# Архитектура современных приложений

## Тема 1 Приложения

### Вопрос 1 Консольные приложения

#### Консольные приложения

прикладные приложения, непредусматривающие свой собственный графический интерфейс.

*Примеры консольных приложений FTP клиенты, некоторые файловые менеджеры могут работать в консольном режиме*

**Текстовый пользовательский интерфейс** - Интерфейс, использующий при вводе-выводе информации исключительно символы (буквы, цифры, символы псевдографики)

**Интерфейс командной строки** - Текстовый интерфейс между человеком и компьютером, в котором инструкции компьютеру даются путём ввода с клавиатуры текстовых строк (команд).

**Преимущества**

* Возможность выстраивать приложения в цепочки посредством конвейера
* Нересурсоёмкие

**Недостатки**

* Отсутствие графического интерфейса
* Высокий порог входа (необходимость прочтение документации)

**Консоль** - интерфейс, который использует стандартные потоки ввода - вывода

**Стандартные потоки данных**

* Stdin - поток ввода данных
* Stdout - поток вывода данных
* Stderr - поток вывода ошибок

#### Конвеер

**Конвеер** - цепочка консольных приложений, так называемые скрипты, где перенаправлен вывод из одного приложения в ввод другого

Два процесса могут обмениваться данными через межпроцессный канал - конвейер. По конвейеру можно передавать данные только в одну сторону.

По умолчанию **потоки ввода-вывода связываются с консолью**, с которой запущен процесс.

Канал вывода Предыдущего процесса связывается с каналом ввода Следующего.

**Поток** - это абстрактное понятие, относящееся к любому передвижению данных отисточника к приемнику.

При запуске консольного приложения операционная система автоматически связывает три потока ввода-вывода с консолью: стандартный входной поток, стандартный поток вывода и стандартный поток вывода ошибок.

Приложение может считывать данные, введенные пользователем из стандартного входного потока. запись обычных данных в стандартный выходной поток; и записывают данные об ошибках в стандартный поток вывода ошибок.

Результат выполнения можно выводить не только на экран. Для перенаправления вывода применяется дескрипторы файлов – число, привязанное к одному из файлов ввода-вывода:

0 – стандартный ввод - зарезервирован для чтения команд пользователя или входных данных.

1 – стандартный вывод - зарезервирован для вывода данных, как правило (хотя и не обязательно) текстовых.

2 – стандартный вывод сообщений об ошибках - зарезервирован для вывода диагностических и отладочных сообщений в текстовом виде.

Консольные функции дают возможность разного уровня доступа к консоли. Высокоуровневые консольные функции ввода-вывода данных (I/O) дают возможность приложению читать из их стандартного ввода данных (stdin), чтобы извлечь введённую информации с клавиатуры, сохраненную во входном буфере консоли. Функции также дают возможность программе записать в стандартный вывод (stout)или показать на экране текст стандартной ошибки (sterr) в экранном буфере консоли. Высокоуровневые функции поддерживают также переадресацию стандартных дескрипторов и управление режимами работы консоли для различных функциональных возможностей ввода - вывода

**Перенаправление** — это возможность командной оболочки перенаправлять стандартные потоки в определенное пользователем место, например в файл.

**Перенаправление потоков** - ОС позволяет подменять эти потоки при старте системы

**Дескриптор** - неотрицательное целое число, идентифицирующее какой-либо объект информационной системы.

**Файловый дескриптор (FD)** — абстрактный индикатор (дескриптор), используемый для доступа к файлу или другому ресурсу ввода/вывода, например к каналу или сетевому сокету.

**Операторы командной строки**

**| - оператор конвейеризации -** передавать результат выполнения из одной команды сразу в другую, из той в третью и так далее.

**Пример** dir | find ".txt" - покажет список всех файлов с расширение .txt в текущем каталоге

**dir -** получить список файлов и папок

**find -** поиск в документах, или в тексте ввода и выводит строки, соответствующие условию

#### () - Одиночная скобка

1. Командная группа. Команды в скобках будут выполняться по порядку, открывая новую подоболочку, переменные в скобках не могут использоваться оставшейся частью скрипта. Несколько команд в скобках разделяются точкой с запятой.
2. Также используется для инициализации массива. Такие как:array=(a b c d)

#### (()) - Двойные скобки

1. При выполнении различных операций переноса (например, двоичной, восьмеричной и шестнадцатеричной) выходные результаты автоматически преобразуются в десятичную. Такие как:echo $((16#5f))Результат 95 (от шестнадцатеричного до десятичного)
2. Просто используйте (()), чтобы переопределить значение переменной, напримерa=5; ((a++))$ A можно переопределить до 6

#### [] - Одиночная скобка

1. внутренние команды bash，их тест эквивалентен, Если мы не указываем абсолютный путь, мы обычно используем собственные команды bash.
2. Единственные операторы сравнения, доступные в test и [], это == и! =, Оба из которых используются для сравнения строк, а не для целочисленного сравнения, целочисленное сравнение может использовать только форму -eq, -gt, Ни сравнение строк, ни целочисленное сравнение не поддерживают знаки «больше» и «меньше».
3. **Диапазон символов**, Используется как часть регулярного выражения для описания диапазона совпадающих символов. Обычные правила нельзя использовать в скобках для целей тестирования.
4. В контексте структуры массива квадратные скобки используются для ссылки на номер каждого элемента в массиве.

#### [[]] - Двойные скобки

1. **[[Является ключевым словом языка программирования bash**, Не команда, структура [[]] является более общей, чем структура []. Все символы между [[и]] не будут иметь расширения имени файла или разбиения по словам, но произойдет расширение параметра и подстановка команд.
2. Поддерживает сопоставление с образцом строк и даже поддерживает регулярные выражения оболочки при использовании оператора = ~. При сравнении строк вы можете использовать правильную как образец, а не просто строку, такую ​​как[[ hello == hell? ]], Результат верный. [[]] Соответствует строкам или групповым символам без кавычек.
3. **Используйте [[…]] условия для оценки структуры вместо […]**, Может предотвратить много логических ошибок в скрипте. Например, операторы &&, ||, <и> обычно могут существовать в структуре условного суждения [[]], но если они появляются в структуре [], будет сообщено об ошибке. Например, его можно использовать напрямуюif [[ $a != 1 && $a != 2 ]], Если двойные скобки не применимы, тоif [ $a -ne 1] && [ $a != 2 ]илиif [ $a -ne 1 -a $a != 2 ]

#### {} - Фигурные скобки

1. Первое: разверните список файлов, разделенных запятыми в фигурных скобках. Такие какtouch {a,b}.txtРезультатом является a.txt b.txt.  
   Второй тип: разверните список последовательных файлов, разделенных точками (...) в фигурных скобках, например:touch {a..d}.txtРезультатом является a.txt b.txt c.txt d.txt
2. ${var:-string}с${var:=string}:Если переменная var пустаИспользуется в командной строкеЗамените $ {var: -string} на строку, В противном случае переменная var не пуста, затем замените $ {var: -string} на значение переменной var, правила замены для $ {var: = string} такие же, как и для $ {var: -string} Место это $ {var: = string}Если var пуст, замените $ {var: = string} на строку и присвойте строку переменной var. Одно из распространенных применений $ {var: = string} - определить, назначена ли переменная, и если нет, назначить ей значение по умолчанию.
3. ${var:+string}Правило замены противоположно приведенному выше,То есть, только когда переменная не пуста, она заменяется строкой, Если переменная пуста, она не заменяется и не заменяется значением переменной var, которое является пустым значением. (Поскольку переменная var в настоящее время пуста, эти два оператора эквивалентны)
4. ${var:?string}Правило замены: если переменная var не пуста, замените $ {var :? String} значением переменной var;Если переменная var пуста, выведите строку со стандартной ошибкой и выйдите из скрипта, Мы можем использовать эту функцию, чтобы проверить, установлено ли значение переменной.

### Вопрос 2 Desktop приложения

Desktop приложения – Прикладные приложения, работающие в среде операционной системы. Приложение инсталлируется на рабочую станцию пользователя и запускается локально или удаленно

**Преимущества**

* Наличие графического интерфейса

**Недостатки**

* Отсутствие возможности перенаправление потоков вывода-ввода данных (т.е. Нельзя автоматизировать клики мыши)

**Графический интерфейс пользователя** - Интерфейс, в котором его элементы (меню, кнопки, значки, списки и т. п.), представленные пользователю на дисплее, исполнены в виде графических изображений.

**Оконный интерфейс** - Способ организации полноэкранного интерфейса программы (разновидность графического интерфейса), в котором каждая часть располагается в окне — собственном субэкранном пространстве, находящемся в произвольном месте «над» основным экраном. Несколько окон, одновременно располагающихся на экране, могут перекрываться, виртуально находясь «выше» или «ниже» друг относительно друга.

**Установка**

* создание конфигурационных файлов
* добавление необходимых библиотек
* получение данных от ос о аппаратном обеспечении
* добавление параметров в реестр

**Portable приложения**

* Приложение не инсталлируемые в среду ос
* Не хранят настройки и параметры в реестре
* Не привязанные к определенному устройству

## Тема 2 Архитектура клиент-сервер

### Вопрос 1 Основные понятия

**модель взаимодействия клиент-сервер**, состоит в разделении приложения на несколько компонентов на клиента и на сервер

**Преимущества**

* данные хранятся на сервере, который защищён лучше клиентов
* на сервере проще организовать контроль полномочий
* несколько клиентов пользуются одним набором данных

**Недостатки**

* Отсутствие соединения может сделать неработоспособной всю вычислительную сеть

Идея архитектуры «клиент-сервер» состоит в разделении сетевого приложения на несколько компонентов, каждый из них реализует специфический набор сервисов

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 сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине.

Все данные хранятся на сервере, который, как правило, защищён гораздо лучше большинства клиентов. На сервере проще организовать **контроль полномочий**, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа"

**Неработоспособность сервера** может сделать неработоспособной всю вычислительную сеть

Поддержка работы данной системы требует отдельного специалиста — **системного администратора**

**Клиент**

Интерфейсный (обычно графический) компонент комплекса, предоставляемый конечному пользователю. Этот уровень не должен иметь прямых связей с базой данных (по требованиям безопасности и масштабируемости), быть нагруженным основной бизнес-логикой (по требованиям масштабируемости) и хранить состояние приложения (по требованиям надёжности). На этот уровень обычно выносится только простейшая бизнес-логика: интерфейс авторизации, алгоритмы шифрования, проверка вводимых значений на допустимость и соответствие формату, несложные операции с данными (сортировка, группировка, подсчёт значений), уже загруженными на терминал.

**Сервер приложений**

Располагается на втором уровне, на нём сосредоточена большая часть бизнес-логики. Вне его остаются только фрагменты, экспортируемые на клиента, а также элементы логики, погруженные в базу данных. Реализация данного компонента обеспечивается связующим программным обеспечением. Серверы приложений проектируются таким образом, чтобы добавление к ним дополнительных экземпляров обеспечивало горизонтальное масштабирование производительности программного комплекса и не требовало внесения изменений в программный код приложения.

**Сервер баз данных**

Обеспечивает хранение данных и выносится на отдельный уровень, реализуется, как правило, средствами систем управления базами данных, подключение к этому компоненту обеспечивается только с уровня сервера приложений.

**Принцип работа клиент-серверной архитектуры**

Клиент отправляет запрос на сервер, сервер обрабатывает запрос, обращается к серверу баз данных, сервер баз данных записывает или отдает данные и сервер возвращает ответ клиенту.

**Двухзвенная архитектура** - сервер отвечает на клиентские запросы напрямую и в полном объеме, при этом используя только собственные ресурсы. Т.е. сервер не вызывает сторонние сетевые приложения и не обращается к сторонним ресурсам для выполнения какой-либо части запроса

### Вопрос 2 Толстый клиент

**Толстый клиент** - Программа-клиент обеспечивающая расширенную функциональность независимо от центрального сервера. Часто сервер в этом случае является лишь хранилищем данных, а вся работа по обработке и представлению этих данных переносится на машину клиента.

**Преимущества**

* Частичная возможность работы без связи с сервером;

**Недостатки:**

* Низкая защищенность данных
* Дублирование кода

### Вопрос 3 Тонкий клиент

**Тонкий клиент** - Программа-клиент которая переносит большую часть задач по обработке информации на сервер

**Преимущества**

* Повышенная защищенность данных
* Отсутствие дублирование кода (основная бизнес логика находится на сервере)

**Недостатки:**

* Полная неработоспособность без связи с сервером

## Тема 3 Web Services, RESTful

**веб-сервисы -** Клиент-серверное приложение, в котором клиентом выступает браузер, а сервером web-сервер

**Протоколы веб-сервисов**

* **SOAP**(Simple Object Access Protocol) — по сути это тройка стандартов SOAP/WSDL/UDDI
* **REST**(Representational State Transfer)
* **XML-RPC** (XML Remote Procedure Call)

**REST** означает REpresentational State Transfer (Википедия: «передача состояния представления»)

Вот как обычно реализуется служба REST:

* **Формат обмена данными**: здесь нет никаких ограничений. JSON — очень популярный формат, хотя можно использовать и другие, такие как XML
* **Транспорт**: всегда HTTP. REST полностью построен на основе HTTP.
* **Определение сервиса**: не существует стандарта для этого, а REST является гибким. Это может быть недостатком в некоторых сценариях, поскольку потребляющему приложению может быть необходимо понимать форматы запросов и ответов. Однако широко используются такие языки определения веб-приложений, как WADL (Web Application Definition Language) и Swagger.

# ООП

## Тема 1 Основные понятия

### Вопрос 1 ООП

**ООП** — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют древовидную структуру наследования.

**Объект** – экземпляр класса.

**Класс** – описание как создать объект, - комплексный тип данных, состоящий из набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями) для оперирования своим содержимым (значениями полей).

### Вопрос 2 Абстракция

**Абстракция -**  обобщение с выделением общих признаков.

Абстракция данных позволяет рассматривать необходимые объекты данных и операции, которые должны выполняться над такими объектами, без необходимости вникать в *несущественные детали.*

***Абстрактный класс –*** *это класс, экземпляр(объект) которого нельзя создать.*

Программист создает абстрактный класс с реализованными методами, если нет смысла создавать класс

### Вопрос 3 Наследование

**Наследование –** инструмент ООП, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью для реализации древовидной структуры. (ключевое слово extends)

(Примечание) Без наследования не работает Абстракция и Полиморфизм

Класс, от которого производится наследование, называется **базовым, родительским или суперклассом.**

Новый класс **— потомком, наследником или производным** классом

### Вопрос 4 Инкапсуляция

**Инкапсуляция –** объединение и управление данными. – данные и методы по работе с ними.

### **Вопрос 5 Полиморфизм**

Полиморфизм

* способность системы изменяться в зависимости от входных данных.
* возможность объектов с одинаковой спецификацией иметь различную реализацию

**Параметрический** **(generic)** подразумевает исполнение одного и того же кода для всех **допустимых** типов аргументов на основе поведения, а не значения применима где тип объекта удовлетворяет заданным требованиям поведения.

*private* void metodTest(Class1 a) //Метод может принять объект класса а, или объекты его наследников

{}

**Подтипов (переопределение)** - возможность **использования** экземпляра класса-наследника там, где требуется экземпляр базового класса.

*class* Class1{

*public* void metodA() {

        }

    }

*class* Class2 *extends* Class1{

*public* void metodA(){

        }

    }

*class* main{

*public* *static* void main(*String*[] args) {

            Class1 a = *new* Class2();//Полиморфизм подтипов так как ссылка одного типа ссылается на объект другого

            a.metodA(); //вызываем метод класса2

        }

    }

**Ad hoc (Изменение сигнатуры метода(неистинный) перегрузка)** - несколько объявлений методов с одинаковым именем но с отличающимися параметрам.

**Перегрузка методов(изменение сигнатуры)** – это объявление в классе методов с одинаковыми именами при этом с различными параметрами.

**Переопределение метода** —возможность, позволяющая дочернему классу обеспечивать специфическую реализацию метода, уже реализованного в родительском классе.

### Вопрос 6 Типы взаимоотношений классов

**Ассоциация** – отношения между объектами.

В объектно-ориентированном программировании ассоциация означает отношение между классами объектов, которое позволяет одному экземпляру объекта вызвать другой, чтобы выполнить действие от его имени. Это структурное отношение, поскольку определяет связь между объектами одного рода и объектами другого рода и не моделирует поведение.

**Агрегация** – это вид ассоциации, при котором объекты не сильно зависят друг от друга. (class A(class B)) "- один класс является коллекцией или контейнером других. Причём по умолчанию, агрегацией называют агрегацию по ссылке, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет. (отдел агрегирует работающих в нем сотрудников)

**Композиция** – это вид ассоциации, при котором объекты сильно зависят друг от друга.

Время существования объекта зависит от времени существования родительского объекта (class A{ class B{}})

**Зависимость** - Отношения между двумя объектами, в которых изменение одного может повлиять на другое.

### Вопрос 7 Интерфейс, абстрактный класс

***Интерфейс (реализовывается implement)–*** *это абстрактный тип, который используется для определения поведения, которое классы должны реализовать.*

Интерфейс владеетсовокупностью методов и правил взаимодействия элементов системы. Класс реализующий интерфес берет на себя обязательства, которые обязан выполнить*.*

С точки зрения моделирования наследование классов моделирует отношение «Является» («Кнопка меню» ЯВЛЯЕТСЯ «Кнопкой»), а базовые классы обычно содержат определенный функционал, тесно связанный с функционалом производного класса.

Интерфейсы же, по своей природе обладают меньшей связностью (low coupling), поскольку Интерфейсы также могут моделировать отношение «Является» («Кнопка меню» ЯВЛЯЕТСЯ «Кнопкой»), но могут определять и менее жесткое отношение «Может выполнять роль»"

**Абстрактный** **класс** может содержать (и не содержать) **абстрактные** методы и свойства. **Абстрактный** метод не реализуется для **класса**, в котором описан, однако должен быть реализован для его неабстрактных потомков.

Отличия абстрактного класса от интерфейса в Java

|  | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Ключевое слово, используемое для описания** | Для определения абстрактного класса используется ключевое слово abstract | Для определения интерфейса используется ключевое слово interface |
| **Ключевое слово, используемое для реализации** | Для наследования от абстрактного класса используется ключевое слово extends | Для реализации интерфейса используется ключевое слово implements |
| **Конструктор** | Определение конструктора разрешено | Объявление/определение конструктора запрещено |
| **Реализация методов по умолчанию** | Допустима | Вплоть до Java 8 разрешено только объявлять методы, но реализация их запрещена. Начиная с Java 8 стала допустима реализация **static** методов, а реализация **non-static** методов стала доступна посредством ключевого слова default |
| **Поля** | Может иметь как **static**, так и **non-static** поля (аналогичное относится и к final) | Любое поля интерфейса по умолчанию является public static final (и иных иметь не может) |
|  | Наследовать 1 класс | Реализация множества интерфейсов |

## Тема 2 Основные паттерны ООП

**Паттерн** (от англ. Pattern) — образец, шаблон.

**Порождающие паттерны -** паттерны, которые создают объекты, или позволяют получить доступ к существующим.

### Вопрос 1 Singleton (Одиночка)

это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

1. **Гарантирует наличие единственного экземпляра класса**. Чаще всего это полезно для доступа к какому-то общему ресурсу, например, базе данных.

Вы создали объект, а через некоторое время пробуете создать ещё один. В этом случае хотелось бы получить старый объект, вместо создания нового.

Такое поведение невозможно реализовать с помощью обычного конструктора, так как конструктор класса всегда возвращает новый объект.

1. **Предоставляет глобальную точку доступа**. Это не просто глобальная переменная, через которую можно достучаться к определённому объекту. Глобальные переменные не защищены от записи, поэтому любой код может подменять их значения без вашего ведома.

Паттерн стал настолько известен, что теперь люди называют «одиночками» даже те классы, которые решают лишь одну из проблем, перечисленных выше.

**Решение**

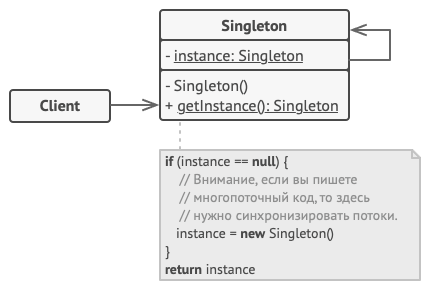
Все реализации одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный статический метод, который и будет контролировать жизненный цикл объекта-одиночки.

Если у вас есть доступ к классу одиночки, значит, будет доступ и к этому статическому методу. Из какой точки кода вы бы его ни вызвали, он всегда будет отдавать один и тот же объект.

**Структура**

Одиночка определяет статический метод getInstance, который возвращает единственный экземпляр своего класса.

Конструктор одиночки должен быть скрыт от клиентов. Вызов метода getInstance должен стать единственным способом получить объект этого класса.



**Пример**

роль **Одиночки** отыгрывает класс подключения к базе данных.

Этот класс не имеет публичного конструктора, поэтому единственный способ получить его объект — это вызвать метод getInstance. Этот метод сохранит первый созданный объект и будет возвращать его при всех последующих вызовах.

// Класс одиночки определяет статический метод `getInstance`,

// который позволяет клиентам повторно использовать одно и то же

// подключение к базе данных по всей программе.

**class** **Database** **is**

// Поле для хранения объекта-одиночки должно быть объявлено

// статичным.

**private** **static** **field** instance: Database

// Конструктор одиночки всегда должен оставаться приватным,

// чтобы клиенты не могли самостоятельно создавать

// экземпляры этого класса через оператор `new`.

**private** **constructor** Database() **is**

// Здесь может жить код инициализации подключения к

// серверу баз данных.

// ...

// Основной статический метод одиночки служит альтернативой

// конструктору и является точкой доступа к экземпляру этого

// класса.

**public** **static** **method** getInstance() **is**

**if** (Database.instance == **null**) **then**

acquireThreadLock() **and** **then**

// На всякий случай ещё раз проверим, не был ли

// объект создан другим потоком, пока текущий

// ждал освобождения блокировки.

**if** (Database.instance == **null**) **then**

Database.instance = **new** Database()

**return** Database.instance

// Наконец, любой класс одиночки должен иметь какую-то

// полезную функциональность, которую клиенты будут

// запускать через полученный объект одиночки.

**public** **method** query(sql) **is**

// Все запросы к базе данных будут проходить через этот

// метод. Поэтому имеет смысл поместить сюда какую-то

// логику кеширования.

// ...

**class** **Application** **is**

**method** main() **is**

Database foo = Database.getInstance()

foo.query("SELECT ...")

// ...

Database bar = Database.getInstance()

bar.query("SELECT ...")

// Переменная "bar" содержит тот же объект, что и

// переменная "foo".

**Применимость**

* Когда в программе должен быть **единственный экземпляр** какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).

Одиночка скрывает от клиентов все способы создания нового объекта, кроме специального метода. Этот метод либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.

* Когда вам хочется иметь **больше контроля над глобальными переменными**.

В отличие от глобальных переменных, Одиночка гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки.

Тем не менее, в любой момент вы можете расширить это ограничение и позволить любое количество объектов-одиночек, поменяв код в одном месте (метод getInstance).

**Шаги реализации**

1. Добавьте в класс приватное статическое поле, которое будет содержать одиночный объект.
2. Объявите статический создающий метод, который будет использоваться для получения одиночки.
3. Добавьте «ленивую инициализацию» (создание объекта при первом вызове метода) в создающий метод одиночки.
4. Сделайте конструктор класса приватным.
5. В клиентском коде замените вызовы конструктора одиночка вызовами его создающего метода.

**Преимущества**

* Гарантирует наличие единственного экземпляра класса.
* Предоставляет к нему глобальную точку доступа.
* Реализует отложенную инициализацию объекта-одиночки.

**Недостатки**

* Нарушает *принцип единственной ответственности класса*.
* Маскирует плохой дизайн.
* Проблемы мультипоточности.
* Требует постоянного создания Mock-объектов при юнит-тестировании.

**Отношения с другими паттернами**

[**Фасад**](https://refactoring.guru/ru/design-patterns/facade) можно сделать [**Одиночкой**](https://refactoring.guru/ru/design-patterns/singleton), так как обычно нужен только один объект-фасад.

Паттерн [**Легковес**](https://refactoring.guru/ru/design-patterns/flyweight) может напоминать [**Одиночку**](https://refactoring.guru/ru/design-patterns/singleton), если для конкретной задачи у вас получилось свести количество объектов к одному. Но помните, что между паттернами есть два кардинальных отличия:

В отличие от Одиночки, вы можете иметь множество объектов-легковесов.

Объекты-легковесы должны быть неизменяемыми, тогда как объект-одиночка допускает изменение своего состояния.

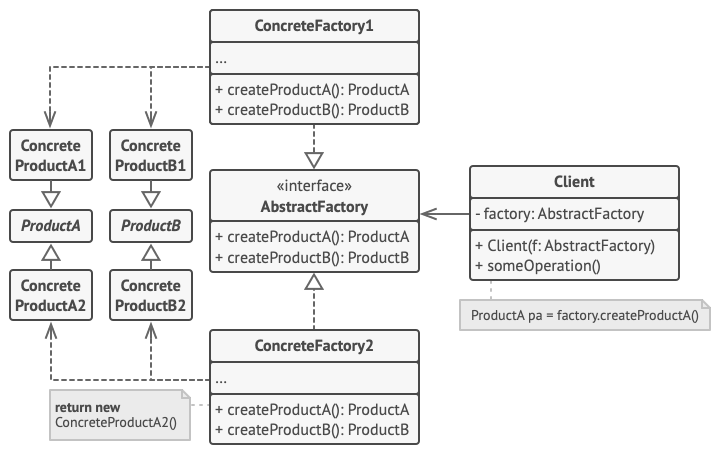
[**Абстрактная фабрика**](https://refactoring.guru/ru/design-patterns/abstract-factory), [**Строитель**](https://refactoring.guru/ru/design-patterns/builder) и [**Прототип**](https://refactoring.guru/ru/design-patterns/prototype) могут быть реализованы при помощи [**Одиночки**](https://refactoring.guru/ru/design-patterns/singleton).

### ****Вопрос 2 Factory (Абстрактная фабрика)****

это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства.

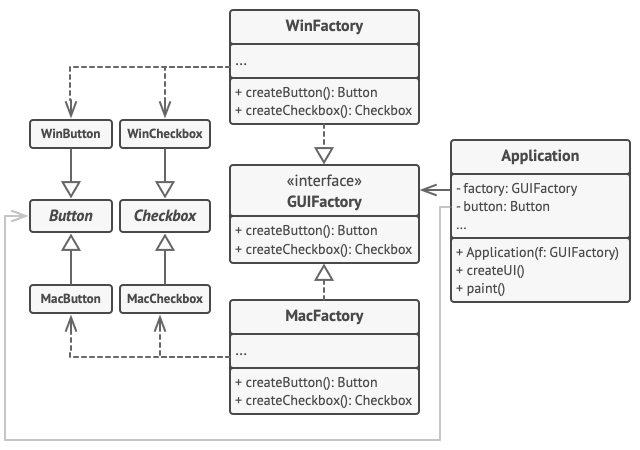
**Структура**



* 1. Абстрактные продукты объявляют интерфейсы продуктов, которые связаны друг с другом по смыслу, но выполняют разные функции.
  2. **Конкретные продукты** — большой набор классов, которые относятся к различным абстрактным продуктам (кресло/столик), но имеют одни и те же вариации (Викторианский/Модерн).
  3. **Абстрактная фабрика** объявляет методы создания различных абстрактных продуктов (кресло/столик).
  4. **Конкретные фабрики** относятся каждая к своей вариации продуктов (Викторианский/Модерн) и реализуют методы абстрактной фабрики, позволяя создавать все продукты определённой вариации.
  5. Несмотря на то, что конкретные фабрики порождают конкретные продукты, сигнатуры их методов должны возвращать соответствующие абстрактные продукты. Это позволит клиентскому коду, использующему фабрику, не привязываться к конкретным классам продуктов. Клиент сможет работать с любыми вариациями продуктов через абстрактные интерфейсы.

**Пример**

 этом примере **Абстрактная фабрика** создаёт кросс-платформенные элементы интерфейса и следит за тем, чтобы они соответствовали выбранной операционной системе.



Пример кросс-платформенного графического интерфейса пользователя.

Кросс-платформенная программа может показывать одни и те же элементы интерфейса, выглядящие чуточку по-другому в различных операционных системах. В такой программе важно, чтобы все создаваемые элементы всегда соответствовали текущей операционной системе. Вы бы не хотели, чтобы программа, запущенная на Windows, вдруг начала показывать чекбоксы в стиле macOS.

Абстрактная фабрика объявляет список создающих методов, которые клиентский код может использовать для получения тех или иных разновидностей элементов интерфейса. Конкретные фабрики относятся к различным операционным системам и создают элементы, совместимые с этой системой.

В самом начале программа определяет, какая из фабрик соответствует текущей операционке. Затем создаёт эту фабрику и отдаёт её клиентскому коду. В дальнейшем клиент будет работать только с этой фабрикой, чтобы исключить несовместимость возвращаемых продуктов.

Клиентский код не зависит от конкретных классов фабрик и элементов интерфейса. Он общается с ними через абстрактные интерфейсы. Благодаря этому клиент может работать с любой разновидностью фабрик и элементов интерфейса.

Чтобы добавить в программу новую вариацию элементов (например, для поддержки Linux), вам не нужно трогать клиентский код. Достаточно создать ещё одну фабрику, производящую эти элементы.

// Этот паттерн предполагает, что у вас есть несколько семейств

// продуктов, находящихся в отдельных иерархиях классов

// (Button/Checkbox). Продукты одного семейства должны иметь

// общий интерфейс.

**interface** **Button** **is**

**method** paint()

// Семейства продуктов имеют те же вариации (macOS/Windows).

**class** **WinButton** **implements** Button **is**

**method** paint() **is**

// Отрисовать кнопку в стиле Windows.

**class** **MacButton** **implements** Button **is**

**method** paint() **is**

// Отрисовать кнопку в стиле macOS.

**interface** **Checkbox** **is**

**method** paint()

**class** **WinCheckbox** **implements** Checkbox **is**

**method** paint() **is**

// Отрисовать чекбокс в стиле Windows.

**class** **MacCheckbox** **implements** Checkbox **is**

**method** paint() **is**

// Отрисовать чекбокс в стиле macOS.

// Абстрактная фабрика знает обо всех абстрактных типах

// продуктов.

**interface** **GUIFactory** **is**

**method** createButton():Button

**method** createCheckbox():Checkbox

// Каждая конкретная фабрика знает и создаёт только продукты

// своей вариации.

**class** **WinFactory** **implements** GUIFactory **is**

**method** createButton():Button **is**

**return** **new** WinButton()

**method** createCheckbox():Checkbox **is**

**return** **new** WinCheckbox()

// Несмотря на то, что фабрики оперируют конкретными классами,

// их методы возвращают абстрактные типы продуктов. Благодаря

// этому фабрики можно взаимозаменять, не изменяя клиентский

// код.

**class** **MacFactory** **implements** GUIFactory **is**

**method** createButton():Button **is**

**return** **new** MacButton()

**method** createCheckbox():Checkbox **is**

**return** **new** MacCheckbox()

// Для кода, использующего фабрику, не важно, с какой конкретно

// фабрикой он работает. Все получатели продуктов работают с

// ними через общие интерфейсы.

**class** **Application** **is**

**private** **field** factory: GUIFactory

**private** **field** button: Button

**constructor** Application(factory: GUIFactory) **is**

**this**.factory = factory

**method** createUI()

**this**.button = factory.createButton()

**method** paint()

button.paint()

// Приложение выбирает тип конкретной фабрики и создаёт её

// динамически, исходя из конфигурации или окружения.

**class** **ApplicationConfigurator** **is**

**method** main() **is**

config = readApplicationConfigFile()

**if** (config.OS == "Windows") **then**

factory = **new** WinFactory()

**else** **if** (config.OS == "Mac") **then**

factory = **new** MacFactory()

**else**

throw **new** Exception("Error! Unknown operating system.")

Application app = **new** Application(factory)

**Применимость**

* Когда бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.

Абстрактная фабрика скрывает от клиентского кода подробности того, как и какие конкретно объекты будут созданы. Но при этом клиентский код может работать со всеми типами создаваемых продуктов, поскольку их общий интерфейс был заранее определён.

* Когда в программе уже используется Фабричный метод, но очередные изменения предполагают введение новых типов продуктов.

