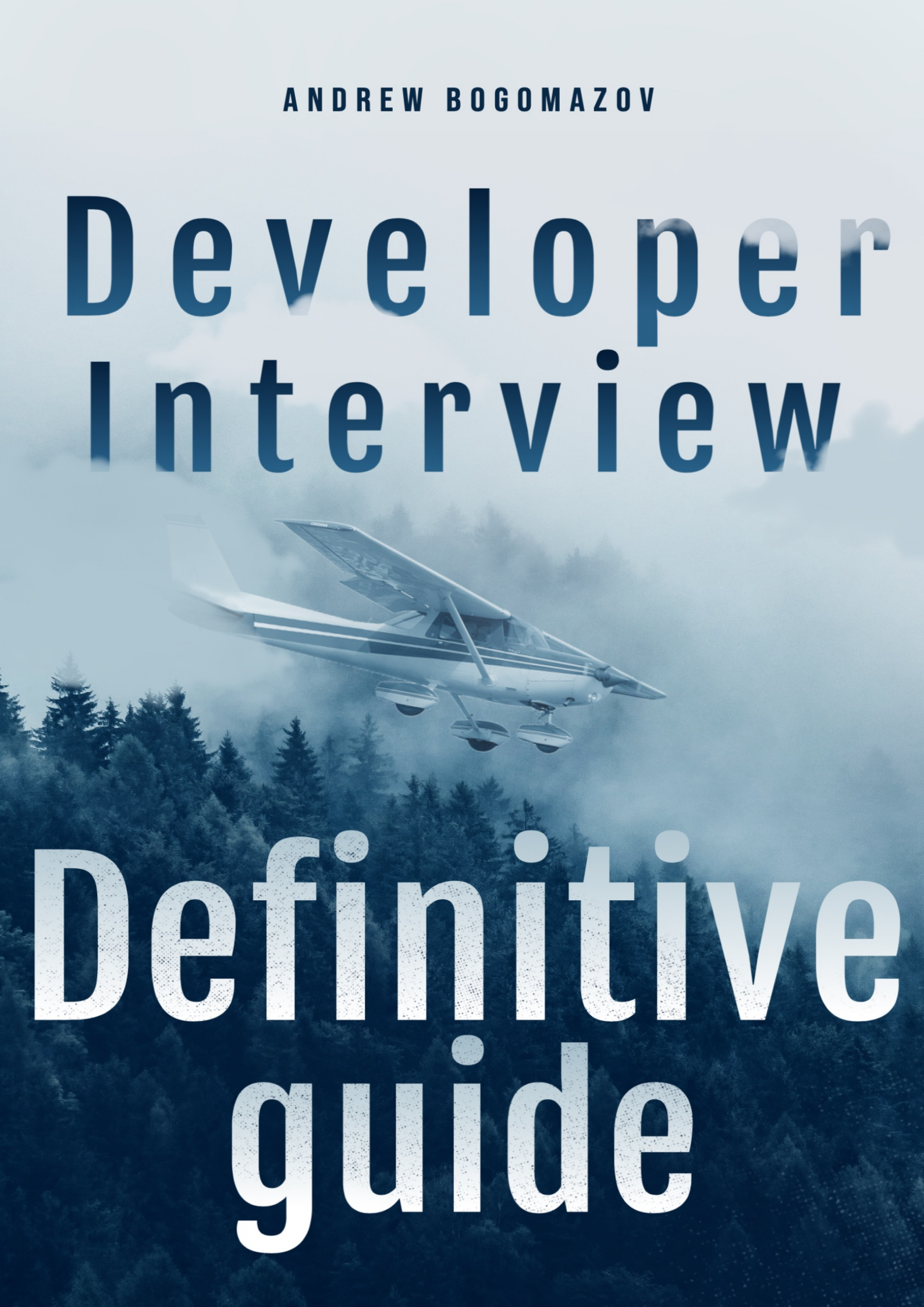
ANDREW BOGOMAZOV

Developer Interview



Definitive guide

Содержание

IDDQD

Матрица Компетентности

PDF EPUB MOBI

SQL

Что такое SQL

SQL (*structured query language* - формальный не процедурный язык программирования, применяемый для создания, модификации и управления данными в произвольной реляционной базе данных, управляемой соответствующей системой управления базами данных (СУБД).

