разделением поддомена от домена и других поддоменов. Это значит домен может начинаться с точки, а поддомен заканчиваться точкой)  — доменная часть заканчивающаяся точкой (test@9m.com.)  — доменная часть заканчивающаяся дефисом (test@9m.com-)  — доменная часть заканчивающаяся нижним подчеркиванием (test@9m.com\_)  — доменная часть содержит точки подряд (test@9m..com)  — доменная часть содержит дефисы подряд (test@9—m.com)  — доменная часть содержит нижние подчеркивания подряд (test@9\_\_m.com)  — доменная часть содержит сочетание разрешенных спецсимволов (test@9-\_m.com)  — домен первого уровня из одного символа (test@9m.c)  — поддомен из одного символа (test@m.com)  — имеет спецзнаки в доменной части (test@#$.com)  — содержит пробелы в доменной части (test@9m .com)  — домен содержит только цифры (test@9m.894)  — поддомен содержит только цифры (test@940.com)  (поддомен из одних цифр на самом деле существует, но здесь мы проверяем на адреса популярных бесплатных почтовиков)  — пробелы перед и после емаила ( test@9m.com )  — одни пробелы | | | |
| Как тестировать поле “Дата рождения”? | **Тестирование осуществляется по следующим этапам:**  1. Вводятся типичные правильные значения, и проверяется связанная с полем функциональность (“05/05/2005” или “05-05-2005”). 2. Вводятся типичные неправильные значения, и проверяется присутствие валидации (“dfg”). 3. Вводятся граничные правильные значения. Каждый элемент (день, месяц, год, час, минута, секунда) тестируются отдельно. Можно совмещать в один тестовый пример тестирование нескольких элементов одновременно, но при нахождении ошибки нужно провести тестирование каждого элемента по отдельности. Это значит, что только один элемент принимает граничное правильное значение, а остальные – типичные правильные. Границы правильных значений: 3.1. Месяц: 01..12. 3.2. День: 01..max\_in\_month, где max\_in\_month – максимальное количество дней в выбранном месяце. Варианты: 02/29/2000, 03/31/2000. 3.3. Год: 1753..9999. 3.4. Час: 00..23. 3.5. Минута: 00..59. 3.6. Секунда: 00..59. 3.7. Если дата не может быть будущей, то вводим сегодня/сейчас. 3.8. Если 2 даты означают диапазон, то вводим одинаковые даты. 4. Дата/время остаются правильными если: 4.1. Месяц/день/час/минута/секунда представлены одной цифрой. 4.2. Секунды отсутствуют. 5. Вводятся граничные неправильные значения. Каждый элемент (день, месяц, год, час, минута, секунда) тестируются отдельно. Нельзя совмещать в один тестовый пример тестирование нескольких элементов одновременно. Это значит, что только один элемент принимает граничное неправильное значение, а остальные – типичные правильные. Границы неправильных значений: 5.1. Минимальное значение -1 (01/01/1752). 5.2. Максимальное значение +1. Особенно месяц. Примеры: 02/29/2001, 04/31/2001, 03/32/2001. 5.3. Нулевое значение для месяца, дня, года. 5.4. Отрицательное значение ("-12/12/2000"). 5.5. Нечисловое значение (“dghf/12/2000”). 5.6. Число, содержащее пробел (“0 1/01/2000”). 5.7. Выражение, начинающееся с правильного числа, но содержащее нечисловые символы (“01/01/2000dhr”). 5.8. Неправильные разделители (-\?.,:;!/ и пробел для даты/времени, в зависимости от того, какие разделители не являются правильными) (01:01:2000). 5.9. Число в неправильном формате ("1e+1/10/2000", 1+1/01/2000). 5.10. Число отсутствует ("/01/2000"). 5.11. Поменять месяц и день местами, чтобы месяц был больше 12 (20/10/2000). 5.12. Не целое число ("01/01/1923.23") 5.13. Число больше 15 символов (01/01/1234567890123456). 5.14. Если дата/время не может быть будущим, то вводим: а) сегодня/сейчас +1. б) значительно большую дату/время. 5.15. Если 2 даты означают диапазон, то вводим а) одинаковые даты в будущем. б) правильный диапазон в будущем. в) начальную дату больше конечной. 6. Производятся атаки на поле: 6.1. DoS атака. Строка содержит очень много символов (0,1-1Мб). 6.2. Строка содержит спецсимволы ("♦♣☺♂"). 6.3. XSS атака. Строка содержит html или php теги. Примеры: <input> <input type="button" title="Hello"> <form action="http://gorod.dp.ua"><input type="submit"></form> <script>alert("Privet")</script> <script>window.open("[http://gorod.dp.ua"](http://gorod.dp.ua/))</script> <script>document.getElementByID("...").disabled=true</script> 6.4. SQL инъекция. Строка содержит SQL команды в правильном формате для исполнения: '01/01/2000"; DROP TABLE users; SELECT \* FROM data WHERE name LIKE "%'. | | | |
| Как протестировать форму регистрации? | Тестирование полей и форм “Дана форма для регистрации. Протестируйте” - вопрос номер один практически на всех собеседованиях на младшую позицию. Он хорош еще и тем, что в зависимости от уровня кандидата будет раскрыт в разной степени. Всегда в первую очередь уточняйте хотя бы какие-то минимальные требования, даже если вначале озвучивают, что требования не формализованы.   * Начальный уровень представляет из себя простые позитивные и негативные кейсы (в основном на валидацию)**:**   + Обязательные поля отмечены \*   + Обязательные поля заполнены/нет   + Галочки на соглашениях проставлены/нет   + Поле password и подтверждение имеет соответствующий тип (в полях формы прописан корректный атрибут TYPE, сообщающий браузеру тип элементов формы.)   + Проверяется, что пароли одинаковы   + Имя пользователя валидируется как минимум на длину и спец. символы, остальное по ТЗ   + Адрес почты валидируется в соответствии со стандартом (наличие символа @, несколько символов @, длины частей до и после @, допустимые символы до и после, наличие пробелов перед адресом и после, корректная доменная часть и т.п.)   + Поля с ожидаемым числовым вводом и текстовым соответственно проверить позитивными и негативными кейсами по типам данных * Следующий уровень:   + Все из предыдущего   + Кроссбраузерность   + Понятность формы. Присутствует описание полей или плейсхолдеры   + Сенситив данные не должны передаваться в URL   + Проверяем, как форма отображается до сабмита и после   + Поведение, если нажать сабмит несколько раз подряд   + Если формы очищаются после сабмита, проверить регистрацию существующего пользователя   + Проверка глобализации - номер телефона, дата, почтовый индекс, валюта, вертикальное или RTL письмо и т.п. (опционально)   + Проверка простых инъекций   + Правильная работа многошаговых форм (Навигация рядом с формой показывает текущий этап и количество оставшихся шагов.)   + Для полей, предполагающих загрузку файлов, прописан атрибут accept, определяющий тип загружаемых документов   + Текстовое многострочное поле при вводе объемного сообщения изменяет высоту либо в правой части появляется скроллбар для просмотра всего содержимого   + Для авторизованного пользователя в поля формы автоматически подставляются все известные о посетителе данные.   + Форма сохраняется в веб-формах (админ-панели) или SQL-таблицах.   + Прописан реальный e-mail лица, отвечающего за обработку заявок (если предполагается ОС)   + Опционально. Пользователь получает уведомление на свой e-mail об успешно полученной заявке и последующих действиях, которые от него требуются.   + Прописан атрибут autocomplete для полей, поддерживающих это значение * Extra:   + Проверяем, отправились ли данные после сабмита   + Проверяем, добавились ли соответствующие записи в бд   + Проверка загрузки формы и сабмита при медленном/нестабильном интернет-соединении   + Корректность cookies/токена и т.п. после сабмита | | | |
| Как работает интернет? | <https://habr.com/ru/post/491540/> | | | |
| Базовые команды Linux | **pwd** - вывести текущую (рабочую) директорию.  **date** - вывести текущую дату и время системы.  **w** – показывает кто авторизован в системе.  **ls** – вывести содержимое директории.  **cd** – изменить текущую директорию.  **mkdir** – создать директорию.  **rm** – удалить файл.  **cp** – копировать файл или директорию.  **mv** – переместить или переименовать файл или директорию.  **cat** – вывести содержимое файла(или файлов).  **tail** – вывести последние измененные строки файла.  **less** - иногда текстовый файл слишком большой, и неудобно выводить его командой cat. Тогда можно открыть его с помощью команды less: файл будет выводиться по частям, доступна навигация по этим частям, поиск и прочий простой функционал.  **ps** – вывести список процессов.  **kill** - послать сигнал процессу. По умолчанию посылается сигнал SIGTERM, который завершает процесс.  **ssh** - openSSH SSH клиент, позволяет подключаться к удаленному хосту.  **scp** - копировать файлы между хостами (для этого используется ssh).  **rsync** - также для синхронизации директорий между хостами можно использовать rsync (-a — archive mode, позволяет скопировать полностью всё содержимое директории «как есть»,  **v** — вывод на консоль дополнительной информации).  **echo** - вывести на экран строку текста. | | | |
| Система контроля версий | **Version Control System, или коротко VCS** - это программное обеспечение, которое помогает отслеживать изменения в программах, текстовых файлах, больших документах, веб-сайтах и так далее (Git, Mercurial, Subversion и другие). Одна из ключевых особенностей современных систем контроля версий — поддержка параллельной работы нескольких пользователей, в том числе над одним файлом. Именно поэтому VCS так популярны у IT-команд. | | | |