В хорошей программе каждый класс отвечает только за одну вещь. Если класс имеет слишком много фабричных методов, они способны затуманить его основную функцию. Поэтому имеет смысл вынести всю логику создания продуктов в отдельную иерархию классов, применив абстрактную фабрику.

**Шаги реализации**

1. Создайте таблицу соотношений типов продуктов к вариациям семейств продуктов.
2. Сведите все вариации продуктов к общим интерфейсам.
3. Определите интерфейс абстрактной фабрики. Он должен иметь фабричные методы для создания каждого из типов продуктов.
4. Создайте классы конкретных фабрик, реализовав интерфейс абстрактной фабрики. Этих классов должно быть столько же, сколько и вариаций семейств продуктов.
5. Измените код инициализации программы так, чтобы она создавала определённую фабрику и передавала её в клиентский код.
6. Замените в клиентском коде участки создания продуктов через конструктор вызовами соответствующих методов фабрики.

**Преимущества**

* Гарантирует сочетаемость создаваемых продуктов.
* Избавляет клиентский код от привязки к конкретным классам продуктов.
* Выделяет код производства продуктов в одно место, упрощая поддержку кода.
* Упрощает добавление новых продуктов в программу.
* Реализует *принцип открытости/закрытости*.

**Недостатки**

* Усложняет код программы из-за введения множества дополнительных классов.
* Требует наличия всех типов продуктов в каждой вариации.

**Отношения с другими паттернами**

* Многие архитектуры начинаются с применения [**Фабричного метода**](https://refactoring.guru/ru/design-patterns/factory-method) (более простого и расширяемого через подклассы) и эволюционируют в сторону [**Абстрактной фабрики**](https://refactoring.guru/ru/design-patterns/abstract-factory), [**Прототипа**](https://refactoring.guru/ru/design-patterns/prototype) или [**Строителя**](https://refactoring.guru/ru/design-patterns/builder) (более гибких, но и более сложных).
* [**Строитель**](https://refactoring.guru/ru/design-patterns/builder) концентрируется на построении сложных объектов шаг за шагом. [**Абстрактная фабрика**](https://refactoring.guru/ru/design-patterns/abstract-factory) специализируется на создании семейств связанных продуктов. Строитель возвращает продукт только после выполнения всех шагов, а Абстрактная фабрика возвращает продукт сразу же.
* Классы [**Абстрактной фабрики**](https://refactoring.guru/ru/design-patterns/abstract-factory) чаще всего реализуются с помощью [**Фабричного метода**](https://refactoring.guru/ru/design-patterns/factory-method), хотя они могут быть построены и на основе [**Прототипа**](https://refactoring.guru/ru/design-patterns/prototype).
* [**Абстрактная фабрика**](https://refactoring.guru/ru/design-patterns/abstract-factory) может быть использована вместо [**Фасада**](https://refactoring.guru/ru/design-patterns/facade) для того, чтобы скрыть платформо-зависимые классы.
* [**Абстрактная фабрика**](https://refactoring.guru/ru/design-patterns/abstract-factory) может работать совместно с [**Мостом**](https://refactoring.guru/ru/design-patterns/bridge). Это особенно полезно, если у вас есть абстракции, которые могут работать только с некоторыми из реализаций. В этом случае фабрика будет определять типы создаваемых абстракций и реализаций.
* [**Абстрактная фабрика**](https://refactoring.guru/ru/design-patterns/abstract-factory), [**Строитель**](https://refactoring.guru/ru/design-patterns/builder) и [**Прототип**](https://refactoring.guru/ru/design-patterns/prototype) могут быть реализованы при помощи [**Одиночки**](https://refactoring.guru/ru/design-patterns/singleton).

### **Вопрос 3 Builder (Строитель)**

это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

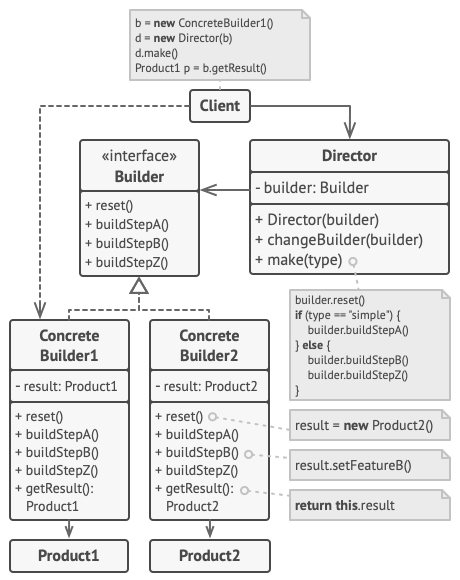
Паттерн Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями.

Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги

Вы можете создать несколько классов строителей, выполняющих одни и те же шаги по-разному. Используя этих строителей в одном и том же строительном процессе, вы сможете получать на выходе различные объекты.

**Структура**

* 1. **Интерфейс строителя** объявляет шаги конструирования продуктов, общие для всех видов строителей
  2. **Конкретные строители** реализуют строительные шаги, каждый по-своему. Конкретные строители могут производить разнородные объекты, не имеющие общего интерфейса.
  3. **Продукт** — создаваемый объект. Продукты, сделанные разными строителями, не обязаны иметь общий интерфейс.
  4. **Директор** определяет порядок вызова строительных шагов для производства той или иной конфигурации продуктов.
  5. Обычно **Клиент** подаёт в конструктор директора уже готовый объект-строитель, и в дальнейшем данный директор использует только его. Но возможен и другой вариант, когда клиент передаёт строителя через параметр строительного метода директора. В этом случае можно каждый раз применять разных строителей для производства различных представлений объектов.



**Пример**

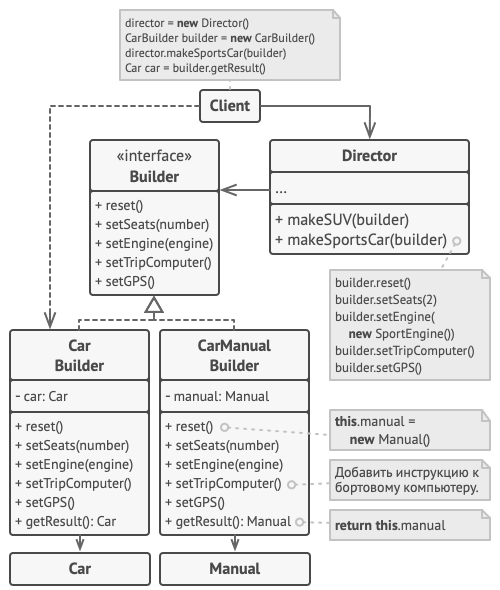
В этом примере **Строитель** используется для пошагового конструирования автомобилей, а также технических руководств к ним.

Автомобиль — это сложный объект, который может быть сконфигурирован сотней разных способов. Вместо того, чтобы настраивать автомобиль через конструктор, мы вынесем его сборку в отдельный класс-строитель, предусмотрев методы для конфигурации всех частей автомобиля.

Клиент может собирать автомобили, работая со строителем напрямую. Но, с другой стороны, он может поручить это дело директору. Это объект, который знает, какие шаги строителя нужно вызвать, чтобы получить несколько самых популярных конфигураций автомобилей.

Но к каждому автомобилю нужно ещё и руководство, совпадающее с его конфигурацией. Для этого мы создадим ещё один класс строителя, который вместо конструирования автомобиля, будет печатать страницы руководства к той детали, которую мы встраиваем в продукт. Теперь, пропустив оба типа строителей через одни и те же шаги, мы получим автомобиль и подходящее к нему руководство пользователя.

Очевидно, что бумажное руководство и железный автомобиль — это две разных вещи, не имеющих ничего общего. По этой причине мы должны получать результат напрямую от строителей, а не от директора. Иначе нам пришлось бы жёстко привязать директора к конкретным классам автомобилей и руководств.



Пример пошагового конструирования автомобилей и инструкций к ним.

// Строитель может создавать различные продукты, используя один

// и тот же процесс строительства.

**class** **Car** **is**

// Автомобили могут отличаться комплектацией: типом

// двигателя, количеством сидений, могут иметь или не иметь

// GPS и систему навигации и т. д. Кроме того, автомобили

// могут быть городскими, спортивными или внедорожниками.

**class** **Manual** **is**

// Руководство пользователя для данной конфигурации

// автомобиля.

// Интерфейс строителя объявляет все возможные этапы и шаги

// конфигурации продукта.

**interface** **Builder** **is**

**method** reset()

**method** setSeats(...)

**method** setEngine(...)

**method** setTripComputer(...)

**method** setGPS(...)

// Все конкретные строители реализуют общий интерфейс по-своему.

**class** **CarBuilder** **implements** Builder **is**

**private** **field** car:Car

**method** reset()

// Поместить новый объект Car в поле "car".

**method** setSeats(...) **is**

// Установить указанное количество сидений.

**method** setEngine(...) **is**

// Установить поданный двигатель.

**method** setTripComputer(...) **is**

// Установить поданную систему навигации.

**method** setGPS(...) **is**

// Установить или снять GPS.

**method** getResult():Car **is**

// Вернуть текущий объект автомобиля.

// В отличие от других порождающих паттернов, где продукты

// должны быть частью одной иерархии классов или следовать

// общему интерфейсу, строители могут создавать совершенно

// разные продукты, которые не имеют общего предка.

**class** **CarManualBuilder** **implements** Builder **is**

**private** **field** manual:Manual

**method** reset()

// Поместить новый объект Manual в поле "manual".

**method** setSeats(...) **is**

// Описать, сколько мест в машине.

**method** setEngine(...) **is**

// Добавить в руководство описание двигателя.

**method** setTripComputer(...) **is**

// Добавить в руководство описание системы навигации.

**method** setGPS(...) **is**

// Добавить в инструкцию инструкцию GPS.

**method** getResult():Manual **is**

// Вернуть текущий объект руководства.

// Директор знает, в какой последовательности нужно заставлять

// работать строителя, чтобы получить ту или иную версию

// продукта. Заметьте, что директор работает со строителем через

// общий интерфейс, благодаря чему он не знает тип продукта,

// который изготовляет строитель.

**class** **Director** **is**

**method** constructSportsCar(builder: Builder) **is**

builder.reset()

builder.setSeats(2)

builder.setEngine(**new** SportEngine())

builder.setTripComputer(**true**)

builder.setGPS(**true**)

// Директор получает объект конкретного строителя от клиента

// (приложения). Приложение само знает, какого строителя нужно

// использовать, чтобы получить определённый продукт.

**class** **Application** **is**

**method** makeCar() **is**

director = **new** Director()

CarBuilder builder = **new** CarBuilder()

director.constructSportsCar(builder)

Car car = builder.getResult()

CarManualBuilder builder = **new** CarManualBuilder()

director.constructSportsCar(builder)

// Готовый продукт возвращает строитель, так как

// директор чаще всего не знает и не зависит от

// конкретных классов строителей и продуктов.

Manual manual = builder.getResult()

**Применимость**

* **Когда вы хотите избавиться от «телескопического конструктора».**

Допустим, у вас есть один конструктор с десятью опциональными параметрами. Его неудобно вызывать, поэтому вы создали ещё десять конструкторов с меньшим количеством параметров. Всё, что они делают — это переадресуют вызов к базовому конструктору, подавая какие-то значения по умолчанию в параметры, которые пропущены в них самих.

**class** **Pizza** {

Pizza(**int** size) { ... }

Pizza(**int** size, **boolean** cheese) { ... }

Pizza(**int** size, **boolean** cheese, **boolean** pepperoni) { ... }

// ...

Такого монстра можно создать только в языках, имеющих механизм перегрузки методов, например, C# или Java.

Паттерн Строитель позволяет собирать объекты пошагово, вызывая только те шаги, которые вам нужны. А значит, больше не нужно пытаться «запихнуть» в конструктор все возможные опции продукта.

* **Когда ваш код должен создавать разные представления какого-то объекта. Например, деревянные и железобетонные дома.**

Строитель можно применить, если создание нескольких представлений объекта состоит из одинаковых этапов, которые отличаются в деталях.

Интерфейс строителей определит все возможные этапы конструирования. Каждому представлению будет соответствовать собственный класс-строитель. А порядок этапов строительства будет задавать класс-директор.

* **Когда вам нужно собирать сложные составные объекты, например, деревья**[**Компоновщика**](https://refactoring.guru/ru/design-patterns/composite)**.**

Строитель конструирует объекты пошагово, а не за один проход. Более того, шаги строительства можно выполнять рекурсивно. А без этого не построить древовидную структуру, вроде Компоновщика.

Заметьте, что Строитель не позволяет посторонним объектам иметь доступ к конструируемому объекту, пока тот не будет полностью готов. Это предохраняет клиентский код от получения незаконченных «битых» объектов.

**Шаги реализации**

1. Убедитесь в том, что создание разных представлений объекта можно свести к общим шагам.
2. Опишите эти шаги в общем интерфейсе строителей.
3. Для каждого из представлений объекта-продукта создайте по одному классу-строителю и реализуйте их методы строительства.
4. Не забудьте про метод получения результата. Обычно конкретные строители определяют собственные методы получения результата строительства. Вы не можете описать эти методы в интерфейсе строителей, поскольку продукты не обязательно должны иметь общий базовый класс или интерфейс. Но вы всегда сможете добавить метод получения результата в общий интерфейс, если ваши строители производят однородные продукты с общим предком.
5. Подумайте о создании класса директора. Его методы будут создавать различные конфигурации продуктов, вызывая разные шаги одного и того же строителя.
6. Клиентский код должен будет создавать и объекты строителей, и объект директора. Перед началом строительства клиент должен связать определённого строителя с директором. Это можно сделать либо через конструктор, либо через сеттер, либо подав строителя напрямую в строительный метод директора.
7. Результат строительства можно вернуть из директора, но только если метод возврата продукта удалось поместить в общий интерфейс строителей. Иначе вы жёстко привяжете директора к конкретным классам строителей.

**Преимущества**

* Позволяет создавать продукты пошагово.
* Позволяет использовать один и тот же код для создания различных продуктов.
* Изолирует сложный код сборки продукта от его основной бизнес-логики.

**Недостатки**

* Усложняет код программы из-за введения дополнительных классов.
* Клиент будет привязан к конкретным классам строителей, так как в интерфейсе директора может не быть метода получения результата.

**Отношения с другими паттернами**

* Многие архитектуры начинаются с применения [**Фабричного метода**](https://refactoring.guru/ru/design-patterns/factory-method) (более простого и расширяемого через подклассы) и эволюционируют в сторону [**Абстрактной фабрики**](https://refactoring.guru/ru/design-patterns/abstract-factory), [**Прототипа**](https://refactoring.guru/ru/design-patterns/prototype) или [**Строителя**](https://refactoring.guru/ru/design-patterns/builder) (более гибких, но и более сложных).
* [**Строитель**](https://refactoring.guru/ru/design-patterns/builder) концентрируется на построении сложных объектов шаг за шагом. [**Абстрактная фабрика**](https://refactoring.guru/ru/design-patterns/abstract-factory) специализируется на создании семейств связанных продуктов. Строитель возвращает продукт только после выполнения всех шагов, а Абстрактная фабрика возвращает продукт сразу же.
* [**Строитель**](https://refactoring.guru/ru/design-patterns/builder) позволяет пошагово сооружать дерево [**Компоновщика**](https://refactoring.guru/ru/design-patterns/composite).
* Паттерн [**Строитель**](https://refactoring.guru/ru/design-patterns/builder) может быть построен в виде [**Моста**](https://refactoring.guru/ru/design-patterns/bridge): директор будет играть роль абстракции, а строители — реализации.
* [**Абстрактная фабрика**](https://refactoring.guru/ru/design-patterns/abstract-factory), [**Строитель**](https://refactoring.guru/ru/design-patterns/builder) и [**Прототип**](https://refactoring.guru/ru/design-patterns/prototype) могут быть реализованы при помощи [**Одиночки**](https://refactoring.guru/ru/design-patterns/singleton).

## Тема 3 Принципы SOLID, DI, IoC

**SOLID**

**SRP-Принцип единственной ответственности.**

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

**OCP-Принцип открытости/закрытости.**

Программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.

Т.е. в методе мы используем не конкретную сущность, а абстрактную, чтобы не пришлось переписывать весь метод

**LSP-Принцип подстановки Барбары Лисков.**

Цель этого принципа заключаются в том, чтобы классы-наследники могли бы использоваться вместо родительских классов, от которых они образованы, не нарушая работу программы. Если оказывается, что в коде проверяется тип класса, значит принцип подстановки нарушается.

Т.е. использовать общий интерфейс, чем наследовать один класс от другого, для того чтобы потомки не противоречили главному поведению родительского класса

**ISP-Принцип разделения интерфейса.**

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

Т.е. используется это потому что мы реализовываем все методы из интерфейса и в случаи дополнение общего интерфейса методами, мы должны переписать все классы, которые наследует этот интерфейс

**DIP-Принцип инверсии зависимостей.**

* Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
* Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Т.е. использовать все классы через интерфейсы, совмещение принципов открытости/закрытости и подстановки Барбары Лисков

**Inversion of Control (инверсия управления)** — это некий абстрактный принцип, набор рекомендаций для написания слабо связанного кода. Суть которого в том, что каждый компонент системы должен быть как можно более изолированным от других, не полагаясь в своей работе на детали конкретной реализации других компонентов.  
**Dependency Injection (внедрение зависимостей)** — это одна из реализаций этого принципа

**внедрением зависимостей (DI)**  сделаем связь между классами более слабой, добавив прослойку в виде интерфейса И будем разрешать ее одним из способов техники DI, а именно **Constructor Injection**  т.е. везде используется интерфейс вместо конкретного класса, например в типе свойства или в типе аргумента метода  
**IoC-контейнер** — это какая-то библиотека, Фреймворк, программа если хотите, которая позволит вам упростить и автоматизировать написание кода с использованием данного подхода на столько, на сколько это возможно  
Они помогают уменьшить количество рутины, позволяя задать соответствие между интерфейсом и его конкретной реализацией, чтобы потом везде этим пользоваться.

# Программирование(Java)

## Тема 1 Основные понятия и типы данных

### Вопрос 1 Объявление

* Это объявление int x;
* Это инициализация x = 2;

**Объявление** (declaration) включает в себя указание идентификатора, типа, а также других аспектов элементов языка, например, переменных и методов.

**Объявление** используется, чтобы уведомить компилятор о существовании элемента.

**Инициализация (**initialization) - создание, активация, подготовка к работе, определение параметров.

**Инициализация переменной** означает явное (или неявное) установление некоторого значения переменной.

**Объявление массива**

* dataType[] arrayName; *//Желательно объявлять так, это Java-стиль*
* dataType arrayName[]; *//Унаследованный от С/С++ способ, работает и в Java*

**Инициализация массива**

* new typeOfArray [length]; *//Где typeOfArray — это тип массива, а length — его длина (то есть, количество ячеек), выраженная в целых числах. Выделили память под массив. После создания массива с помощью new, в его ячейках записаны значения по умолчанию*
* String[] seasons = new String[] {"Winter", "Spring", "Summer", "Autumn"};
* String[] seasons = {"Winter", "Spring", "Summer", "Autumn"};

**Многомерный массив**

* Int[][] myTwoDimentionalArray = new int [8][8];
* int [][] twoDimArray = {{5,7,3,17}, {7,0,1,12}, {8,1,2,3}};

**Инициализация объекта**

Оператор **new** создает экземпляр указанного класса, имеющего конструктор и возвращает ссылку на вновь созданный объект.

new constructor[([arguments])]

constructor - Функция, задающая тип объекта.

Arguments - Список параметров, с которыми будет вызван конструктор.

Point р = new Point();

### Вопрос 2 Области видимости (модификаторы доступа)

* **public**: публичный, общедоступный класс или член класса.

Поля и методы видны другим классам из текущего пакета и из внешних пакетов.

* **protected**: класс или член класса доступен из любого места в текущем классе или пакете, или в производных классах, даже если они находятся в других пакетах
* **Модификатор по умолчанию**. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете.
* **private**: закрытый класс или член класса, противоположность модификатору public. Закрытый класс или член класса доступен только из кода в том же классе.

### Вопрос 3 Boxing и Unboxing

**Автоупаковка**  
Это автоматическая инкапсуляция примитивного типа в эквивалентную ему класс-обёртку всякий раз, когда требуется объект данного типа.

**Autoboxing** происходит:

* При присвоении значения переменной примитивного типа соответствующему классу-обёртки.
* При передаче примитивного типа в параметр метода, ожидающего соответствующий ему класс-обёртку.

**Пример**:

    Integer iOb = 7;

**Автораспаковка**  
Это преобразование класса-обёртки в соответствующий ему примитивный тип.

**Unboxing** происходит:

* При присвоении экземпляра класса-обёртки переменной соответствующего примитивного типа.
* В выражениях, в которых один или оба аргумента являются экземплярами классов-обёрток (кроме операции == и! =).
* При передаче объекта класса-обёртки в метод, ожидающий соответствующий примитивный тип.

**Пример:**

    int i = iOb;

Если в классе обертке null, то при попытке распаковать в примитивный тип будет NullPointerException

### Вопрос 4 Примитивные и Ссылочный типы

**Примитивные типы данных:**   
• **целочисленные**

**byte** (1 байт, -2^7 .. 2^7-1),

**short** (2 байта, -2^15 .. 2^15-1),

**int** (4 байта, -2^31 .. 2^31-1),

**long** (8 байт, -2^63 .. 2^63-1)

*1бит используется под хранение знака, 0 – положительное число*

*по умолчанию все константы в компиляторе int (для long в конце L)*

*C Java 7 между цифрами в числовом литерале может стоять любое количество символов подчеркивания (\_) для читаемости кода**long x = 1\_000\_000L*

**• с плавающей точкой**

**float** (4 байта),

**double** (8байт, двойной точности)

*по умолчанию все константы в компиляторе double (для float в конце F)*

**• логические**

**boolean** (1 бит, true/false)

**• символьные**

**char** (2 байта, 2^16 -1),

*Инициализация ‘a’ – символ, 12 – код символа в десятичной, 0x12 – код символа в шестнадцатеричной, '\u0042' - \u", за которой следует четырехзначное шестнадцатеричное число (в диапазоне от 0000 до FFFF), представляющее собой код символа в Unicode*

*Некоторые символы, отсутствующие на клавиатуре, можно задавать с помощью так называемых escape-последовательностей, содержащих символ "\", за которым следует буквенный символ, идентифицирующий escape-последовательность*

**Ссылочные типы данных**

это любой экземпляр класса, а также массивы: String, Scanner, Random, Die, int [], String [], и т.д.

#### Вопрос 4.1 Типы ссылок

**Сильные ссылки (Strong reference):** Это тип / класс ссылочного объекта по умолчанию. Любой объект, имеющий активную сильную ссылку, не подходит для сборки мусора. Объект является сборщиком мусора, только когда переменная, на которую есть сильные ссылки, указывает на ноль.

    MyClass obj = *new* MyClass();

Здесь объект 'obj' является сильной ссылкой на вновь созданный экземпляр MyClass, в настоящее время объект obj является активным объектом, поэтому не может быть сборщиком мусора.

**Слабые ссылки (Weak Reference):** Слабые ссылочные объекты не являются типом / классом ссылочных объектов по умолчанию, и их следует явно указывать при их использовании.

* Этот тип ссылки используется в WeakHashMap для ссылки на объекты ввода.
* Если JVM обнаруживает объект только со слабыми ссылками (т.е. без сильных или мягких ссылок, связанных с каким-либо объектом), этот объект будет помечен для сборки мусора.
* Для создания таких ссылок используется класс [java.lang.ref.WeakReference](https://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html).
* Эти ссылки используются в приложениях реального времени при установлении DBConnection, который может быть очищен сборщиком мусора, когда приложение, использующее базу данных, закрывается.

Два разных уровней слабости могут быть зачислены: мягкий и фантомный

**Мягкие ссылки (Soft Reference):** в ссылке Soft, даже если объект свободен для сборки мусора, он также не собирает мусор, пока JVM не нуждается в памяти. Объекты удаляются из памяти, когда в JVM заканчивается память. Для создания таких ссылок Используется класс [java.lang.ref.SoftReference](https://docs.oracle.com/javase/7/docs/api/java/lang/ref/SoftReference.html) .

**Фантомные ссылки(Phantom Reference):** Фантомные ссылки возникают, когда объект уже признан мусором, финализирован и находится в процессе зачистки, объекты, на которые ссылаются фантомные ссылки, имеют право на сборку мусора. Но перед удалением их из памяти JVM помещает их в очередь, называемую «ссылочной очередью». Они помещаются в очередь ссылок после вызова метода finalize () для них. Для создания таких ссылок используется класс [java.lang.ref.PhantomReference](https://docs.oracle.com/javase/7/docs/api/java/lang/ref/PhantomReference.html).

#### Вопрос 4.2 Не изменяемы типы данных - Иммутабельность

**Иммутабельность классов**

**Иммутабельный** (неизменяемый, immutable) класс — это класс, который после инициализации не может изменить свое состояние. То есть если в коде есть ссылка на экземпляр иммутабельного класса, то любые изменения в нем приводят к созданию нового экземпляра.

Чтобы класс был иммутабельным, он должен соответствовать следующим требованиям:

* Должен быть объявлен как final, чтобы от него нельзя было наследоваться. Иначе дочерние классы могут нарушить иммутабельность.
* Все поля класса должны быть приватными в соответствии с принципами инкапсуляции.
* Для корректного создания экземпляра в нем должны быть параметризованные конструкторы, через которые осуществляется первоначальная инициализация полей класса.
* Для исключения возможности изменения состояния после инстанцирования, в классе не должно быть сеттеров.
* Для полей-коллекций необходимо делать глубокие копии, чтобы гарантировать их неизменность.

**Иммутабельность строк**

Класс String, представляющий набор символов, вероятно, самый популярный класс в Java. Его назначение — упростить работу со строками, предоставляя различные методы для их обработки.

Например, в классе String есть методы для получения символов, выделения подстрок, поиска, замены и многие другие. Как и другие классы-обертки в Java (Integer, Boolean и т.д.), класс String является иммутабельным.

Иммутабельность строк дает следующие преимущества:

* Строки потокобезопасны.
* Для строк можно использовать специальную область памяти, называемую "пул строк". Благодаря которой две разные переменные типа String с одинаковым значением будут указывать на одну и ту же область памяти.
* Строки отличный кандидат для ключей в коллекциях, поскольку они не могут быть изменены по ошибке.
* Класс String кэширует хэш-код, что улучшает производительность хеш-коллекций, использующих String.
* Чувствительные данные, такие как имена пользователей и пароли, нельзя изменить по ошибке во время выполнения, даже при передаче ссылок на них между разными методами.

**StringBuffer**

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

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

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

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

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

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

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

String domain = ".ru";

// *создадим буфер с помощью String объекта*

StringBuffer buffer = *new* StringBuffer("habrahabr"); // *"habrahabr"*

// *вставим домен в конец*

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

// *удалим домен*

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

// *вставим домен в конец на этот раз используя insert*

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

**StringBuilder** — класс, представляет изменяемую последовательность символов (введен в Java 5) полностью идентичный API с **StringBuffer**. Единственное отличие — **StringBuilder** не синхронизирован (не потокобезопасный). В многопоточности - **StringBuffer**, иначе **StringBuilder** (работает быстрее в большинстве реализаций).

### Вопрос 5 Модель памяти. Сборка мусора GC

Разделение памяти JVM

Память процесса делится на Stack (стек) и Heap (куча) и включает 5 областей:

**Stack**

* **Code Cache** — используемая JVM память при включенной JIT-компиляции; в этой области памяти кэшируется скомпилированный платформенно-зависимый код.

**Heap**

* **Eden Space** — в этой области выделяется память под все создаваемые программой объекты. Жизненный цикл большей части объектов, к которым относятся итераторы, объекты внутри методов и т.п., недолгий.
* **Survivor Space** — здесь хранятся перемещенные из Eden Space объекты после первой сборки мусора. Объекты, пережившие несколько сборок мусора, перемещаются в следующую сборку Tenured Generation.
* **Tenured Generation** хранит долгоживущие объекты. Когда данная область памяти заполняется, выполняется полная сборка мусора.

**Stack**

Стек работает по схеме LIFO (последним вошел, первым вышел). Всякий раз, когда вызывается новый метод, содержащий примитивные значения или ссылки на объекты, то на вершине стека под них выделяется блок памяти.

Когда метод завершает выполнение, блок памяти (frame), отведенный для его нужд, очищается, и пространство становится доступным для следующего метода. При этом поток выполнения программы возвращается к месту вызова этого метода с последующим переходом к следующей строке кода.

* Он заполняется и освобождается по мере вызова и завершения новых методов
* Переменные в стеке существуют до тех пор, пока выполняется метод в котором они были созданы
* Если память стека будет заполнена, Java бросит исключение java.lang.StackOverFlowError
* Доступ к этой области памяти осуществляется быстрее, чем к куче
* Является потокобезопасным, поскольку для каждого потока создается свой отдельный стек

**Metaspace** – новая область памяти, появившаяся в 8 версии Java и заменившая устаревшую PermGen. Основное их отличие заключается в способе распределения памяти.

По умолчанию, Metaspace увеличивается автоматически. Однако и здесь у нас есть возможность управления памятью:

При помощи MetaspaceSize и MaxMetaspaceSize можно задать границыMetaspace области.

В результате появления Metaspace, процесс очистки памяти получил некоторые преимущества. Теперь сборщик мусора автоматически удаляет из памяти ненужные классы, когда емкость, выделенная для хранения метаданных, достигает максимального значения. Вместе с этим, уменьшилась вероятность получения ошибки OutOfMemoryError.

**Область памяти Heap**

Куча Heap является основным сегментом памяти, где хранятся создаваемые объекты. Heap делится на два под сегмента: Tenured (Old) Generation и New Generation. New Generation в свою очередь делится на Eden Space и Survivor.

При создании нового объекта, когда используется оператор 'new', например byte [] data = new byte[1024], этот объект создаётся в сегменте Eden Space. Кроме, собственно данных для массива байт, создается также ссылка (указатель) на эти данные. Если места в сегменте Eden Space уже нет, то JVM выполняет сборку мусора. При сборке мусора объекты, на которые имеются ссылки, не удаляются, а перемещаются из одной области в другую. Так, объекты со ссылками перемещаются из Eden Space в Survivor Space, а объекты без ссылок удаляются.

Если количество используемой Eden Space памяти превышает некоторый заданный объем, то Garbage Collection может выполнить быструю (minor collection) сборку мусора. По сравнению с полной сборкой мусора данный процесс занимает немного времени, и затрагивает только область Eden Space — устаревшие объекты без ссылок удаляются, а выжившие перемещаются в область Survivor Space.

**В чем отличие между сегментами Stack и Heap?**

* Heap (куча) используется всеми частями приложения, а Stack используется только одним потоком исполнения программы.
* Новый объект создается в Heap, а в памяти Stack'a размещается ссылка на него. В памяти стека также размещаются локальные переменные примитивных типов.
* Объекты в куче доступны из любого места программы, в то время, как стековая память не доступна для других потоков.
* Если память стека полностью занята, то Java Runtime вызывает исключение java.lang.StackOverflowError, а если память кучи заполнена, то вызывается исключение java.lang.OutOfMemoryError: Java Heap Space.
* Размер памяти стека, как правило, намного меньше памяти в куче. Из-за простоты распределения памяти (LIFO), стековая память работает намного быстрее кучи.

**Массив основных типов данных**

*Примеры*

Основываясь на рассмотренной ранее информации, рассмотрим пример кода и разберемся, как происходит управление памятью:

*class* Person {

        int id;

        String name;

*public* Person(int id, String name) {

*this*.id = id;

*this*.name = name;

        }

    }

*public* *class* PersonBuilder {

*private* *static* Person buildPerson(int id, String name) {

*return* *new* Person(id, name);

        }

*public* *static* void main(*String*[] args) {

            int id = 23;

            String name = "John";

            Person person = null;

            person = buildPerson(id, name);

        }

    }

**Рассмотрим выполнение кода по шагам:**

1. До начала выполнения метода *main()*, в стеке будет выделено пространство для хранения примитивов и ссылок этого метода:

* примитивное значение ***id*** типа ***int*** будет храниться непосредственно в стеке;
* ссылочная переменная **name** типа **String** будет создана в стеке, но сама строка "John" будет храниться в области, называемой String Pool (является частью Кучи);
* ссылочная переменная ***person*** типа ***Person*** будет также создана в памяти стека, но будет указывать на объект, расположенный в куче;

2. Для вызова конструктора с параметрами *Person (int, String)* из метода *main()* в стеке, поверх предыдущего вызова метода *main(),* будет выделен блок памяти*,* который будет хранить:

* ***this*** — ссылка на текущий объект;
* примитивное значение ***id ;***
* ссылочную переменную ***name*** типа ***String***, которая указывает на объект строки из пула строк;

3. В методе ***main*** дополнительно вызывается метод ***buildPerson*** для которого будет выделен блок памяти в стеке поверх предыдущего вызова. Этот блок снова сохранит переменные способом, описанным выше.