Что такое DLL

DDL(Data Definition Language) - Команды определения структуры данных. В состав DDL-группы входят команды, позволяющие определять внутреннюю структуру базы данных. Перед тем, как сохранять данные в БД, необходимо создать в ней таблицы и, возможно, некоторые другие сопутствующие объекты

Пример некоторых DDL-команд:

- `CREATE TABLE` - Создание новой таблицы
- `DROP TABLE` - Удалить существующую таблицу

Какие есть типы `JOIN`

- **INNER JOIN** - внутреннее соединение. В результирующем наборе присутствуют только записи, значения связанных полей в которых совпадают.
- **LEFT JOIN** - левое внешнее соединение. В результирующем наборе присутствуют все записи из первой таблицы и соответствующие им записи из второй таблицы. Если соответствия нет, поля из второй таблицы будут пустыми.
- **RIGHT JOIN** - правое внешнее соединение. В результирующем наборе присутствуют все записи из второй таблицы и соответствующие им записи из первой таблицы. Если соответствия нет, поля из первой таблицы будут пустыми.
- **FULL JOIN** - полное внешнее соединение. Комбинация двух предыдущих. В результирующем наборе присутствуют все записи из первой таблицы и соответствующие им записи из второй таблицы. Если соответствия нет - поля из второй таблицы будут пустыми. Записи из второй таблицы, которым не нашлось пары в первой таблице, тоже будут присутствовать в результирующем наборе. В этом случае поля из первой таблицы будут пустыми.
- **CROSS JOIN** - Результирующий набор содержит все варианты комбинации строк из первой таблицы и второй таблицы. Условие соединения при этом не указывается.

Примеры различий `LEFT JOIN` и `RIGHT JOIN`

Table1

Key1	Field1
1	A
2	B
3	C

Table2

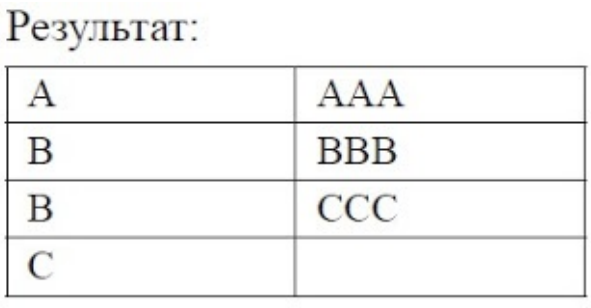
Key2	Field2
1	AAA
2	BBB
2	CCC
4	DDD

LEFT JOIN

```
SELECT Table1.Field1, Table2.Field2
```

```
FROM Table1
LEFT JOIN Table2
ON Table1.Key1 = Table2.Key2
```

Результат:



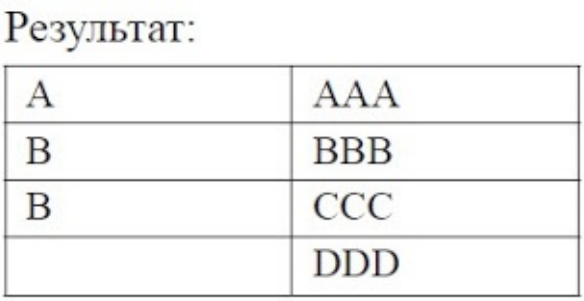
A	AAA
B	BBB
B	CCC
C	

Результат:

RIGHT JOIN

```
SELECT Table1.Field1, Table2.Field2
FROM Table1
RIGHT JOIN Table2
ON Table1.Key1 = Table2.Key2
```

Результат:



A	AAA
B	BBB
B	CCC
	DDD

Результат:

Для чего используется слово `HAVING`

Ключевое слово `HAVING` определяет условие, которое затем применяется к групам строк. Следовательно, это предложение имеет тот же смысл для группы строк, что и предложение `WHERE` в отношении соодержимого соответствующей таблицы. Синтаксис предложения `HAVING`:

```
HAVING [condition]
```

где condition содержит агрегатные функции или константы.

Важно понимать, что секции `HAVING` и `WHERE` взаимно дополняют друг друга. Сначала с помощью ограничений `WHERE` формируется итоговая выборка, затем выполняется разбивка на группы по значениям полей, заданных в `GROUP BY`. Далее по каждой группе вычисляется групповая функция и в заключение накладывается условие `HAVING`.

Пример:

```
SELECT DeptNum, MAX(SALARY)
FROM Employees
GROUP BY DeptNum
HAVING MAX(SALARY) > 1000
```

В приведенном примере в результат попадут только отделы, максимальная зарплата в которых превышает 1000

JSON

Что такое JSON

JSON (JavaScript Object Notation) - простой формат обмена данными, удобный для чтения и написания как человеком, так и компьютером. Он основан на JavaScript. JSON - текстовый формат, полностью независимый от языка реализации, но он использует соглашения, знакомые программистам C-подобных языков, таких как C, C++, C#, Java, JavaScript, Perl, Python и многих других. Эти свойства делают JSON идеальным языком обмена данными.