| Команды git | **Инициализация репозитория**  git init (от англ. initialize, «инициализировать») — инициализируй репозиторий.  **Синхронизация локального и удалённого репозиториев**  git remote add origin https://github.com/YandexPracticum/first-project.git (от англ. remote, «удалённый» + add, «добавить») — привяжи локальный репозиторий к удалённому с URL https://github.com/YandexPracticum/first-project.git;  git remote -v (от англ. verbose, «подробный») — проверь, что репозитории действительно связались;  git push -u origin main (от англ. push, «толкать») — в первый раз загрузи все коммиты из локального репозитория в удалённый с названием origin.  💡 Ваша ветка может называться master, а не main. Подправьте команду, если это необходимо.  git push (от англ. push, «толкать») — загрузи коммиты в удалённый репозиторий после того, как он был привязан с помощью флага -u.  **Подготовка файла к коммиту**  git add todo.txt (от англ. add, «добавить») — подготовь файл todo.txt к коммиту;  git add --all (от англ. add, «добавить» + all, «всё») — подготовь к коммиту сразу все файлы, в которых были изменения, и все новые файлы;  git add . — подготовь к коммиту текущую папку и все файлы в ней.  Создание и публикация коммита  git commit -m "Комментарий к коммиту." (от англ. commit, «совершать», фиксировать» + message, «сообщение») — сделай коммит и оставь комментарий, чтобы было проще понять, какие изменения сделаны;  git push (от англ. push, «толкать») — добавь изменения в удалённый репозиторий.  **Просмотр информации о коммитах**  git log (от англ. log, «журнал [записей]») — выведи подробную историю коммитов;  git log --oneline (от англ. log, «журнал [записей]» + oneline, «одной строкой») — покажи краткую информацию о коммитах: сокращённый хеш и сообщение.  **Просмотр состояния файлов**  git status (от англ. status, «статус», «состояние») — покажи текущее состояние репозитория.  **Добавление изменений в последний коммит**  git commit --amend --no-edit (от англ. amend, «исправить») — добавь изменения к последнему коммиту и оставь сообщение прежним;  git commit --amend -m "Новое сообщение" — измени сообщение к последнему коммиту на Новое сообщение.  **«Откат» файлов и коммитов**  git restore --staged hello.txt (от англ. restore, «восстановить») — переведи файл hello.txt из состояния staged обратно в untracked или modified;  git restore hello.txt — верни файл hello.txt к последней версии, которая была сохранена через git commit или git add;  git reset --hard b576d89 (от англ. reset, «сброс», «обнуление» + hard, «суровый») — удали все незакоммиченные изменения из staging и «рабочей зоны» вплоть до указанного коммита.  **Просмотр изменений**  git diff (от англ. difference, «отличие», «разница») — покажи изменения в «рабочей зоне», то есть в modified-файлах;  git diff a9928ab 11bada1 — выведи разницу между двумя коммитами;  git diff --staged — покажи изменения, которые добавлены в staged-файлах. | | | |
| Жизненный цикл файла в Git |  | | | |
| Fork | **Fork** (англ. «развилка», «ответвление»), или **«форк»**, — это GitHub-операция; напрямую с Git она не связана. «Форк» создаёт копию репозитория в аккаунте GitHub. Такая копия будет полностью независима. Изменения, которые вы внесёте, не будут синхронизированы с исходным репозиторием.  В процессе «форка» создаётся копия всех файлов, истории коммитов и веток. Эта копия сохраняется в вашей учётной записи GitHub.  Вот некоторые из распространённых причин использования «форков»:   * Вы хотите внести свой вклад в проект (например, open source), но не имеете прав на изменение исходного репозитория. Тогда вы можете сделать «форк», добавить нужные правки, а затем отправить запрос на включение этих изменений в оригинальный проект. * Вы хотите развивать проект независимо от исходного. Допустим, создатели проекта решили, что не будут добавлять функциональность, которая вам необходима. В таком случае вы можете сделать «форк» и добавить её самостоятельно. | | | |
| Ветки | **Ветка** (англ. branch) — это изолированный поток разработки проекта. В таком потоке можно проверять разные идеи, тестировать новую функциональность и так далее.  Основная, стабильная версия проекта хранится в главной ветке main или master. Она появляется автоматически при создании репозитория. Часто все новые ветки в репозитории отходят от main, хотя это не является правилом.  **Клонирование чужого репозитория**  git clone git@github.com:YandexPraktikum/first-project.git (от англ. clone, «клон», «копия») — склонируй репозиторий с URL first-project.git из аккаунта YandexPraktikum на мой локальный компьютер.  **Создание веток**  git branch feature/the-finest-branch (от англ. branch, «ветка») — создай ветку от текущей с названием feature/the-finest-branch;  git checkout -b feature/the-finest-branch — создай ветку feature/the-finest-branch и сразу переключись на неё.  **Навигация по веткам**  git branch (от англ. branch, «ветка») — покажи, какие есть ветки в репозитории и в какой из них я нахожусь (текущая ветка будет отмечена символом \*);  git branch -a — покажи все известные ветки, как локальные (в локальном репозитории), так и удалённые (в origin, или на GitHub).  git checkout feature/br — переключись на ветку feature/br.  **Сравнение веток**  git diff main HEAD (от англ. difference, «отличие», «разница») — покажи разницу между веткой main и указателем на HEAD;  git diff HEAD~2 HEAD — покажи разницу между тем коммитом, который был два коммита назад, и текущим.  **Удаление веток**  git branch -d br-name — удали ветку br-name, но только если она является частью main;  git branch -D br-name — удали ветку br-name, даже если она не объединена с main.  **Слияние веток**  git merge main (от англ. merge, «сливать», «поглощать») — объедини ветку main с текущей активной веткой.  **Работа с удалённым репозиторием**  git push -u origin my-branch (от англ. push, «толкнуть», «протолкнуть») — отправь новую ветку my-branch в удалённый репозиторий и свяжи локальную ветку с удалённой, чтобы при дополнительных коммитах можно было писать просто git push без -u;  git push my-branch — отправь дополнительные изменения в ветку my-branch, которая уже существует в удалённом репозитории;  git pull (от англ. pull, «вытянуть») — подтяни изменения текущей ветки из удалённого репозитория. | | | |
| Подходы в работе с ветками | **Подходы** - это правила, которые описывают, когда и для чего создаются ветки, какие в них коммиты и в какой момент происходит слияние веток.  Рассмотрим три самых популярных подхода — их также принято называть workflow (англ. «рабочий процесс») или сокращённо: flow.  **Feature branch workflow** — простой и самый популярный вариант. Если коротко, в нём для каждого нового изменения создаётся новая ветка, которая позже вливается в main с помощью git merge.  **Git flow** — более сложный вариант. Подход похож на feature branch workflow, но в нём создаётся больше веток, а изменения (коммиты) делят на разные типы: исправление, новая функциональность и так далее. Разные типы коммитов попадают в разные ветки.  **Trunk-based** — популярный в больших компаниях (таких как Яндекс, Google и прочих) подход, который обещает бо́льшую скорость работы в крупных командах.  Этот подход тоже похож на feature branch workflow. Главное отличие в том, что участники проекта вливают (merge) свой код в основную ветку максимально часто. Например, каждый день. | | | |
| Какие существуют парадигмы программирования? | Существует несколько парадигм программирования, каждая из которых представляет различный подход к решению задач программирования. Вот некоторые из наиболее распространенных парадигм программирования:   1. Императивное (процедурное) программирование (с): это парадигма программирования, которая основана на последовательном выполнении команд и изменении состояния программы. В императивном программировании разработчик должен явно указать, что нужно делать, а также как и когда делать это. 2. Декларативное программирование (HTML, SQL): это парадигма программирования, которая позволяет описывать желаемый результат, а не порядок его достижения. В декларативном программировании разработчик описывает, что нужно сделать, а не как это делать. 3. Функциональное программирование(F#): это парадигма программирования, которая основана на функциях как на основных элементах программы. В функциональном программировании функции могут быть переданы в качестве аргументов другим функциям, а также могут быть возвращены из функций. 4. Объектно-ориентированное программирование: это парадигма программирования, которая основана на объектах, каждый из которых имеет свойства и методы. В объектно-ориентированном программировании объекты могут наследовать свойства и методы от других объектов, а также могут быть созданы новые объекты на основе существующих. 5. Логическое программирование: это парадигма программирования, которая основана на формальной логике и математических концепциях. В логическом программировании разработчик описывает, какие факты и правила применять к данным для получения нужных результатов. 6. Реактивное программирование: это парадигма программирования, которая основана на работе с потоками данных и событиями. В реактивном программировании программа реагирует на изменения внешней среды и входных данных, а также уведомляет об изменениях другие части программы. | | | |
| Что такое ООП? | **ООП, или объектно-ориентированное программирование**, это парадигма программирования, которая основана на концепции объектов, каждый из которых имеет свои свойства и методы. ООП используется для разработки программного обеспечения, которое легко масштабируется и поддерживается.  В ООП программа рассматривается как набор объектов, которые взаимодействуют друг с другом для выполнения определенных задач. Каждый объект имеет свои свойства, которые описывают его состояние, и методы, которые определяют его поведение. Объекты могут взаимодействовать друг с другом, обмениваясь сообщениями, вызывая методы других объектов или получая доступ к их свойствам.  Преимущества объектно-ориентированного программирования включают:  -Модульность: ООП позволяет разбить программу на независимые объекты, что упрощает разработку и поддержку программного обеспечения.  -Переиспользование: Объекты могут использоваться в разных частях программы, что уменьшает дублирование кода и упрощает разработку новых функций.  -Наследование: Объекты могут наследовать свойства и методы от других объектов, что позволяет создавать новые объекты на основе существующих.  -Полиморфизм: Объекты могут иметь различные реализации одного и того же метода, что позволяет программе адаптироваться к различным условиям и изменениям входных данных. Абстракция: ООП позволяет скрыть детали реализации объектов, что упрощает использование программы и уменьшает вероятность ошибок. | | | |