4. Для вновь созданного объекта ***person*** типа Person все переменные будут сохранены в памяти кучи.

**Garbage Collector**

Сборщик мусора Garbage Collector выполняет всего две задачи, связанные с поиском мусора и его очисткой. Для обнаружения мусора существует два подхода:

* Reference counting – учет ссылок;
* Tracing – трассировка.

**Reference counting**

Суть подхода «Reference counting» связана с тем, что каждый объект имеет счетчик, который хранит информацию о количестве указывающих на него ссылок. При уничтожении ссылки счетчик уменьшается. При нулевом значении счетчика объект можно считать мусором.

Главным недостатком данного подхода является сложность обеспечения точности счетчика и «невозможность» выявлять циклические зависимости. Так, например, два объекта могут ссылаться друг на друга, но ни на один из них нет внешней ссылки. Это сопровождается утечками памяти. В этой связи данный подход не получил распространения.

**Tracing**

Главная идея «Tracing» связана с тем, что до «живого» объекта можно добраться из корневых точек (GC Root). Всё, что доступно из «живого» объекта, также является «живым». Если представить все объекты и ссылки между ними как дерево, то необходимо пройти от корневых узлов GC Roots по всем узлам. При этом узлы, до которых нельзя добраться, являются мусором.

Данный подход, обеспечивающий выявление циклических ссылок, используется в виртуальной машине HotSpot VM. Теперь, осталось понять, а что представляет из себя корневая точка (GC Root)? «Источники» говорят, что существуют следующие типы корневых точек:

* Основной Java поток.
* Локальные переменные в основном методе.
* Статические переменные основного класса.

Таким образом, простое java-приложение будет иметь следующие корневые точки:

* Параметры main метода и локальные переменные внутри main метода.
* Поток, который выполняет main.
* Статические переменные основного класса, внутри которого находится main метод.

**Очистка памяти**

Имеется несколько подходов к очистке памяти, которые в совокупности определяют принцип функционирования Garbage Collection.

**Copying collectors**

При использовании «Copying collectors» область памяти делится на две части: в одной части размещаются объекты, а вторая часть остается чистой. На время очистки мусора приложение останавливает работу и запускается сборщик мусора, который находит в первой области объекты со ссылками и переносит их во вторую (чистую) область. После этого, первая область очищается от оставшихся там объектов без ссылок, и области меняются местами.

Главным достоинством данного подхода является плотное заполнение памяти. Недостатком «Copying collectors» является необходимость остановки приложения и размеры двух частей памяти должны быть одинаковыми на случай, когда все объекты остаются «живыми».

**Mark-and-sweep**

При использовании «mark-and-sweep» все объекты размещаются в одном сегменте памяти. Сборка мусора также приостанавливает приложение, и Garbage Collection проходит по дереву объектов, помечая занятые ими области памяти, как «живые». После этого, все не помеченные участки памяти сохраняются в «free list», в которой будут, после завершения сборки мусора, размещаться новые объекты.

К недостаткам данного подхода следует отнести необходимость приостановки приложения. Кроме этого, время сборки мусора, как и время приостановки приложения, зависит от размера памяти. Память становится «решетчатой», и, если не применить «уплотнение», то память будет использоваться неэффективно.

**Generational Garbage Collection**

JVM HotSpot использует алгоритм сборки мусора типа «Generational Garbage Collection», который позволяет применять разные модули для разных этапов сборки мусора. Всего в HotSpot реализовано четыре сборщика мусора:

* Serial Garbage Collection
* Parallel Garbage Collection
* CMS Garbage Collection
* G1 Garbage Collection

**Serial Garbage Collection** относится к одним из первых сборщиков мусора в HotSpot VM. Во время работы этого сборщика приложение приостанавливается и возобновляет работу только после прекращения сборки мусора. В *Serial Garbage Collection* область памяти делится на две части («young generation» и «old generation»), для которых выполняются два типа сборки мусора:

* minor GC – частый и быстрый c областью памяти «young generation»;
* mark-sweep-compact – редкий и более длительный c областью памяти «old generation».

**Алгоритм работы minor GC**

Алгоритм работы *minor GC* очень похож на описанный выше «Copying collectors». Отличие связано с дополнительным использованием области памяти «Eden». Очистка мусора выполняется в несколько шагов:

* приложение приостанавливается на начало сборки мусора;
* «живые» объекты из Eden перемещаются в область памяти «To»;
* «живые» объекты из «From» перемещаются в «To» или в «old generation», если они достаточно «старые»;
* Eden и «From» очищаются от мусора;
* «To» и «From» меняются местами;
* приложение возобновляет работу.

### Доп вопрос 1 JVM

Java — мультиплатформенный язык программирования, это значит, что можно выполнять на любой платформе, где установлена Java Virtual Machine   
  
Java virtual machine (JVM) — это спецификация программного обеспечения, которое выполняет код и предоставляет среду выполнения для этого кода   
  
Предназначение JVM   
• Позволяет запускать Java-приложения на любых устройствах или операционных системах   
• Управляет и оптимизирует память, используемую приложением   
  
В состав JVM входят   
• загрузчик классов (Classloader)   
• сборщик мусора (Garbage Collector)   
• интерпретатор   
• компилятор   
• компоненты управления потоками   
  
Загрузка классов — это поиск и загрузка типов (классов и интерфейсов) динамически во время выполнения программы. Данные о типах находятся в бинарных class-файлах.   
Используется для поставки в JVM скомпилированного байт-кода   
Этапы загрузки классов   
• Загрузка (loading) — поиск и импорт бинарных данных для типа по его имени, создание класса или интерфейса из этого бинарного представления.   
• Связывание, линковка (linking)   
• Верификация (verification) — проверка корректности полученного байт-кода.   
• Подготовка (preparation) — выделение памяти для статических переменных класса и инициализация памяти значениями по умолчанию.   
• Разрешение (resolution) — преобразование символьных ссылок типов в прямые ссылки.   
• Инициализация (initialization) — вызов Java-кода, который инициализирует переменные класса их правильными начальными значениями.   
  
  
  
Интерпретатор - читает исходный код программы и выполняет его. Преобразование исходного кода в бинарный и выполнение происходит построчно.   
Достоинства интерпретаторов:   
• Независимость от ОС (переносимость кода).   
• При внесении изменений НЕ требуется перекомпиляция кода.   
Недостатки интерпретаторов:   
• Для запуска программы требуется наличие интерпретатора.   
• Низкая скорость работы.   
  
компиляция - полностью преобразовывает исходный код программы в бинарный, а не построчно.   
Достоинства компиляторов:   
• Быстрота работы программ;   
• Отсутствие надобности компилятора на компьютере пользователя.   
Недостатки компиляторов:   
• Программа зависит от ОС, под которую была скомпилирована.   
• При внесении изменений требуется перекомпиляция кода.   
  
JDK - комплект разработчика приложений на языке Java, включающий в себя JRE, компилятор, стандартные библиотеки классов Java   
  
JRE — это программа, которая берет ваш Java-код, объединяет его с необходимыми библиотеками и запускает JVM для его выполнения.   
  
Javac - компилятор java проверяет код на синтаксис, лексику, семантику, оптимизирует его, и в конечном итоге, генерирует файл с расширением .class содержащий байт-код.   
  
Принцип работы Java   
Для компиляции исходного кода в байт-код существует компилятор javac, входящий в JDK на вход компилятор принимает файл с расширением .java, содежащий исходный код программы, а на выходе выдает файл с расширением .class, содержащий байт-код, необходимый для исполнения программы виртуальной машиной.   
После того, как программа была скомпилирована в байт-код, она может быть выполнена с помощью виртуальной машины.

разобраться с числами с фиксированной и плавающей точкой в Java, разобраться с перегрузкой

## Тема 2 Знание синтаксиса и конструкций языка

### Вопрос 1 Синтаксис

Когда мы рассматриваем java-программу, она может быть определена как совокупность объектов, которые взаимодействуют с помощью вызова методов друг друга.

**Объект** — объекты имеют состояние и поведение. Например: собака может иметь состояние — цвет, имя, а также и поведение — кивать, бежать, лаять, кушать. Объект является экземпляром класса.

**Класс** — может быть определен как шаблон, который описывает поведение объекта.

**Интерфейс** — Задает поведение класса

**Перечисления** — Они ограничивают переменную, чтобы выбрать только одно из нескольких предопределенных значений. Значения в этом перечисляемом списке называются **перечисления**.

**Переменные экземпляра** — каждый объект имеет свой уникальный набор переменных экземпляра. Состояние объекта создается значениями, присвоенными этим переменным экземпляра.

**Метод** — является в основном поведением. Класс может содержать несколько методов. Именно в методах логически записанные данные манипулируют и выполняют все действия.

###### (пометка для сережи)

Параметры в метод передаться в круглых скобках

Автокомплит - функция в программах, предусматривающих интерактивный ввод текста (редакторы, оболочки командной строки, браузеры и т. д.), по дополнению текста по введённой его части.

**Нативные методы** – методы написанные на другом языке программирования

**Блоки инициализации**

Существует два типа блоков инициализации обычный и статический что бы в них инициализировать переменные

Пример**:**

*class* MyClass

    {

        int a;

        { // *блок инициализации*

            a = 12;

        }

    }

Обычный блок инициализации существует для удобства.

Статический блок существует для объявления статических переменных так как выполняется раньше и один раз, выполняется в одном из двух случаев:

1. При создании первого объекта класса в процессе работы программы, перед запуском конструктора.
2. При первом вызове статической функции, перед выполнением.

Порядок выполнения следующий:

1. Статический блок
2. Конструктор родителя
3. Обычный блок инициализации
4. Конструктор класса

### Вопрос 2 Операторы

Операторы — это специальные символы, которые сообщают транслятору о том, что вы хотите выполнить операцию с не¬которыми операндами.

Все операторы Java можно разделить на следующие группы:

* арифметические операторы;
* операторы сравнения;
* побитовые операторы;
* логические операторы;
* операторы присваивания;
* прочие операторы.

**Арифметические операторы** — используются в математических выражениях

* + Складывает значения по обе стороны от оператора
* - Вычитает правый операнд из левого операнда
* Умножает значения по обе стороны от оператора
* / Оператор деления делит левый операнд на правый операнд
* % Делит левый операнд на правый операнд и возвращает остаток
* ++ Инкремент - увеличивает значение операнда на 1
* -- Декремент - уменьшает значение операнда на 1

**Операторы сравнения**

* == Проверяет, равны или нет значения двух операндов, если да, то условие становится истинным
* ! = Проверяет, равны или нет значения двух операндов, если значения не равны, то условие становится истинным
* > Проверяет, является ли значение левого операнда больше, чем значение правого операнда, если да, то условие становится истинным
* < Проверяет, является ли значение левого операнда меньше, чем значение правого операнда, если да, то условие становится истинным
* >= Проверяет, является ли значение левого операнда больше или равно значению правого операнда, если да, то условие становится истинным
* <= Проверяет, если значение левого операнда меньше или равно значению правого операнда, если да, то условие становится истинным

**Побитовые операторы**

Java определяет несколько побитовых операторов, которые могут быть применены для целочисленных типов: int, long, short, char и byte.

* & (побитовое и) Бинарный оператор AND копирует бит в результат, если он существует в обоих операндах.
* | (побитовое или) Бинарный оператор OR копирует бит, если он существует в любом из операндов
* ^ (побитовое логическое или) Бинарный оператор XOR копирует бит, если он установлен в одном операнде, но не в обоих.
* ~ (побитовое дополнение) Бинарный оператор дополнения и имеет эффект «отражения» бит. Инвертирует все биты
* << (сдвиг влево) Бинарный оператор сдвига влево. Значение левых операндов перемещается влево на количество бит, заданных правым операндом.
* >> (сдвиг вправо) Бинарный оператор сдвига вправо. Значение правых операндов перемещается вправо на количество бит, заданных левых операндом.
* >>> (нулевой сдвиг вправо)) Нулевой оператор сдвига вправо. Значение левых операндов перемещается вправо на количество бит, заданных правым операндом, а сдвинутые значения заполняются нулями.

**Пример**

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

a<<b = 1111 0000

a<<b = 1111

a>>>b = 0000 1111

Отрицательные числа

В Java байты знаковые. Крайний левый разряд/бит (его ещё называют "старший бит") сделан своего рода "маркером", отвечающим на вопрос: "Это число отрицательное?". Если ответ да, значит маркер имеет значение 1. А иначе — 0.

### Вопрос 3 Логические выражения

**Логические операторы**

! Логическое “не” (отрицание)

!x означает “не x”. Возвращает true если операнд является false. Возвращает false если операнд является true.

Пример

boolean x = true;

!x==false

& Логическое И (AND, умножение)

Возвращает true если оба операнда равны true.

Пример

    a=true;

    b=false;

    тогда a&b==false

| Логическое ИЛИ (OR, сложение)

Возвращает true если хотя бы один из операндов равен true.

Пример

    a=true;

    b=false;

    тогда a|b==true

^ Логическое исключающее ИЛИ (XOR)

Возвращает true, если один и только один из операндов равен true. Возвращает false, если оба операнда равны true или false. По сути, возвращает true, если операнды — разные.

Пример

    a=true;

    b=false;

    тогда a^b==true

### Вопрос 4 Управляющие конструкции

Управляющие конструкции языка - это наборы служебных слов, позволяющие изменять ход выполнения скрипта

**Оператор if**

Инструкция if(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода.

**[Блок «else»](https://learn.javascript.ru/ifelse" \l "blok-else)**

Инструкция if может содержать необязательный блок «else» («иначе»). Он выполняется, когда условие ложно.

**[Несколько условий: «else if»](https://learn.javascript.ru/ifelse" \l "neskolko-usloviy-else-if)**

Иногда, нужно проверить несколько вариантов условия. Для этого используется блок else if.

Пример

    int num1 = 6;

    int num2 = 8;

if(num1>num2)

    {

        System.out.println("Первое число больше второго");

    }else if(num1<num2)

    {

        System.out.println("Первое число меньше второго");

    }else

    {

        System.out.println("Числа равны");

    }

**Конструкция switch**

Конструкция **switch/case** аналогична конструкции if/else,

После ключевого слова **switch** в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после операторов **сase**. И если совпадение найдено, то будет выполняет соответствующий блок **сase**.

В конце блока сase ставится оператор **break**, чтобы избежать выполнения других блоков.

Если мы хотим также обработать ситуацию, когда совпадения не будет найдено, то можно добавить блок **default**

**Пример**

int num = 2;

    switch(num)

    {

*case* 1*:*

        System.out.println("число равно 1");

*break*;

*case* 2*:*

        System.out.println("число равно 2");

*break*;

*default:*

        System.out.println("число не равно 1, 2");

    }

[**Циклы**](https://learn.javascript.ru/while-for#tsikl-while)

Цикл — разновидность управляющей конструкции, предназначенная для организации многократного исполнения набора инструкций.

[**Цикл «while»**](https://learn.javascript.ru/while-for#tsikl-while)

**while**(Логическое выражение) {

// Тело цикла - периодически выполняемые оператор(ы)

}

Выполняется следующим образом (по шагам):

* Вычисляем Логическое условие, следующее в скобках за while.
* Если логическое условие истинно, то выполняются операторы в теле цикла, после выполнения последнего оператора в теле цикла, переходим на шаг 1
* Если логическое условие ложно, то переходим к первому оператору за пределами цикла while.

**Цикл do while**

*do*

    {

        // *Тело цикла - периодически выполняемые оператор(ы)*

    }while(Логическое выражение);

Выполняется следующим образом (шаги):

* Выполняется тело цикла (сразу после ключевого слова do).
* Вычисляем Логическое условие, следующее в скобках за while.
* Если логическое условие истинно, то переходим на шаг 1
* Если логическое условие ложно, то переходим к первому оператору за пределами цикла while.

Два основных отличия от предыдущего вида цикла: тело цикла как минимум выполняется один раз и логическое условие проверяется после выполнения тела цикла.

**Управляющие команды в теле цикла**

* *continue* – прекращает выполнение тела текущего цикла и осуществляет переход к логическому выражению оператора while. Если вычисленное выражение будет истинно – выполнение цикла будет продолжено.
* *break* – немедленно прекращает выполнение текущего цикла и осуществляет переход к первой команде за его пределами. Таким образом, выполнение текущего цикла прерывается.

**Цикл for**

    for(<начальная точка>;<условие выхода>;<операторы счетчика>)

    {

        // *Тело цикла*

    }

В цикле фор если не задать условие выхода цикл будет отрабатывать бесконечно, или если ничего не задать

**Цикл forEach**

    for(<Тип элемента><Имя переменной, куда будет записан очередной элемент>:<Название массива>)

    {

        // *Тело цикла*

    }

### Вопрос 5 Интерфейсы, Классы, Абстрактные классы

#### **Интерфейс** – Контракт, обязанность реализовать все стандартные методы интерфеса

*public* *interface* NameOfInterface // *создание интерфейса*

    {

        // *поля автоматически помечаются модификатором final и static*

        // *Любое количество объявлений абстрактных методов*

    }

**Чем отличается класс от интерфейса?**

* Вы не можете создать экземпляр интерфейса.
* В интерфейсе не содержатся конструкторы.
* Все методы в интерфейсе абстрактные.
* Интерфейс не может содержать поля экземпляров. Поля, которые могут появиться в интерфейсе, обязаны быть объявлены и статическими, и final.
* Интерфейс не расширяется классом, он реализуется классом.
* Интерфейс может расширить множество интерфейсов.

**интерфейсы имеют следующие свойства:**

* Интерфейс абстрактный косвенно. Вам не нужно использовать ключевое слово abstract во время объявления интерфейса.
* Каждый метод в интерфейсе косвенно абстрактным, поэтому ключевое слово abstract не нужно.
* Методы в интерфейсе косвенно публичны.

**Реализация интерфейса**

Когда класс реализует интерфейс, вы можете представить себе, что класс словно подписывает контракт с интерфейсом, соглашаясь совершить конкретные его поведения. Если класс не исполняет все поведения интерфейса, то класс должен объявить себя абстрактным.

Класс использует ключевое слово implements для реализации интерфейса. Ключевое слово **implements** появляется при объявлении класса в его расширенной части.

**Расширение интерфейсов**

Интерфейс может расширять другой интерфейс так же, как класс другой класс. Ключевое слово **extends** используется для расширения интерфейса, и дочерний интерфейс наследует методы родительского интерфейса.

#### Класс

Класс определяется с помощью ключевого слова **сlass**:

*class* Person {

        // *поля*

        // *методы*

    }

После названия класса идут фигурные скобки, между которыми помещается тело класса - то есть его поля и методы.

Любой объект может обладать двумя основными характеристиками: состояние - некоторые данные, которые хранит объект, и поведение - действия, которые может совершать объект.

#### Методы

**Метод** — является в основном поведением. Класс может содержать несколько методов. Именно в методах логически записанные данные манипулируют и выполняют все действия.

##### **Вопрос Ранее позднее связывание(Динамическая статическая типизация)**

**Раннее** **связывание** - это когда метод, который будет вызван, известен во время компиляции, например, вызов статического метода. Кстати, то что вы называете **поздним** связываением, есть скорее dynamic dispatch.

**Позднее** **связывание** - это когда вызов метода может быть осуществлен только во время выполнения и у компилятора нет информации, чтобы проверить корректность такого вызова.

##### **Вопрос передача параметров в методы**

Передача параметров в метод если передаем объект, то передаться ссылка на объект, если передаем примитив, то передаться копия переменной

Так же в методы можно передавать varArgs это необходимо для того что бы мы могли передать в него массив не определенной длинны

Запись имеет вид

*public* *static* void metod(T... a){}

Из данной записи видно, что тип можно параметризировать

VarArgs должен находить в конце метода при объявлении метода

то есть:

*public* *static* void metod(String s, int b, T... a) {}

#### Конструкторы

Кроме обычных методов классы могут определять специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта. С определённым количеством аргументов которые мы зададим или ни одного если используем в конструктор по умолчанию.

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор без параметров.

    Person(параметры) {

//*поля*

}

Если мы создали конструктор с параметрами, то класс не создает конструктор по умолчанию и при необходимости нам самим нужно его создать.

В конструкторе всегда вызывается родительский конструктор super();

В super(); можно передавать параметры конструктору родительского класса.

**Ключевое слово this**

Ключевое слово this представляет ссылку на текущий экземпляр класса. Через это ключевое слово мы можем обращаться к переменным, методам объекта, а также вызывать его конструкторы.

#### Абстрактные классы.

При определении абстрактных классов используется ключевое слово **abstract**:

*public* *abstract* *class* Person {

        // *поля*

        // *методы*

    }

Но главное отличие состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта. Например, следующим образом:

    Person р = *new* Person();

Кроме обычных методов абстрактный класс может содержать **абстрактные методы**. Такие методы определяются с помощью ключевого слова abstract и не имеют никакой реализации:

*public* *abstract* void display();

производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе.

Классы в одном .class файле будут package-private, т.е. будут видимы только в рамках пакета.

1. Почему НЕ стоит пихать интерфейс в тот же файл - потому что интерфейс сам по себе подразумевает, что его будут использовать другие люди, иначе толку от него?!
2. Почему НЕ стоит пихать классы в тот же файл - потому что люди не будут знать, как найти этот класс (это только в университетах код пишет и фиксит 1 человек).
3. Почему стоит пихать классы в тот же файл - если вы посмотрите в исходный код Жава, то вы найдете много примеров, когда абстрактный класс обьявлен как public, и в этом же файле лежит его дефолтная реализация. Нет смысла выносить этот класс в отдельный файл т.к. не предусмотрено, что кто-то будет им пользоватся, он нужен только этому паблик классу.

#### Класс Object

Хотя мы можем создать обычный класс, который не является наследником, но фактически все классы наследуются от класса **Object**. Все остальные классы, даже те, которые мы добавляем в свой проект, являются неявно производными от класса Object. Поэтому все типы и классы могут реализовать те методы, которые определены в классе Object. Рассмотрим эти методы.

##### Метод toString

Метод toString служит для получения представления данного объекта в виде строки. При попытке вывести строковое представления какого-нибудь объекта, как правило, будет выводиться полное имя класса.

##### Метод hashCode

Метод hashCode позволяет задать некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты.

##### Метод getClass

Метод getClass позволяет получить тип данного объекта

##### Метод equals

Метод equals сравнивает два объекта на равенство, так как при сравнении ссылок двойным равно сравниваться сами ссылки, а не объекты

Метод equals принимает в качестве параметра объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса.

Оператор instanceof позволяет выяснить, является ли переданный в качестве параметра объект объектом определенного класса, в данном случае класса Person. Если объекты принадлежат к разным классам, то их сравнение не имеет смысла, и возвращается значение false.

Затем сравниваем по именам. Если они совпадают, возвращаем true, что будет говорить, что объекты равны.

#### Отличия абстрактного класса от интерфейса в Java

|  | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Ключевое слово, используемое для описания** | Для определения абстрактного класса используется ключевое слово abstract | Для определения интерфейса используется ключевое слово interface |
| **Ключевое слово, используемое для реализации** | Для наследования от абстрактного класса используется ключевое слово extends | Для реализации интерфейса используется ключевое слово implements |
| **Конструктор** | Определение конструктора разрешено | Объявление/определение конструктора запрещено |
| **Реализация методов по умолчанию** | Допустима | Вплоть до Java 8 разрешено только объявлять методы, но реализация их запрещена. Начиная с Java 8 стала допустима реализация **static** методов, а реализация **non-static** методов стала доступна посредством ключевого слова default |
| **Поля** | Может иметь как **static**, так и **non-static** поля (аналогичное относится и к final) | Любое поля интерфейса по умолчанию является public static final (и иных иметь не может) |
|  | Наследовать 1 класс | Реализация множества интерфейсов |

### Вопрос 6 Внутренние и вложенные классы

Класс называется вложенным (nested), если он определен внутри другого класса

Существует четыре категории вложенных классов:

* Статические вложенные классы и не статические вложенные классы. Вложенные классы, объявленные статически, называются вложенными статическими классами.
* Внутренние классы — когда объект внутреннего класса связан с объектом обрамляющего класса. Не статические вложенные классы называются внутренними классами, если они связанны с внешним классом.
* Локальные классы — объявленные внутри блока кода и не являющиеся членом обрамляющего класса. В этом случае можно рассматривать класс как локальную переменную типа класс.
* Анонимные классы – наследуемые, от какого либо класса, классы в которых при объявлении не задано имя класса.

**Статические Вложенные Классы**  
Причины использования статических вложенных классов:

Для случая, когда связь между объектом вложенного класса и объектом внешнего класса не нужна, можно сделать вложенный класс статическим(static).  
  
Так как внутренний класс связан с экземпляром, он не может определить в себе любые статические члены.  
  
Статические вложенные классы не имеют ограничений по объявлению своих данных и полей как static.  
  
Из вложенного статического класса мы не имеем доступа к внешней не статической переменной внешнего класса.

**Внутренние классы**  
  
Внутренний класс связан с экземпляром его обрамляющего класса

внутренний класс связан с внешним классом через экземпляр, или через объект класса.  
  
Чтобы создать экземпляр внутреннего класса, нам нужно сначала создать экземпляр внешнего класса. Затем создать внутренний объект, в пределах внешнего объекта, таким образом:

*class* OuterClass {

    ...

*class* InnerClass {

    }

}

    OuterClass.InnerClass innerObject = outerObject.new InnerClass();

во внутреннем классе мы можем использовать весь унаследованный функционал внешнего класса

**Локальный класс**

это вложенный класс, объявленный внутри другого класса и некоторого блока кода этого класса, то есть объявленный между фигурными скобками {}. Этот блок может быть статическим блоком, циклом, телом if и т.д.  
  
Локальный класс объявленный внутри блока кода другого класса не является членом класса, к которому относится блок, а принадлежит самому блоку, точно так же, как обычная локальная переменная. Такие классы недоступны за пределами внешнего класса, поскольку нет никаких способов обращения к ним, но их экземпляры – это обычные объекты

* Объекты локального класса могут создаваться в блоке кода, котором они описаны.
* Локальные классы не имеют модификаторов доступа private или public, потому что они принадлежат не классу, а тому блоку кода, в котором они описаны.
* Локальные классы, объявленные в статичном блоке, могут обращаться только к статичным полям внешнего класса.
* Локальные классы могут иметь модификаторы доступа final
* Локальные классы не могут быть статичными, за исключением static final.
* В java8 мы можем обращаться из локального класса не только к финальным переменным внешнего класса, и к не финальным методам внешнего класса, если они не были изменены до момента инициализации класса.
* Локальные классы имеют доступ к закрытым переменным внешнего класса, в контексте экземпляра.

**Анонимные (безымянные) классы**  
  
Анонимным классом называется класс, в котором программист не задает явно имя экземпляру.  
Пример:

*new* AnyClass();

В анонимном классе явно не задается конструктор класса, а используется конструктор базового класса.  
Мы можем создать анонимный класс, передав некоторое значение в конструктор базового класса

*new* AnyClass(){

// *Тело нового анонимного класса. Новый функционал вашего класса*

}

**Случаи использования анонимного класса:**

* Тело класса является очень коротким.
* Нужен только один экземпляр класса.
* Класс используется в месте его создания или сразу после него.
* Имя класса не важно и не облегчает понимание кода.

#### (Пометка)

При попытке обраться из абстрактного к полям и методам внешнего мы получаем ошибку effctivly final

Это ограничение, которое ввели разработчики спецификации: при создании анонимного класса происходит захват объектов по значению, а не по ссылке. В анонимном классе создаются закрытые поля, в которые копируются значения переданных переменных. То есть по сути реализована эмуляция замыканий.

Добавление модификатора final как бы говорит программисту о том, что при использовании такого рода замыканий не будут наблюдаться побочные эффекты (представьте, что вы передали в созданный объект анонимного класса переменную, а затем изменили её "снаружи" - вы в праве ожидать, что это изменение отразится и внутри замыкания, но это не так в силу пресловутой эмуляции; однако изменение внутреннего состояния mutable объектов будет видно и там и там).

Для реализации полноценных замыканий необходимо расширять жизненный цикл захватываемых объектов (это касается объектов, расположенных в стеке, которые при выходе из области видимости, в которой они объявлены, уничтожаются).

### Вопрос 7 Ключевые слова static и final

**final.**

* Оно может применяться к классам, методам, переменным (в том числе аргументам методов).
* Для класса это означает, что класс не сможет иметь подклассов, т.е. запрещено наследование.
* Для метода final означает, что он не может быть переопределен в подклассах.
* Для переменных примитивного типа это означает, что однажды присвоенное значение не может быть изменено.

**Static**

модификатор, применяемый к полю, блоку, методу или внутреннему классу. Данный модификатор указывает на привязку субъекта к текущему классу.

* **Статические поля**

При обозначении переменной уровня класса мы указываем на то, что это значение относится к классу. Если этого не делать, то значение переменной будет привязываться к объекту, созданному по этому классу.

* **Статический метод**

Статические методы отличаются от обычных тем, что они также привязаны к классу, а не к объекту. Важным свойством статического метода является то, что он может обратиться только к статическим переменным/методам.

* **Статический блок**

Есть два блока инициализации — обычный и статический. Блок предназначен для инициализации внутренних переменных. Если блок обычный, то им инициализируют внутренние переменные объекта, если же статический, соответственно, им задают статические переменные (то есть переменные класса).

* **Статический класс**

Статическим классом может быть только внутренний класс. Опять же, этот класс привязан к внешнему классу, и если внешний наследуется другим классом, то этот не будет наследован. При этом данный класс можно наследовать, как и он может наследоваться от любого другого класса и имплементировать интерфейс.

**Основные свойства**

* Вы не можете получить доступ к не статическим членам класса, внутри статического контекста, как вариант, метода или блока.
* В отличие от локальных переменных, статические поля и методы не потока безопасны
* Статические методы имеют преимущество в применении, т.к. отсутствует необходимость каждый раз создавать новый объект для доступа к таким методам. Статический метод можно вызвать, используя тип класса, в котором эти методы описаны.
* Другим важным моментом является то, что вы НЕ можете переопределять (Override) статические методы.
* Вовремя сериализации, статические поля не сериализуются.

### Вопрос 8 enum

Перечисления представляют набор логически связанных констант. Перечисление фактически представляет новый тип Объявление перечисления происходит с помощью оператора enum, после которого идет название перечисления. Затем идет список элементов перечисления через запятую:

*enum* Day {

        MONDAY,

        TUESDAY,

        WEDNESDAY,

        THURSDAY,

        FRIDAY,

        SATURDAY,

        SUNDAY

    }

Создание объекта ENUM

    Day sonDay = Day.MONDAY;

**Методы перечислений**

Каждое перечисление имеет статический метод **values().** Он возвращает массив всех констант перечисления

Метод **ordinal()** возвращает порядковый номер определенной константы

Перечисления, как и обычные классы, могут определять конструкторы, поля и методы.

Поля в Enum применяться сразу ко всем перечислениям;

Статические методы в enum работают так же как и в обычном классе

Конструкторы всегда только privat и их может быть несколько тогда запись может быть в виде:

*enum* people{

    Sergey1, Sergey2(123), Sergey3(15,"Africa");

    people(){}

    people(int i){}

    people(int i, String a){}

}

### Вопрос 9 Пакеты

**Что такое пакет для класса**

Для класса его пакет — это его местоположение в проекте, относительно других классов. Благодаря разделению классов на несколько пакетов — мы организовываем структуру программы.