Что такое JSON Schema

JSON Schema - это стандарт описания структур данных в формате JSON. Использует синтаксис JSON. Схемы, описанные этим стандартом, имеют MIME "application/schema+json". Стандарт удобен для использования при валидации и документировании структур данных, состоящих из чисел, строк, массивов и структур типа ключ-значение.

Что такое JSON объект

JSON объект - неупорядоченный набор пар ключ/значение. Объект начинается с { (открывающей фигурной скобки) и заканчивается } (закрывающей фигурной скобкой). Каждое имя сопровождается : (двоеточием), пары ключ/значение разделяются , (запятой).

Какие есть правила синтаксиса JSON объекта (массива)

Есть несколько основных правил для создания строки JSON:

- Строка JSON содержит либо массив значений, либо объект (ассоциативный массив пар имя/значение).
- Массив заключается в квадратные скобки ([и]) и содержит разделенный запятой список значений.
- Объект заключается в фигурные скобки ({ и }) и содержит разделенный запятой список пар имя/значение.
- Пара имя/значение состоит из имени поля, заключенного в двойные кавычки, за которым следует двоеточие (:) и значение поля.

Чтобы включить двойные кавычки в строку, нужно использовать обратную косую черту: \". Так же, как и во многих языках программирования, можно помещать управляющие символы и шестнадцатеричные коды в строку, предваряя их обратной косой чертой.

Ниже приводится пример оформления заказа в формате JSON:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Какие типы данных, поддерживаются в JSON

Значение в массиве или объекте может быть:

- Числом (целым или с плавающей точкой)
- Строкой (в двойных кавычках)
- Логическим значением (true или false)
- Другим массивом (заключенным в квадратные скобки)
- Другой объект (заключенный в фигурные скобки)
- Значение null

Каковы недостатки JSON

Недостатками JSON являются:

- Трудно читается и анализируется пользователем, нет визуальности
- Нет синтаксиса для задания типа объекта
- Много синтаксического мусора

Что такое JSONP

JSONP или "JSON with padding" является расширением JSON, позволяющим выполнять в единообразном стиле асинхронные запросы к сервисам, расположенным на другом домене - операцию, запрещённую в типичных веб-браузерах из-за политики ограничения домена.

Какой MIME-тип в JSON

`application/json`

Для чего используется JSON

Наиболее частое распространенное использование JSON - пересылка данных между сервером и браузером, а так же для хранения данных. Обычно данные JSON доставляются с помощью AJAX, который позволяет обмениваться данными браузеру и серверу без необходимости перезагружать страницу.

Какие преимущества использования JSON

Сравнительные преимущества JSON:

- В разы меньший объем данных (экономия трафика, плюс к скорости работы сайта)
- Меньшая загрузка процессора и клиента, и сервера
- Почти неограниченные возможности расширения (т.к. можно выполнить ф-цию)
- Его предложения легко читаются и составляются как человеком, так и компьютером.
- Его легко преобразовать в структуру данных для большинства языков программирования (числа, строки, логические переменные, массивы и так далее)
- Многие языки программирования имеют функции и библиотеки для чтения и создания структур JSON.

Какая функция используется для преобразования текста JSON в объект

Чтобы преобразовать текст JSON в объект используется функция `eval()`.

Что такое JSON Parser

Вызов `JSON.parse(str)` превратит строку с данными в формате JSON в JavaScript-объект/массив/значение, возможно с преобразованием получаемого в процессе разбора значения.

Что такое JSON-RPC

JSON-RPC(сокр. от англ. JavaScript Object Notation Remote Procedure Call - JSON-вызов удалённых процедур) - протокол удалённого вызова процедур, использующий JSON для кодирования сообщений. Это очень простой протокол (очень похожий на XML-RPC), определяющий только несколько типов данных и команд. JSON-RPC поддерживает уведомления (информация, отправляемая на сервер, не требует ответа) и множественные вызовы.

Что такое JSON-RPC-Java

Реализация протокола JSON-RPC на Java

Какова роль `JSON.stringify()`

Метод `JSON.stringify()` преобразует объекты JavaScript в строку в формате JSON, возможно с заменой значений, если указана функция замены, или с включением только определённых свойств, если указан массив замены. Используется, когда нужно из JavaScript передать данные по сети.