| 4 столпа ООП? | Существует несколько принципов объектно-ориентированного программирования (ООП), которые помогают разработчикам создавать программы, которые легко поддерживать и масштабировать. **Вот четыре основных принципа ООП:**   1. Инкапсуляция: Инкапсуляция является одним из основных принципов объектно-ориентированного программирования (ООП).   Есть две трактовки:  Инкапсуляция — это объединение данных и методов, которые управляют этими данными в единый компонент.  Инкапсуляция – это механизм языка, позволяющий ограничить доступ одних компонентов программы к другим (т.е. изменять состояние объекта возможно только через его методы), но не всегда т.к. существует рефлексия.  Важно, что если мы сможем из любого места кода обратиться к публичному полю, то в случае его изменения полетит много всего, а если мы будем использовать приватное поле + гет + сет, то при изменении будут минимальные поломки.  Также есть нюанс с сокрытием он заключается в том, что инкапсуляция обеспечивает сокрытие, но не является сокрытием.   1. Наследование: Этот принцип заключается в том, что объекты могут наследовать свойства и методы от других объектов. Наследование позволяет создавать новые объекты на основе существующих и уменьшает дублирование кода. Также наследование облегчает создание иерархий объектов и упрощает изменение реализации методов. 2. Полиморфизм: Этот принцип заключается в том, что объекты могут иметь различные реализации одного и того же метода. Полиморфизм позволяет программе адаптироваться к различным условиям и изменениям входных данных. Также полиморфизм упрощает разработку кода и повышает его гибкость. 3. Абстракция: Этот принцип заключается в том, что детали реализации объекта должны быть скрыты от пользователя. Абстракция позволяет разрабатывать программы на более высоком уровне абстракции, что упрощает понимание и использование кода. Абстракция также позволяет скрыть детали реализации объекта, что уменьшает вероятность ошибок и упрощает поддержку программного обеспечения. | | | |
| SOLID | **SOLID** - это сокращение от первых букв пяти принципов объектно-ориентированного программирования, которые были предложены Робертом Мартином (Robert C. Martin) в начале 2000-х годов. Эти принципы помогают разработчикам создавать более гибкие, расширяемые и поддерживаемые системы.  1) Принцип единственной ответственности (Single Responsibility Principle - SRP) - класс должен иметь только одну причину для изменения. Это означает, что класс должен отвечать только за одну обязанность или задачу. Если класс имеет несколько обязанностей, то изменение одной из них может повлиять на другие, что приведет к неожиданным побочным эффектам.  2) Принцип открытости/закрытости (Open-Closed Principle - OCP) - классы должны быть открыты для расширения, но закрыты для изменения. Это означает, что поведение класса можно расширить, добавив новые функции, но не изменяя существующий код.  3) Принцип подстановки Барбары Лисков (Liskov Substitution Principle - LSP) - объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы. Это означает, что любой экземпляр подкласса должен иметь такое же поведение, как и его суперкласс, и не должен изменять контракт, установленный в суперклассе.  4) Принцип разделения интерфейса (Interface Segregation Principle - ISP) - клиенты не должны зависеть от интерфейсов, которые они не используют. Это означает, что интерфейсы должны быть максимально специализированы, чтобы клиенты могли реализовывать только те методы, которые им нужны.  5) Принцип инверсии зависимостей (Dependency Inversion Principle - DIP) - модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба должны зависеть от абстракций. Это означает, что модули должны зависеть от абстракций (интерфейсов или абстрактных классов), а не от конкретных реализаций.  Соблюдение этих принципов помогает создавать гибкие, расширяемые и поддерживаемые системы. | | | |
| KISS | **KISS** — это аббревиатура от фразы "Keep It Simple, Stupid" (Делайте это просто, дурак), которая описывает принцип проектирования программного обеспечения, который заключается в том, чтобы создавать системы и решения как можно проще и понятнее.  Принцип KISS включает в себя несколько идей:  1) Простота - система должна быть настолько простой, насколько это возможно. Она должна решать только те задачи, которые ей были заданы, и не иметь никакого избыточного кода или функциональности.  2) Ясность - система должна быть ясной и понятной для всех, кто работает с ней. Код должен быть написан так, чтобы его можно было легко читать и понимать.  3) Минимализм - система должна иметь минимальное количество зависимостей и компонентов. Это позволит упростить ее сопровождение и разработку.  4) Надежность - система должна быть надежной и стабильной. Это достигается путем тщательного тестирования и проверки кода на ошибки.  Принцип KISS помогает улучшить качество и производительность программного обеспечения, уменьшить риски и упростить сопровождение и разработку. Когда разработчики следуют принципу KISS, они создают более эффективные и удобные системы, которые легко сопровождать и развивать в будущем. | | | |
| DRY | **DRY** - это аббревиатура от фразы "Don't Repeat Yourself" (не повторяйся), которая описывает принцип проектирования программного обеспечения, который заключается в том, чтобы избегать дублирования кода и логики в разных частях системы.  Принцип DRY включает в себя несколько идей:  1) Избегание дублирования кода - вся логика, которая используется в нескольких местах системы, должна быть вынесена в отдельную функцию, класс или модуль.  2) Однозначность - каждый фрагмент кода должен выполнять только одну функцию. Это позволяет легче понимать и поддерживать систему.  3) Надежность - изменения в одном месте не должны влиять на работу других частей системы. Это позволяет избежать ошибок и повысить надежность системы.  Принцип DRY позволяет создавать более эффективный, поддерживаемый и масштабируемый код. Избегая дублирования, разработчики могут быстрее и безопаснее вносить изменения в систему, не боясь нарушить ее работу. Также DRY помогает улучшить качество кода и уменьшить объем работы при поддержке и доработке системы в будущем. | | | |
| YAGNI | **YAGNI** - это аббревиатура от фразы "You Ain't Gonna Need It" (вы не понадобитесь этого), которая описывает принцип проектирования программного обеспечения, который заключается в том, чтобы избегать добавления ненужной функциональности в систему.  Принцип YAGNI включает в себя несколько идей:  1) Не добавляйте функциональность, которая не требуется на текущем этапе разработки - разработчики должны сосредоточиться только на том, что необходимо для выполнения текущей задачи и избегать добавления функциональности, которая может понадобиться в будущем, но не требуется сейчас.  2) Не добавляйте сложности, которые не требуются - разработчики должны избегать создания сложных структур или архитектуры, которые не являются необходимыми для решения текущих задач.  3) Используйте простые решения - разработчики должны выбирать наиболее простые и прямолинейные решения для решения текущих задач, а не создавать сложные и избыточные конструкции.  Принцип YAGNI помогает разработчикам создавать более простые и понятные системы, которые легче сопровождать и развивать. Он также помогает избежать излишней сложности и избыточности, которые могут привести к ошибкам и увеличению затрат на разработку и поддержку системы. | | | |
| Что такое паттерны-проектирования? | **Паттерны проектирования** — это повторяемые решения для типичных проблем в проектировании программного обеспечения. Они обеспечивают определенную степень стандартизации в разработке ПО и позволяют разработчикам использовать уже известные и проверенные способы решения конкретных задач, вместо того, чтобы каждый раз изобретать велосипед. | | | |
| Зачем нужны паттерны проектирования? | **Вот несколько причин, почему паттерны проектирования полезны:**  1)Улучшают качество кода: Паттерны проектирования помогают создавать более чистый, гибкий и поддерживаемый код, что приводит к уменьшению количества ошибок и более быстрой разработке.  2)Сокращают время разработки: Использование готовых решений позволяет сэкономить время на разработке новых компонентов, так как разработчики уже знакомы с применением паттернов и могут быстро применять их в своей работе.  3)Увеличивают переносимость: Использование паттернов проектирования может помочь сделать код более переносимым, что означает, что он может быть легко перенесен на другие платформы или архитектуры.  4)Содействуют коммуникации: Паттерны проектирования предоставляют общий язык и соглашения для команды разработчиков, что упрощает коммуникацию и повышает эффективность работы.  5)Упрощают сопровождение: При использовании паттернов проектирования код становится более понятным и легче сопровождать, что делает его более надежным и устойчивым к изменениям.  6)Улучшают расширяемость: Использование паттернов проектирования позволяет создавать более гибкие и расширяемые приложения, что упрощает добавление новых функций в будущем. | | | |
| Page Object | <https://testengineer.ru/bolshoj-gajd-po-page-object-model/> | | | |
| SCREENPLAY | <https://habr.com/ru/companies/arcadia/articles/434122/> | | | |
| Singleton | <https://refactoring.guru/ru/design-patterns/singleton>  Где его применить в атотестировании?  Конкретный пример объекта, который может быть создан с использованием паттерна Singleton в автотестировании — это объект WebDriver.  WebDriver — это интерфейс для работы с браузером в автоматизированном тестировании. Он используется для управления браузером и выполнения действий на веб-страницах.  Создание объекта WebDriver может быть дорогостоящим и занимать много времени. Поэтому создание одного экземпляра WebDriver и его использование во всех тестах может ускорить выполнение тестов.  Получено сообщение. Конкретный пример объекта, который может быть создан с использованием паттерна Singleton в автотестировании — это объект WebDriver. WebDriver — это интерфейс для работы с браузером в автоматизированном тестировании. Он используется для управления браузером и выполнения действий на веб-страницах. Создание объекта WebDriver может быть дорогостоящим и занимать много времени. Поэтому создание одного экземпляра WebDriver и его использование во всех тестах может ускорить выполнение тестов. | | | |
| Builder | <https://refactoring.guru/ru/design-patterns/builder> | | | |
| Decorator | <https://refactoring.guru/ru/design-patterns/decorator> | | | |
| Facade | <https://refactoring.guru/ru/design-patterns/facade> | | | |
| Observer (Наблюдатель) | <https://refactoring.guru/ru/design-patterns/observer> | | | |
| Iterator | <https://refactoring.guru/ru/design-patterns/iterator> | | | |
| Что такое алгоритмы? | В широком смысле ***алгоритм*** — это последовательность действий, которые нужно выполнить, чтобы получить определённый результат. | | | |