**protected** перед методом или полем — означает, что данный член класса будет виден не только классам текущего пакета, как с package-private, но и классам-наследникам

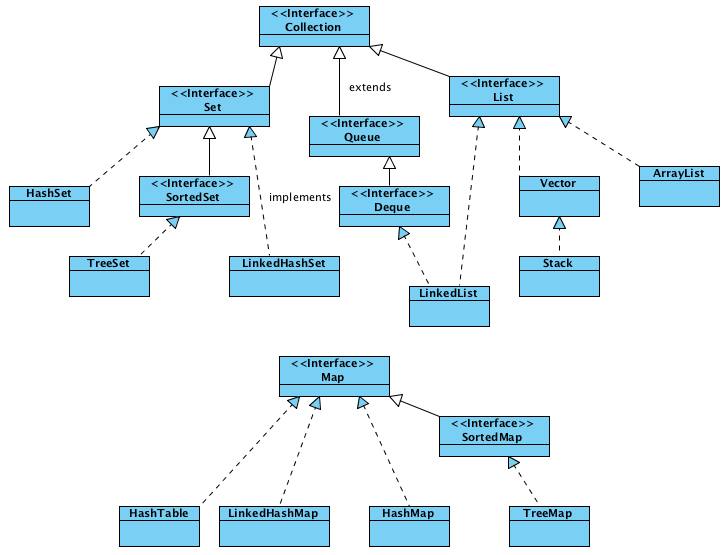
**Импортирование пакетов**

Для того чтобы не писать длинные пути, к каждому конкретному классу, при каждом его использовании — нужно его импортировать. Для этого применяется ключевое слово **import**.

Разобраться с: взаимными ссылками внутреннего и внешнего класса, перекрытием статических методов, final параметрами и effctivly final

## Тема 3 Коллекции

### Вопрос 1 Иерархия коллекций



### Вопрос 2 Iterable, Collection

**Итератор** – объект, позволяющий перебирать элементы коллекции. Например foreach реализован с использованием итератора. Одним из ключевых методов интерфейса Collection является метод Iterator<E> iterator(). Он возвращает итератор – то есть объект, реализующий интерфейс Iterator.

Интерфейс Iterator имеет следующее определение:

public interface Iterator <E>{

    E next();

    boolean hasNext();

    void remove();

}

**Интерфейс Collection** является базовым для всех коллекций, определяя основной функционал:

public interface Collection<E> extends Iterable<E>{

    // определения методов

}

Интерфейс Collection является обобщенным и расширяет интерфейс Iterable, поэтому все объекты коллекций можно перебирать в цикле по типу for-each.

### Вопрос 3 List и его реализации

**List** – это упорядоченный список. Объекты хранятся в порядке их добавления в список. Доступ к элементам списка осуществляется по индексу.

**ArrayList** реализован внутри в виде обычного массива. Поэтому при вставке элемента в середину, приходится сначала сдвигать на один все элементы после него, а уже затем в освободившееся место вставлять новый элемент. Зато в нем быстро реализованы взятие и изменение элемента – операции get, set, так как в них мы просто обращаемся к соответствующему элементу массива.

**LinkedList** реализован внутри по-другому. Он реализован в виде связного списка: набора отдельных элементов, каждый из которых хранит ссылку на следующий и предыдущий элементы. Чтобы вставить элемент в середину такого списка, достаточно поменять ссылки его будущих соседей. А вот чтобы получить элемент с номером 130, нужно пройтись последовательно по всем объектам от 0 до 130.

### Вопрос 3 Set и его реализации

**Set** – множество **неповторяющихся** объектов. В коллекции этого типа разрешено наличие только одной ссылки типа null.

**Класс HashSet** реализует интерфейс Set, основан на хэш-таблице HashSet

* В HashSet элементы не упорядочены, нет никаких гарантий, что элементы будут в том же порядке спустя какое-то время. Операции добавления, удаления и поиска будут выполняться за константное время при условии, что хэш-функция правильно распределяет элементы по «корзинам»
* HashSet также реализует интерфейсы Serializable и Cloneable.
* производительность итерации по контейнеру зависит от емкости и “коэффициента загрузки” (рекомендуется оставлять load factor значением по умолчанию равным 0.75, что является хорошим компромиссом между временем доступа и объемом хранимых данных)

**TreeSet**

Класс TreeSet реализует интерфейс [NavigableSet](https://www.examclouds.com/java/java-core-russian/navigable-set), который поддерживает элементы в отсортированном по возрастанию порядке. Объекты сохраняются в отсортированном порядке по возрастанию. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

**LinkedHashSet**

* Класс LinkedHashSet расширяет класс HashSet, не добавляя никаких новых методов. Класс поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор.
* **LinkedHashSet** в некотором смысле занимает промежуточное положение между HashSetи TreeSet. Реализованный как хэш-таблица со связанным списком, проходящим через нее, **она обеспечивает итерацию с упорядоченной вставкой, которая отличается от отсортированного обхода**

### Вопрос 4 SortedSet, NavigableSet

**SortedSet**

Интерфейс SortedSet предназначен для создания коллекций, который хранят элементы в отсортированном виде (сортировка по возрастанию). SortedSet расширяет интерфейс Set, поэтому такая коллекция опять же хранит только уникальные значения.

**NavigableSet**

Интерфейс NavigableSet расширяет интерфейс SortedSet и позволяет извлекать элементы на основании их значений.

### Вопрос 5 Map и ее реализации

**Map в java** — это структура данных, которая хранит данные в виде <Ключ, Значение>. Каждое значение можно найти по его ключу. Немного фактов о мап:

* карта не может содержать дубликаты ключей;
* каждый ключ может отображать только одно значение (далее мы посмотрим реализации map и что бывает, если попробовать добавить несколько объектов с одним ключом);
* Map не расширяет интерфейс Collection.

**HashMap** реализована на основе массива (а если точнее — динамического, так как таблица может расширяться) односвязных списков. По сути, мы получаем хеш-код ключа в результате работы метода hashCode(), который затем модифицируется (как рассмотрим далее), а внутри с помощью дополнительного метода полученные значения распределяются по нужным ячейкам. Элементы массива (ячейки) еще называются **корзинами «bucket»**, которые используются для хранения отдельно взятых узлов. Каждый из бакетов представляет из себя коллекцию (список или дерево)

**TreeMap** в Java реализует [интерфейс Map](http://proglang.su/java/map-interface), используя дерево. Обеспечивает эффективное средство хранения пар ключ/значение в отсортированном порядке и позволяет быстро извлекать данные.

Следует отметить, что, в отличие от хэш-карты, карта деревьев гарантирует, что ее элементы будут отсортированы в порядке возрастания ключа.

**LinkedHashMap** расширяет [HashMap](http://proglang.su/java/hashmap-class) и поддерживает связанный список записей на [Map](http://proglang.su/java/map-interface) в том порядке, в котором они были вставлены. Это позволяет итерации ввода-вывода по Map. То есть при итерировании LinkedHashMap элементы будут возвращены в том порядке, в котором они были вставлены.

### Вопрос 6 SortedMap, NavigableMap

Интерфейс **SortedMap** расширяет Map и создает отображение, в котором все элементы отсортированы в порядке возрастания их ключей.

Интерфейс **NavigableMap** расширяет интерфейс SortedMap и обеспечивает возможность получения элементов отображения относительно других элементов

### Вопрос 7 Другие коллекции (Queue, Dequeue, Stack, Vector, Hashtable)

**Интерфейс Queue** расширяет Collection и объявляет поведение очередей, которые представляют собой список с дисциплиной "первый вошел, первый вышел" (FIFO). Существуют разные типы очередей, в которых порядок основан на некотором критерии. Очереди не могут хранить значения null.

**Интерфейс Deque** появился в Java 6. Он расширяет Queue и описывает поведение двунаправленной очереди. Двунаправленная очередь может функционировать как стандартная очередь FIFO либо как стек LIFO.

**Stack** является подклассом класса **Vector**, который реализует простой механизм типа "последний вошёл - первый вышел" (LIFO).

**Класс Vector** реализует динамический массив. Он похож на [ArrayList](http://proglang.su/java/arraylist-class), но с двумя отличиями:

* Vector синхронизирован.
* Vector содержит много устаревших методов, которые не являются частью структуры коллекций.

## Тема 4 Обработка исключений (ошибок)

### Вопрос 1 Что такое Exception

Exception — это проблема, которая может возникнуть в runtime. Это исключительная ситуация, которая возникает из-за каких-то причин. Диаграмма наследования исключений.

В целом все исключения делятся на две группы — exceptions и error.

Error — используются JVM для отображения ошибок, после которых работа приложения уже не имеет смысла. Например StackOverFlowError, которая говорит, что стек заполнен и программа уже не может работать.

Exception — исключения, которые генерируются программно в коде.

Есть разные исключения, проверяемые и непроверяемые, но главное, что они есть, и их можно перехватить и продолжить работу приложения.

Exceptions, в свою очередь, еще делятся на тех, кто наследуется от RuntimeException и других наследников Exception.

### Вопрос 2 Как JVM обрабатывает исключения?

Как это работает? Как только где-то создается исключение, runtime создает Exception Object (обозначим как ExcObj). В нем хранится вся необходимая для работы информация — само исключение, которое вызывалось и место, где это произошло. Создание ExcObj и передача в runtime есть ничто иное как “выбрасывание исключения”.

ExcObj содержит методы, по которым можно дойти до место создания исключения. Это множество методов называется Call Stack. Далее, runtime система ищет метод в Call Stack, который сможет обработать наше исключение. Если он таки находит соответствующий обработчик, то есть тип исключения совпадает с типом в обработчике, все хорошо. Если не находит, то runtime передает всё в default exception handler, который подготавливает ответ и завершает работу.

### Вопрос 3 Ключевые слова try … catch … finally.

**Обработка исключений в Java основана на использовании в программе следующих ключевых слов:**

* **try** – определяет блок кода, в котором может произойти исключение;
* **catch** – определяет блок кода, в котором происходит обработка исключения;
* **finally** – определяет блок кода, который является необязательным, но при его наличии выполняется в любом случае независимо от результатов выполнения блока try.
* **throw** – используется для возбуждения исключения;
* **throws** – используется в сигнатуре методов для предупреждения, о том, что метод может выбросить исключение.

#### Вопррос 3.1 Всегда ли выполнится finally

Нет. Finally не выполниться:

Если вы вызываете System.exit();

Если сначала произойдет сбой JVM;

Если, Finally, блок будет выполняться потоком демона (сервисы, работающие в фоновом режиме, не являются частью программы). Потоки демоны прерываются внезапно.

Если в Finally исключение и нет обработчика, тооставшийся код в Finally не выполниться.

#### Вопрос 3.2 throw и throws

throw используют в случае, когда нужно явно создать новое исключение. Применяют его для создания и выбрасывания пользовательских исключений. Использовать эту конструкцию можно только тем, что наследуется от Throwable.

throws — механизм, при помощи которого метод может выбрасывать одно или более исключений. Добавляются они через запятую. Могут быть как проверяемые таки непроверяемые исключения.

Если это проверяемые, то используя метод, который их генерирует, нужно как-то его обработать. Есть два варианта:

Написать try-catch с соответствующим и выше по наследованию исключением.

Использовать throws точно так же, чтобы эта проблема была уже у кого-то другого

#### Вопрос 3.3 Что такое try-with-resources?

Это механизм, при помощи которого нужно правильно закрывать все ресурсы.

Ресурс — это объект, после работы с которым нужно закрыть его, то есть вызвать метод close(). Ресурсом называются все объекты, которые реализуют интерфейс AutoClosable, который, в свою очередь реализует интерфейс Closeable. Для нас важно понять, что все InputStream, OutpuStream являются ресурсами и их нужно правильно и успешно высвобождать. Вот как раз для этого и нужно нам использовать try-with-resource конструкцию.

Определять можно больше одной переменной, разделяют их точкой с запятой.

Но можно использовать finally блок? Нет.

OutputStream stream = openOutputStream();*try*

    {

        // *что-то делаем со stream*

    }finally

    {

        stream.close();

    }

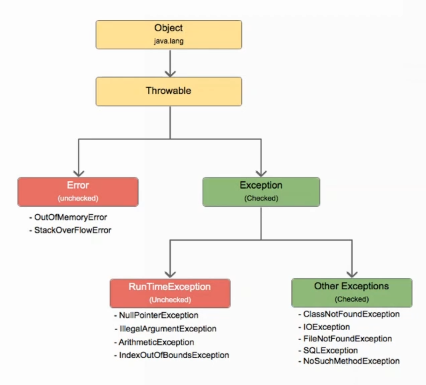
Проблема следующая. Метод close() может сгенерировать исключение. И если при этом основной код работы с ресурсом тоже выбросит исключение, то оно перезатрется исключением из close(). Информация об исходной ошибке пропадёт: мы никогда не узнаем, что было причиной исходного исключения.

### Вопрос 4 Иерархия исключений

Исключения делятся на несколько классов, но все они имеют общего предка — класс **Throwable.** Его потомками являются подклассы **Exception** и **Error.**

**Исключения (Exceptions)** являются результатом проблем в программе, которые в принципе решаемые и предсказуемые. Например, произошло деление на ноль в целых числах.

**Ошибки (Errors)** представляют собой более серьёзные проблемы, которые, согласно спецификации Java, не следует пытаться обрабатывать в собственной программе, поскольку они связаны с проблемами уровня JVM. Например, исключения такого рода возникают, если закончилась память, доступная виртуальной машине. Программа дополнительную память всё равно не сможет обеспечить для JVM.



### Вопрос 5 Checked и Unchecked исключения

Все исключительные ситуации делятся на «проверяемые» **(checked)** и «непроверяемые» **(unchecked)** Это свойство присуще «корневищу» (Throwable, Error, Exception, RuntimeException) и передается по наследству.   
Throwable и Exception и все их наследники (за исключением наследников Error-а и RuntimeExceptionа) — checked  
— Error и RuntimeException и все их наследники — unchecked  
checked exception = проверяемое исключение, проверяемое компилятором.

* Checked исключения, это те, которые должны обрабатываться блоком catch или описываться в сигнатуре метода throws. Unchecked могут не обрабатываться и не быть описанными.
* Unchecked исключения в Java – наследованные от RuntimeException, checked – от Exception (не включая unchecked).

Конструкцию **try-with-resources** ввели в Java 7. Дает возможность объявлять один или несколько ресурсов в блоке try, которые будут закрыты автоматически без использования finally блока.

В качестве ресурса можно использовать любой объект, класс которого реализует интерфейс java.lang.AutoCloseable или java.io.Closable.

Если try блок также выбрасывает исключение, оно побеждает, а исключение из close() метода подавляется(игнорируется).

Получить подавленные исключения - вызов метода **Throwable.getSuppressed** из исключения, созданного блоком try.

### Вопросы 6 Дополнительные

#### Вопрос 6.1 Как написать собственное (“пользовательское”) исключение?

Необходимо унаследоваться от базового класса требуемого типа исключений (например от Exception или RuntimeException).

#### Вопрос 6.2 Что вы знаете о OutOfMemoryError?

OutOfMemoryError выбрасывается, когда виртуальная машина Java не может выделить (разместить) объект из-за нехватки памяти, а сборщик мусора не может высвободить ещё.

Область памяти, занимаемая java процессом, состоит из нескольких частей. Тип OutOfMemoryError зависит от того, в какой из них не хватило места.

#### Вопрос 6.2 Что вы знаете о SQLException

SQLException предоставляет информацию об ошибках доступа к базе данных или других ошибках связанных с работой с базами данных.

SQLException относится к checked исключениям, а значит проверяется на этапе компиляции. Сделав SQLException проверяемым — это попытка заставить разработчиков обработать исключение и обернуть его в новом уровне абстракции.

#### Вопрос 6.3 Вызов исключений

Способ 1

    Exception x = *new* Exception(); //*СОЗДАЕМ ОБЬЕКТ ОШИБКИ*

*throw* x; //*ВЫБРАСЫВАЕМ ОБЬЕКТ ОШИБКИ*

Способ 2

throw new Exception("asdasd"); //*выбрасываем ПРОСТО ОШИБКУ*

#### Тех вопрос:

**Предположим, есть блок try-finally. В блоке try возникло исключение и выполнение переместилось в блок finally. В блоке finally тоже возникло исключение. Какое из двух исключений “выпадет” из блока try-finally? Что случится со вторым исключением?**

Будет обработано в finally блоке. Если было выброшено два исключения — одно в try, второе в finally, то исключение в finally «проглотит» исключение выше. Если до блока finally исключение было обработано, то мы можем получить информацию об исключении в блоке try и тем самым не потерять исключение, которое впоследствии может быть перезаписано в finally другим исключением.

## Тема 5 Общие понятия о многопоточном исполнении кода

Вопрос 1 Гонка ресурсов. Голодание. Deadlock(Взаимная блокировка)  
Семафоры, мьютексы, мониторы.

**Гонка ресурсов**

Состояние гонки — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода.

**Голодание**

В Java имеется планировщик потоков (Thread Scheduler), который контролирует все запущенные потоки и решает, какие потоки должны быть запущены и какая строка кода должна выполняться. Решение основывается на приоритете потока. Поэтому потоки с меньшим приоритетом получают меньше процессорного времени по сравнению с потоками с бо́льшим приоритет. Данное разумное решение может стать причиной проблем при злоупотреблении. То есть, если бо́льшую часть времени исполняются потоки с высоким приоритетом, то низкоприоритетные потоки начинают «голодать», поскольку не получают достаточно времени для того, чтобы выполнить свою работу должным образом.

**Deadlock(Взаимная блокировка)**

Возникают deadlock'и между двумя потоками, когда каждый из потоков захватывает один объект и начинают выполнять синхронизируемый блок кода, внутри которого обращается к другому объекту, захваченному параллельным потоком. В этом случае оба потока переходят в режим ожидания освобождения объектов, блокированных параллельными потоками.

**Семафоры**

Семафоры представляют еще одно средство синхронизации для доступа к ресурсу. В Java семафоры представлены классом **Semaphore**, который располагается в пакете **java.util.concurrent**.

Для управления доступом к ресурсу семафор использует счетчик, представляющий количество разрешений. Если значение счетчика больше нуля, то поток получает доступ к ресурсу, при этом счетчик уменьшается на единицу. После окончания работы с ресурсом поток освобождает семафор, и счетчик увеличивается на единицу. Если же счетчик равен нулю, то поток блокируется и ждет, пока не получит разрешение от семафора.

Установить количество разрешений для доступа к ресурсу можно с помощью конструкторов класса Semaphore:

* Semaphore(int permits)
* Semaphore(int permits, boolean fair)

**Мьютекс**

это в основном взаимное исключение. Только один поток может получить ресурс одновременно. Когда один поток получает ресурс, никакому другому потоку не разрешается получать ресурс, пока поток, владеющий ресурсом, не освободит его. Все потоки, ожидающие получения ресурса, будут заблокированы.

Для получения одного разрешения применяется первый вариант, а для получения нескольких разрешений - второй вариант.

После вызова этого метода пока поток не получит разрешение, он блокируется.

После окончания работы с ресурсом полученное ранее разрешение надо освободить с помощью метода **release()**:

* void release()
* void release(int permits)

Первый вариант метода освобождает одно разрешение, а второй вариант - количество разрешений, указанных в permits.

**монитор**

Несколько потоков могут мешать друг другу при обращении к одним и тем же данным. Для решения этой проблемы придуман мьютекс (он же монитор). Он имеет два состояния – объект занят и объект свободен. Монитор(мьютекс) — высокоуровневый механизм взаимодействия и синхронизации потоков, обеспечивающий доступ к неразделяемым ресурсам.

Когда какой-то нити нужен общий для всех нитей объект, она проверяет мьютекс, связанный с этим объектом. Если мьютекс свободен, то нить блокирует его (помечает как занятый) и начинает использование общего ресурса. После того, как она сделала свои дела, мьютекс разблокируется (помечается как свободен).

### Вопрос 2 Создание потоков (Thread, Runnable)

Для создания нового потока мы можем создать новый класс, либо наследуя его от класса Thread, либо реализуя в классе интерфейс **Runnable**.

**Наследование от класса Thread**

И также в классе переопределяется метод **run()**, код которого собственно и будет представлять весь тот код, который выполняется в потоке.

В методе main для запуска потока у него вызывается метод **start()**, после чего начинается выполнение того кода, который определен в методе run:

**Реализация интерфейса Runnable**

Другой способ определения потока представляет реализация интерфейса **Runnable**. Этот интерфейс имеет один метод **run**:

interface Runnable{

    void run();

}

В методе run() собственно определяется весь тот код, который выполняется при запуске потока.

После определения объекта Runnable он передается в один из конструкторов класса Thread:

### Вопрос 3 Синхронизация между потоками (synchronized, wait, notify, notifyAll; Thread.sleep)

**синхронизации потоков**: это настройка взаимодействия потоков между собой.

ключевым словом — **synchronized**. Им помечается определенный кусок нашего кода. Если блок кода помечен ключевым словом synchronized, это значит, что блок может выполняться только одним потоком одновременн

**метод переводит поток в режим ожидания**

Метод wait().

**функциональность методов notify и notifyAll**

Метод notify пробуждает один из потоков, который вызвал метод wait() у этого монитора. Метод notifyAll пробуждает все потоки. Очередность выполнения в этом случае будет определяться приоритетом потока.

Методы должны вызываться на объекте-мониторе только из синхронизированного кода. Поток, который вызывает эти методы должен владеть монитором, иначе будет выдано исключение java.lang.IllegalMonitorStateException.

Мы можем приостановить выполнение потока на заданное время с помощью статического метода **Thread.sleep().**

### Вопрос 4 Потокобезопасные concurrent коллекции

Пакет **java.util.concurrent** включает несколько потокобезопасных коллекций :

* [**ConcurrentHashMap**](http://java-online.ru/concurrent-collections.xhtml#concurrenthashmap) коллекция типа HashMap, реализующая интерфейс ConcurrentMap;
* [**CopyOnWriteArrayList**](http://java-online.ru/concurrent-collections.xhtml#copyonwritearraylist) коллекция типа ArrayList с алгоритмом CopyOnWrite;
* [**CopyOnWriteArraySet**](http://java-online.ru/concurrent-collections.xhtml#copyonwritearrayset) реализация интерфейса Set, использующая за основу CopyOnWriteArrayList;
* **ConcurrentNavigableMap** расширяет интерфейс NavigableMap;
* **ConcurrentSkipListMap** аналог коллекции TreeMap с сортировкой данных по ключу и с поддержкой многопоточности;
* **ConcurrentSkipListSet** реализация интерфейса Set, выполненная на основе класса ConcurrentSkipListMap.

## Тема 6 Основы технологии взаимодействия приложения с БД (JDBC)

**JDBC** (Java Database Connectivity)– это стандарт взаимодействия приложения с различными СУБД. JDBC основан на концепции драйверов, позволяющей получать соединение с БД по специальному url. JDBC API находятся в пакетах java.sql и javax.sql. С помощью JDBC API можно создавать соединения с БД, выполнять SQL запросы, хранимые процедуры и обрабатывать результаты. JDBC API упрощает работу с базами данных из Java программ.

В ходе развития языка и средств JDBC, поставщики баз данных сошлись на том, что лучше предоставить единый интерфейс API только на Java для доступа к базам данных средствами SQL, а также диспетчер драйверов, который позволил бы подключать к базам драйверы независимых производителей.

### Вопрос 1 Driver, DriverManager, Connection

JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает.

JDBC – Java DataBase Connectivity – соединение с базами данных на Java. Это стандарт взаимодействия с БД. Его интерфейс поставляется в стандартной JDK в виде пакета java.sql. Какой бы библиотекой для работы с базой данных вы ни пользовались, в ее основе почти всегда лежит низкоуровневый JDBC.

#### **JDBC Driver**

реализация JDBC для определенной базы данных. В приложении может быть зарегистрировано несколько разных драйверов. При соединении к базе, нужный выбирается исходя из URL соединения.

Различают четыре типа драйверов.Java программа работает с БД в двух частях. Первая часть это JDBC API, а вторая – драйвер, который и выполняет всю работу. Каждый тип определяет реализацию JDBC драйвера по возрастающей степени независимости от платформы, производительности и легкости администрирования. Эти четыре типа следующие:

* **Тип 1: JDBC-ODBC бридж** (**JDBC-ODBC Bridge plus ODBC Driver)**– транслирует JDBC в ODBC и для взаимодействия с базой данных использует драйвер ODBC. Компания Sun включила в состав JDK один такой драйвер — мост JDBC/ODBC. Сейчас имеются более удачные реализации. Платформо зависимый

Не поддерживается с Java 8..

* **Тип 2: Нативный API/частично Java драйвер (Native API partly Java technology-enabled driver)** – переводит вызовы JDBC в вызовы специфичные к СУБД таких как например SQL Server, Informix, Oracle или Sybase. Драйвер 2-го типа общается напрямую с сервером базы данных, следовательно он требует, чтобы какой то бинарный код был на стороне клиентской машины.

Делегирует работу с базой библиотеке, установленной в системе. Платформозависимый. Библиотеки в отличие от ODBC специфичны для конкретной базы, поэтому такой драйвер обычно работает быстрее.

* **Тип 3: Сетевой протокол/«чистый» Java драйвер (Pure Java Driver for Database Middleware)** – Работает с промежуточным слоем (JavaEE сервером приложений), который транслирует запросы в сетевой протокол, с которым работает конкретная БД. Такой драйвер полностью реализован на Java.

**Тип 4: Нативный протокол/«чистый» Java драйвер (Direct-to-Database Pure Java Driver)** – Полная реализация протокола взаимодействия с базой данных. Отличается от типа 3 тем, что здесь логика протокола находится не на внешнем промежуточном слое, а прямо в самом драйвере.

#### JDBC Connection

соединение, установленное с сервером базы данных. Это своего рода сессия или Socket Connection. Для создания JDBC connection требуется всего два шага:

* Зарегистрировать и загрузить драйвер с помощью Class.forName(). Класс драйвера будет зарегистрирован для DriverManager и загружен в память.

Class.forName("com.mysql.cj.jdbc.Driver");

* Используя DriverManager.getConnection() получить объект Connection. В метод необходимо передать URL базы данных, имя и пароль.

Connection conn = DriverManager.getConnection(url, userName, pass);

#### JDBC DriverManager

фабрика, через которую можно получить объект Database Connection. После загрузки JDBC драйвера в память, он регистрирует себя в DriverManager. DriverManager используется для получения подключения с помощью зарегистрированных драйверов (метод getConnection()).

### Вопрос 2 Statement, PreparedStatement, CallableStatement

Connection – создание и управление соединениями

Statement – выполнение и управление запросами к базе данных

ResultSet – получение и управление результирующими наборами (Все интерфейсы реализуются в драйверах JDBC поставщиков баз данных)

#### **Statement**

используется для выполнения SQL запросов к базе данных.

Объект Statement можно получить с помощью метода Connection.getStatement().

Statement staitment = connection.createStatement()

Вызывая методы execute(), executeQuery(), executeUpdate() и др., можно выполнять различные статичные SQL запросы.

Созданный объект можно использовать для выполнения запроса SQL, передавая его в один из методов:

методов командам SQL:

executeUpdate(String sql) – CREATE, DROP, INSERT, UPDATE, DELETE

executeQuery(String sql) – SELECT

execute(String sql) – универсальный метод

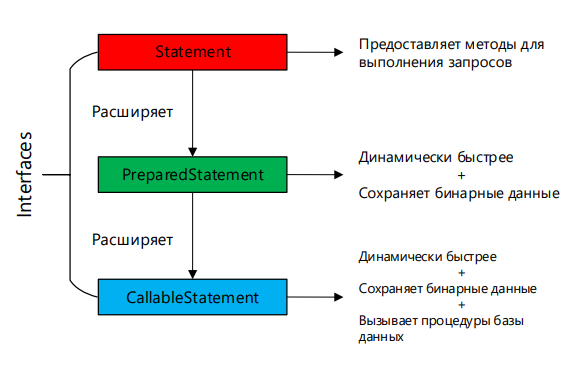
По умолчанию только один объект ResultSet для каждого Statement может быть открыт в одно и то же время. Таким образом, если необходимо работать с несколькими объектами ResultSet одновременно, мы должны использовать различные объекты Statement. Все execute() методы в интерфейсе Statement закроют текущий открытый объект ResultSet при выполнении.

Для представления запросов существует еще два типа объектов PreparedStatement и CallableStatement.

PreparedStatement – используются для часто повторяющихся запросов SQL.

CallableStatement – нужен для выполнения хранимых процедур, созданных средствами CУБД.

Оператор подготовленного запроса (PreparedStatement) предварительно готовится и хранится в объекте, что ускоряет обмен информацией с базой данных при многократном выполнении однотипных запросов.



#### PreparedStatement

Используется для выполнения прекомпилированных SQL-запросов с или без входных (IN) параметров.

 PreparedStatement preparableStatement = connection.prepareStatement(sql)

Мы можем использовать сеттеры для установки значений в запрос. Т.к. PreparedStatement является предкомпилированным, то он может быть эффективно использован множество раз. PreparedStatement считается лучшим выбором нежели Statement, т.к. он автоматически обрабатывает специальные символы, а так же предотвращает, так называемые, SQL injection attack (когда в запрос можно подставить свой код).

Каждая главная переменная в запросе обозначается знаком вопроса (?). Подготовка запроса с конкретными отсутствующими параметрами осуществляется вызовом метода prepareStatement(String sql) интерфейса Connection возвращающим объект типа PreparedStatement:

PreparedStatement ps = connection.prepareStatement(“INSERT INTO Books (name, price) VALUES (*?*, *?*)”);

Установка входных значений конкретных параметров объекта осуществляется вызовом одного из методов setXxx(int index, Xxx x), предусмотренных для каждого из типов Java:

ps.setString(1, ‘Book’);

ps.setDouble(2, ’00.000’);

#### CallableStatement

Подобно тому, как объект Connection создает объекты Statement и PreparedStatement, он также создает объект CallableStatement, который будет использоваться для выполнения вызова хранимой процедуры базы данных.

Для выполнения хранимых процедур которые находятся непосредственно в БД служит интерфейс CallableStatement. Особенностью этого процесса есть то, что он способен обрабатывать входные (IN), выходные (OUT) и смешанные (INOUT) параметры. Пример выполнения хранимых процедур:

CallableStatement callStat = connection.prepareCall(“{call PROCEDURE(*?*, *?*)}”); callStat.setXxx(1, …);

callStat.setXxx(2, …);

callStat.registerOutParameter(2, Types.TYPE);

callStat.execute();

**Множественные результаты**

По запросу могут быть возвращены множественные результаты. Получить все результирующие наборы можно следующим образом

execute()

getResultSet()

getMoreResults()

### Вопрос 3 ResultSet, RowMapper

#### ResultSet

**JDBC ResultSet** – интерфейс, объект которого создается в результате запроса к базе данных. Его можно представить в виде таблицы данных, которая была сформирована в ответ на запрос.

Объект ResultSet поддерживает курсор, который указывает на текущую строку данных. При инициализации курсор устанавливается до первой строки. Для движение по строкам используется метод next(). При наличии строк после текущей позиции, метод next() возвращает true, что можно использовать для итерации по таблице полученных результатов.

Выполнение метода executeQuery(String sql) возвращает объект типа ResultSet, в который помещаются результаты выполнения запроса.

ResultSet rs = statement.executeQuery("SELECT \* FROM TableName");

Для анализа результирующего набора организуется цикл:

*while*(rs.next()) { // *проанализировать строку }*

Прокручиваемые результирующие наборы позволяют произвольно перемещаться по всему набору на любую позицию. Для получения такого набора необходимо установить специальные параметры для метода createStatement() или prepareStatement(String sql) :

Statement statement = connection.createStatement(type, concurrency);

PreparedStatement prepStat = connection.preapreStatement(command, type, concurrency);

**concurrency**

* CONCURE\_READ\_ONLY
* CONCURE\_UPDATABLE

**Type**

* TYPE\_FORWARD\_ONLY
* TYPE\_SCROLL\_INSENSITIVE
* TYPE\_SCROLL\_SENSITIVE

Чтобы изменить полученные в результирующем наборе данные, а также автоматически обновить БД, необходимо для параметра concurrency установить значение CONCUR\_UPDATABLE. Для обновления значений полей базы данных существуют методы updateXxx(param), предусмотрены для всех типов SQL. Эти методы принимают на вход два параметра: индекс (имя) поля таблицы и его новое значение.