Как парсить JSON в JavaScript

```
var json = '{"name": "Andrew", "sn": "Cohen", "age": "24"}';
var obj = JSON.parse(json);
```

Как создать JSON объект из JavaScript

```
var obj = {};
    obj['name'] = 'Andrew';
    obj['sn'] = 'Cohen';
    obj['age'] = 24;
```

Валидациия JSON в JavaScript

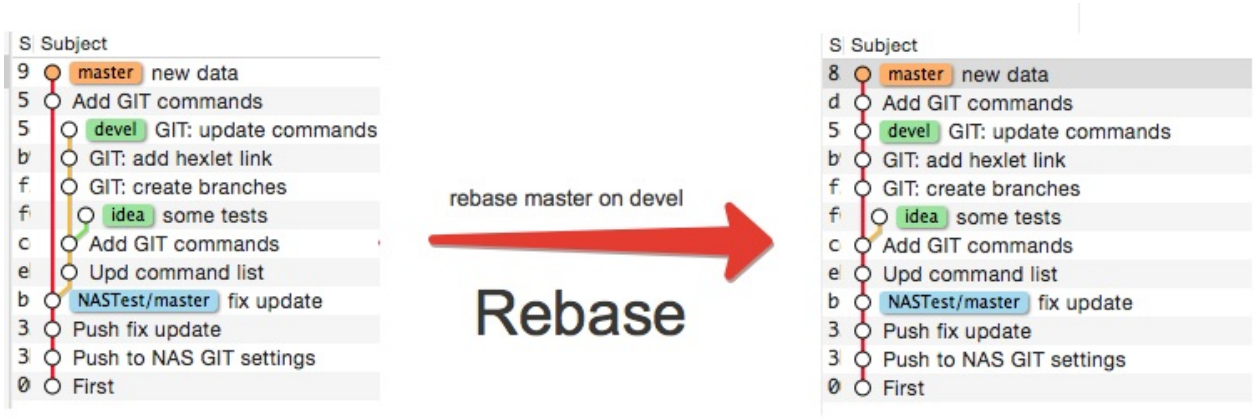
```
function isValidJSON(jsonData) {
    try {
        JSON.parse(jsonData)
        return true;
    }
    catch (e) {
        return false;
    }
}

var json = '{"name": "Andrew", "sn": "Cohen", "age": "24"}';
isValidJSON(json);
```