| Оценка сложности алгоритма или асимптотика? | Наверняка вы не раз сталкивались с обозначениями вроде O(log n) или слышали фразы типа **«логарифмическая вычислительная сложность»** в адрес каких-либо алгоритмов.  ***Оценка сложности или асимптотика***  Сложность алгоритмов обычно оценивают по времени выполнения или по используемой памяти. В обоих случаях сложность зависит от размеров входных данных: массив из 100 элементов будет обработан быстрее, чем аналогичный из 1000. При этом точное время мало кого интересует: оно зависит от процессора, типа данных, языка программирования и множества других параметров. Важна лишь асимптотическая сложность, т. е. сложность при стремлении размера входных данных к бесконечности.  Допустим, некоторому алгоритму нужно выполнить 4n3 + 7n условных операций, чтобы обработать n элементов входных данных. При увеличении n на итоговое время работы будет значительно больше влиять возведение n в куб, чем умножение его на 4 или же прибавление 7n. Тогда говорят, что временная сложность этого алгоритма равна О(n3), т. е. зависит от размера входных данных кубически.  Использование заглавной буквы О (или так называемая О-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально O(f(n)) означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на f(n).  ***Примеры***  ***O(n) — линейная сложность***  Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройтись по всем n элементам массива, чтобы понять, какой из них максимальный.  ***O(log n) — логарифмическая сложность***  Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим log n элементов.  ***O(n2) — квадратичная сложность***  Такую сложность имеет, например, алгоритм сортировки вставками. В канонической реализации он представляет из себя два вложенных цикла: один, чтобы проходить по всему массиву, а второй, чтобы находить место очередному элементу в уже отсортированной части. Таким образом, количество операций будет зависеть от размера массива как n \* n, т. е. n2.  Бывают и другие оценки по сложности, но все они основаны на том же принципе.  Также случается, что время работы алгоритма вообще не зависит от размера входных данных. Тогда сложность обозначают как O(1). Например, для определения значения третьего элемента массива не нужно ни запоминать элементы, ни проходить по ним сколько-то раз. Всегда нужно просто дождаться в потоке входных данных третий элемент и это будет результатом, на вычисление которого для любого количества данных нужно одно и то же время.  Аналогично проводят оценку и по памяти, когда это важно. Однако **алгоритмы могут использовать значительно больше памяти при увеличении размера входных данных, чем другие, но зато работать быстрее. И наоборот. Это помогает выбирать оптимальные пути решения задач исходя из текущих условий и требований.**  ***Наглядно***  Время выполнения алгоритма с определённой сложностью в зависимости от размера входных данных при скорости 106 операций в секунду:  [сложность алгоритмов](https://media.tproger.ru/uploads/2017/09/complexity-table.png) | | | |
| Алгоритм бинарного поиска | Алгоритм бинарного поиска используется для более быстрого поиска какого, либо объекта в **отсортированном** списке. Его оценка сложности или асимптотика равна О(log n) – это означает, что нам понадобится 7 шагов т.к. log2 100 = 7.  Алгоритм бинарного поиска работает следующим образом:   1. Определить середину отсортированного массива. 2. Сравнить искомое значение с элементом в середине массива. 3. Если искомое значение меньше, чем элемент в середине массива, то продолжить поиск в левой половине массива. 4. Если искомое значение больше, чем элемент в середине массива, то продолжить поиск в правой половине массива. 5. Если искомое значение равно элементу в середине массива, то поиск завершен.   Вот пример кода на Java для реализации алгоритма бинарного поиска:  public class BinarySearch {  public static int binarySearch(int[] arr, int key) {  int first = 0;  int last = arr.length - 1;  while (first <= last) {  int mid = (first + last) / 2;  if (arr[mid] == key) {  return mid;  } else if (arr[mid] > key) {  last = mid - 1;  } else {  first = mid + 1;  }  }  return -1;  }  } | | | |
| Алгоритм поиска в ширину | Поиск в ширину – алгоритм поиска и работает с графами.  Он отвечает на два вопроса:  -Существует ли путь от узла А к узлу В?  Пример: Наша задача найти продавца манго. Мы начинаем искать среди друзей в вк и если у друга нет, то в список тех, у кого мы будем проверять добавляются друзья нашего друга. Таким образом рано или поздно мы пройдем по всей сети, пока не отыщем продавца манго. Так и работает алгоритм поиска в ширину.  -Как выглядит кратчайший путь от узла А к узлу В? Пример:  Допустим, что ваши друзья – это связи первого уровня, а друзья друзей - связи второго уровня. Связи первого уровня предпочтительнее связей второго уровня, связи второго уровня предпочтительнее связей третьего уровня и т.д. Отсюда следует, что пока вы не проверите все контакты первого уровня вы не должны переходить на второй уровень.  Также это можно объяснить так: Связи первого уровня добавляются в список поиска раньше связей второго уровня. Вы двигаетесь вниз по списку и проверяете каждого человека. Связи первого уровня будут проверены до связей второго уровня, так что вы найдете продавца манго, ближайшего к вам. Поиск в ширину находит не только путь из А в, но и кратчайший путь. Для операций такого рода существует специальная структура данных, которая называется очередью.  Асимптотика О(V+E) (V – количество вершин, E – количество ребер) | | | |
| Алгоритм k ближайших соседей | Алгоритм k-ближайших соседей (kNN) — это метрический алгоритм классификации, основанный на оценивании сходства объектов. Классифицируемый объект относится к тому классу, которому принадлежат ближайшие к нему объекты обучающей выборки.  Например, если мы хотим классифицировать новый объект (красный круг) на рисунке ниже, мы можем использовать kNN для определения его класса. Если мы выберем k = 3 (три ближайших соседа), мы увидим, что два круга и один треугольник находятся ближе всего к новому объекту. Поэтому мы можем классифицировать новый объект как круг. | | | |
| Жадные алгоритм | Жадный алгоритм — это алгоритм, который на каждом шаге выбирает локально оптимальное решение в надежде получить глобально оптимальное решение. Он не всегда дает оптимальное решение, но часто работает достаточно хорошо.  Например, если мы хотим найти минимальное количество монет для сдачи определенной суммы денег, то жадный алгоритм будет выбирать монеты наибольшего номинала, пока не достигнет нужной суммы. Это может не дать минимального количества монет в некоторых случаях, но обычно работает достаточно хорошо.  Например, если у нас есть рюкзак вместимостью 15 кг и следующие предметы:   | Предмет | Вес (кг) | Стоимость (руб.) | | --- | --- | --- | | A | 10 | 60 | | B | 20 | 100 | | C | 30 | 120 |   Тогда жадный алгоритм выберет сначала предмет C (так как он имеет наибольшую удельную стоимость), затем A. В итоге мы получим решение со стоимостью 180 рублей и весом 40 кг. | | | |
| Алгоритм Дейкстры | Алгоритм Дейкстры — это алгоритм на графах, который находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только с графами без рёбер отрицательного веса.  Алгоритм Дейкстры работает следующим образом:   1. Выбирается начальная вершина. 2. Все остальные вершины помечаются как не посещённые. 3. Для каждой не посещённой вершины вычисляется ее расстояние от начальной вершины. 4. Выбирается не посещённая вершина с наименьшим расстоянием и помечается как посещенная. 5. Для каждой не посещённой соседней вершины текущей вершины вычисляется расстояние от начальной вершины через текущую вершину. Если это расстояние меньше, чем уже известное расстояние до этой вершины, то оно заменяется на новое значение. 6. Повторяем шаги 4 и 5 до тех пор, пока все вершины не будут помечены как посещенные | | | |
| Отличие массива и списка? | ***Отличие массива и списка?***   1. Память компьютера напоминает огромный шкаф с ящиками. 2. Если вам потребуется сохранить набор элементов, воспользуйтесь массивом или списком. 3. В массиве все элементы хранятся в памяти рядом друг с другом. 4. В списке элементы распределяются в произвольных местах памяти, при этом в одном элементе хранится адрес следующего элемента. 5. Массивы обеспечивают быстрое чтение. 6. Списки обеспечивают быструю вставку и удаление (Важно, что это выполняется только в случае, если мы получим мгновенный доступ к объекту, но на практике мы знаем адрес только первого и последнего объекта в списке). 7. Все элементы массива должны быть однотипными (только целые числа, только вещественные числа и т. д.).  |  |  |  | | --- | --- | --- | |  | Массивы | Списки | | Чтение | О(1) | О(n) | | Вставка | O(n) | O(1) | | Удаление | O(n) | O(1) |   ***Как происходит удаление или добавление элементов в массив и список?*** При удалении или добавлении элементов в массиве все последующие элементы должны быть сдвинуты на одну позицию влево или вправо, что может занять много времени и памяти. В связном списке каждый элемент содержит ссылку на следующий элемент, поэтому при удалении или добавлении элемента ссылки могут быть обновлены без необходимости перемещения других элементов. Это может быть более эффективным в использовании памяти и времени, если вы часто добавляете или удаляете элементы из списка. | | | |
| С вызовов | Стек — это структура данных, которая работает по принципу “последним пришел - первым вышел” (LIFO). Это означает, что последний элемент, добавленный в стек, будет первым элементом, который будет удален из стека.  **Стек используется для хранения локальных переменных методов, параметров функций, адресов возврата и временных данных, он управляется компилятором (Гарбичколлектором).**  Концепция очень проста и довольно много предметов из нашей повседневной жизни “реализованы” как стек. Переменные внутри методов хранятся в стеке.  Представь себе простую ситуацию: ты живешь в гостинице, и в течение дня тебе поступали деловые письма. Поскольку тебя в это время не было в номере, служащий гостиницы просто складывал приходящие письма на твой стол. Сначала он положил на стол первое письмо. Потом пришло второе, и он положил его поверх первого. Третье пришедшее письмо он положил поверх второго, а четвертое — поверх третьего.  А теперь, ответь на простой вопрос: какое письмо ты прочтешь первым, когда придешь в номер и увидишь стопку на столе? Правильно, ты прочитаешь верхнее письмо. То есть то, которое пришло последним по времени. Именно так и работает стек. | | | |