ResultSet resultSet = statement.executeQuery(query);

*while*(resultSet.next) {

    String newName = “Something”;

    String name = resultSet.getString(“name”);

    resultSet.updateString(“name”, newName);

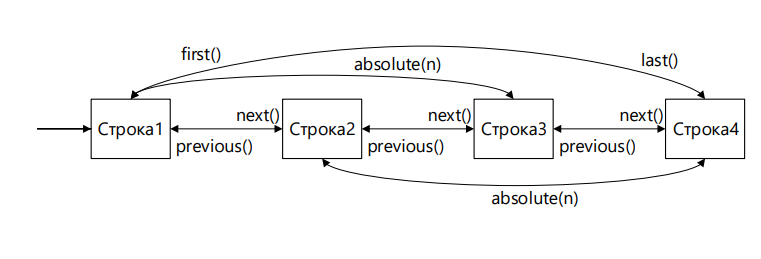
    resultSet.updateRow();

}

Обновляемые результирующие наборы позволяют обновлять, вставлять и удалять записи с таблиц БД используя методы: updateRow(), insertRow(), deleteRow().

ПРИМЕР

Для перемещения по результирующему набору предназначены методы:



#### RowSet

Наборы строк используются для переноса данных куда либо. Их главное преимущество – отсутствие необходимости постоянного подключения к БД. Получение набора строк (аналогичны для всех интерфейсов):

RowSetFactory factory = RowSetProvider.newFactory();

CachedRowSet crs = factory.createCachedRowSet();

Интерфейс CachedRowSet расширяет интерфейс ResultSet. Кроме того, CachedRowSet позволяет изменять данные, которые находятся в нём. Чтобы изменения были внесены в БД, необходимо задать явное подключение к ней для объекта типа CachedRowSet. Для просмотра и изменения данных в кэшированном наборе предназначены те же методы, что и для результирующего набора. Для внесения изменений в БД необходимо осуществить один из следующих вызовов:

crs.acceptChanges(connection);

или

crs.acceptChanges();

Также можно создавать «страницы», используя методы setPageSize(int n) и перелистываться по них методом nextPage().

**Метаданные** – информация о базе данных, её компонентах и структуре. Для получения таких сведений нужен объект типа **DatabaseMetaData**, который получается следующим образом:

DatabaseMetaData metadata = connection.getMetaData();

Для получения сведений о всех таблицах базы данных необходимо вызвать метод getTables():

ResultSet resultSet = metadata.getTables(null, null, null, *new* *String*[] {"TABLE"});

Информацию о результирующем наборе предоставляет интерфейс ResultSetMetaData. Получив результирующий набор по запросу, можно определить количество столбцов, их имена, типы данных и ширину.

ResultSetMetaData rsmd = resultSet.getMetadata();

**Управление соединениями, командами и результирующими наборами**

Каждый объект типа Connection может создать один или несколько объектов типа Statement. Но для такого объекта допускается наличие не более одного открытого результирующего набора.

По окончанию работы с объектом типа ResultSet, Statement и Connection необходимо вызвать метод close() для освобождения ресурсов памяти.

Начиная с версии Java 7 для гарантированного закрытия соединений можно использовать конструкцию try with resources.

Для доступа к данным полей строки используют методы, предусмотренные для каждого типа данных Java: getString(int index), getInt(int index), getDouble(int index), … , getXxx(int index) или такие же методы со строковым параметром.

**RowMapper**

Чаще всего при работе со списками в restful-сервисах, построенных на Spring Boot, вы будете использовать RowMapper. Этот класс обрабатывает отдельно каждую запись, полученную из БД, и возвращает уже готовый объект - модель данных.

### Вопрос 4 Управление транзакциями

По умолчанию, при создании подключения к базе данных будет выбран auto-commit mode. Это означает, что при каждом выполнении запроса он будет подтвержден автоматически по завершению. Каждый SQL запрос является транзакционным и выполняя какие-либо DML или DDL запросы по их завершению изменения будут приняты (сохранены) базой данных. Если у нас есть необходимость отказываться от сохранения выполнения какого-либо запроса (или групп запросов) в случае, если что-то пошло не так, то мы можем воспользоваться поддержкой транзакций в JDBC API.

С помощью метода setAutoCommit(boolean flag) можно отключить авто коммит в конкретном соединении. Следует отметить, что при отключении auto-commit не одно изменение не будет сохранено в базе данных до вызова метода commit() и за этим необходимо следить. Сервер базы данных будет блокировать необходимую часть базы данных до подтверждения транзакции, а так как это ресурсоёмкая задача, то подтверждать транзакцию необходимо сразу после выполнения задачи.

**Транзакция** – группа логических взаимосвязанных SQL запросов, таких, что только их общее выполнение приводит к внесению изменений в базу данных. Если хотя бы одна команда не будет успешной, то вся транзакция откатывается.

По умолчанию соединение с базой данных находится в режиме **автоматической фиксации**. Для отключения этого режима предназначен метод setAutoCommit(boolean x) вызываемый объектом типа Connection. После отключения автоматической фиксации вызывается метод executeUpdate(String sql) столько раз, сколько необходимо. И в случае успешного завершения всех команд осуществляется фиксация результатов выполнения методом commit().

В случае, если хотя бы одна команда не выполнилась успешно, осуществляется откат всей транзакции. Для этого предназначен метод rollback().

Некоторые драйверы СУБД позволяют создавать точки сохранения. Они позволяют вернуться на определённую точку выполнения команд не отменяя всю транзакцию. Для получения точки сохранения необходим объект типа **Savepoint**, который получается следующим образом:

Savepoint spt = conn.setSavepoint();

Чтобы вернуться в случае отката на точку сохранения, необходимо передать объект Savepoint в метод rollback(). После окончания работы с точкой сохранения, её следует освободить:

connection.releaseSavepoint(spt);

**Групповые обновления**

Для выполнения большого количества команд INSERT можно использовать групповые обновления. Для группового обновления можно использовать команды управления данными (INSERT, UPDATE, DELETE) и команды определения данных (CREATE TABLE и DROP TABLE).

Чтобы добавить команду в группу, необходимо вызвать метод addBatch(String sql):

String command = “INSERT INTO Table (field1, field2) VALUES (value1, value2)”;

statement.addBatch(command);

После добавления, все команды выдаются вместе для группового обновления методом executeBatch().

#### Уровни изоляции транзакций

Для блокирования различных типов чтения, предусмотрены уровни изоляции транзакций, которые определено в виде констант интерфейса Connection

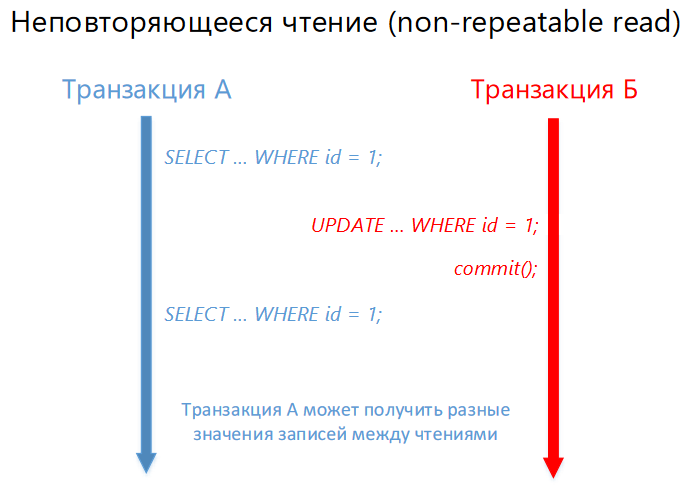
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Уровень изоляции | Транзакции | Dirty reads | Non-repeatable reads | Phantom reads |
| TRANSACTION\_NONE | Не поддерживает | Не применяется | Не применяется | Не применяется |
| TRANSACTION\_READ\_COMMITTED | поддерживает | Предотвращает | Разрешает | Разрешает |
| TRANSACTION\_READ\_UNCOMMITTED | поддерживает | Разрешает | Разрешает | Разрешает |
| TRANSACTION\_REPEATABLE\_READ | поддерживает | Предотвращает | Предотвращает | Разрешает |
| TRANSACTION\_SERIALIZABLE | поддерживает | Предотвращает | Предотвращает | Предотвращает |

Установить уровень изоляции транзакций можно с помощью метода setTransactionIsolation(int level), а получить методом int getTransactionIsolation(). Эти методы находятся в интерфейсе Connection.

#### Управление подключением к базам данных в веб-приложениях и производственных приложениях

* При установке приложения JDBC в корпоративной среде, управление подключениями осуществляется с помощью интерфейса JNDI (Java Naming and Directory Interface – интерфейс для служб каталогов и именования в Java );
* Вместо DriverManager для поиска источника данных применяется служба JNDI. В качестве источника данных служит интерфейс DataSource;
* Сведения для настройки базы данных (имя службы JNDI, URL, имя пользователя и пароль) хранятся в конфигурационном файле или задаются через ГПИ администратора;
* Для рационального использования ресурсов используется пул соединений.

#### Типы чтения транзакций



### Вопрос 5 Выполнение различных видов SQL-запросов через JDBC

*public* void update(ProductsEntity productsEntity) *throws* SQLException

    {

        String sql = "UPDATE products SET name=?,  prise=? WHERE id=?";

*try*(Connection connection = getConnection())

        {

*try*(PreparedStatement preparableStatement = connection.prepareStatement(sql))

            {

                preparableStatement.setString(1, productsEntity.getName());

                preparableStatement.setInt(2, productsEntity.getPrise());

                preparableStatement.setLong(3, productsEntity.getId());

                preparableStatement.executeUpdate();

            }

        }

    }

*public* ProductsEntity getById(Long id) *throws* SQLException

    {

        String sql = "SELECT \* FROM products WHERE  id = ?";

        ProductsEntity prod = *new* ProductsEntity();

*try*(Connection connection = getConnection())

        {

*try*(PreparedStatement preparableStatement = connection.prepareStatement(sql))

            {

                preparableStatement.setLong(1, id);

                ResultSet resultSet = preparableStatement.executeQuery();

*if*(resultSet.next())

                {

                prod.setId(resultSet.getLong("id"));

                prod.setName(resultSet.getString("name"));

                prod.setIdCategory(resultSet.getLong("id\_category"));

                prod.setPrise(resultSet.getInt("prise"));

                }

            }

        }

*return* prod;

    }

## Тема 7 Lambda

Основу лямбда-выражения составляет **лямбда-оператор**, который представляет стрелку **->**. Этот оператор разделяет лямбда-выражение на две части: левая часть содержит список параметров выражения, а правая собственно представляет тело лямбда-выражения, где выполняются все действия.

Лямбда-выражение не выполняется само по себе, а образует реализацию метода, определенного в **функциональном интерфейсе**. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

**функциональный интерфейс**  
Если интерфейс в Java содержит один и только один абстрактный метод, то он называется функциональным. Этот единственный метод определяет назначение интерфейса.

В Java, лямбда-выражения имеют следующий синтаксис:

(parameter list) -> lambda body

## Тема 8 Generics

### Вопрос 1 Назначение и синтаксис

**Дженерики** (обобщения) —средства языка Java для реализации обобщённого программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания.

В угловых скобках используется **T** — имя параметра типа. Это имя используется в качестве заполнителя, куда будет подставлено имя реального типа, переданного классу **MyClass** при создании реальных типов. То есть параметр типа **T** применяется в классе всякий раз, когда требуется параметр типа. Угловые скобки указывают, что параметр может быть обобщен. Сам класс при этом называется обобщенным классом или параметризованным типом.

### Вопрос 2 Использование в классах(включая наследование), методах, полях, переменных.

Синтаксис в классах и переменные

*public* *class* Box<T> // *Параметризация класса типом Т*

    {

*private* T t; // *Создание поля типа Т которая принимает тип, переданный в класс*

*public* T getSomething(T... a) //*Метод принимает и возвращает тип Т так как он объявлен в классе*

        {

            T b;

*return* b;

        }

    }

*public* *class* Box2<T> *extends* Box<T> // *Пробрасываем параметризацию дальше*

    {

    }

*public* *class* Box3 *extends* Box2<Number> // *Принимаем только box2 параметризированный типом Number*

    {

    }

*public* *class* Box4 // *не параметризируем класс*

    {

*private* T t; // *Не правильная запись так как Т не известный тип*

*public*<T> T getSomething(T... a) //*Мы можем отдельно параметризировать Метод*

        {

            T b; //*Параметризированная переменная в методе*

*return* b;

        }

    }

**Ограничения**

К типам, указываемым в дженериках мы можем применить ограничение. Например, мы хотим, чтобы контейнер принимал на вход только Number

*public* *static* *class* NumberContainer<T *extends* Number> {}

### Вопрос 3 Wildcards - ?

Wildcard — это дженерик вида <?>, что означает, что тип может быть чем угодно. Используется, например, в коллекциях, где для всех коллекций базовым типом является Сollection<?>.

#### Вопрос 3.1 **extends и super в Generic'ах**

**Lower Bounded Wildcards** - <[? super Integer](https://docs.oracle.com/javase/tutorial/java/generics/lowerBounded.html)>Запись вида "? extends ..." или "? super ..." — называется или символом подстановки, с верхней границей (extends) или с нижней границей (super). List<? extends Number> может содержать объекты, класс которых является Number или наследуется от Number. List<? super Number> может содержать объекты, класс которых Number или  у которых Number является наследником (супертип от Number).

Простыми словами:

Чтобы наложить ограничение на wildcard необходимо использовать конструкции типа:

* ? extends SomeClass — означает, что может быть использован любой класс-наследник SomeClass
* ? super SomeClass — означает, что может быть использован класс SomeClass, либо класс-родитель (или интерфейс) SomeClass

### Вопрос 4 PECS

**PECS — Producer Extends Consumer Super.(Продюсер производит потребитель принимает)**

Если мы объявили ***wildcard с extends*,** то это *producer*. Он только «продюсирует», предоставляет элемент из контейнера, а сам ничего не принимает.

Если же мы объявили ***wildcard с super*** — то это *consumer*. Он только принимает, а предоставить ничего не может.

#### Вопрос 4.1 ****ковариантность и контравариантность****

[Формально](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B2%D0%B0%D1%80%D0%B8%D0%B0%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B8_%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%B0%D0%B2%D0%B0%D1%80%D0%B8%D0%B0%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), ковариантность/контравариантность типов – это сохранение/обращение порядка наследования для производных типов. Проще говоря, когда у ковариантных сущностей типами-параметрами являются родитель и наследник, они сами становятся как бы родителем и наследником. Контравариантные наоборот, становятся наследником и родителем.

**Ковариантностью** называется сохранение иерархии наследования исходных типов в производных типах в том же порядке. Так, если класс Cat наследуется от класса Animal, то естественно полагать, что перечисление IEnumerable<Cat> будет потомком перечисления IEnumerable<Animal>. Действительно, «список из пяти кошек» — это частный случай «списка из пяти животных». В таком случае говорят, что тип (в данном случае обобщённый интерфейс) IEnumerable<T> ковариантен своему параметру-типу T.

**Контравариантностью** называется обращение иерархии исходных типов на противоположную в производных типах. Так, если класс String наследуется от класса Object, а делегат Action<T> определён как метод, принимающий объект типа T, то Action<Object> наследуется от делегата Action<String>, а не наоборот. Действительно, если «все строки — объекты», то «всякий метод, оперирующий произвольными объектами, может выполнить операцию над строкой», но не наоборот. В таком случае говорят, что тип (в данном случае обобщённый делегат) Action<T> контравариантен своему параметру-типу T.  
  
**Инвариантность** — отсутствие наследования между производными типами.

Простыми словами:

* **Ковариантность**: List<Integer> можно присвоить в переменную типа List<? extends Number> (как будто он наследник List<Number>).

**Ковариантностью** называется сохранение иерархии наследования исходных типов в производных типах в том же порядке.

* **Контравариантность**: в качестве параметра метода List<Number>#sort типа Comparator<? super Number> может быть передан Comparator<Object> (как будто он родитель Comparator<Number>)

Отношение типов «можно присвоить» – не совсем наследование, такие типы называются *совместимыми* (отношение «is a»).

### Доп Вопрос 1 Стирание типов

Внутри класса-дженерика не хранится информация о его типе параметре. Это и называется стиранием типов. На стадии компиляции происходит приведение объекта класса к типу, который был указан при объявлении.

Компилятор удаляет из байткода класс-файла информацию о типах-дженериках. Этот процесс и называется [**стирание типов**](https://docs.oracle.com/javase/tutorial/java/generics/erasure.html) (type erasure). Он появился в Java 5 вместе с самими дженериками. Такое решение позволило сохранить обратную совместимость без перекомпилляции кода Java 4.

Стирание состоит из трех действий:  
***🔘*** Если параметры ограничены (bounded), вместо типа-параметра в местах использования подставляется верхняя граница, иначе Object;  
***🔘*** В местах присвоения значения типа-параметра в переменную обычного типа добавляется каст к этому типу;  
***🔘*** Генерируются *bridge-методы*.

Информация о типах стирается только из методов и полей, но остается в метаинформации самого класса. Получить эту информацию в рантайме можно с помощью рефлекшна, методом Field#getGenericType.

Тип-параметр может иметь несколько верхних границ, то есть границу-пересечение типов: <T extends Comparable & Serializable>. Стирание произойдет до первой из границ, остальные послужат только ограничением вариантов фактического типа. Поэтому граница-класс, при наличии, должна быть указана раньше границ-интерфейсов.

### Доп Вопрос 2 ****Что такое и когда нужно использовать raw types?****

*raw type*. В Java так называют generic-типы без указания типа-параметра. Такая языковая конструкция валидна, но в большинстве случаев приводит к предупреждению компилятора.  
  
Предупреждение связано с риском получения проблемы heap pollution. Ей мы уже посвящали публикации [ранее](https://itsobes.ru/JavaSobes/chto-takoe-heap-pollution). Использование raw types никогда не оправдано – спецификация языка явно говорит: их поддержка остается только для обратной совместимости.  
  
Есть всего три случая, когда использовать обобщенный тип без параметра правильно:  
• Целевая версия Java < 5.0 (2002 год и ранее – вряд ли это ваш случай);  
• В литерале класса. List<String>.class не сработает, нужно писать List.class;  
• В операторе instanceof. Вместо instanceof Set<Integer> должно быть instanceof Set.

### Доп Вопрос ****3 Что такое heap pollution****

Массивы Java ковариантны. А значит, можно обратиться к объекту типа String[] через переменную типа Object[], и положить туда например Integer. Такой код скомпилируется, но в момент записи произойдет ArrayStoreException.  
  
Дженерики защищены инвариантностью. Если попытаться положить List<Object> в List<String>, эта же по сути ошибка произойдет уже на этапе компиляции.  
  
[Heap pollution](https://en.wikipedia.org/wiki/Heap_pollution) – ситуация, когда эта защита не срабатывает, и переменная параметризованного типа хранит в себе объект, параметризованный другим типом.

Heap pollution может произойти в двух случаях: при использовании массивов дженериков и при смешивании параметризованных и raw-типов.  
  
Raw types – это параметризованные типы без указания параметра. Пример с raw types, приводящий к heap pollution, уже был описан выше:

    List<String> strings = (List) *new* ArrayList<Integer>();

### Тех вопросы: **Отличается ли List<?>; от List<? extends Object>;?**

Все классы без исключения наследуются от Object. Поэтому [неограниченный](https://itsobes.ru/JavaSobes/kak-ogranichivaetsia-tip-generic-parametra) wildcard <?> всегда подразумевает его в качестве верхней границы. Оба этих типа в рантайме [сотрутся](https://itsobes.ru/JavaSobes/chto-takoe-type-erasure) в List<Object>, функциональных отличий нет.  
  
Не смотря на одинаковое поведение, существует одно синтаксическое различие. Неограниченный дженерик – reifiable тип. Это значит, что он представлен в рантайме. Такой тип можно использовать в операторе instanceof, тогда как синтаксическая конструкция x instanceof List<? extends Object> приведет к ошибке компиляции.

## Тема 9 Аннотации и их виды

### **Вопрос 1** Назначение и синтаксис

**Аннотации** - дескрипторы, включаемые в текст программы, и используются для хранения метаданных программного кода, необходимых на разных этапах жизненного цикла программы.

это метаданные, привязанные к элементам исходного кода программы и не влияющие на работу кода, с которым они работают.

Их типичными случаями использования являются:

* **Информация для компилятора** – с помощью аннотаций компилятор может обнаруживать ошибки или подавлять предупреждения
* **Обработка во время компиляции и развертывания** -программные средства могут обрабатывать аннотации и генерировать код, файлы конфигурации и т. Д.
* **Обработка во время выполнения** – аннотации могут быть проверены во время выполнения, чтобы настроить поведение программы

Информация, хранимая в аннотациях, может использоваться соответствующими обработчиками для создания необходимых вспомогательных файлов или для маркировки классов

##### (Пометка)

Аннотация — эквивалент маркерного интерфейса. С одним отличием в использовании - интерфейсы маркеров следуют логике полиморфизма. Все классы, реализующие маркерный интерфейс, наследуют всю функциональность

### Вопрос 2 Какие есть анотации

#### Вопрос 2.1 **Мета-Аннотации**

Мета-Аннотации **-** аннотации, которые применяются к другим аннотациям.

Вообще мета-аннотациями можно назвать любую аннотацию с таргетом ANNOTATION\_TYPE пример:

    @*Target*(ElementType.ANNOTATION\_TYPE)

но основных в Java существует 5. Они определяют для аннотации:

**Аннотация**[**@Retention**](http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Retention.html) позволяет указать жизненный цикл аннотации: будет она присутствовать только в исходном коде, в скомпилированном файле, или она будет также видна и в процессе выполнения. Выбор нужного типа зависит от того, как вы хотите использовать аннотацию, например, генерировать что-то побочное из исходных кодов, или в процессе выполнения стучаться к классу через reflection.

**Тип хранения**

Под "типом хранения" понимается стадия до которой "доживает" наша аннотация внутри класса. Каждая аннотация имеет только один из возможных "типов хранения" указанный в классе RetentionPolicy:

* **SOURCE** - используется только при написании кода и игнорируется компилятором (не сохраняется после компиляции). Обычно используется для препроцессоров (условно), либо указаний компилятору
* **CLASS** – (выбрана по умолчанию) сохраняется после компиляции, игнорируется JVM (т.е. не может быть использована во время выполнения). Обычно используется для сторонних сервисов, подгружающих ваш код в качестве plug-in приложения
* **RUNTIME** - сохраняется после компиляции и подгружается JVM (т.е. может использоваться во время выполнения самой программы). Используется в качестве меток в коде, которые напрямую влияют на ход выполнения программы

**Аннотация**[**@Target**](http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Target.html) указывает, что именно мы можем пометить этой аннотацией, это может быть поле, метод, тип и т.д.  
**Тип объекта над которым указывается**

Данное описание стоит понимать практически буквально, т.к. в Java аннотации могут указываться над чем угодно (Поля, класса, функции, т.д.) и для каждой аннотации указывается, над чем конкретно она может быть задана. Здесь уже нет правила "что-то одно", аннотацию можно указывать над всем ниже перечисленным, либо же выбрать только нужные элементы класса ElementType:

* **ANNOTATION\_TYPE** - другая аннотация
* **CONSTRUCTOR**- конструктор класса
* **FIELD**- поле класса
* **LOCAL\_VARIABLE**- локальная переменная
* **METHOD**- метод класса
* **PACKAGE**- описание пакета package
* **PARAMETER**- параметр метода public void hello(@Annontation String param){}
* **TYPE** - указывается над классом

*–– С Java 8 ––*

* **TYPE\_PARAMETER** – Обявление generic типа-параметра.
* **TYPE\_USE** – Любое использование типа. Например приведение: (@NonNull String) myObject.  
  *–– С Java 9 ––*
* **MODULE** – Объявление модуля.

**Аннотация** **@Inherited** помечает аннотацию, которая будет унаследована потомком класса, отмеченного такой аннотацией.

**Аннотация** **@Repeatable** - чтобы сделать аннотацию повторяющейся

**Аннотация**[**@Documented**](http://docs.oracle.com/javase/7/docs/api/java/lang/annotation/Documented.html)указывает, что помеченная таким образом аннотация должна быть добавлена в javadoc поля/метода и т.д.

##### (Пометка)

**Можно ли расширить аннотации -** через extends нет. Если мы попытаемся использовать предложение extends в объявлении аннотации, мы получим ошибку компиляции

#### Вопрос 2.2 Аннотации, применяемые к исходному коду:

**Аннотация  @Override** — сообщает компилятору, что мы собираемся переопределить метод родительского класса.

**Аннотация  @Deprecated** — отмечает, что метод устарел и не рекомендуется к использованию.

опциональные параметры – строка since и булево значение forRemoval. В since указывается версия *вашего приложения*, начиная с которой элемент считается устаревшим. Значение true в forRemoval предупреждает, что со следующей [мажорной](https://semver.org/lang/ru/) версии этот элемент будет удален.

**Аннотация  @SuppressWarnings** —для устранения предупреждений, создаваемых компилятором.

**Аннотация  @SafeVarargs** — никакие небезопасные действия, связанные с параметром переменного количества аргументов, недопустимы. Применяется только к методам и конструкторам с переменным количеством аргументов, которые объявлены как **static** или **final**.

**Аннотация  @functional Interface** – введено в Java 8, указывает, что объявление типа является функциональным интерфейсом и реализация которого может быть предоставлена с помощью лямбда-выражения

### Вопрос 3 Создание своей аннотации.

Аннотации создаются с использование механизма, основанного  
на интерфейсе.

Все поля в аннотациях не явно объявляются public abstract

Все поля в аннотациях работают как методы

Все default значения являются константами

Пример объявление аннотации:

@*Target*(value=ElementType.FIELD)

@*Retention*(value= RetentionPolicy.RUNTIME)

*public* @*interface* *Name* {

     String name();

     String type() *default*  “string”; //Нельзя создавать объекты -> default new String(“string”)

}

**Пример** **@Repeatable**

    // *сделать аннотацию MyAnno повторяющейся*

    @*Retention*(RetentionPolicy.RUNTIME)

    @*Repeatable*(MyRepeatedAnnos.class)

    @*interface* *MyAnno* {

        String str() *default* "Тестирование";

        int val() *default* 9000;

    }

    // *Это контейнерная аннотация*

    @*Retention*(RetentionPolicy.RUNTIME)

    @*interface* *MyRepeatedAnnos* {

*MyAnno*[] value();

    }

*class* RepeatAnno {

        // *повторить аннотацию MyAnno в методе myMeth()*

    @*MyAnno*(str = "Первая аннотация", val = -1)

    @*MyAnno*(str = "Вторая аннотация", val = 100)

*public* *static* void myMeth(String str, int i) {

….

}

    }

### Доп Вопрос 1 Какие Элементы Программы Могут Быть Аннотированы

Аннотации могут быть применены в нескольких местах по всему исходному коду. Они могут быть применены к объявлениям классов, конструкторов и полей

Методы и их параметры:

@*SimpleAnnotation*

*public* void aMethod(@*SimpleAnnotation* String param) {

Локальные переменные, включая цикл и переменные ресурсов*:*

@*SimpleAnnotation*

int i = 10;

*for* (@*SimpleAnnotation* int j = 0; j < i; j++) {

}

*try* (@*SimpleAnnotation* FileWriter writer = getWriter()) {

### Доп Вопрос 2 Какие Типы объектов Могут Быть Возвращены из Объявления Метода Аннотации?

Возвращаемый тип должен быть примитивом, Строкой , Классом , Перечислением или массивом одного из предыдущих типов. В противном случае компилятор выдаст ошибку.

### Доп Вопрос 3 поле типа Class

при обработке аннотаций в зависимости от указанного в поле класса вызывать нужный обработчик

Экземпляры класса Class представляют классы и интерфейсы в запущенном Java-приложении. Перечисление — это своего рода класс, а аннотация — это своего рода интерфейс. Каждый массив также принадлежит классу, который отражается как объект Class, который используется всеми массивами с одинаковым типом элемента и числом измерений. Примитивные типы Java (boolean, byte, char, short, int, long, float и double) и ключевое слово void также представлены как объекты класса.

Класс не имеет открытого конструктора. Вместо этого объекты Class создаются виртуальной машиной Java автоматически по мере загрузки классов и вызовов метода defineClass в загрузчике классов.

## Тема 10 Stream API

### Вопрос 1 способы создания Stream:

Stream API. Его задача - упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Вся основная функциональность данного API сосредоточена в пакете **java.util.stream**.

* Пустой стрим: Stream.empty()
* Стрим из List: list.stream()
* Стрим из Map: map.entrySet().stream()
* Стрим из массива: Arrays.stream(array)
* Стрим из указанных элементов: Stream.of("1", "2", "3")

### Вопрос 2 Конвейерные и терминальные операции

**Операторы можно разделить на две группы:**

* *Промежуточные* (“intermediate”, ещё называют “lazy”) — обрабатывают поступающие элементы и возвращают стрим. Промежуточных операторов в цепочке обработки элементов может быть много.
* *Терминальные* (“terminal”, ещё называют “eager”) — обрабатывают элементы и завершают работу стрима, так что терминальный оператор в цепочке может быть только один.

Важные моменты:

* Обработка не начнётся до тех пор, пока не будет вызван терминальный оператор. list.stream().filter(s -> s > 5) (не возьмёт ни единого элемента из списка);
* Экземпляр, стрима нельзя использовать более одного раза

**промежуточные операторы:**

* filter(Predicate predicate) фильтрует стрим, пропуская только те элементы, что проходят по условию (Predicate встроенный функциональный интерфейс, добавленный в Java SE 8 в пакет java.util.function. Проверяет значение на “*true*” и “*false*”);
* map(Function mapper) даёт возможность создать функию с помощью которой мы будем изменять каждый элемент и пропускать его дальше (Функциональный интерфейс Function<T,R> представляет функцию перехода от объекта типа T к объекту типа R)
* flatMap(Function<T, Stream<R>> mapper) — как и в случае с map, служат для преобразования в примитивный стрим.

**Терминальные операторы:**

* forEach(Consumer action) – аналог for each (Consumer<T> выполняет некоторое действие над объектом типа T, при этом ничего не возвращая);
* count() – возвращает количество елементов стрима:

### Вопрос 3 Тип Optional

**Optional**- новый класс в пакете**java.util**, является контейнером (оберткой) для значений которая также может безопасно содержать **null**.

Благодаря опциональным типам можно забыть про проверки на **null** и **NullPointerException**.

### Вопрос 4 Parallel Stream

Чтобы сделать обычный последовательный поток параллельным, надо вызвать у объекта Stream метод **parallel**. Кроме того, можно также использовать метод **parallelStream()** интерфейса Collection для создания параллельного потока из коллекции.

**Вопросы производительности в параллельных операциях**

* Фактически применение параллельных потоков сводится к тому, что данные в потоке будут разделены на части, каждая часть обрабатывается на отдельном ядре процессора, и в конце эти части соединяются, и над ними выполняются финальные операции. Рассмотрим некоторые критерии, которые могут повлиять на производительность в параллельных потоках:
* Размер данных. Чем больше данных, тем сложнее сначала разделять данные, а потом их соединять.
* Количество ядер процессора. Теоретически, чем больше ядер в компьютере, тем быстрее программа будет работать. Если на машине одно ядро, нет смысла применять параллельные потоки.
* Чем проще структура данных, с которой работает поток, тем быстрее будут происходить операции. Например, данные из ArrayList легко использовать, так как структура данной коллекции предполагает последовательность несвязанных данных. А вот коллекция типа LinkedList - не лучший вариант, так как в последовательном списке все элементы связаны с предыдущими/последующими. И такие данные трудно распараллелить.
* Над данными примитивных типов операции будут производиться быстрее, чем над объектами классов

**Упорядоченность в параллельных потоках**

Как правило, элементы передаются в поток в том же порядке, в котором они определены в источнике данных. При работе с параллельными потоками система сохраняет порядок следования элементов. Исключение составляет метод forEach(), который может выводить элементы в произвольном порядке. И чтобы сохранить порядок следования, необходимо применять метод **forEachOrdered**:

## Тема 11 Потоки данных

### Вопрос 1 Типы и назначение

Базовый класс **InputStream** представляет классы, которые получают данные из различных источников:  
– массив байтов  
– строка (String)  
– файл  
– канал (pipe): данные помещаются с одного конца и извлекаются с другого  
– последовательность различных потоков, которые можно объединить в одном потоке  
– другие источники (например, подключение к интернету)

Класс **OutputStream** – это абстрактный класс, определяющий потоковый байтовый вывод. В этой категории находятся классы, определяющие, куда направляются ваши данные: в массив байтов (но не напрямую в String; предполагается что вы сможете создать их из массива байтов), в файл или канал.

Символьные потоки имеют два основных абстрактных класса **Reader** и **Writer**, управляющие потоками символов Unicode. Класс Reader – абстрактный класс, определяющий символьный потоковый ввод. Класс **Writer** – абстрактный класс, определяющий символьный потоковый вывод. В случае ошибок все методы класса передают исключение **IOException**.