| Куча | **Куча** — это область памяти, в которой хранятся объекты, созданные с помощью оператора new, а также поля класса. Когда вы создаете объект с помощью оператора new, он выделяется в куче и ссылка на этот объект возвращается из оператора new. Объекты в куче не удаляются автоматически, и вы должны явно удалить их с помощью оператора delete. | | | |
| Очередь | **Очередь** (или, по-английски, “Queue”) — еще одна распространенная структура данных. Также как стек она реализована во многих языках программирования, в том числе и в Java. В чем же отличие очереди от стека? Ее очередь основана не на LIFO, а на другом принципе — FIFO (“first in — first out”, “первым вошел — первым вышел”). Его легко понять, взяв для примера...да хотя бы обычную, настоящую очередь из реальной жизни! Например, очередь в магазин.  Если в очереди стоит пять человек, первым в магазин попадет тот, кто встал в очередь первым. Если еще один человек (помимо пяти в очереди) захочет что-то купить и встанет в очередь, то в магазин он попадает последним, то есть шестым. При работе с очередью новые элементы добавляются в конец, а если ты хочешь получить элемент, он будет взят из начала. Принцип работы очереди очень легко понять интуитивно, поскольку она часто встречается в реальной жизни. Стоит отдельно заметить, что в Java очередь представлена не классом, а интерфейсом — Queue. Но вместе с тем, очередь в Java — это интерфейс, у которого есть очень много реализаций. | | | |
| Хэш таблицы | **Хеш-таблица** - это структура данных, которая позволяет хранить пары ключ-значение и выполнять операции вставки, поиска и удаления элементов за время, близкое к константе O(1).  Хэш-таблица – это объединение массива и хэш функции.  Хеш-таблица состоит из массива, где каждый элемент массива является указателем на связанный список. Даже если в списке находится только один элемент, это все равно будет связанный список. При вставке нового элемента, вычисляется хеш-код ключа, который определяет индекс массива, по которому будет производиться поиск нужного списка. Если в списке уже есть элемент с таким же ключом, то значение этого элемента будет заменено новым значением. Если такого элемента нет, то новый элемент будет добавлен в конец списка.  Для того, чтобы быстро находить нужный список, используется хеш-функция, которая преобразует ключ в индекс массива. Хорошая хеш-функция должна равномерно распределять элементы по ячейкам массива, чтобы избежать коллизий (ситуации, когда два разных ключа имеют одинаковый хеш-код и попадают в одну ячейку массива).  Если коллизии не избежать, то элементы с одинаковыми ключами будут храниться в одном списке, что может снижать производительность при поиске. Для решения этой проблемы используются различные методы разрешения коллизий, например, метод цепочек, когда элементы с одинаковым хеш-кодом хранятся в связанном списке, или метод открытой адресации, когда при коллизии производится поиск следующей свободной ячейки массива.  Хеш-таблицы часто используются в различных приложениях, таких как базы данных, поисковые системы, хранение паролей и т.д., где необходимо быстро находить и обрабатывать данные по ключу. По мимо этого их используют в кэшировании страниц т.е. в хэш таблицу сохраняется ключ значение, где ключ – это урла страницы, а значение это код самой страницы.  Хэш таблицы в среднем случае работают быстро, они ищут элемент также быстро как массивы, а удаляют и вставляют также быстро, как связанные списки. НО! В худшем случае они выполняют эти операции медленно, а худший случай — это наличие тех самых коллизий. Чтобы их избежать нужно: -соблюдать низкий коэф заполнения (Он рассчитывается, как кол-во элементов в хэш таблице разделить на общее количество элементов и если коэф больше 0,7, то пора расширять хэш таблицу);  -хорошая хэш функция; | | | |
| Графы | Граф используют для моделирования связей между объектами. Он состоит из узлов и ребер. Узел может быть напрямую соединен с несколькими другими узлами такие узлы называются соседям.  Графы бывают: Неориентированный граф, это подмножество пар (V, E), где V — это подмножество вершин, а Е — подмножество ребер графа, которые соединяют эти вершины. Чтобы откинуть всякие сложные описания и термины, легче всего представить граф как две точки A и B, соединённые ребром. В неориентированном графе по ребру из точки A можно попасть в точку B, и из точки B в точку А.  В ориентированном графе все тоже самое, кроме того, что ребра здесь уже называются дуги, и они четко соединяют одну точку с другой. К примеру, точку A с точкой B, что значит, что с точки A в точку B попасть по дуге можно, а вот из точки B в точку A — тут уж шиш. Конечно ж из точки B в точку A тоже может быть дуга :)  Если у ребер графа есть веса, то такой граф называется взвешенным, если нет, то невзвешенным. | | | |
| Бинарное дерево | Бинарное дерево — это структура данных, которая представляет собой дерево, состоящее из узлов, где каждый узел имеет не более двух потомков (детей). Эти узлы связаны друг с другом в определенном порядке, таким образом, что каждый узел имеет максимум двух потомков, которые называются левым и правым потомками.  Как правило, узел дерева содержит некоторую информацию, которая может быть числом, строкой или объектом. Узлы дерева с числовой информацией обычно используются в алгоритмах сортировки и поиска данных.  Для добавления элемента в бинарное дерево нужно сначала определить, в каком порядке будут располагаться элементы. Если элемент меньше узла, то он ставится влево, иначе - вправо. Для поиска элемента нужно начать с корня дерева и двигаться по нему влево или вправо, сравнивая значения элемента с текущим значением узла, пока не будет найден искомый элемент.  Примером бинарного дерева может быть структура хранения данных в компьютере, где каждый узел представляет файл или директорию, а левый потомок - это содержимое папки, а правый потомок - это содержимое подпапки. | | | |
| Какие есть структуры данных в java? | Вот список структур данных в Java:   1. Массивы 2. Списки 3. Стеки 4. Очереди 5. Деревья 6. Графы 7. Хэш таблицы   Некоторые из них являются примитивными типами данных в Java, такими как массивы и строки. В Java также есть множество классов коллекций, которые предоставляют реализации различных структур данных. Например, класс ArrayList представляет собой динамический массив, а LinkedList - двусвязный список. | | | |
| Рекурсия | Рекурсия – это когда функция вызывает самого себя.  В каждой рекурсивной функции обязательно должны быть два случая: 1) Базовый – это условие выхода из рекурсии.  2) Рекурсивный – это условие, когда рекурсия продолжается.  Стек поддерживает две операции (извлечение и занесение) и все вызовы рекурсивной функции сохраняются в стеке вызовов. Из-за этого в некоторых ситуациях это может быть более затратно по памяти чем итеративный подход. | | | |
| Как работает Docker? | Docker – это инструмент, который упаковывает код, системные инструменты, среду выполнения, библиотеки, зависимости и файлы конфигурации необходимые для его запуска и упаковывает их в виртуальный контейнер, который может работать на любом ПК, где установлен docker.  Отличие докера от виртуалок в том, что мы не виртуализируем железо и избавляемся от гипервизора и виртуализируем только ОС. Также у нас нет необходимости в гостевых ОС т.к. контейнер использует ядро ОС того сервера, где мы работаем при помощи приложения Docker Engine (Соответственно мы тратим меньше ресурсов).  Важные элементы Docker:  -Докер файл (из докер файла собирается образ) — это инструкция из кода, которая говорит докеру, как создать образ и что там должно быть  -Образ (на основе образа запускается контейнер) – это готовое к запуску приложение, в котором лежит исходный код, библиотеки, инструменты и т.д., они необходимы для его запуска. Образами можно делится и образ какого-нибудь приложения или ОС мы можем взять на DockerHub.  -Контейнер – это экземпляр нашего приложения, который мы запускаем на основании образа. | | | |
| Трансляторы | Транслятор – выполняет функцию перевода в машинный код.  Существует два вида трансляторов: 1) Компилятор – это транслятор, который читает код полностью и создает его эквивалент на машинном языке.  2) Интерпретатор – это транслятор, который читает исходную программу по частям и сразу выполняет прочитанные части. | | | |
| Как работает Java? | WRITE ONCE, RUN ANYWHERE!  1) Создадим класс Human. 2) Сохраним этот файл с расширением “.java” в результате мы получаем sourcefile “[Human.java](https://vk.com/away.php?to=http%3A%2F%2FHuman.java&cc_key=)”. 3) Запускаем код. 4) Он компилируется с помощью javac. 5) На выходе мы имеем byte код, который находится в файле “Human.class”. 6) JVM использует интерпретатор и переводит byte код (Байт-код во многом похож на машинный код, только он использует набор инструкций не реального процессора, а виртуального) в машинный код (JVM выполняет код в зависимости от ОС на которую она установлена т.е. если мы запускаем код на linux, то и JVM нужно скачать для linux). 7) JVM выполняет машинный код. | | | |
| Примитивные типы данных | Типы данных – это множество допустимых значений, а также совокупность операций над ними.  8 примитивных типов данных: -byte; -short; -int (по умолчанию); -long; -float; -double (по умолчанию); -char; -boolean; | | | |
| Ссылочные типы данных | В Java ссылочные типы данных являются одним из двух основных типов данных, вторым являются примитивные типы данных.  Ссылочные типы данных представляют объекты и содержат ссылку на память, где хранятся фактические данные объекта. Вот некоторые особенности ссылочных типов данных в Java:  -Объектное создание: Ссылочные типы данных создаются с помощью ключевого слова new, после которого следует вызов конструктора класса. Например: String str = new String("Hello");  -Передача по ссылке: когда вы передаете ссылочный тип данных в метод или присваиваете его другой переменной, передается сама ссылка на объект, а не его копия. Это означает, что любые изменения, сделанные с помощью одной ссылки, отразятся на всех других ссылках, указывающих на тот же объект.  -Null-ссылки: Ссылочные типы данных могут содержать значение null, которое указывает на отсутствие ссылки на объект. Если переменная ссылочного типа содержит null, попытка доступа к ее методам или полям приведет к возникновению исключения NullPointerException.  -Управление памятью: В Java автоматическое управление памятью осуществляется сборщиком мусора (garbage collector). Он отслеживает ссылки на объекты и автоматически освобождает память, занимаемую объектами, которые больше не доступны. Это позволяет избежать утечек памяти и упрощает работу с объектами.  -Работа с методами и полями: Ссылочные типы данных могут иметь методы и поля, к которым можно обращаться с помощью оператора точки (.). Например, для объекта типа String можно вызывать методы, такие как length(), charAt(), substring() и многие другие.  -Наследование и полиморфизм: Ссылочные типы данных в Java могут использоваться для реализации наследования и полиморфизма. Это означает, что переменная ссылочного типа может хранить объект любого класса, который является подклассом этого типа. Это позволяет создавать гибкие и расширяемые программы.  -Массивы ссылочных типов: В Java можно создавать массивы ссылочных типов данных. Например, можно создать массив объектов типа String или других пользовательских классов. Массивы позволяют хранить и обрабатывать коллекции объектов одного типа.  Ссылочные типы данных в Java предоставляют мощные средства для создания и манипулирования объектами. Они играют важную роль в объектно-ориентированном программировании и позволяют создавать сложные и эффективные приложения. | | | |
| Перегрузка методов | **Перегрузка методов** — это приём программирования, который позволяет разработчику в одном классе для методов с разными параметрами использовать одно и то же имя. В этом случае мы говорим, что метод перегружен.  При перезагрузке можно изменить: 1. Количество параметров  **Public class** **Calculator {}**  **void** **calculate** (**int** number1, **int** number2) **{}**  **void** **calculate** (**int** number1, **int** number2, **int** number3) **{}**  2. Типы параметров  **Public class** **Calculator {}**  **void** **calculate** (**int** number1, **int** number2) **{}**  **void** **calculate** (**double** number1, **double** number2) **{}**  3. Порядок параметров  **Public class** **Calculator {}**  **void** **calculate** (**double** number1, **int** number2) **{}**  **void** **calculate** (**int** number1, **double** number2) **{}**  Правила перегрузки методов:  1. Если изменить модификатор доступа или return type, то метод не overloading.  2. Главное отличие в разных параметрах.  3. Перегрузка конструкторов работает аналогично. | | | |
| Модификаторы доступов | Public – это означает, что элемент виден в любом другом месте нашей программы.  Private – это означает, что элемент виден только внутри класса в котором он находится.  (default) – если не указать модификатор доступа, то устанавливается модификатор доступа по умолчанию в соответствии с ним наш элемент виден только в рамках пакета.  Protected – это означает, что элемент будет виден внутри пакета и в классах наследниках. | | | |
| Модификаторы final и static | **Final** - оно предполагает, что значение не может быть изменено в будущем. Ключевое слово final может применяться для переменной, метода и класса.  Если final применен к переменной, то она становится константой т.е. не может быть изменена и не имеет дефолтного значения.  Если final применен к классу, то его нельзя использовать, как родителя.  Если final применен к методу, то его нельзя переопределить в классах наследниках.  **Static** – этот элемент принадлежит всему классу, а не отдельному объекту. Существуют и могут быть использованы без создания объекта класса.  Особенности: -Статические переменные могут быть созданы только, как переменные класса.  -К статическим полям класса можно получить доступ без создания объекта, используя имя класса (ссылка на объект не нужна).  -Статические поля не сериализуются.  -Static элементы не могут обращаться к instance переменным и методам. | | | |
| Разновидности переменных | **Локальные** – переменные, созданные внутри метода. Область их видимости — это блок кода, в котором они созданы.  **Параметр** – переменная созданная в параметре метода, может использоваться только внутри блока метода в котором была создана.  **Instance** – переменная, которая принадлежит объекту класса и может быть использована внутри класса и методов.  **Статическая** – переменная общая для всего класса, а не для конкретного объекта, может быть использована внутри класса и метода. | | | |
| Ключевое слово this | -this указывает на текущий экземпляр класса, при использовании с переменными и методами.  -this нельзя использовать для обращения к переменным и методам внутри static метода или при присваивании значения static переменной. | | | |
| Что такое Garbage collector? | **Garbage collector** – удаляет объекты из памяти нашего ПК.  **Особенности:**  -Мы можем определить какие объекты будут удалены.  -Мы не знаем конкретный момент удаления объекта из памяти.  -Управление GC полностью на стороне JVM. | | | |
| Когда использовать интерфейс, а когда абстрактный класс? | Если объекты схожи по свойствам (полям класса), то используем – Абстракцию.  Если объекты схожи по поведению (методы), то используем – Интерфейс. | | | |
| Класс String | **Если при объявлении переменной мы используем синтаксис String s1=”asd”; то:**  -Если объект есть в пуле, то объект не будет создан, а переменная будет ссылаться на ранее созданный объект.  -Если объекта не будет в пуле, то будет создан новый объект.  **Если при объявлении переменной мы используем синтаксис String s1 = new String(“asd”); то:**  -Всегда создается новый объект.  -Если такого объекта нет в пуле, то он будет создан и в куче, и в пуле.    String pool является частью кучи, это по сути отдельный контейнер в куче.  **String иммутабельный класс т.е. его значение никак не может быть изменен это обеспечено тем что:**  -String хранит значение в final массиве соответственно его можно инициализировать только один раз.  -String хранит значение в private char массиве, который имеет фиксированную длину.  -Ни один из методов класса String не изменяет индивидуальные элементы массива.  **Методы String:**   |  |  | | --- | --- | | **№** | **Методы с описанием** | | 1 | [char charAt(int index)](https://proglang.su/java/strings-charat) Возвращает символ по указанному индексу. | | 2 | [int compareTo(Object o)](https://proglang.su/java/strings-compareto) Сравнивает данную строку с другим объектом. | | 3 | int compareTo(String anotherString) Сравнивает две строки лексически. | | 4 | [int compareToIgnoreCase(String str)](https://proglang.su/java/strings-comparetoignorecase) Сравнивает две строки лексически, игнорируя регистр букв. | | 5 | [String concat(String str)](https://proglang.su/java/strings-concat) Объединяет указанную строку с данной строкой, путем добавления ее в конце. | | 6 | [boolean contentEquals(StringBuffer sb)](https://proglang.su/java/strings-contentequals) Возвращает значение true только в том случае, если эта строка представляет собой ту же последовательность символов как указанно в буфере строки (StringBuffer). | | 7 | [static String copyValueOf(char[] data)](https://proglang.su/java/strings-copyvalueof) Возвращает строку, которая представляет собой последовательность символов, в указанный массив. | | 8 | static String copyValueOf(char[] data, int offset, int count) Возвращает строку, которая представляет собой последовательность символов, в указанный массив. | | 9 | [boolean endsWith(String suffix)](https://proglang.su/java/strings-endswith) Проверяет заканчивается ли эта строка указанным окончанием. | | 10 | [boolean equals(Object anObject)](https://proglang.su/java/strings-equals) Сравнивает данную строку с указанным объектом. | | 11 | [boolean equalsIgnoreCase(String anotherString)](https://proglang.su/java/strings-equalsignorecase) Сравнивает данную строку с другой строкой, игнорируя регистр букв. | | 12 | [byte getBytes()](https://proglang.su/java/strings-getbytes) Кодирует эту строку в последовательность байтов с помощью платформы charset, сохраняя результат в новый массив байтов. | | 13 | byte[] getBytes(String charsetName Кодирует эту строку в последовательность байтов с помощью платформы charset, сохраняя результат в новый массив байтов. | | 14 | [void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)](https://proglang.su/java/strings-getchars) Копирует символы из этой строки в массив символов назначения. | | 15 | [int hashCode()](https://proglang.su/java/strings-hashcode) Возвращает хэш-код для этой строки. | | 16 | [int indexOf(int ch)](https://proglang.su/java/strings-indexof) Возвращает индекс первого вхождения указанного символа в данной строке. | | 17 | int indexOf(int ch, int fromIndex) Возвращает индекс первого вхождения указанного символа в данной строке, начиная поиск с указанного индекса. | | 18 | int indexOf(String str) Возвращает индекс первого вхождения указанной подстроки в данной строке. | | 19 | int indexOf(String str, int fromIndex) Возвращает индекс первого вхождения указанной подстроки в данной строке, начиная с указанного индекса. | | 20 | [String intern()](https://proglang.su/java/strings-intern) Возвращает каноническое представление для строкового объекта. | | 21 | [int lastIndexOf(int ch)](https://proglang.su/java/strings-lastindexof) Возвращает индекс последнего вхождения указанного символа в этой строке. | | 22 | int lastIndexOf(int ch, int fromIndex) Возвращает индекс последнего вхождения указанного символа в этой строке, начиная обратный поиск с указанного индекса. | | 23 | int lastIndexOf(String str) Возвращает индекс последнего вхождения указанной подстроки в данной строке. | | 24 | int lastIndexOf(String str, int fromIndex) Возвращает индекс последнего вхождения указанной подстроки в этой строке, начиная обратный поиск с указанного индекса. | | 25 | [int length()](https://proglang.su/java/strings-length) Возвращает длину строки. | | 26 | [boolean matches(String regex)](https://proglang.su/java/strings-matches) Сообщает, соответствует ли или нет эта строка заданному регулярному выражению. | | 27 | [boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)](https://proglang.su/java/strings-regionmatches) Проверяет равны ли две области строки. | | 28 | boolean regionMatches(int toffset, String other, int ooffset, int len) Проверяет равны ли две области строки. | | 29 | [String replace(char oldChar, char newChar)](https://proglang.su/java/strings-replace) Возвращает новую строку, в результате, заменив все вхождения oldChar в этой строке на newChar. | | 30 | [String