### Вопрос 2 Правильное открытие и закрытие потоков

Чтобы работать с потоком, его необходимо открыть. Для открытия потока применяется функция **fopen()**, которая имеет следующий прототип:

FILE \* fopen(имя\_файла, режим\_открытия);

Первый параметр представляет имя открываемого файла, а второй задает режим открытия, от которого зависит, как файл может быть обработан.

Функция возвращает указатель на структуру, которая имеет тип **FILE**, определенный в том же файле Этот указатель идентифицирует поток в программе и через него мы сможем обращаться к открытому файлу.

После завершения работы с файлом его следует закрыть. Для этого применяется функция **fclose()**:

int fclose(указатель\_на\_поток);

Единственный параметр функции представляет ранее полученный при открытии файла указатель на структуру FILE, связанный с файлом.

Функция возвращает число: 0 - в случае успешного выполнения и встроенное значение **EOF** в случае ошибки.

### Вопрос 3 Сериализация объектов в поток. Десериализация.

* *Сериализация* это процесс сохранения состояния объекта в последовательность байт;
* *десериализация* это процесс восстановления объекта, из этих байт. Java Serialization API предоставляет стандартный механизм для создания сериализуемых объектов.

В Java за процессы сериализации отвечает интерфейс **Serializable**.

## Тема 12 Maven

**Задачи**

* Управление зависимостями
* Компиляция
* Выполнение тестов
* Сборка в архив
* Выполнение тестов
* Деплой на сервер приложений

**Основная архитектура mvn проекта**

**Main**

* Java
* resources

**Test**

* Java
* resources

**Maven** реализует принцип ‘Convention over Configuration’, что позволяет добиться существенного сокращения объёмов конфигурационных файлов и возможности повторного использования существующих конфигураций вместо генерации новых.

**Project Object Model** – это детальное описание проекта в виде xml, содержащее детальную информацию о структуре проекта, зависимостях, особенностях сборки.

**Обязательные теги**

* **groupId** - указывает на разработчика ПО, там вы можете указать доменное имя своей компании.
* **artifactId** - название проекта. Maven оперирует так называемыми артефактами. Это приложения, плагины, архетипы и другие проекты. И ваша разработка также будет считаться артефактом.
* **archetypeVersion**-  указывает на версию артефакта «архетип».

**Репозиторий** - это место, где хранятся файлы jar, pom, javadoc, исходники и т.д. В проекте могут быть использованы :

* центральный репозиторий, доступный на чтение для всех пользователей в интернете;
* внутренний «корпоративный» репозиторий - дополнительный репозиторий группы разработчиков;
* локальный репозиторий, по умолчанию расположен в ${user.home}/.m2/repository - персональный для каждого пользователя.

### Вопрос 1 Жизненный цикл maven проекта

Жизненный цикл **maven** проекта – это чётко определённая последовательность фаз. Когда *maven* начинает сборку проекта, он проходит через определённую последовательность фаз, выполняя задачи, указанные в каждой из фаз. *Maven* имеет 3 стандартных жизненных цикла :

* clean — жизненный цикл для очистки проекта;
* default — основной жизненный цикл;
* site — жизненный цикл генерации проектной документации.

**Фазы жизненного цикла clean**

* pre-clean;
* clean;
* post-clean.

По умолчанию, используется только фаза clean (удаление target директории) • При желании, мы можем дополнить стандартное поведение, добавив цели к любой директории • Рекомендуется запускать перед каждой сборкой, чтобы быть уверенным, что все данные в проекте обновились

**Фазы жизненного цикла default**

* validate - выполнение проверки, является ли структура проекта полной и правильной;
* generate-sources - включение исходного кода в фазу;
* process-sources - подготовка/обработка исходного кода; например, фильтрация определенных значений;
* generate-resources - генерирование ресурсов, которые должны быть включены в пакет;
* process-resources - копирование ресурсов в указанную директорию (перед упаковкой);
* compile - компиляция исходных кодов проекта;
* process-test-sources - обработка исходных кодов тестов;
* process-test-resources - обработка ресурсов для тестов;
* test-compile - компиляция исходных кодов тестов;
* test - собранный код тестируется, используя приемлемый фреймворк типа [JUnit](http://java-online.ru/blog-junit.xhtml);
* package - упаковка откомпилированных классов и прочих ресурсов в дистрибутивный формат;
* integration-test - программное обеспечение в целом или его крупные модули подвергаются интеграционному тестированию. Проверяется взаимодействие между составными частями программного продукта;
* install - установка программного обеспечения в maven-репозиторий, чтобы сделать его доступным для других проектов;
* deploy - стабильная версия программного обеспечения копируется в удаленный maven-репозиторий, чтобы сделать его доступным для других пользователей и проектов;

Конечное предназначение – собрать конкретный артефакт • Содержит 23 фазы • Используемые фазы могут изменяться в зависимости от вида упаковки (packaging в pom.xml)

**Фазы жизненного цикла site**

* pre-site;
* site;
* post-site;
* site-deploy;

Нужен для генерации проектной документации в читабельном виде • В основном применяется при разработке публичных фреймворков и библиотек

### Вопрос 2 Зависимости, dependency

**Зависимость,** эта связь, которая говорит, что для некоторых фаз жизненного цикла *maven* проекта, требуются некоторые артефакты. Зависимости проекта описываются в секции *<dependencies>* файла *pom.xml*. Для каждого используемого в проекте артефакта необходимо указать GAV (groupId, artifactId, version), где

* **groupId** - идентификатор производителя объекта. Часто используется схема принятая в обозначении пакетов Java. Например, если производитель имеет домен domain.com, то в качестве значения *groupId* удобно использовать значение *com.domain*. То есть, *groupId* это по сути имя пакета.
* **artifactId**- идентификатор объекта. Обычно это имя создаваемого модуля или приложения.
* **version**- версия описываемого объекта. Для незавершенных проектов принято добавлять суффикс SNAPSHOT. Например 1.0-SNAPSHOT.

**Область действия зависимости, scope**

Область действия scope определяет этап жизненного цикла проекта, в котором эта зависимость будет использоваться.

**test**  
Если зависимость junit имеет область действия *test*, то эта зависимость будет использована maven'ом при выполнении компиляции той части проекта, которая содержит тесты, а также при запуске тестов на выполнение и построении отчета с результатами тестирования кода. Попытка сослаться на какой-либо класс или функцию библиотеки *junit* в основной части приложения (каталог src/main) вызовет ошибку.

**compile**  
К наиболее часто используемой зависимости относится *compile* (используется по умолчанию). Т.е. *dependency*, помеченная как *compile*, или для которой не указано *scope*, будет доступна как для компиляции основного приложения и его тестов, так и на стадиях запуска основного приложения или тестов. Чтобы инициировать запуск тестов из управляемого maven-проекта можно выполнив команду "mvn test", а для запуска приложения используется плагин exec.

**provided**  
Область действия зависимости *provided* аналогична *compile*, за исключением того, что артефакт используется на этапе компиляции и тестирования, а в сборку не включается. Предполагается, что среда исполнения (JDK или WEB-контейнер) предоставят данный артефакт во время выполнения программы. Наглядным примером подобных артефактов являются такие библиотеки, как hibernate или jsf, которые необходимы на этапе разработки приложения.

**runtime**  
Область действия зависимости *runtime* не нужна для компиляции проекта и используется только на стадии выполнения приложения.

**system**  
Область действия зависимости *system* аналогична provided за исключением того, что содержащий зависимость артефакт указывается явно в виде абсолютного пути к файлу, определенному в теге *systemPath*. Обычно к таким артефактам относятся собственные наработки, и искать их в центральном репозитории, куда Вы его не размещали, не имеет смысла :

### Вопрос 3 Плагины, plugins

Объявление плагина в проекте похоже на объявление зависимости. Плагины также идентифицируется с помощью GAV (groupId, artifactId, version)

**Основные исполняемые цели**

* validate — проверка корректности метаинформации о проекте;
* compile — компилиляция исходников.
* test — прогонка тестов классов;
* package — упаковка скомпилированнных классов в заданный формат (jar или war, к примеру);
* integration-test — отправка упакованных классов в среду интеграционного тестирования и прогонка тестов;
* verify — проверка корректности пакета и удовлетворение требованиям качества;
* install — отправка пакета в локальный репозиторий, где он будет доступен для использования как зависимость в других проектах;
* deploy — отправка пакета на удаленный production сервер, где доступ к нему будет открыт другим разработчикам.

### Вопрос 4 Секция свойств maven проекта, properties

Отдельные настройки проекта можно определить в переменных. Это может быть связанно, к примеру, с тем, что требуется использовать семейство библиотек определенной версии. Для этого в проектном файле используется секция <properties>, в которой объявляются переменные. Обращение к переменной выглядит следующим образом : ${имя переменной}.

### Вопрос 5 Секция build

Секция <build> является необязательной, т. к. существует значение по умолчанию. Данная секция содержит информацию по самой сборке, т.е. где находятся исходные файлы, файлы ресурсов, какие плагины используются.

* <finalName> - наименование результирующего файла сборки (jar, war, ear..), который создаётся в фазе package. Значение по умолчанию — «artifactId-version»;
* <sourceDirectory> - определение месторасположения файлов с исходным кодом. По умолчанию файлы располагаются в директории «${basedir}/src/main/java», но можно определить и в другом месте;
* <outputDirectory> - определение месторасположения директории, куда компилятор будет сохранять результаты компиляции - \*.class файлы. По умолчанию определено значение «target/classes»;
* <resources> и вложенные в неё тэги <resource> определяют местоположение файлов ресурсов. Ресурсы, в отличие от файлов исходного кода, при сборке просто копируются в директорию, значение по умолчанию которой равно «src/main/resources».

### Вопрос 6 Наследование проектов в maven

**Настройка родительского pom.xml**

В pom.xml родительского проекта необходимо определить параметры GAV (groupId, artifactId, version) и в теге <packaging> указать значение «pom». Дополнительно вводится секция <modules>, в которой перечисляются все дочерние проекты.

**Настройка дочерних pom.xml**

В pom.xml дочерних проектов необходимо ввести секцию <parent> и определить GAV-параметры родительского проекта.

## Тема 13 ORM (на примере Hibernate или Spring Data)

**JPA(Java Persistence API)** —спецификация. Предоставляет возможность сохранять в удобном виде Java-объекты в базе данных

**Hibernate** —библиотека, предназначена для задач объектно-реляционного отображения. hibernate позволяет разработчику работать с базой данных не напрямую, как мы это делали с помощью библиотеки JDBC, а с помощью представления таблиц баз данных в виде классов java.

**Spring Data JPA** — это библиотека, которая добавляет дополнительный уровень абстракции поверх ORM реализации JPA. По умолчанию Spring Data JPA использует Hibernate, в качестве ORM провайдера (чтобы выполнять запросы). Это, кстати, можно изменить используя настройки Spring.

## Тема 14 Тестирование

**JUnit** — библиотека для модульного тестирования программ Java.

**JUnit** – это Java фреймворк для тестирования, т. е. тестирования отдельных участков кода, например, методов или классов.

## Тема 15 Поддержка версионирования БД Liquibase

**Migration(Миграция)** - любые изменения в БД, вследствие запуска одного или нескольких скриптов

chaneSet - набор команд и инструкция, которые должны быть применены к БД, при запуске

changelog – фаил который содержит набор команд(changeSet)

mvn liquibase:update – команда для добавления

## Тема 16 Логирование

**Логирование** - процесс записи каких-либо событий, которые происходят в коде.

**лог** — файл или то место, куда программа производит необходимые записи.

Уровни логирования применяются в программах на различных языках программирования, в том числе и на Java.

debug — выводится информация, которая пригодится для отладки программы;

info — обычные и стандартные сообщения;

warning — нефатальное предупреждение;

error — записи ошибок;

fatal — фатальная ошибка;

off — сообщения не выводятся;

trace — информация для точной отладки;

all — выводится вся информация.

## Тема 17 Default и Static методы в интерфейсах

### Вопрос 1. Определение, синтаксис, как используется?

### Вопрос 1.1 Статические методы

Static методы в интерфейсе работают точно так же как и static методы в классе.

Наследовать static методы в интерфейсе нельзя, как и static методы в классах.

Так же нельзя вызывать static метод интерфейса из класса наследника

а) Статический метод в интерфейсе принадлежит только интерфейсу. Вызов ИмяИнтрефеса.ИмяМетода

б) Класс, который реализует интерфейс НЕ может переопределить статический метод интерфейса

в) Если создать объект класса, то нельзя вызвать статический метод интерфейса

### Вопрос 1.2 Наследование и default методы

С одной стороны, нужно было добавить новую функциональность: лямбды, Stream API, с другой стороны, нужно было оставить обратную совместимость. Для этого нужно было ввести уже готовые решения в интерфейсы. Так к нам и пришли дефолтные методы.

То есть, дефолтный метод — это реализованный метод в интерфейсе, у которого есть ключевое слово default.

default методы по умолчанию public, потому public писать НЕ надо.

Класс который имплементит данный метод может иметь реализацию (@Override), а может и не иметь.

**default** (реализация по умолчанию) метод ДОЛЖЕН иметь реализацию.

Класс который реализовывает данный интерфейс может переопределить default метод.

Если класс реализует более 2 интерфейсов, которые содержат одинаковые методы по умолчанию, то сам класс должен содержать данный переопределенный метод, или интерфейс должен наследоваться (переопределять метод).

Есть конструкция такого вида: класс.super.метод, если надо использовать из определенного интерфейса.

Если класс реализует более 2 интерфейса, которые содержат одинаковые методы по умолчанию,  
то сам класс должен содержать данный метод.

## Тема 18 Time API (Java 8)

## Тема 19 Пакет java.nio.

## Тема 20 Знакомство с реактивным программированием (один из)

## Тема 21 Docker

## Тема 22 Java Concurrency

## Тема 23 Сервлеты/JSP

## Тема 24 Spring Core

## Тема 25 Spring и DB

## Тема 26 Spring Web

## Тема 27 Безопасность приложения, Авторизация и Аутентификация (опционально, один из)

## Тема 28 Spring Boot

## Тема 29 Hibernate работы кешей

## Тема 30 Java EE fundamental

## Тема 31 Java EE specifications

## Тема 32 Java EE (Один из серверов приложений)

## Тема 33 Message Queue (опционально, один из)

# WEB

# **HTTP**

## Тема 1 Основы HTTP

HyperText Transfer Protocol – «протокол передачи гипертекста» -  протокол прикладного уровня передачи данных. Изначально задумывался для передачи документов в HTML-формате, но на данный момент используется для передачи произвольных данных

Гипертекст – текстовые страницы, имеющие внутри себя ссылки на другие текстовые страницы(перекрёстные ссылки).

Тег – метка

HTTP основан на архитектуре «клиент-сервер».

Клиентское приложение формирует **запрос** и отправляет его на сервер. Серверное программное обеспечение обрабатывает данный запрос, формирует **ответ**, которые возвращается клиенту.

Клиент в качестве отправляющей стороны отправляет HTTP-запрос на прикладном уровне (протокол HTTP) для просмотра определенной веб-страницы. Затем для удобства передачи данные (сообщение с запросом HTTP), полученные от прикладного уровня, разделяются на транспортном уровне (протокол TCP), а порядковый номер и номер порта отмечаются в каждом сообщении и передаются на сетевой уровень. На сетевом уровне (протокол IP) добавьте MAC-адрес в качестве пункта назначения связи и перенаправьте его на канальный уровень. Таким образом, запрос связи с сетью готов. Сервер на принимающей стороне получает данные на канальном уровне и отправляет их на верхний уровень по порядку, вплоть до прикладного уровня. Когда он передается на прикладной уровень, его можно рассматривать как фактическое получение HTTP-запроса, отправленного клиентом.

#### (ПОМЕТКА)

Достоинства:

* **Простота -** прост в реализации - лёгко создавать клиентские приложения.
* **Расширяемость -** внедрение своих собственных заголовков, сохраняя совместимость с другими клиентами и серверами. Они будут игнорировать неизвестные им заголовки, но при этом можно получить необходимую функциональность при решении специфической задачи.
* **Распространенность**При выборе протокола HTTP для решения конкретных задач немаловажным фактором является его распространенность. Как следствие, это обилие различной документации по протоколу на многих языках мира, включение удобных в использовании средств разработки в популярные IDE, поддержка протокола в качестве клиента многими программами и обширный выбор среди хостинговых компаний с серверами HTTP.

Минусы:

**Большой размер сообщений.**

Вследствие того, что протокол является текстовым, то и все передаваемые данные тоже в текстовом формате. Это порождает существенно больший размер по сравнению с двоичными данными. Из-за этого возрастает нагрузка на оборудование при формировании, обработке и передаче сообщений.

Для решения этой проблемы протокол уже содержит встроенные средства:

* кэширование на стороне клиента.
* компрессия передаваемого содержимого.

**Отсутствие «навигации»**

Хотя протокол разрабатывался как средство работы с ресурсами сервера, у него отсутствуют в явном виде средства навигации среди этих ресурсов.Клиент не может явным образом запросить список доступных файлов, как это доступно в протоколе FTP.

Предполагалось, что конечный пользователь уже знает URI необходимого ему документа, получив который, он будет производить навигацию благодаря гиперссылкам.  
Это вполне нормально и удобно для человека, но затруднительно для автоматической обработки и анализ ресурсов. Решение этой проблемы лежит полностью на плечах разработчиков приложений, использующих данный протокол.

### Вопрос 2 Структура

#### Вопрос 2.1 Структура URL

URI - Унифицированный (единообразный) идентификатор ресурса. 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

примеры:

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

Ресурс – конечная точка от куда берутся данные или производится бизнес логика.

*Длина URI*

*Протокол HTTP не устанавливает ограничений на длину URI. Серверы ДОЛЖНЫ иметь возможность обрабатывать URI любого ресурса, который они обслуживают, и ДОЛЖНЫ иметь возможность обрабатывать URI неограниченной длины, если они предоставляют формы на основе GET, которые могут генерировать такие URI. Сервер ДОЛЖЕН вернуть статус*

*414 (Request-URI Too Long), если URI длиннее, чем может обработать сервер.*

*Максимальная длина URL в адресной строке популярных браузеров составляет 2048 символов.*

#### Вопрос 2.2 Структура запроса

Структура протокола - Каждое HTTP-сообщение состоит из трёх частей, которые передаются в указанном порядке:

1. Стартовая строка - определяет тип сообщения

2. **Заголовки** - характеризует тело сообщения, параметры передачи и прочие сведения **(**Строки, содержащие разделенную двоеточием пару параметр-значение**)**

3. Тело сообщения - непосредственно данные сообщения. Отделяются от заголовков пустой строкой

Заголовки и тело могут отсутствовать, стартовая строка является обязательным элементом.

Стартовая строка запроса состоит из:

* Метод
* URI
* Версия

Стартовая строка ответа состоит из:

* Версия
* Код состояния (Status Code) - 3 цифры
* Пояснения (Reason Phrase) - короткое текстовое описание кода состояния

Стартовая строка запроса - Метод URI HTTP/Версия

пример - GET /wiki/HTTP HTTP/1.1

Стартовая строка ответа - HTTP/Версия КодСостояния Пояснение

пример - HTTP/1.1 200 OK

### Вопрос 3 Методы

**Методы** - Последовательность из символов, кроме управляющих и разделителей, указывающая на основную операцию над ресурсом. Обычно короткое английское слово, записанное заглавными буквами. (чувствительно к регистру).

Спецификация HTTP 1.1 не ограничивает количество разных методов, которые могут быть использованы.

Сервер может использовать любые методы. Не существует обязательных методов для клиента или сервера.

Однако в целях соответствия общим стандартам и сохранения совместимости с максимально широким спектром программного обеспечения как правило используются лишь некоторые, наиболее стандартные методы, смысл которых однозначно раскрыт в спецификации протокола.

**OPTIONS** - определение возможностей веб-сервера или параметров соединения для конкретного ресурса. В ответе сервера заголовок Allow со списком поддерживаемых методов и может включаться информация о поддерживаемых расширениях.

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.

**GET** - для запроса содержимого указанного ресурса. С помощью 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 «?» между параметрами “&”:

Согласно стандарту HTTP, запросы типа GET считаются идемпотентными.

Идемпотентный метод - многократное повторение одних и тех же запросов возвращает одинаковые результаты.

На get влияет заголовок Cache-Control.

**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.

* проверка наличия ресурса (валидация URL).
* узнать, не изменился ли он с момента последнего обращения.
* если ресурс является файлом, можно узнать его размер и затем определиться со стратегией его загрузки.

**POST** - для передачи пользовательских данных заданному ресурсу. Передаваемые данные включаются в тело запроса. POST обычно используют для загрузки файлов на сервер.

Не идемпотентный - 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.

**PUT** - Применяется для загрузки содержимого запроса на указанный в запросе URI. Если по заданному URI не существует ресурс, то сервер создаёт его и возвращает статус 201 (Created). Если же был изменён ресурс, 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 не допустим при текущих условиях, то необходимо вернуть код ошибки 501 (Not Implemented).

**PATCH** Аналогично PUT, но применяется только к фрагменту ресурса.

**DELETE** Удаляет указанный ресурс. (Идемпотентный метод)

**TRACE** Возвращает полученный запрос так, что клиент может увидеть, какую информацию промежуточные серверы добавляют или изменяют в запросе. Сервер возвращает в теле ответа все, что было отправлено в запросе. УТОЧНИТЬ

**CONNECT** Преобразует соединение запроса в прозрачный TCP/IP-туннель, обычно чтобы содействовать установлению защищённого SSL-соединения через нешифрованный прокси.

***LINK*** - устанавливает связь указанного ресурса с другими.

**UNLINK** - убирает связь указанного ресурса с другими.

### Вопрос 4 различие методов POST и PUT

POST предполагает, что по указанному URI будет производиться обработка передаваемого клиентом содержимого.

PUT, клиент предполагает, что ресурс существует и загружаемое содержимое соответствует находящемуся по данному URI ресурсу.

Сообщения ответов сервера на метод PUT не кэшируются.

Ресурс – конечная точка от куда берутся данные или производится бизнес логика.

### Вопрос 5 Коды состояния

Коды состояния HTTP Код состояния целое число из трёх цифр. Первая цифра указывает на класс состояния. За кодом обычно отделённая пробелом поясняющая фраза на английском.

1xx Информационный

2xx Успех

3xx Перенаправление

4xx Ошибка клиента

5xx Ошибка сервера

Знать по 3 с каждого класса

### Вопрос 6 Заголовки HTTP

Заголовки HTTP - это строки в HTTP-сообщении, содержащие разделенную двоеточием пару параметр-значение.

Каждый заголовок пишется с новой строки, иными словами, заголовки разделяются символом переноса строки(CRLF).

После всех заголовков должна быть хотя бы одной пустая строка, чтобы отделить их от тела сообщения.

Тело сообщения Тело HTTP-сообщения (message-body), если оно присутствует, используется для передачи тела объекта, связанного с запросом или ответом. Тело сообщения отличается от тела объекта (entity-body) только в том случае, когда применяется кодирование передачи, что указывается полем заголовка Transfer-Encoding.

Правила, устанавливающие допустимость тела сообщения в сообщении, отличны для запросов и ответов.

Все заголовки разделяются на четыре основных группы:

1. **General Headers** («Основные заголовки»)  
   Являются основными для запросов клиента и ответов сервера. Большая часть из них являются обязательными.
2. **Request Headers** («Заголовки запроса»)  
   Используются только в запросах клиента.  
   Например: Host, Referer, User-Agent
3. **Response Headers** («Заголовки ответа»)  
   Только для ответов от сервера. Например Location, Server, Allow
4. **Entity Headers** («Заголовки сущности») Сопровождают каждую сущность сообщения.  
   Content-Length, Content-Language, Content-Encoding

*Заголовки сущности - сопровождают каждую сущность сообщения. В отдельный класс заголовки сущности выделены для того, чтобы не путать их с заголовками запроса или заголовками ответа при передаче множественного содержимого (multipart/\*). Заголовки запроса и ответа как и основные заголовки описывают всё сообщение в целом и размещаются только в начальном блоке заголовков, в то время как заголовки сущности характеризуют содержимое каждой части в отдельности располагаясь непосредственно перед её телом.*

Заголовки, такие как Content-Length, Content-Language, Content-Encoding являются заголовками сущности.

General Headers «Основные заголовки»

Connection - определяет, остаётся ли сетевое соединение активным после завершения текущей транзакции (запроса). Если в запросе отправлено значение **keep-alive**, то соединение остаётся и не завершается, позволяя выполнять последующие запросы на тот же сервер.

[запрещен в HTTP/2](https://tools.ietf.org/html/rfc7540#section-8.1.2.2).

Cache-Control - Основные директивы для управления кэшированием.

Для КЛИЕНТА может существовать установка, в которой реализована цепочка кэша. В этом случае, если запрос поступает в первый кэш, где он не выполняется, он может перейти в дополнительный цепной кэш.

Таким образом, чтобы всегда получать ответ от сервера, мы включаем контроль кэша в заголовки запроса. Это гарантирует, что ответ всегда от сервера.

**public**

Указывает, что ответ может быть закеширован в любом кеше.

**private**

Указывает, что ответ предназначен для одного пользователя и не должен помещаться в разделяемый кеш. Частный кеш может хранить ресурс.

**no-cache**

Указывает на необходимость отправить запрос на сервер для валидации ресурса перед использованием закешированных данных.

**only-if-cached**

Указывает на необходимость использования только закешированных данных. Запрос на сервер не должен посылаться.

Управление временем жизни

**max-age=<seconds>**

Задаёт максимальное время в течение которого ресурс будет считаться актуальным. В отличие от Expires, данная инструкция является относительной по отношению ко времени запроса.

**s-maxage=<seconds>**

Переопределяет max-age или заголовок Expires, но применяется только для разделяемых кешей (например, прокси) и игнорируется частными кешами.

**max-stale[=<seconds>]**

Указывает, что клиент хочет получить ответ, для которого было превышено время устаревания. Дополнительно может быть указано значение в секундах, указывающее, что ответ не должен быть просрочен более чем на указанное значение.

**min-fresh=<seconds>**

Указывает, что клиент хочет получить ответ, который будет актуален как минимум указанное количество секунд.

**stale-while-revalidate=<seconds>**

Указывает, что клиент хочет получить просроченный ответ, одновременно осуществляя фоновую проверку наличия свежих данных. Значение в секундах обозначает, какое время клиент желает получать просроченный ответ.

**stale-if-error=<seconds>**

...

Управление ревалидацией и перезагрузкой

**must-revalidate**

Кеш должен проверить статус устаревших ресурсов перед их использованием. Просроченные ресурсы не должны быть использованы.

**proxy-revalidate**

То же самое, что must-revalidate, но применимо только к разделяемым кешам (например, прокси) и игнорируется частными кешами.

**Другие инструкции**

**no-store**

Кеш не должен хранить никакую информацию о запросе и ответе

**no-transform**

Никакие преобразования не должны применяться к ресурсу. Заголовки Content-Encoding, Content-Range, Content-Type не должны изменяться прокси. Непрозрачный прокси может, например, конвертировать изображения из одного формата в другой для сохранения дискового пространства или уменьшения трафика. Инструкция no-transform запрещает это.

Date - дату и время, в которое сообщение было создано

Transfer-Encoding - Список способов кодирования, которые были применены к сообщению для передачи.

Chunked Данные отправляются сериями. Заголовок Content-Length в этом случае опускается, и в начале каждого фрагмента вам нужно добавить длину текущего фрагмента в шестнадцатеричном формате, за которым следует '\ r \ n', а затем сам фрагмент, за которым следует еще один '\ г \ п '. Завершающий фрагмент - это обычный фрагмент, за исключением того, что его длина равна нулю. За ним следует трейлер, который состоит из (возможно, пустой) последовательности полей заголовка.

Compress Формат, использующий алгоритм Лемпеля-Зива-Велча (LZW). Имя значения было взято из программы сжатия UNIX, в которой реализован этот алгоритм. Подобно программе сжатия, которая исчезла из большинства дистрибутивов UNIX, это кодирование содержимого сегодня почти не используется ни в одном браузере, отчасти из-за проблемы с патентом (срок действия которого истек в 2003 году).

Deflate - Использование структуры zlib (определенной в RFC 1950) с алгоритмом сжатия deflate (определенным в RFC 1951).

Gzip - Формат, использующий кодирование Лемпеля-Зива (LZ77) с 32-битным CRC. Изначально это формат программы gzip для UNIX. Стандарт HTTP / 1.1 также рекомендует, чтобы сервер поддерживающих это кодирование содержимого, распознавали x-gzip как псевдоним для целей совместимости.

#### Вопрос 6.1 Ответить несколько заголовков

Общие поля HTTP заголовка

|  |  |  |
| --- | --- | --- |
| **Поля HTTP заголовка** | **Описание поля HTTP заголовка** | **Пример** |
| Cache-Control | Общее поле HTTP заголовка Cache-Control определяет директивы для управления кэшем, которым должны следовать все кэширующие механизмы | Cache-Control: no-cache  **Cache-Control: no-store Cache-Control: max-age=3600 Cache-Control: max-stale=0 Cache-Control: min-fresh=0 Cache-Control: no-transform Cache-Control: only-if-cached Cache-Control: cache-extension** |
| Connection | Общее поле HTTP заголовка Connection позволяет управлять HTTP соединением | Connection: close |
| Date | Общее поле HTTP заголовка хранит дату и время создания HTTP сообщения | Date: Tue, 15 Nov 1994 08:12:31 GMT |

HTTP заголовка запроса пользователя

|  |  |  |
| --- | --- | --- |
| **Поля HTTP заголовка** | **Описание поля HTTP заголовка** | **Пример** |
| Accept | Поле заголовка запроса Accept используется, чтобы определить тип информации, который должен содержаться в ответе [HTTP сервера](https://zametkinapolyah.ru/servera-i-protokoly/http-server-ili-veb-server-naznachenie-funkcii-i-rol-servera-v-http.html) | Accept: text/plain; q=0.5, text/html,  text/x-dvi; q=0.8, text/x-c |
| Host | Поле заголовка запроса Hostиспользуется для указания доменного имени и порта запрашиваемого ресурса. | Host: zametkinapolyah.ru |
| Authorization | Поле заголовка запроса Authorization используется для отправки данных авторизации на сервер | Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ== |
| Range | Поле заголовка запроса Range указывает [байтовый диапазон](https://zametkinapolyah.ru/servera-i-protokoly/edinicy-izmereniya-diapazonov-v-http.html) фрагмента ресурса, который требуется клиенту, чтобы не тянуть весь ресурс целиком | Range: bytes=50000-99999,250000-399999,500000- |

Поля HTTP заголовка ответа сервера

|  |  |  |
| --- | --- | --- |
| **Поля HTTP заголовка** | **Описание поля HTTP заголовка** | **Пример** |
| Age | Поле заголовка ответа Age хранит в себе количество секунд с момента последней модификации ресурса |  |
| Location | Поле заголовка ответа Location указывает URI, на котором хранится запрошенный ресурс | Location :absoluteURI |

Заголовки сущности

|  |  |  |  |
| --- | --- | --- | --- |
| **Поля HTTP заголовка** | **Описание поля HTTP заголовка** | **Пример** | **Тип HTTP сообщения** |
| Content-Location | Поле заголовка Content-Location используется для идентификации исходного местоположения объекта на сервере. | «Content-Location» «:»absoluteURI | relativeURI ) | Запрос и ответ |
| Content-Language | Поле заголовка Content-Language указывает клиенту на каком языке  информация, находящаяся в теле объекта. | Content-Language: mi, en | Запрос и ответ |
| Content-Type | Поле заголовка Content-Type используется для указания медиа типа данных в теле сообщения. | Content-Type: text/html;charset=utf-8 | Запрос и ответ |

### Вопрос 7 Cookie

Cookie - Небольшой фрагмент данных, отправленный веб-сервером и хранимый на компьютере пользователя. Веб-клиент (обычно веб-браузер) всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в составе HTTP-запроса. Применяется для сохранения данных на стороне пользователя, на практике обычно используется для:

* аутентификации пользователя;
* хранения персональных предпочтений и настроек пользователя;
* отслеживания состояния сеанса доступа пользователя;
* ведения статистики о пользователях.