replaceAll(String regex, String replacement)](https://proglang.su/java/strings-replaceall) Заменяет каждую подстроку строки, соответствующей заданному регулярному выражению с данной заменой. | | 31 | [String replaceFirst(String regex, String replacement)](https://proglang.su/java/strings-replacefirst) Заменяет первые подстроки данной строки, которая соответствует заданному регулярному выражению с данной заменой. | | 32 | [String[] split(String regex)](https://proglang.su/java/strings-split) Разделяет эту строку, окружая данным регулярным выражением. | | 33 | String[] split(String regex, int limit) Разделяет эту строку, окружая данным регулярным выражением. | | 34 | [boolean startsWith(String prefix)](https://proglang.su/java/strings-startswith) Проверяет, начинается ли эта строка с заданного префикса. | | 35 | boolean startsWith(String prefix, int toffset) Проверяет, начинается ли эта строка с указанного префикса, начиная с указанного индекса. | | 36 | [CharSequence subSequence(int beginIndex, int endIndex)](https://proglang.su/java/strings-subsequence) Возвращает новую последовательность символов, которая является подпоследовательностью этой последовательности. | | 37 | [String substring(int beginIndex)](https://proglang.su/java/strings-substring) Возвращает новую строку, которая является подстрокой данной строки. | | 38 | String substring(int beginIndex, int endIndex) Возвращает новую строку, которая является подстрокой данной строки. | | 39 | [char[] toCharArray()](https://proglang.su/java/strings-tochararray) Преобразует эту строку в новый массив символов. | | 40 | [String toLowerCase()](https://proglang.su/java/strings-tolowercase) Преобразует все символы в данной строке в нижний регистр, используя правила данного языкового стандарта. | | 41 | String toLowerCase(Locale locale) Преобразует все знаки в данной строке в нижний регистр, используя правила данного языкового стандарта. | | 42 | [String toString()](https://proglang.su/java/strings-tostring) Этот объект (который уже является строкой!) возвращает себя. | | 43 | [String toUpperCase()](https://proglang.su/java/strings-touppercase) Преобразует все символы в строке в верхний регистр, используя правила данного языкового стандарта. | | 44 | String toUpperCase(Locale locale) Преобразует все символы в строке в верхний регистр, используя правила данного языкового стандарта. | | 45 | [String trim()](https://proglang.su/java/strings-trim) Возвращает копию строки с пропущенными начальными и конечными пробелами. | | 46 | [static String valueOf(primitive data type x)](https://proglang.su/java/strings-valueof) Возвращает строковое представление переданного типа данных аргумента. | | | | |
| Класс StringBuilder | Объект этого класса используется, когда мы знаем заранее, что наша строка будет часто меняться. С помощью обычного String это будет очень затратно по памяти т.к. класс String иммутабельный и такая операция str1+str2 будет создана новая строка, а не изменена старая.  Класс StringBuilder мутабельный т.к. в его основе лежит не final массив и его методы направлены на изменения объекта класса.  По дефолту капасити (вместимость) объекта класса StringBuilder, если в конструктор ничего не передано, то выделяется массив размером 16 ячеек и по необходимости он расширяется до нужного размера путем создания нового массива.  **Методы StringBuilder:**   |  |  | | --- | --- | | 1 | [public StringBuffer append(String s)](https://proglang.su/java/strings-stringbuilder-stringbuffer-append) Обновляет значение объекта, который вызывает метод. Этот метод принимает boolean, char, int, long, Strings и т.д. | | 2 | [public StringBuffer reverse()](https://proglang.su/java/strings-stringbuilder-stringbuffer-reverse) Этот метод изменяет значение объекта StringBuffer, который вызывает метод. | | 3 | [public delete(int start, int end)](https://proglang.su/java/strings-stringbuilder-stringbuffer-delete) Удаляет строку, начиная с начального индекса до конечного индекса. | | 4 | [public insert(int offset, int i)](https://proglang.su/java/strings-stringbuilder-stringbuffer-insert) Этот метод вставляет строку s в позицию упомянутую по смещению. | | 5 | [replace(int start, int end, String str)](https://proglang.su/java/strings-stringbuilder-stringbuffer-replace) Этот метод заменяет символы в подстроке данного StringBuffer символами в указанной строке. |   Список других методов (за исключением методов set), которые очень похожи на [методы класса строк](https://proglang.su/java/strings):   |  |  | | --- | --- | | **№** | **Описание** | | 1 | int capacity() Возвращает текущую вместимость буфера String. | | 2 | char charAt(int index) Возвращается указанный символ последовательности, в настоящее время представленный буфером строки, указанный индексом аргумент. | | 3 | void ensureCapacity(int minimumCapacity) Гарантирует вместимость буфера, по крайней мере равным указанному минимуму. | | 4 | void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Символы копируются из этого буфера строки в символьный массив назначения dst. | | 5 | int indexOf(String str) Возвращает индекс в данной строке первого вхождения указанной подстроки. | | 6 | int indexOf(String str, int fromIndex) Возвращает индекс в данной строке первого вхождения указанной подстроки, начиная с указанного индекса. | | 7 | int lastIndexOf(String str) Возвращает индекс в данной строке последнего вхождения указанной подстроки. | | 8 | int lastIndexOf(String str, int fromIndex) Возвращает индекс в данной строке последнего вхождения указанной подстроки, начиная с указанного индекса. | | 9 | int length() Возвращает длину строкового буфера (количество символов). | | 10 | void setCharAt(int index, char ch) Символ с указанным индексом этого буфера строки имеет значение ch. | | 11 | void setLength(int newLength) Устанавливает длину буфера строки (Stringbuffer). | | 12 | CharSequence subSequence(int start, int end) Возвращает новую последовательность символов, которая является подпоследовательностью этой последовательности. | | 13 | String substring(int start) Возвращает новую строку, которая содержит подпоследовательность символов в данный момент содержащихся в StringBuffer. Подстрока начинается с указанного индекса и продолжается до конца StringBuffer. | | 14 | String substring(int start, int end) Возвращает новую строку, которая содержит подпоследовательность символов в данный момент содержащихся в этом StringBuffer. | | 15 | String toString() Преобразование в строку, представляющую данные в этой строке буфера. | | | | |
| **Класс StringBuffer** | StringBuffer отличается от StringBuiler тем, что StringBuffer безопасен для потоков, но при этом он медленнее чем StringBuiler. Рекомендуется использовать StringBuilder всякий раз, когда это возможно, потому что он быстрее, чем StringBuffer в Java. Однако, если необходима безопасность потоков, наилучшим вариантом являются объекты StringBuffer. | | | |
| **Массивы** | Массив хранит определенное количество элементов одного типа данных.  **Особенности:**  -Длину массива изменить нельзя.  -В памяти объекты хранятся друг за другом.  -Массив — это объект.  **Методы:**  -sort() – сортировка массива.  -binarySearch(array, value) – выведет индекс value, но только если массив отсортирован.  -length() – возвращает длину массива. | | | |
| **Класс ArrayList** | **LinkedList** – это структура данных, которая содержит в себе не упорядоченные, дублированные элементы. Каждый элемент(Нода) содержит ссылку на следующий.  **ArrayList** – это массив, который может изменять своб длину. Его нужно использовать, когда необходим массив и при этом нужно проводить CRUD с его элементами. Под капотом массив Object. Хранит в себе все кроме примитивов.  Методы ArrayList:   |  |  | | --- | --- | | **№** | **Метод и описание** | | 1 | **void add(int index, Object element)** Вставляет указанный элемент в указанный индекс позиции в этом списке. Выбрасывает IndexOutOfBoundsException, если указанный индекс выходит за допустимые пределы (index < 0 || index > size()). | | 2 | **boolean add(Object o)** Добавляет указанный элемент в конец этого списка. | | 3 | **boolean addAll(Collection c)** Добавляет все элементы в указанной коллекции в конец этого списка в том порядке, в котором они возвращаются итератором указанной коллекции. Вызывает NullPointerException, если указанная коллекция имеет значение null (нулевая). | | 4 | **boolean addAll(int index, Collection c)** Вставляет все элементы указанной коллекции в этот список, начиная с указанной позиции. Вызывает NullPointerException, если указанная коллекция имеет значение null. | | 5 | **void clear()** Удаляет все элементы из этого списка. | | 6 | **Object clone()** Возвращает неглубокую копию этого массива. | | 7 | **boolean contains(Object o)** Возвращает true, если этот список содержит указанный элемент. Если точнее, то возвращает true тогда и только тогда, когда этот список содержит хотя бы один элемент e такой, что (o==null ? e==null : o.equals(e)). | | 8 | **void ensureCapacity(int minCapacity)** Увеличивает емкость этого экземпляра ArrayList, если это необходимо, чтобы гарантировать, что он может удерживать, по крайней мере, количество элементов, заданных аргументом минимальной емкости. | | 9 | **Object get(int index)** Возвращает элемент в указанной позиции в этом списке. Вызывает IndexOutOfBoundsException, если указанный индекс выходит за допустимые пределы (index < 0 || index >= size()). | | 10 | **int indexOf(Object o)** Возвращает индекс в этом списке первого упоминания указанного элемента или -1, если List не содержит этот элемент. | | 11 | **int lastIndexOf(Object o)** Возвращает индекс в этом списке последнего упоминания указанного элемента или -1, если List не содержит этот элемент. | | 12 | **Object remove(int index)** Удаляет элемент в указанной позиции в этом списке. Вызывает IndexOutOfBoundsException, если индекс выходит за допустимые пределы (index < 0 || index >= size()). | | 13 | **protected void removeRange(int fromIndex, int toIndex)** Удаляет из этого списка все элементы, индекс которых находится между fromIndex, включительно, и toIndex, исключительно. | | 14 | **Object set(int index, Object element)** Заменяет элемент в указанной позиции в этом списке указанным элементом. Вызывает IndexOutOfBoundsException, если указанный индекс выходит за допустимые пределы (index < 0 || index >= size()). | | 15 | **int size()** Возвращает количество элементов в этом списке. | | 16 | **Object[] toArray()** Возвращает массив, содержащий все элементы в этом списке в правильном порядке. Вызывает NullPointerException, если указанный массив равен null (нулевой). | | 17 | **Object[] toArray(Object[] a)** Возвращает массив, содержащий все элементы в этом списке в правильном порядке; тип выполнения возвращаемого массива - тип указанного массива. | | 18 | **void trimToSize()** Обрезает емкость этого экземпляра ArrayList до его текущих размеров. | | | | |
| **Методы Arrays** | -asList() – создает List из массива.  -compare() – сравнивает массивы по длине.  -mismath() – возвращает индекс первого расхождения.  -sort() - выполняет сортировку массива.  -соруOf() – копирует массив. | | | |
| Garbage Collector | **Garbage Collector** – механизм сборки мусора.  **Мы знаем какие объекты будут удалены, но когда не знаем.**  **Методы:**  -System.gc() – советуем GC чтобы он сработал.  -finalize() – вызывается перед удалением и говорит что нужно сделать перед этим. | | | |
| Super | **Super** – вызывает конструктор superкласса, который заканчивает свою работу всегда раньше конструктора subкласса.  **Особенности:**  -всегда стоит на первой строке конструктора.  -выражения this и super не могут находится в теле конструктора одновременно. | | | |
| Overriding | **Overriding** – это изменение не статичного, не финализированного метода в subклассе, который унаследован от superкласса.  **Метод считается Overriding:**  -Имя в subклассе такое же, что и в superклассе.  -Список аргументов в subклассе такое же, что и в superклассе.  -Return type в subклассе такой же, что и в superклассе.  -Метод в subклассе тоже должен быть non-static.  -Access modifier в subклассе должен быть такой же или менее строгий, чем в superклассе.  **Все методы, которые не являются static, final или private можно override.** | | | |
| Отличие абстрактного класса и интерфейса? | 1. **Абстрактный класс** может содержать как абстрактные, так и неабстрактные методы. **Интерфейс** содержит только абстрактные методы. 2. **Абстрактный класс** может содержать поля и реализованные методы. В **интерфейсе** можно объявлять только константы. 3. Класс может расширять только один **абстрактный класс**. Класс может реализовывать несколько **интерфейсов**. 4. **Абстрактный класс** может иметь конструктор. **Интерфейс** не имеет конструктора. 5. От **абстрактного класса** можно создать объект с помощью наследования и реализации всех абстрактных методов. От интерфейса нельзя создать объект, его можно только реализовать. 6. Абстрактные методы в **абстрактном классе** могут иметь модификатор доступа. В **интерфейсе** методы абстрактны по умолчанию и не могут иметь модификатор доступа. 7. **Абстрактный класс** используют, когда нужно предоставить частичную реализацию и определить поведение базового класса. **Интерфейс** - когда нужно только определить контракт класса, не затрагивая реализации.   Таким образом, абстрактный класс более гибкий и функциональный, а интерфейс используется для стандартизации и полиморфизма. Выбор между ними зависит от конкретных потребностей проектирования. | | | |
| Casting | **Casting** в Java - это приведение типа одного объекта к другому типу. Основные виды приведения типов:   1. Приведение вверх (upcasting) - приведение subtype к superclass. Происходит неявно, без явного приведения. Например:   Animal animal = new Cat();  Здесь объект класса Cat неявно приводится к типу Animal.   1. Приведение вниз (downcasting) - приведение superclass к subtype. Требует явного приведения типа. Например:   Animal animal = new Cat(); Cat cat = (Cat) animal;  Здесь объект класса Animal явно приводится к типу Cat.   1. Приведение типов между несвязанными классами. Также требует явного приведения. Например:   String str = "123"; Integer i = (Integer) str;  Основные моменты при приведении типов:   * Приведение вниз может привести к ошибке ClassCastException, если объект не является объектом целевого типа. * Приведение между несвязанными типами опасно и часто является ошибкой проектирования. * Приведение типов активно используется в полиморфизме для работы с объектами по ссылке на их базовый тип. | | | |
| Оператор Instanceof | Оператор **instanceof** в Java используется для проверки, является ли объект экземпляром указанного класса или интерфейса.  **Синтаксис:**  object instanceof Class  Где object - проверяемый объект, Class - класс или интерфейс.  Оператор возвращает значение true, если объект является экземпляром данного класса или интерфейса, и false - если не является.  **Пример:**  String str = "Hello";  if(str instanceof String) { System.out.println("Это строка"); } else if (str instanceof Integer) { System.out.println("Это integer"); } else { System.out.println("Неизвестный тип"); }  Здесь проверяется, является ли объект str экземпляром класса String. Так как это строка, будет выведено "Это строка".  Оператор instanceof часто используется перед приведением типов, чтобы избежать ошибки ClassCastException. Также он позволяет проверить, реализует ли объект указанный интерфейс. | | | |
| Методы equals() и toString() | Методы **equals()** и **toString()** используются в Java для сравнения объектов и представления объекта в виде строки соответственно.  **Метод equals():**   * По умолчанию сравнивает ссылки на объекты. * Может быть переопределен в классе для сравнения объектов по содержимому. * Должен быть рефлексивным, симметричным и транзитивным.   **Пример переопределения:**  public class Person {  String name;  public boolean equals(Object o) { if (o instanceof Person) { Person p = (Person) o; return this.name.equals(p.name); } return false; } }  **Метод toString():**   * По умолчанию возвращает строку с именем класса и хэш-кодом объекта. * Часто переопределяется, чтобы возвращать информативное текстовое представление объекта.   **Пример:**  public class Person {  String name;  public String toString() { return "Person " + this.name; }  }  **Переопределение этих методов важно для удобства отладки и вывода информации об объектах класса.** | | | |
| Wrapper классы | **Wrapper классы (обёртки) в Java** - это классы, которые инкапсулируют примитивные типы данных (int, boolean, char и т.д.). Каждый примитивный тип имеет соответствующий Wrapper класс:   * int -> Integer * double -> Double * boolean -> Boolean * char -> Character   **Основные причины использования Wrapper классов:**   * Предоставление дополнительных полезных методов для примитивных типов, например: conversion between types, validation, math operations. * Возможность использования примитивных типов как объектов, например, для хранения в коллекциях. Коллекции в Java не могут хранить примитивные типы, только объекты. * Возможность работы со значениями примитивных типов как с объектами (передача в методы и т.д.). * Автоматическое преобразование между примитивными типами и Wrapper-объектами (autoboxing/unboxing).   Таким образом, использование Wrapper классов даёт больше возможностей при работе с примитивными типами данных в Java. | | | |
| Лямбда выражения и функциональные интерфейсы | **Лямбда выражения в Java** - это анонимные функции, которые позволяют передавать поведение методов как аргументы другим методам.  Основные черты лямбда выражений:   * Не имеют имени * Могут захватывать внешние переменные * Компактны и элегантны * Передаются как объекты функциональных интерфейсов   **Функциональный интерфейс** - это интерфейс, содержащий только ОДИН абстрактный метод. Лямбда выражения реализуют это единственный метод.  **Пример лямбда выражения:**  (x, y) -> x + y  Это выражение принимает два аргумента x и y, и возвращает их сумму. Оно реализует функциональный интерфейс:  @FunctionalInterface interface Adder { int add(int x, int y); }  И может быть присвоено переменной этого типа:  Adder adder = (x, y) -> x + y;  **Таким образом, лямбда выражения позволяют создавать анонимные функции "на лету", не объявляя отдельные классы. Это упрощает код и делает его более лаконичным.** | | | |
| Generics | **Generics** (обобщения) в Java - это возможность определять классы, интерфейсы и методы с типизированными параметрами. Это позволяет создавать гибкие решения, которые могут работать с разными типами данных.  **Основные возможности generics:**   * Определение классов с параметром типа: class Box<T> { private T data;   public void set(T data) { this.data = data; }  public T get() { return data; } }  Теперь Box может хранить данные любого типа.   * Определение интерфейсов с параметром типа. * Определение методов с параметрами типа.   Преимущества generics:   * Безопасность типов - компилятор проверяет корректность типов. * Устранение необходимости приведения типов. * Возможность пере использования кода для разных типов.   **Generics** улучшают строгость типизации в Java и делают код более универсальным и гибким к расширению. | | | |

***Полезные ссылки***

|  |  |
| --- | --- |
| Сайт О. Назиной | <http://testbase.ru/> |
| ия тестировщика | <https://vladislaveremeev.gitbook.io/qa_bible/> |
| Гайд, как вкатиться в тестирование? | [Гайд, как вкатиться в тестирование?](https://medium.com/@vrode_v_prode/%D0%B3%D0%B0%D0%B9%D0%B4-%D0%BA%D0%B0%D0%BA-%D0%B2%D0%BA%D0%B0%D1%82%D0%B8%D1%82%D1%8C%D1%81%D1%8F-%D0%B2-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-d174a289bf87) |
| Гайд, как вкатиться в автоматизацию? | [Гайд, как вкатиться в автоматизацию?](https://medium.com/@vrode_v_prode/%D0%B3%D0%B0%D0%B9%D0%B4-%D0%BA%D0%B0%D0%BA-%D0%B2%D0%BA%D0%B0%D1%82%D0%B8%D1%82%D1%8C%D1%81%D1%8F-%D0%B2-%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8E-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F-90e3c422da6c#10a4) |
| Как понять ООП от ulbiTV? | <https://www.youtube.com/watch?v=-6DWwR_R4Xk&t=1560s&ab_channel=UlbiTV> |
| Как понять SOLID от ulbiTV? | <https://www.youtube.com/watch?v=TxZwqVTaCmA&ab_channel=UlbiTV> |
| Алгоритмы и структуры данных | <https://www.youtube.com/watch?v=2kSx7wMAhcc&list=PLAma_mKffTOT_qpTFv4KdD9DhOAUd5Rqy&ab_channel=alishev> |
| Основы Java для автоматизации | <https://stepik.org/course/119500/syllabus> |
| Задачи по SQL | <https://sql-academy.org/ru/trainer> |
| Как строить xpath? | <https://www.youtube.com/watch?v=zGYdbHACmLI&ab_channel=OlehPendrak> |
| Собеседования на java junior dev (На QAA примерно такие вопрос) | <https://www.youtube.com/@Jetbulb> |