#### Вопрос 7.1 Сессия

Написать определение

### Вопрос 8 Base64

Base64 — стандарт кодирования двоичных данных при помощи только 64 символов ASCII. Алфавит кодирования содержит текстово-цифровые латинские символы A-Z, a-z и 0-9 (62 знака) и 2 дополнительных символа, зависящих от системы реализации. (символы «+» и «/», с символом «=» в качестве специального кода суффикса).

### Вопрос 9 CORS (Cross-origin)

Cross-origin resource sharing (CORS; с англ. — «совместное использование ресурсов между разными источниками») — технология современных браузеров, которая позволяет предоставить веб-страницам доступ к ресурсам другого домена. CORS дает контролировать доступ к тегам на веб странице по сети. Механизм классифицируется на три разные категории доступа тегов:

1. Запись из разных источников
2. Вставка из разных источников
3. Считывание из разных источников

Запись из разных источников — это ссылки, переадресации и отправка форм. С активным CORS в браузере все эти операции разрешены. Существует также под названием **предварительный запрос**, которая настраивает запись из разных источников. Таким образом, если некоторые записи могут быть разрешены, это не означают, что они будут выполнены.

Вставки из разных источников — это теги, загружаемые через **<script>**, **<link>**, **<img>**, **<video>**, **<audio>**, **<object>**, **<embed>**, **<iframe>** и т.п. Все они разрешены по умолчанию. **<iframe>** выделяется на их фоне, так как он используется для загрузки другой страницы внутри фрейма. Его обрамление в зависимости от источника может регулироваться посредством использования заголовка  X-Frame-options.

Что касается **<img>** и других вставных тегов, то они устроены так, что сами инициируют запросы из разных источников cross-origin запроса. Именно поэтому в CORS существует различие между вставкой из разных источников и считыванием из разных источников.   
  
Считывание из разных источников — это теги, загружаемые через вызовы AJAX/ fetch. Все они по умолчанию заблокированы вашим браузером. Существует обходной путь для вставки таких тегов на странице, но такие трюки регулируются другой политикой, которая соблюдается в современных браузерах.

GET /cors.txt HTTP/1.1

Host: www.b.com

Origin: [www.a.com](http://www.a.com)

**Origin - домена сайта, с которого происходит запрос**

[**Синтаксис**](https://developer.mozilla.org/ru/docs/Web/HTTP/Headers/Origin#%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81)

Origin: ""

Origin: <протокол> "://" <имя\_хоста> [ ":" <порт> ]

Заголовок ответа Access-Control-Allow-Origin показывает, может ли ответ сервера быть доступен коду, отправляющему запрос с данного источника origin.

Сервер отправляет следующее в своем заголовке ответа, чтобы установить поле cookie.

Set-Cookie:имя =значение [ ; expires=дата ] [ ; domain=домен ] [ ; path=путь ] [ ; secure]

Однако не все эти данные передаются обратно на сервер клиентом при выполнении следующего HTTP-запроса.

Вы также можете установить HttpOnly

Если есть набор файлов cookie, браузер отправляет следующее в заголовке запроса.

Cookie:имя =значение

**Сессии** — более эфемерное понятие, которое не привязано к какой-то конкретной реальной технологии. Это просто некая методика, которая позволяет отличить одного клиента от другого, и, как правило, где-то хранить связанные с каждым клиентом данные.

Как правило, сессии реализуются используя cookies и идентификаторы сессий. Т.е. сервер со своей стороны создает уникальный идентификатор, например, «1a2b3c» (session\_id про который вы справшивали), а клиента просит его запомнить. Обычно — при помощи cookies, говоря что-то в духе Set-Cookie: PHPSESSID=1a2b3c (где «PHPSESSID» — **имя сессии**, обычно, оно только одно, вести параллельно несколько сессий нужно редко). Со своей стороны сервер где-то (зависит от реализации, иногда это файл, например, /tmp/1a2b3c, иногда запись в БД, иногда еще что-то) хранит различные данные, которые ему приказано связывать с этой сессией. Например, имя пользователя.

Разобраться c передачей бинарной информации в HTTP, разобраться со статус-кодами 100 группы, разобраться с Cookie.

Рассмотреть ограничения и проблемы при отправки данных через GET запросы.

## Тема 2 Основы HTML

### Вопрос 1 Основные понятия

**HTML (HyperText Markup Language/** **Язык гипертекстовой разметки) -** стандартизированный язык разметки документов во Всемирной паутине. Интерпретируется браузерами.

### Вопрос 2 Структура HTML-документа

Тег <!DOCTYPE> - предназначен для указания типа текущего документа DTD (document type definition, описание типа документа)

Синтаксис:

<!DOCTYPE *[Элемент* *верхнего* *уровня]* *[Публичность]* "[Регистрация]//[Организация]//[Тип] [Имя]//[Язык]" "[URL]">

Тег **<html>** является контейнером для всего содержимого страницы. Другими словами, весь код страницы находиться между открывающимся **<html>** и закрывающимся **</html>**.

В тег **<head>** заключается служебная информация для браузера и поисковых систем. Эта информация никак не отображается браузером на экране монитора. Исключение составляет тег **<title>** **</title>**. И favicon <link rel="shortcut icon" href="/images/favicon.png" type="image/png"> (**Favicon** (сокр. от англ. FAVorites ICON — «значок для избранного», от названия папки с закладками) — значок веб-сайта или веб-страницы.)

Тег **<body>** - это основное тело нашего документа. Все, что находится между тегами **<body>** и **</body>** будет выводиться браузером на экран монитора.

**Мета-теги** — часть программного HTML-кода, заключенная между тегом <head>... </head> (заголовок страницы). Эти атрибуты не являются обязательными: они не видны пользователю, но сообщают определенную информацию поисковой системе (описание, заголовок, ключевые слова и т. д.)

**Назначение метатега viewport**

Метатег viewport был разработан компанией Apple для того, чтобы указывать браузерам на то, в каком масштабе необходимо отображать пользователю видимую область веб-страницы.

**Настройка meta viewport для адаптивных веб-страниц**

Включение поддержки тега meta viewport для адаптивных сайтов осуществляется посредством добавления всего одной строчки в раздел head веб-страницы:

<meta *name*="viewport" *content*="width=device-width, initial-scale=1">

Тег <script> предназначен для описания скриптов, может содержать ссылку на программу или ее текст на определенном языке. Скрипты могут располагаться во внешнем файле и связываться с любым HTML-документом. Такой подход позволяет использовать одни и те же общие функции на многих веб-страницах и ускоряет их загрузку, т.к. внешний файл кэшируется при первой загрузке, и скрипт вызывается быстрее при последующих вызовах.

<script> может располагаться в заголовке или теле HTML-документа в неограниченном количестве. В большинстве случаев местоположение скрипта никак не сказывается на работу программы. Однако скрипты, которые должны выполняться в первую очередь, обычно помещают в заголовок документа.

**Синтаксис**

<script *type*="тип">

  ...

</script>

<script *type*="тип" *src*="URL"></script>

**Атрибуты**

**async** - Загружает скрипт асинхронно.

**defer** - Откладывает выполнение скрипта до тех пор, пока вся страница не будет загружена полностью.

**language** - Устанавливает язык программирования на котором написан скрипт.

**src** - Адрес скрипта из внешнего файла для импорта в текущий документ.

**type** - Определяет тип содержимого тега <script>.

**Спец символы**

&lt; <

&gt; >

&nbsp; пробел

Контейнер <noscript> показывает свое содержимое, если браузер не поддерживает работу со скриптами или их поддержка отключена пользователем.

### Вопрос 3 Тэги

**парный** тег - контейнеры, состоят из двух частей— открывающий и закрывающий тег.

**непарный** - состоят из одного тега (нет закрывающего тега).

<br>, <img>, <hr>

Внутрь этого тега нельзя положить контент

|  |  |
| --- | --- |
| *Декларация (HTML 5)* | <!DOCTYPE html> |
| *</a>* | Гиперссылка, Якорь <a href="filename" target="\_self">название ссылки</a> |
| *</H1> …. </H6>* | Заголовки 1-6 уровня (от крупных к мелким) |
| *<P>* | Новый абзац |
| *<BR>* | Новая строка |
| *<HR>* | Горизонтальная линия |
| *</BLOCKQUOTE>* | Цитата |
| *</PRE>* | Режим превью (все теги кроме PRE игнорируются) |
| *</DIV>* | Блок(обычно используется для применения стилей CSS) |
| *</SPAN>* | Строка (обычно используется для применения стилей CSS) |
| *</EM>* | Логическое ударение (обычно отображается курсивным шрифтом) |
| *</STRONG>* | Усиленное логическое ударение (обычно отображается жирным шрифтом) |
| *</I>* | Выделение текста курсивом |
| *</B>* | Выделение текста жирным шрифтом |
| *</U>* | Подчёркивание текста |
| *</S> или </STRIKE>* | Зачёркивание текста |
| *</BIG>* | Увеличение шрифта |
| *</SMALL>* | Уменьшение шрифта |
| *</BLINK>* | Мигающий текст |
| *</MARQUEE>* | Сдвигающийся по экрану текст |
| *</SUB>* | Подстрочный текст |
| *</SUP>* | Надстрочный текст |
| *<FONT параметры> … </FONT>* | Задание параметров шрифта COLOR=цвет FACE=шрифт SIZE=размер SIZE=+-изменение <FONT SIZE="+2">больших</FONT> <FONT COLOR="green">зелёных</FONT> |
| *</UL>* | Ненумерованный список <UL> <LI> первый элемент </LI> <LI> второй элемент </LI> <LI> третий элемент </LI> </UL> |
| *</OL>* | Нумерованный список <ol type="i"> <li> Первое </li> <li> Второе </li> <li> И т.д. </li> </ol> |
| *</DL>* | Список определений <DL> <DT> Кошка </DT> <DD> мяукающее домашнее животное </DD> <DT> Кот </DT> <DD> муж кошки </DD> <DT> Крокодил </DT> <DD> большой африканский зверь с острыми зубами </DD> </DL> |
| *</EMBED>* | Вставка различных объектов: не-HTML документов и media-файлов |
| *</APPLET>* | Вставка Java-апплетов |
| *</SCRIPT>* | Вставка скриптов |
| *<IMG>* | Вставка изображения. Этот тег не закрывается SRC - имя или URL ALT - альтернативное имя TITTLE - краткое описание WIDTH, HEIGHT - размеры ALIGN - обтекание текстом VSPACE, HSPACE - пространство вокруг <IMG SRC=url ALT="текст" TITLE="текст" WIDTH="размер (пикс, %)" HEIGHT="размер (пикс, %)"> |
| *</MAP>* | Создание карты изображений - графического объект языка разметки HTML, связанного с изображением и содержащем специальные области (активные зоны), при нажатии на которые происходит переход по определённому URL (при помощи javascript можно установить другие действия). Использование карт изображений позволяет хранить несколько ссылок в одном изображении. |
| *</TABLE>* | Создание таблицы BORDER - толщина линий CELLSPACING - расстояние между клетками CELLPADDING - отступ от рамки до содержимого ячейки CAPTION - заголовок таблицы TR - строка таблицы TH - заголовок столбца таблицы TD - ячейка таблицы height - высота таблицы width - ширина таблицы <TABLE BORDER="1" CELLSPACING="0"> <CAPTION> Улов крокодилов в Мумбе-Юмбе </CAPTION> <TH> Год </TH>  <TH> Улов </TH> <TR> <TD> 1997 </TD>  <TD> 214 </TD> </TR> <TR> <TD> 1998 </TD>  <TD> 216 </TD> </TR> <TR> <TD> 1999 </TD>  <TD> 207 </TD> </TR> </TABLE> |

### **Вопрос 4 Атрибуты**

Теги могут иметь атрибуты, определяющие какие-либо свойства. Атрибуты указываются в открывающем теге.

Все атрибуты состоят из двух частей — это имя = “значение”. Значение помещается в кавычки.

### Вопрос 5 Работа с текстом

**Тег <p> - параграф.**

<p align="center">Привет мир</p>

По левому краю:

<p align="left">Привет мир</p>

По правому краю:

<p align="right">Привет мир</p>

Или же обоим краям документа:

<p align="justify">Привет мир</p>

Тег **<p>** подразумевает в себе перенос строки

Тег <div> **- параграф без переноса строки.**

данный тег не будет переносить текст на следующую строку и в него можно вставлять тег **<p>**

<div>

    <p align="left">Пишем слева</p>

    <p align="right">Пишем справа</p>

</div>

тег <center> **- выравнивания текста по центру**

**В наборе тегов html языка имеется шесть типов заголовков:**

<h1> Привет мир!!! </h1>

<h2> Привет мир!!! </h2>

<h3> Привет мир!!! </h3>

<h4> Привет мир!!! </h4>

<h5> Привет мир!!! </h5>

<h6> Привет мир!!! </h6>

**тег <font>  - "шрифт".**

Тег **<font>** помимо прочих атрибутов, о которых ещё пойдет речь, имеет атрибут **size** - размер.

Пишется и выглядит это так:

<font *size*="+4">Привет мир!!!</font>

 у него есть еще один атрибут - **color.**

<font *color*="#ff0000">Привет мир!!!</font>

<font *color*="red">Привет мир!!!</font>

Тег **<body></body>** "тело" - имеет атрибут **text**

<body *text*="#ff8c40 ">

Теперь весь текст у нас стал оранжевым кроме заголовка

атрибут тега **<body> bgcolor**и его значение задает цвет фона страницы

**<body bgcolor="#40caff">** - залили всё голубым

**Стиль текста**

|  |  |
| --- | --- |
| **<b> </b>** | - **Полужирный текст** |
| **<i> </i>** | -*Наклонный текст* |
| **<u> </u>** | - Подчеркнутый текст |
| **<strike> </strike>** | - ~~Перечеркнутый~~ |
| **<s> </s>** | - ~~Перечеркнутый~~ (второй вариант, от первого ничем не отличается) |
| **<tt> </tt>** | - моноширинный шрифт |
| **<small> </small>** | - Малый |
| **<big> </big>** | - Большой |
| **<sup> </sup>** | - Верхний индекс |
| **<sub> </sub>** | - Нижний индекс |

<big><b><u>Научная статья.</u></b></big>

**Шрифт**

используем всё тот же тег **<font>** и его атрибут **face**

**<font face="arial">**Эта строчка будет написана с помощью шрифта Arial**</font>**

тег **<pre>**, текст заключённый в данный тег выводится браузерами на экран в том виде в котором он был набран, т.е. со всеми пробелами и переносами строк

### Вопрос 6 Form

Тег <form> устанавливает форму на веб-странице. Форма предназначена для обмена данными между пользователем и сервером.

method - Метод протокола HTTP.

action - Указывает обработчик, к которому обращаются данные формы при их отправке на сервер.

Если атрибут action отсутствует, текущая страница перезагружается, возвращая все элементы формы к их значениям по умолчанию.

Enctype - Способ кодирования данных формы.

application/x-www-form-urlencoded

Вместо пробелов ставится +, символы вроде русских букв кодируются их шестнадцатеричными значениями (например, %D0%90%D0%BD%D1%8F вместо Аня).

multipart/form-data

Данные не кодируются. Это значение применяется при отправке файлов.

text/plain

Пробелы заменяются знаком +, буквы и другие символы не кодируются.

Тег <input> - разносторонний элемент формы

name - Имя поля, предназначено для того, чтобы обработчик формы мог его идентифицировать.

type - Сообщает браузеру, к какому типу относится элемент формы.

value- Значение элемента.

|  |  |
| --- | --- |
| button | Кнопка. |
| checkbox | Флажки. Позволяют выбрать более одного варианта из предложенных. |
| file | Поле для ввода имени файла, который пересылается на сервер. |
| hidden | Скрытое поле. Оно никак не отображается на веб-странице. |
| image | Поле с изображением. При нажатии на рисунок данные формы отправляются на сервер. |
| password | Обычное текстовое поле, но отличается от него тем, что все символы показываются звездочками. Предназначено для того, чтобы никто не подглядел вводимый пароль. |
| radio | Переключатели. Используются, когда следует выбрать один вариант из нескольких предложенных. |
| reset | Кнопка для возвращения данных формы в первоначальное значение. |
| submit | Кнопка для отправки данных формы на сервер. |
| text | Текстовое поле. Предназначено для ввода символов с помощью клавиатуры. |

Группа радиокнопок определяется путём присвоения каждой радиокнопке в данной группе одного и того же значения атрибута (name). Выбор любой радиокнопки в этой группе автоматически отменяет выбор другой радиокнопки в той же группе.

Поле <textarea> представляет собой элемент формы для создания области, в которую можно вводить несколько строк текста. В отличие от тега <input> в текстовом поле допустимо делать переносы строк, они сохраняются при отправке данных на сервер.

Между тегами <textarea> и </textarea> можно поместить любой текст, который будет отображаться внутри поля.

cols - Ширина поля в символах.

rows - Высота поля в строках текста.

Тег <select> позволяет создать элемент интерфейса в виде раскрывающегося списка.

name - Имя элемента для отправки на сервер или обращения через скрипты.

Тег <option> определяет отдельные пункты списка, создаваемого с помощью контейнера <select>.

value - Значение пункта списка, которое будет отправлено на сервер или прочитано с помощью скриптов.

Если не задан value то значение будет текст из option.

### Доп Вопрос 1 Обработка массива на сервере

в HTML:

<input *type*="hidden" *name*="cars" *value*="Volvo">

<input *type*="hidden" *name*="cars" *value*="Saab">

<input *type*="hidden" *name*="cars" *value*="Mercedes">

в get:

?cars=Volvo&cars=Saab&cars=Mercedes

в Java:

например, коллекция, со значениями: Volvo,Saab,Mercedes

по name инпута идет формирование запроса, с одинаковыми параметрами (в данном случае "cars=") идет формирование в коллекцию.

Связь label через input

**Тег** <label> **устанавливает связь между определенной меткой, в качестве которой обычно выступает текст, и элементом формы (<input>, <select>, <textarea>)**

for - Идентификатор элемента, с которым следует установить связь.

Пример

<input *type*="checkbox" *id*="check2"><label *for*="check2">Ipsum</label><Br>

<input *type*="checkbox" *id*="check3"><label *for*="check3">Dolor</label><Br>

### Доп Вопрос 2 работа с файлами

**тег <img>** для того чтобы теперь вставить фото в нашу страничку к ней нужно указать путь.

делается это так:

<img *src*="foto.jpg">

**<img src="foto.jpg" vspace="15">** - Атрибут **vspace** задаёт расстояние по вертикали от рисунка до текста, в данном случае мы задали расстояние в 15 пикселей  
**<img src="foto.jpg" hspace="25">** - Расстояние по горизонтали соответственно  
**<img src="foto.jpg" width="180">** - Ширина непосредственно самого изображения  
**<img src="foto.jpg" height="240">** - Высота изображения. Если атрибуты **width** и **height** не использовать, то ширина и высота изображения по умолчанию будет равна реальным её размерам, без каких либо искажений.  
**<img src="foto.jpg" border="5">**- Бордюр, рамка вокруг изображения и её толщина в пикселях.  
**<img src="foto.jpg" border="5" bordercolor="#008000 "> - bordercolor** - это цвет рамки.  
**<img src="foto.jpg" alt="Это моя фотка!!!">** -Атрибут **alt** - это описание изображения. Если навести курсор на наше фото и подержать его там несколько секунд, выскочит надпись -Это моя фотка!!!  
**<img src="foto.jpg" title="Это моя фотка!!!">** - альтернатива **alt** в данном случае.

А еще изображение можно сделать фоном страницы.. для этого используем атрибут **background**

### Доп Вопрос 3 Работа с таблицами

Тег <table> задаёт начало и конец таблицы, любая таблица, как известно, состоит из строк и столбцов, для этого есть ещё два тега:

**<tr>** - строка таблицы  
**<td>** - столбец таблицы

Вместе эти теги пишутся так:

<table>

    <tr>

        <td>ячейка</td>

    </tr>

</table>

атрибуты **colspan** и **rowspan**

**colspan** - определяет какое количество столбцов будет занимать данная ячейка

**rowspan** - количество рядов занимаемое ячейкой

Пример

<tr>

    <td>строка3 ячейка1</td>

    <td *colspan*="2">строка3 ячейка2</td>

</tr>

Значения атрибутов **width** и **height** в таблице могут указываться не только в пикселях, но и в процентах

<tr *height*="25%">

    <td *width*="15%"> строка1 ячейка1</td>

    <td *width*="25%">строка1 ячейка2</td>

    <td *width*="60%">строка1 ячейка3</td>

</tr>

атрибут **valign** - Вертикальное выравнивание.

**Cellspacing** - задаёт расстояние в пикселях между ячейками таблицы. Задав значение **cellspacing="0"** можно избавиться от "зазора" между ячейками.

Атрибут **cellpadding** (не путайте с **cellspacing**) - в пикселях задаёт поля ячеек (отступ от границ ячеек до текста)

### Доп Вопрос 4 работа с ссылками

тег **<a>** (от anchor- якорь), в него можно заключить текст или рисунок, которые станут ссылкой на те или иные документы. Атрибут тега **<a> href** задаёт имя и путь к документу на который указывает ссылка.

<a *href*="primer.html">Здесь мои фотки!!</a>

**link** - цвет ссылки.  
**alink** - цвет нажатой, активной ссылки.  
**vlink** - цвет посещенной ссылки.

Пишется так:

<body *link*="#008000" *alink*="#ff0000" *vlink*="#ffff00">

 при переходе по ссылке открыть документ в новом окне браузера, до этого если Вы обратили внимание он у нас открывался в текущем, что не всегда удобно. Решает эту проблему атрибут **target** (цель) и его значение **\_blank.**

Пишется так:

<a *href*="primer3.html" *target*="\_blank">открыть в новом окне</a>

Для того что бы сделать текст или рисунок ссылкой на e-mail - почтовый ящик его нужно заключить в тег **<a>**, но не простой, а с использованием **mailto**

Пишем так:

<a *href*="mailto:karlson@kruha.ru"> Напишите мне письмо.. </a>

**?subject=** - Тема пиcьма  
**&Body=** - Текст сообщения  
**&cc=** maluh@kruha.ru,maluh2@kruha.ru - Копии письма  
**&bcc=** freken\_bok@kruha.ru,freken\_bok2@kruha.ru - Скрытые копии письма

Адреса ящиков для копий и скрытых копий письма пишутся через запятую.

**Закладки или якоря** - это особый вид ссылок. Данные ссылки, как правило, не ведут к какому либо документу, а предназначены для навигации внутри страницы.

<a *href*="#glava1"> Глава1</a>

<a *href*="#glava2"> Глава3</a>

<a *href*="#glava3"> Глава3</a>

## Тема 3 Продвинутый HTML

## Тема 4 Основы CSS

### Вопрос 1 Что такое стили.

**CSS - Cascading Style Sheets (каскадные таблицы стилей)** - формальный язык описания внешнего вида документа, написанного с использованием языка разметки. Преимущественно используется как средство описания, оформления внешнего вида веб-страниц, написанных с помощью HTML, но может также применяться к любым XML-документам, например, к SVG или XUL.

**Цель создания** - Основной целью разработки CSS являлось разделение описания логической структуры веб-страницы (которое производится с помощью HTML или других языков разметки) от описания внешнего вида этой веб-страницы (которое теперь производится с помощью формального языка CSS. CSS позволяет представлять один и тот же документ в различных стилях: экранное представление, печатное представление, чтение голосом (специальным голосовым браузером или программой чтения с экрана), или при выводе устройствами, использующими шрифт Брайля.

### Вопрос 2 Наиболее существенные плюсы использования стилей

**Преимущества CSS**

1. Несколько дизайнов страницы для разных устройств просмотра
2. Уменьшение времени загрузки страниц сайта за счет переноса правил представления данных в отдельный CSS-файл. В этом случае браузер загружает только структуру документа и данные, хранимые на странице, а представление этих данных загружается браузером только один раз и может быть закэшировано.
3. Простота последующего изменения дизайна. Не нужно править каждую страницу, а достаточно лишь изменить CSS-файл.
4. Дополнительные возможности оформления. Например сделать так, чтобы меню было всегда видно при прокрутке страницы."

**Недстатки CSS**

1. "Различное отображение вёрстки в различных браузерах (особенно устаревших), которые по-разному интерпретируют одни и те же данные CSS
2. Часто встречающаяся необходимость на практике исправлять не только один CSS-файл, но и теги HTML, которые сложным и ненаглядным способом связаны с селекторами CSS, что иногда сводит на нет простоту применения единых файлов стилей и значительно увеличивает время редактирования и тестирования."

### Вопрос 3 Способы описания/добавления стилей на страницу.

### Вопрос 4 Подключение css

Подключение CSS могут располагаться как в самом веб-документе, так и в отдельных файлах, имеющих формат CSS

могут быть подключены четырьмя различными способами"

1. способ link

Когда таблица стилей находится в отдельном файле, она может быть подключена к веб-документу посредством тега <link>, располагающегося в этом документе между тегами <head> и </head>. (Тег <link> будет иметь атрибут href, имеющий значением адрес этой таблицы стилей). Все правила этой таблицы действуют на протяжении всего документа

<link *rel*="" *stylesheet*"" *href*="" *style.css*"">"

Атрибут rel определяет отношения между текущим документом и файлом, на который делается ссылка. Это необходимо, чтобы браузер знал, как использовать подключаемый документ.

Значение stylesheet Определяет, что подключаемый файл хранит таблицу стилей (CSS).

1. способ style + import Когда таблица стилей находится в отдельном файле, она может быть подключена к веб-документу посредством директивы @import, располагающейся в этом документе между тегами <style> и </style> (которые, в свою очередь, располагаются в этом документе между тегами <head> и </head>) сразу после тега <style>, которая также указывает (в своих скобках, после слова url) на адрес этой таблицы стилей. Все правила этой таблицы действуют на протяжении всего документа

<style *media*="" *all*"">

    @*import* url(style.css);

</style>"

1. Внутренняя таблица стилей ко всему документу "Когда таблица стилей описана в самом документе, она может располагаться в нём между тегами <style> и </style> (которые, в свою очередь, располагаются в этом документе между тегами <head> и </head>). Все правила этой таблицы действуют на протяжении всего документа

<style>

    body {

        color: red;

    }

</style>"

1. Внтурення таблица стилей к тэгу Когда таблица стилей описана в самом документе, она может располагаться в нём в теле какого-то отдельного тега (посредством его атрибута style) этого документа. Все правила этой таблицы действуют только на содержимое этого тега

<body>

    <p *style*="" *font-size:* *20px;* *color:* *green;* *font-family:* *arial,* *helvetica,* *sans-serif*"">

        .....

    </p>

</body>"

### Вопрос 5 Группирование, наследование, каскадирование

**Группирование -** При создании стиля для сайта, когда одновременно используется множество селекторов, возможно появление повторяющихся стилевых правил. Чтобы не повторять дважды одни и те же элементы, их можно сгруппировать для удобства представления и сокращения кода

H1, H2, H3 {

  font-family: Arial, Helvetica, sans-serif;

}

H1 {

  font-size: 160*%*;

  color: #003;

}

H2 {

  font-size: 135*%*;

  color: #333;

}

H3 {

  font-size: 120*%*;

  color: #900;

}

**Принцип каскадирования** - применяется в случае, когда какому-то элементу HTML одновременно поставлено в соответствие более одного правила CSS, когда происходит конфликт значений этих правил. Чтобы разрешить такие конфликты, вводятся правила приоритета.

**Правила приоритета**

**Каскадирование** - Принцип каскадирования применяется в случае, когда какому-то элементу HTML одновременно поставлено в соответствие более одного правила CSS, то есть, когда происходит конфликт значений этих правил. Чтобы разрешить такие конфликты, вводятся правила приоритета.

Приоритет:

1 Стиль браузера

2 Пользовательские настройки браузера

3 Стили подключённые к хтмл страничке

1 Тег

2 Класс

3 Идентификатор

4 inline-стиль

! important вне приоритетов, всегда важнее всего

Основной целью разработки CSS являлось разделение описания логической структуры веб-страницы (которое производится с помощью HTML или других языков разметки) от описания внешнего вида этой веб-страницы (которое теперь производится с помощью формального языка CSS). Такое разделение может увеличить доступность документа, предоставить большую гибкость и возможность управления его представлением, а также уменьшить сложность и повторяемость в структурном содержимом.

**Наследование** - это перенос правил **CSS** от элементов-"родителей" к элементам-"детям" (находящимся внутри "родителей"). Простейший пример наследования - если прописать в правилах для тега body определённый цвет текста (например, green), то этот цвет будет у всего текста на странице

Под каскадом в данном случае понимается одновременное применение разных стилевых правил к элементам документа — с помощью подключения нескольких стилевых файлов, наследования свойств и других методов. Чтобы в подобной ситуации браузер понимал, какое в итоге правило применять к элементу, и не возникало конфликтов в поведении разных браузеров, введены некоторые приоритеты.

### Вопрос 6 Виды селекторов

|  |  |
| --- | --- |
| Виды селекторов |  |
| *Универсальный* | \* { margin: 0; применяется ко всем атрибутам padding: 0;  } |
| *Селектор тегов* | p { font-family: arial, helvetica, sans-serif; применяется к тегам p } |
| *Селектор классов* | .note { color: red;  background-color: yellow; применяется к классам note font-weight: bold;  } |
| *Селектор идентификаторов* | #paragraph1 { margin: 0; применяется к идентификаторам paragraph1 } |
| *Селектор атрибутов* | a[href="http://www.somesite.com"] { font-weight: bold; применяется ко всем ссылкам со значением в скобках } |
| *Селектор потомков (контекстный селектор)* | H1 EM { color: red; применится ко всем EM являющихся потомками H1 } |
| *Селектор дочерних элементов* | p.note > b { color: green; для всех элементов b являющихся дочерними к p.note } |
| *Селектор сестринских элементов* | h1 + p { font-size: 24pt; выбор p следующего сразу за h1 если у них общий родитель } |
| *Селектор псевдоклассов* | a:active { псевдокласс - определяет динамическое состояние элемента, active - применится при активации пользователем элемента color: blue; :link - применится к непосещенным ссылкам :focus - к элементу при получении им фокуса, :hover - курсор в пределах элемента } :visited, :first-child, :lang ит.п. |
| *Селектор псевдоэлементов* | p:first-letter { псевдоэлемент - позволяет задать стиль элементов, не определенных в дереве элементов и генерировать содержимое которого font-size: 32px; нет в исходном коде текста. :after - вставка контента после элемента, :before, :first-letter - стиль первого символа в тексте эл-та  } :first-line, |
| Классы и идентификаторы элементов | Класс или идентификатор может быть присвоен какому-нибудь элементу (тегу) HTML посредством атрибутов class или id этого элемента (тега) <div id="First"> <p class="Big"> |
| *Основные отличия* | Идентификатор предназначен для одного элемента, тогда как класс обычно присваивают сразу нескольким. Тем не менее, современные браузеры, как правило, корректно отображают множественные элементы с одинаковым идентификатором. Также отличие в том, что могут существовать множественные классы (когда класс элемента состоит из нескольких слов, разделённых пробелами). Для идентификаторов такое невозможно. Идентификаторы широко используются в JavaScript для нахождения уникального элемента в документе. |

### Доп Вопрос 1 Свойства текста.

**Выравнивание текста**.

Свойство **text-align**, которое выравнивает текстовое содержание относительно элемента родителя (например, блока **div**) или же окна браузера.

**text-align** (так же как и htmlловский атрибут **align**) имеет следующие значения:

**left** - Выровнять текст по левому краю элемента (по умолчанию).

**right** - Выровнять текст по правому краю.

**center** - Выровнять текст по центру.

**justify** - Выровнять текст по обоим краям.

Теперь для того чтобы выровнять текст того же параграфа по центру следует писать так:

**<p style="text-align: center">текст по центру </p>**

**Оформление текста.**

Свойство **text-decoration** позволяет декорировать текст, присвоив ему одно или несколько значений из ниже представленных вариантов оформления текста.

Возможные значения:

**blink** - Текст будет мигать.

**line-through** - Делает текст перечеркнутым.

**overline** - Надчёркивание текста.

**underline** - Подчеркивание текста.

**none** - Текст без оформления.

Пишется так:

<a *href*="index.html" *style*="text-decoration:none">Ссылка без подчёркивания</a>

**Отступ первой строки.**

Свойство **text-indent** - задаёт отступ первой строки в текстовом блоке с левой стороны, проще говоря делает "красную строку".

p {text-align: justify; text-indent: 20*px*}

**Трансформация текста**

Своийство **text-transform** трансформирует символы в указанном текстовом блоке, делая их заглавными или прописными по одному из правил в зависимости от присужденного значения данному свойству.

Значения:

**none** - Текст отображается без каких-либо изменений.(по умолчанию)

**capitalize** - Каждое слово в тексте отображается с заглавного символа.

**lowercase** - Все символы преобразуются в нижний регистр.

**uppercase** - Все символы преобразуются в верхний регистр.

<p *style*="text-transform: capitalize">текст</p>

**Вертикальное выравнивание**

Вертикальное выравнивание текста в строке устанавливает свойство **vertical-align**

Возможные значения свойства **vertical-align:**

**baseline** - Выравнивает базовую линию элемента по базовой линии родителя.

**bottom** - Выравнивает элемент по нижней части строки.

**middle** - Выравнивает средину элемента по базовой линии родителя и прибавляет половину высоты родительского элемента.

**sub** - Нижний индекс (размер шрифта не меняется).

**super** - Верхний индекс (размер шрифта не меняется).

**text-bottom** - Нижняя граница элемента выравнивается по нижнему краю строки.

**text-top** - Верхняя граница элемента выравнивается по верхнему краю строки.

**top** - Выравнивает элемент по верхней части строки.

Так же вертикальное выравнивание элемента относительно строки может выражаться в процентах, пикселях или любых других принятых в CSS единицах измерения, причем эти единицы могут принимать как положительные, так и отрицательные значения.

**Пробелы и перенос строки.**

Набранный текст, в каком либо текстовом редакторе браузерами по умолчанию выводится на экран в виде сплошного текста, где переносы строк расставляются автоматически, а так же убираются лишние (более одного) пробелы между символами.

Свойство **white-space** имитирует работу тега **<pre>**, определяя показывать или нет пробелы между символов, если таковых больше чем один, а так же разрешает или запрещает перенос строки.

Может иметь следующие значения:

**normal** - текст выводится как обычно (лишние пробелы убираются), переносы строк определяются автоматически. (по умолчанию)

**nowrap** - запрещает автоматический перенос строки.

**pre** - показывает текст в том виде в котором он был набран. пробелы и переносы строки не удаляются.

**Расстояние между словами.**

Свойство **word-spacing** задаёт расстояние между словами (группами символов не разделенными пробелом) в строке.

Значения:

**normal** - Нормальное расстояние. (по умолчанию)

**px** - Расстояние задаётся в пикселях или любых других единицах измерения принятых в CSS.

**Межсимвольное расстояние.**

А вот свойство **letter-spacing** определяет расстояние между символами в тексте и так же как и может **word-spacing** быть задано следующими значениями:

**normal** - Нормальное расстояние. (по умолчанию)

**px** - Расстояние задаётся в пикселях или любых других единицах измерения принятых в CSS.

**Интерлиньяж**

Интерлиньяж - это расстояние между строками текста.

Расстояние между строками текста можно задать используя свойство **line-height**, сделать это можно следующими способами:

**normal** - Норма (по умолчанию).

**%** - Проценты. за сто процентов берется высота шрифта

**0.5** - Множитель. Может быть использовано любое число больше ноля. Так, например, множитель 0.5 будет равняться половинному межстрочному расстоянию, а 2 - двойному.

**px** - Пиксели и любые другие единицы измерения, принятые в CSS.

### Доп Вопрос 2 Хаки

**Хаки**

**Кросс-браузерность** — свойство сайта отображаться и работать во всех популярных браузерах идентично.

**Условные комментарии работают только в IE под Windows**, для других браузеров они являются обычными комментариями, поэтому их можно безболезненно использовать. Синтаксис такой:

<[if условие]> <link *href*= "styles.css" *rel*= "stylesheet" *media*= "all" /> <![endif]>

<[if !условие]> <link *href*= "styles.css" *rel*= "stylesheet" *media*= "all" /> <![endif]>

**Хак** — специальная CSS конструкция, которую понимают одни браузеры и игнорируют другие.

**Хак** — исправление ошибки или добавление новой функции посредством использования другой недокументированной или некорректно реализованной особенности.

**Плюс хаков**

Это быстрое решение проблемы. Прописали дополнительных строчек и все отлично, верстаем дальше.

**Недостатки хаков**

Нет уверенности как поведет себя сайт с выходом новых версий браузеров. Например, написали код с хаком для Firefox, через месяц выходит новая версия Safari и оказывается она тоже воспринимает этот хак! Приходится переделывать.

**Многие хаки не проходят валидацию**

По возможности старайтесь избегать использования хаков. Для исправления ошибок IE используйте отдельный CSS-файл, котjрый подключается с помощью условных комментариев.

## Тема 5 Продвинутый CSS

## Тема 6 Основы JavaScript

## Тема 7 Продвинутый JavaScript

## Тема 8 JavaScript framework

# Базы данных

## Тема 1 основные понятия

### Вопрос 1 Основные понятия

**Реляционная база данных** – это набор данных с предопределенными связями между ними, основана на реляционной модели данных.

**Реляционная модель** **данных** – модель данных которая включает в себя:

* Структурный аспект – информация храниться в виде набора таблиц
* Обработки аспект - Реляционная модель данных поддерживает операторы манипулирования отношениями.
* Целостности аспект – отношения отвечают определенным условиям целостности:

Структурная целостность:

* + - Наличие первичного ключа
    - Отсутствие дубликатов строк
    - работу с однородными структурами

Ссылочная целостность:

* + Удаляем в родители удаляется в дочернем
  + Удаляем в родители в дочернем NULL

Семантическая целостность: данные соответствуют всем ограничениям что создал пользователь:

* + по типу данных
  + по диапазону
  + по уникальность

**Нерялиционные** **база данных** – это база данных в которой в отличии от большинства систем баз данных не используется табличная схема строк и столбцов, в этих базданных применяется модель хранение, оптимизированная под конкретные требования типа хранимых данных

### Вопрос 2 Типы данных

**Строковые типы данных**

VARCHAR – хранит символьные строки переменной длины до (255)

CHAR (5) – текст с фиксированной длинной, всегда использует одинаковый объем памяти для каждой записи, значения дополняются пробелами, необходимыми для сравнения

NCHAR/NVARCHAR –хранит строку в кодировке Unicode

BLOB – текст более (255) символов

**Числовые типы данных**

INT – числовые значение

BIT/ TINYINT/ SMALLINT/BIGINT

DEC (2.1) – число с фиксированной точностью, обладавший двумя параметрами первый сколько чисел, второй сколько чисел после запятой

SMALLMONEY/ MONEY – хранит дробные значения, предназначены для хранение денежных величин

FLOAT/REAL –хранит дробные числа

**Типы данных, представляющие дату и время**

DATE – дата

TIME - время

DATETIME - задает дату и время

DATETIME2/ SMALLDATETIME/ DATETIMEOFFSET – различные диапазоны

**Бинарные типы данных**

BINARY/VARBINARY – хранит бинарные данные в виде последовательности байтов

### Вопрос 3 Нормализация и денормализация

**Нормализация** - Процесс преобразования отношений базы данных к виду, отвечающему нормальным формам.

**Нормальная форма** - Требование, предъявляемое к структуре таблиц для устранения из базы избыточных функциональных зависимостей между атрибутами (полями таблиц).

**Денормализация** - Намеренное приведение структуры базы данных в состояние, не соответствующее критериям нормализации

**Первая нормализованная форма**

1НФ - неделимости значения в каждом поле (столбце) строки – атомарность значений. Все его атрибуты являются простыми, все используемые домены должны содержать только скалярные значения. Не должно быть повторений строк в таблице.

* Каждая запись должна содержать атомарные значения

**вторая нормализованная форма**

2НФ - находится в 1НФ и каждый не ключевой атрибут функционально полно зависит от ключа, т.е. функционально зависит от всего ключа в целом, но не находится в функциональной зависимости от какого-либо из входящих в него атрибутов.

* Находится в первой нормальной форме
* Отсутствует частично функциональная зависимость (каждый не ключевой атрибут  [зависит](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%B7%D0%B0%D0%B2%D0%B8%D1%81%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) от  [ключа](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%B5%D0%BD%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BB%D1%8E%D1%87))

#### (Пометка)

**Функциональная зависимость** - Если даны два атрибута X и Y некоторого отношения, то говорят, что Y функционально зависит от X, если в любой момент времени каждому значению X соответствует ровно одно значение Y. Функциональная зависимость обозначается X -> Y

**Избыточная функциональная зависимость** - Зависимость, заключающая в себе такую информацию, которая может быть получена на основе других зависимостей, имеющихся в базе данных.

**Третья нормализованная форма**

3НФ - "Отношение находится в 3НФ, когда находится во 2НФ и каждый не ключевой атрибут нетранзитивно зависит от потенциального ключа. Проще говоря, каждый неключевой атрибут «должен предоставлять информацию о ключе, полном ключе и ни о чём, кроме ключа.

* Находится во второй нормальной форме
* Отсутствует транзитивные зависимости (не-ключевой столбец не зависит от не-ключевого столбца)

#### (Пометка)

Транзитивная функциональная зависимость "Пусть X, Y, Z - три атрибута некоторого отношения. При этом X --> Y и Y --> Z, но обратное соответствие отсутствует, т.е. Z -/-> Y и Y -/-> X. Тогда Z транзитивно зависит от X. "

### Вопрос 4 Ключи

**Первичный ключ** - атрибут или подмножество атрибутов, отвечающий за уникальность строки.

* **Естественный** – атрибут или подмножество атрибутов, которые при этом содержат в себе какую-то информацию.
* **Суррогатный** – мы делаем его сами, например, типичный ID.

Ограничение для ключа **CONSTRAINT**- уникальный и NOT NULL

**Индекс** – объект базы данных который отвечает за быстрый поиск элементов, создаётся таблица, которая содержит индексы и ссылки на элементы.

**Индекс бывает в виде**

* хэш данных (для операции сравнивания)
* красно черное дерево автобалансировка (для поиска)

**Внешний ключ** -столбец таблицы, в котором хранятся значения первичного ключа другой таблицы

**Потенциальный ключ** - "Подмножество атрибутов отношения, удовлетворяющее требованиям уникальности и минимальности (несократимости).

**Уникальность** - нет и не может быть двух кортежей данного отношения, в которых значения этого подмножества атрибутов совпадают

**Минимальность** - в составе потенциального ключа отсутствует меньшее подмножество атрибутов, удовлетворяющее условию уникальности. Иными словами, если из потенциального ключа убрать любой атрибут, он утратит свойство уникальности."

**Пример**

CREATE TABLE group (

Id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

id\_user INT NOT NULL

CONSTRAINT user\_id\_fk //ограничение

FOREIGN KEY(id\_user)

REFERENCES user (id)

);

Создаем новую таблицу group, создаем столбец id\_user который будет являться внешним ключом, с помощью команды RAFERENCES указывает из какой таблицы был взят внешний ключ и как он там назывался

Ограничению присваивается имя, по которому можно определить, из какой таблицы взят ключ (user), как он называется (id) и что ключ является внешним (fk

### Вопрос 5 транзакции

**Целостное состояние БД -** "Любая база данных годна к использованию только тогда, когда ее состояние соответствует состоянию предметной области. Такие состояния называют целостными. Очевидно, что при изменении данных БД должна переходить от одного целостного состояния к другому. Однако, в процессе обновления данных возможны ситуации, когда состояние целостности нарушается.

Пример - перевод денег с одного счета на другой производится за 2 операции, и если произойдет сбой между ними - целостное состояние БД будет утеряно"

**COMMIT и ROLLBACK**

COMMIT означает фиксацию всех изменений в транзакции. ROLLBACK означает отмену (откат) изменений, произошедших в транзакции.

(ПОМЕТКА)

В MySQL не существует механизма вложенных транзакций. Одно соединение с БД — одна транзакция. Новая транзакция в пределах одного соединения может начаться только после завершения предыдущей.

По умолчанию в MySQL установлен уровень изоляции № 2 (Repeatable Read).

**Транзакции** – последовательность операций, которые должны отвечать следующим требованиям ACID:

**ACID** описывает требования к транзакционной системе

* **Атомарность** - транзакция должна быть либо вся выполнена, либо не выполнена. Другими словами, это гарантирует, что все операции внутри рабочего блока успешно завершены; в противном случае транзакция прерывается в точке сбоя, а предыдущие операции возвращаются в прежнее состояние.
* **Согласованность** – данные до транзакции должны быть согласованны с данными после транзакции.Т.е. каждая успешная транзакция по определению фиксирует только допустимые результаты. Другими словами, условие, при котором данные, записываемые в базу данных в рамках транзакции, должны соответствовать всем правилам и ограничениям, включая ограничения целостности, каскады и триггеры.
* **Изолированность** – на результат транзакции не должны влиять другие транзакции. Другими словами, позволяет транзакциям работать независимо друг от друга и быть прозрачными друг для друга.
* **Долговечность** – гарантирует, что результат или результат совершенной транзакции сохраняется в случае сбоя системы.

**Уровни изолированности транзакций**

**Read uncommitted (чтение незафиксированных данных)**

Если несколько параллельных транзакций пытаются изменять одну и ту же строку таблицы, то в окончательном варианте строка будет иметь значение, определенное всем набором успешно выполненных транзакций

**Read committed (чтение фиксированных данных)**

обеспечивается защита от чернового, «грязного» чтения, тем не менее, в процессе работы одной транзакции другая может быть успешно завершена и сделанные ею изменения зафиксированы

**Repeatable read (повторяющееся чтение)**

читающая транзакция «не видит» изменения данных, которые были ею ранее прочитаны. При этом никакая другая транзакция не может изменять данные, читаемые текущей транзакцией, пока та не окончена.

**Serializable (упорядочиваемость)**

транзакции полностью изолируются друг от друга, каждая выполняется так, как будто параллельных транзакций не существует.

**Проблемы одновременного выполнения нескольких транзакций** - **Грязное чтение** - транзакция Т1 модифицировала некий элемент данных. После этого другая транзакция Т2 прочитала содержимое этого элемента данных до завершения транзакции Т1. Если Т1 заврешается операцией ROLLBACK(откат в SQL), то получается, что транзакция Т2 прочитала не существующие данные.

**Неповторяемое (размытое) чтение** - транзакция Т1 прочитала содержимое элемента данных. После этого другая транзакция Т2 модифицировала или удалила этот элемент. Если Т1 прочитает содержимое этого элемента заново, то она получит другое значение или обнаружит, что элемент данных больше не существует.

**Фантом (фиктивные элементы)** - транзакция Т1 прочитала содержимое нескольких элементов данных, удовлетворяющих некому условию. После этого Т2 создала элемент данных, удовлетворяющий этому условию и зафиксировалась. Если Т1 повторит чтение с тем же условием, она получит другой набор данных."

**Сериализуемость параллельной обработки транзакций - Чередующееся (параллельное) -** выполнение заданного множества транзакций будет верным, если при его выполнении будет получен такой же результат, как и при последовательном выполнении тех же транзакций.

### Вопрос 6 блокировки

**Механизм блокировок** обеспечивает принудительное упорядочение транзакций. Если для выполнения некоторой транзакции необходимо, чтобы некоторый объект базы данных (кортеж, набор кортежей, отношение, набор отношений…) не изменялся непредсказуемо и без ведома этой транзакции, такой объект блокируется.

**Блокировка** — отметка о захвате объекта [транзакцией](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))

**Виды блокировок**

**блокировка со взаимным доступом**- S-блокировкой (блокировкой по чтению)

**монопольная блокировка**- X-блокировкой (блокировкой по записи)

**Если транзакция налагает на объект X-блокировку**

то любой запрос другой транзакции с блокировкой этого объекта будет отвергнут.

**Если транзакция налагает на объект S-блокировку**

То запрос со стороны другой транзакции с **X-блокировкой** на этот объект будет отвергнут, а с **S-блокировкой** принят

в чем разница между числами с фиксированной и плавающей точкой

Разобраться с режимами работы внешнего ключа. Лучше разобраться с физическими блокировками

Разобраться с: char, nchar, nvarchar; минимальностью PK;

## Тема 2 обьекты бд

## Тема 3 простые sql запросы

**Команда HAVING** позволяет фильтровать результат группировки, сделанной с помощью команды [**GROUP BY**](http://old.code.mu/sql/group-by.html).

SELECT поля FROM таблица GROUP BY поле HAVING поля=> значение

**Права доступа**

"GRANT SELECT, INSERT ON таблица TO пользователь1, пользователь2;

дать права на SELECT и INSERT в таблице пользователю1 и пользователю2

GRANT ALL ON таблица TO пользователь1;

дать все права в таблице пользователю1

GRANT UPDATE (столбец1, столбец2) ON таблица TO пользователь1;

дать права на UPDATE столбец1 и столбец2 в таблице пользователю1

GRANT SELECT ON таблица TO PUBLIC;

дать права на SELECT в таблице всем

GRANT INSERT ON таблица TO пользователь1 WITH GRANT OPTION;

дать права с возможностью передачи прав

REVOKE INSERT ON таблица FROM пользователь1;

забрать права на INSERT в таблице у пользователя1

Запрет действий DENY SELECT ON таблица TO пользователь1;

**Добавление в таблицу значений**

INSERT INTO имя таблицы (столбец 1, столбец 2) VALUES (“значение 1’, “значение 2”);

NOT NULL столбец не будет иметь NULL значение

DEFAULT задает значение по умолчанию

CREATE TABLE user (name VARCHAR (10) NOT NULL DEFAULT serg, group INTEGER (10));

Создали таблицу с двумя столбцами первый из которых не может иметь значение NULL и по умолчанию если не задать значение, то будет serg

**Выборка данных**

SELECT \* FROM user; – просмотреть все данные из таблицы user

SELECT выбрать столбцы FROM из таблицы WHERE по критерию столбец=значение

SELECT \* FROM user WHERE name =ivan;

В таблице user будет искать по столбцу name значение ivan и выводим все строки, в которых есть ivan

SELECT name, group FROM user WHERE name =ivan;

По столбцам name, group в таблице user ищем по столбцу name значение ivan и выводим все строки из выбранных столбцов в которых есть ivan

SELECT name FROM user;

Вывести из таблицы user столбец name

**Ключевое слово AND – и**

SELECT name, group FROM user WHERE name =’ivan’ AND group = 1;

Выведет всех Иванов из 1 группы

**OR (или) объединение двух запросов**

SELECT name, group FROM user WHERE name =’ivan’ OR name =’serg’;

Если истинными должны быть все условия, используйте AND. Если истинным должно быть хотя бы одно из условий, используйте 0R.

**Для выборки и поиска значение NULL используется IS NULL**

SELECT name FROM user WHERE name IS NULL;

**ключевое слово LIKE**, которое в сочетании со специальными символами ищет часть текстовой строки и возвращает совпадения.

SELECT \* FROM name WHERE user LIKE '%ivan’;

**%** — обозначает любое количество произвольных символов

**— знак подчеркивания** (\_) — представляет ровно один произвольный символ.

**ключевое слово BETWEEN** для проверки вхождения значений в диапазон можно воспользоваться

SELECT group FROM user WHERE group BETWEEN 1 AND 5;

Выведет все строки с 1-5 столбца group

**ключевого слова IN**

Вместо того чтобы строить длинные цепочки ОR, мы можем упростить запрос при помощи ключевого слова IN. После IN следует набор значений в круглых скобках. Если значение столбца совпадает с одним из значений набора, то запись или заданное подмножество столбцов включаются в результат запроса

SELECT group FROM user WHERE user IN (‘ivan ‘, serg ');

Данная запись выведет нам всех ivan, serg со всех group

**С конструкцией NOT IN** в выборку включаются записи, у которых значение столбца не входит в заданный набор.

SELECT group FROM user WHERE user NOT IN (‘ivan ‘, serg ');

Данная запись выведет всех кроме inav, serg

При использовании с AND и OR ключевое слово NOT ставится после них:

SELECT \* FROM user WHERE NOT name = 'ivan' AND NOT name = ‘serg’;

**Удаление и изменение данных в таблицы**

**Команда DELETE**

DELETE FROM user WHERE name = ‘serg’;

Удалит из таблицы user столбца name значение serg

* Команда delete не позволяет удалить значение одного столбца или группы столбцов.
* Команда delete удаляет из таблицы одну или несколько записей
* Также возможно удаление сразу нескольких записей. Для этого критерий выбора удаляемых записей определяется при помощи условия WHERE.
* следующая команда удаляет из таблицы все записи: DELETE FROM user

**Команда UPDATE**

UPDATE user SET name =’ivan’ WHERE name = ‘serg’;

Из таблицы user, столбца name значение serg измениться на ivan

UPDATE user SET name =’ivan’ WHERE group = 1;

Из таблицы user, столбца group со значением 1 все значение name поменяются на ivan

UPDATE user SET group = group + 1 WHERE name = ‘ivan’;

Из таблицы user столбца name со значением ivan измениться значение столбца group на +1, т.е если ivan был в 1 группе, то теперь он во 2

* Команда UPDATE может использоваться для изменения значения одного столбца или группы столбцов.

Включите дополнительные пары столбец = значение в условие SET и поставьте запятую после каждой пары: UPDATE your\_table SET первый\_столбец= 'новое\_значение', второй\_столбец = 'старое\_значение';

* Команда UPDATE может изменять одну запись или несколько записей в зависимости от условия where.

**Изменение таблицы**

**команда ALTER**

ALTER TABLE user

ADD COLUMN id INT NOT NULL AUTO\_INCREMENT FIRST

ADD PRIMARY KEY(id);

Добавляем в таблицу user новый столбец с помочью ADD COLUMN и присваиваем ему первое место в таблице с помощью FIRST

**Ключевое слова AFTER.**

задать местонахождение нового столбца в таблице после указанного столбца

**Ключевое слово BEFORE**

задать местонахождение нового столбца в таблице перед указанным столбцом

**Ключевое слово LAST**

задать местонахождение нового столбца в таблице в конце

**Ключевое слово FIRST**

задать местонахождение нового столбца в таблице в начале

**Ключевое слово SECOND**

задать местонахождение нового столбца в таблице после первого столбца

ALTER TABLE user ADD COLUMN family VARCHAR (20) AFTER name;

В таблице user создать новый столбец family после столбца name

CHANGE - изменение имени и типа данных столбцов

MODIDY- изменение типа данных или позиции столбцов

ADD - добавление столбцов в таблицу

DROP- удаление столбцов из таблицы

**Команда RENAME**

Позволяет переименовать таблицу

ALTER TABLE user RENAME TO children;

Переименовали таблицу user в children

**Конструкция CHANGE COLUMN**

Позволять менять имя и тип данных

ALTER TABLE user

CHANGE COLUMN group people VARCHAR (20);

Изменит столбец user на people и поменяет тип данных на символьный

**Команда MODIFY**

Изменяет тип данных

ALTER TABLE user

MODIFY COLUMN group VARCHAR (20);

В столбце user поменяет тип данных на символьный

**Команда DROP**

Используется для удаления архитектуры

ALTER TABLE user DROP COLUMN name;

Удаляет из таблицы user столбец name

**Функции RIGHT () и LEFT ()**

выделения заданного количества символов в столбце

SELECT RIGHT(name,4) FROM user;

Выделит с правого бока 4 символа в столбе name

**Функция SUBSTRING\_INDEX ()**

находит все символы текстового значения, предшествующие заданному символу или подстроке.

SELECT SUBSTRING\_INDEX (name, ‘,’ ,1) FROM user;

Возвращает все символы до 1 запятой

**Функция SUBSTRING (текст, начало, длина)**

возвращает часть строкового значения текст, начиная с буквы в позиции начало. Параметр длина определяет длину возвращаемой строки.

SELECT SUBSTRING {'Сан-Антонио, ТХ', 5, 3);

**Функции UPPER (текст) и LOWER (текст)** преобразуют все символы строки к верхнему или нижнему регистру

SELECT UPPER('сШа'); SELECT LOWER('спаГЕтти');

**Функция REVERSE (текст)**

переставляет символы строки в обратном порядке.

SELECT REVERSE('спаГЕтти');

**функции LTRIM(TeKCT) и RTRlM(TeKCT)**

возвращают строку, полученную удалением лишних пробелов в начале (у левого края) или в конце (у правого края) строки.

SELECT LTRIMC (‘собака ');

SELECT RTRIMC (‘кошка ');

**функция LENGTH (текст)**

возвращает количество символов в строке.

SELECT LENGTH ('Сан-Антонио, ТХ ');

**Объедение функций и команд**

UPDATE user SET family = RIGHT (name, 2);

В таблице user в столбец family и заполнит его выделенными значениями из столбца name

**Выражение CASE**

объединяет множество команд UPDATE, проверяя значение существующего столбца по условию. Если условие выполняется, то новый столбец заполняется заданным значением

UPDATE user

SET новый столбец =

CASE

WHEN столбец 1 = значение 1 //если выполнится это условие

THEN новое значение 1 // то новому столбцу присвоиться новое знач. 1

WHEN столбец 2 = значение 2 // если выполняется другое условие

THEN новое значение 2 // то новому столбцу присвоиться новое знач. 2

ELSE новое значение 3 // если не одно условие не выполниться то новому столбцу присвоится новое значение 3

END; //завершает выполнение CASE

### Основные команды

**DESCRIBE** позволяет посмотреть какая архитектура у таблицы

**SHOW CREATE\_TABLE** возвращает команду CREATE TABLE, которая была использована для создания таблицы

### Функции

**Ключевые слова ORDER BY**

Упорядочивает записи

SELECT name FROM user WHERE name ORDER BY name;

Упорядочивает записи в столбце name по алфавиту

**Ключевое слово DESC**

Указывается после имени столбца изменяя порядок по убыванию

SELECT name FROM user WHERE name ORDER BY name DESC;

**Ключевое слово ASC**

Указывается после имени столбца изменяя порядок по возрастанию

SELECT name FROM user WHERE name ORDER BY name ASC ;

**Функция SUM**

суммирует значения столбца

SELECT SUM (group) FROM user;

Суммирует значение таблицы group

**функцию AVG**.

суммирует значения столбца и находит среднее значение

**функция MIN**

выводит минимальное значение

**функция MAX**

выводит максимальное значение

**условие GROUP BY**

группирует записи и суммирует

**функция COUNT**

возвращает количество записей в столбце

**Ключевое слово DISTINT**

возвращает записи в столбце без дубликатов

SELECT DISTINT name FROM user;

Выведет все значение из столбца name без дубликатов

**Ключевое слово LIMIT**

позволяет указать количество записей, возвращаемых запросом из итогового набора

так же есть LIMIT 0,2 позволяет вернуть записи от 1 до 2

### Многотабличные базы данных

### Перекрестное соединение

**Псевдонимы столбцов**

указывается после первого использования имени столбца в запросе с другим ключевым словом AS

**Ключевое слово AS**

заполняет новую таблицу результатами SELECT

CREATE TABLE user(

id INT(10) NOT NULL AUTO\_INCREMENT PRIMARY KEY,

user VARCHAR(20) )

AS

SELECT user AS usr FROM baz\_dan

GROUP BY usr

ORDER BY usr;

В таблица user создали стобец id и столбец user пометивего псевдонимом usr с помощью команды AS

**Перекрестное соединение (CROSS JOIN)** возвращает комбинации каждой записи первой таблицы с каждой записью второй таблицы.

SELECT учителя.имя, ученики.номер\_учителя FROM учителя CROSS JOIN ученики;

**Внутренним соединением (INNER JOIN)** называется перекрестное соединение, из результатов которого часть записей исключается по условию запроса.

**Эквивалентное соединение** **=** внутреннее соединение с проверкой равенства.

**Неэквивалентное соединение <>** возвращает записи, у которых заданные значения столбцов не равны

**Естественные соединения** **(NATURAL JOIN)** возможны только в том случае, если столбец, по которому выполняется соединение, имеет одинаковые имена в обеих таблицах

**LEFT OUTER JOIN**

В левом внешнем соединении для каждой записи левой таблицы ищется соответствие среди записей правой таблицы.

внешние соединения в большей степени зависят от отношений между двумя таблицами, чем все рассмотренные ранее типы соединений.

Левое внешнее соединение **(LEFT OUTER JOIN**) перебирает все записи левой таблицы и ищет для каждой соответствие среди записей правой таблицы.

В левом внешнем соединении таблица, следующая

после FROM, но ДО JOIN, считается ≪левой≫, а таблица, следующая ПОСЛЕ JOIN, считается ≪правой≫.

внешнее соединение возвращает запись независимо от того, есть у нее совпадение в другой таблице или нет.

Значение NULL в результатах левого

внешнего соединения означает, что правая таблица не содержит значений, соответствующих левой таблице.

**RIGHT OUTER JOIN**

Правое внешнее соединение ищет в левой таблице соответствия для правой таблицы

Одна и та же таблица может использоваться в качестве левой и правой таблицы соединения.

### Подзапросы

запрос внутри другого запроса

SELECT столбец 1, столбец 2 FROM table WHERE column =(SELECT column FROM table);

* Подзапрос всегда представялет собой команду SELECT
* Позапросы всегда заключаеются в круглые скобки
* Подзапросы не имеют собсвтенного символа точка с запятой, как обычно , один такой символ обозначает конец всего запроса
* Подзапросы могут использоваться в четырех местах запросов SELECT, SELECT список столбцов как один из столбцов, условие FROM и условие WHERE
* Подзапросы могут использоватсья в командах INSERT. DELETE. UPDATE. SELECT

Подзапрос, используемый в качестве выражения столбца в команде SELECT, может возвращать только одно значение из одного столбца

Внутренний запрос обрабатывается первым, а полученный

результат используется в условии WHERE внешнего запроса

Автономный подзапрос, не содержащий ссылок на данные внешнего запроса, называется некоррелированным подзапросом.

ключевые слова IN или NOT IN проверяют соответственно присутствие или отсутствие результатов внешнего запроса среди результатов подзапроса.

В модели с некоррелированным подзапросом внутренний запрос (то есть подзапрос) интерпретируется РСУБД после внешнего запроса.

в модели с коррелированным подзапросом выполнение внутреннего запроса возможно только при условии выполнения внешнего.

Внешние соединения возвращают все записи одной из таблиц вместе со всеми соответствующими данными из другой таблицы. При внутреннем соединении сравниваются записи двух таблиц, причем порядок следования этих двух таблиц неважен.

**РЕФЛЕКСИВНЫЙ внешний ключ**

Первичный ключ таблицы, используемый в той же таблицедля других целей.

При само соединении запрос к одной таблице строится так, как если бы она была двумя таблицами, содержащими одинаковую информацию

**ключевое слово UNION**.

Союз объединяет в одну таблицу результаты двух и более запросов на основании того, что указано в запросе SELECT. Союзы можно трактовать как ≪пересекающиеся≫ значения всех запросов.

При использовании UNION допускается только одно условие ORDER BY в концекоманды *.* Это объясняется тем, что союз объединяет и группирует результаты нескольких команд SELECT

Количество столбцов в командах SELECT должно быть одинаковым.Нельзя выбрать два столбца одной командой и еще один столбец другой. Команды SELECT должны содержать одинаковые выражения и агрегатные функции. Команды SELECT могут следовать в любом порядке; на результаты это не влияет.

По умолчанию SQL исключает дубликаты из результатов союзов Типы данных в столбцах должны либо совпадать, либо быть совместимыми. Если вы по какой-то причине хотите получить список со всеми дубликатами**,** используйте **оператор UNION ALL .**Он возвращает все совпадения, не только уникальные.

UNION ALL работает точно так же, как UNION, если не считать того, что он возвращает все значения из столбцов — вместо одного экземпляра из каждой группы дубликатов.

**Конструкции INTERSECT и EXCEPT**, в отличие от UNION, используются для поиска перекрывающихся результатов запросов. INTERSECT возвращает данные из результатов первого запроса, которые также присутствуют во втором запросе.

EXCEPT возвращает только те значения, которые присутствуют в результатах первого запроса, но не встречаются в результатах второго.

разобраться что такое NULL в SQL,

разобраться какие JOIN-ы бывают.

Разобраться с конкатенацией строк в SQL запросов.

Разобраться лучше с конкатенацией строк в SQL, разобраться что такое NULL, разобраться с alias.

## Тема 4 Продвинутый SQL

## Тема 5 Оптимизация производительности

# Другие знания

## Тема 1 Командная оболочка CMD (Windows)

## Тема 2 Командная оболочка Bash (Linux)

## Тема 3 Языки разметки и структуризации данных

## Тема 4 Регулярные выражения

## Тема 5 Тесты

## Тема 6 Безопасность и методы защиты приложений

## Тема 7 VCS