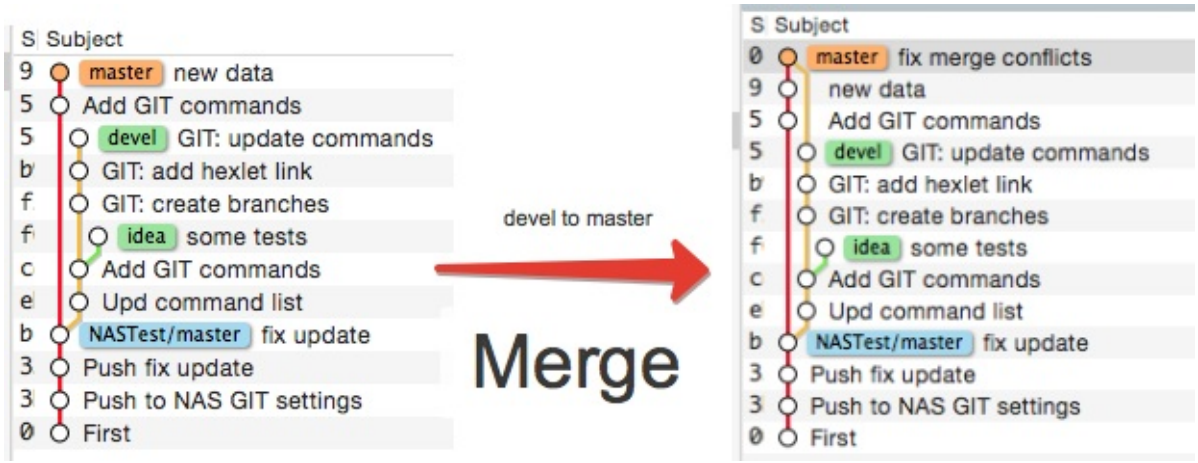
Git

Чем отличается REBASE, MERGE, CHERRY-PICK

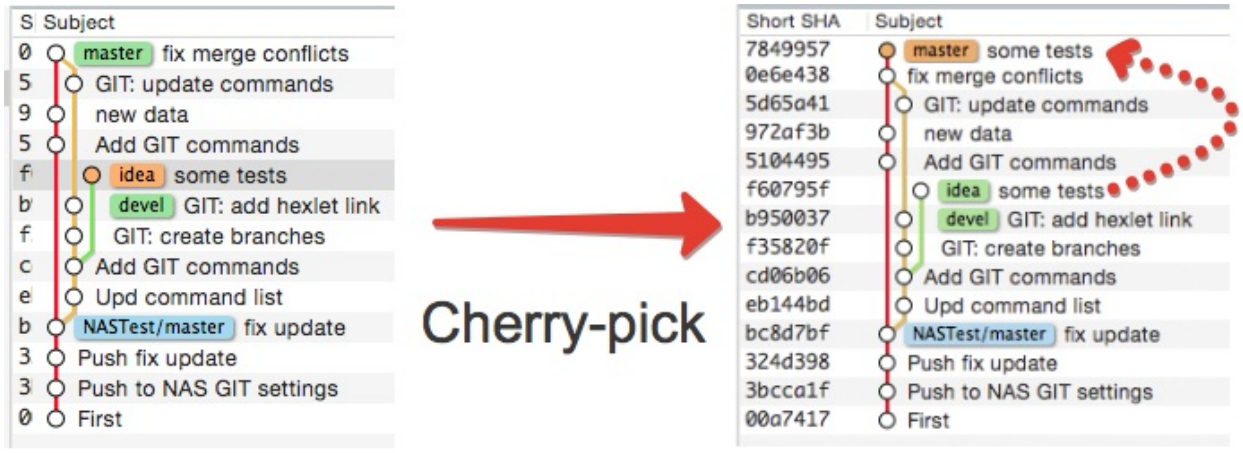
`git rebase <branch>` - "сшивает" коммиты по дате их создания



`git merge <branch>` - Добавляет коммиты в конец графа



`git chery-pick <branch>` - Забрать коммиты из ветки в свою



Maven

Как запустить сборку без выполнения тестов

Для запуска сборки без выполнения тестов нужно добавить параметр: `-Dmaven.test.skip=true`

Как запустить только один тест

Для запуска только одного теста нужно добавить параметр: `-Dtest=[Имя класса]`

Например: `mvn install -Dtest=ru.apache-maven.utils.ConverterTest`

JUnit Interview Questions

Что такое Unit Testing

Модульное тестирование или *Unit Testing* - процесс проверки на корректность функционирования отдельных частей исходного кода программы путем запуска тестов в искусственной среде.

Что такое Fixture

Фикстура или Fixture - состояние среды тестирования, которое требуется для успешного выполнения тестового метода. Может быть представлено набором каких-либо объектов, состоянием базы данных, наличием определенных файлов, соединений и прочее.

Какие аннотации используются для создания фикстур

Предусмотрено четыре аннотации две для фикстур уровня класса и две для уровня метода.

- `@BeforeClass` - Запуск перед выполнением класса
- `@Before` - Запуск перед каждым тестовым методом
- `@After` - Запуск после каждого тестого метода
- `@AfterClass` - Запуск после выполнения класса

Для чего нужна аннотация `@Ignore`

Аннотация `@Ignore` заставляет инфраструктуру тестирования проигнорировать данный тестовый метод. Аннотация предусматривает наличие комментария о причине игнорирования теста.

Object Oriented Programming

Что такое ООП

ООП - методология программирования, основанная на представлении программного продукта в виде совокупности объектов, каждый из которых является экземпляром конкретного класса. ООП использует в качестве базовых элементов взаимодействие объектов.

Что такое объект

Объект - именованная модель реальной сущности, обладающая конкретными значениями свойств и проявляющая свое поведение, обладающая именем, набором данных (полей и свойств объекта), физически находящихся в памяти компьютера, и методов, имеющих доступ к ним.

Объект - конкретный экземпляр класса.

Назовите основные принципы ООП

Принято считать, что объектно-ориентированное программирование строится на 4 основных принципах (раньше их было всего 3). Эти принципы:

- Абстракция
- Инкапсуляция
- Наследование
- Полиморфизм

Что такое наследование

Наследование это процесс благодаря которому один объект может приобрести свойства другого объекта (наследование всех свойств одного объекта другим) и добавлять новые.

```
class Dog extends Animal
{...}
```

Терминология: Суперкласс -> Подкласс Родительский -> Дочерний

Что такое полиморфизм? Каковы проявления полиморфизма в Java

Полиморфизм (от греческого polymorphos) - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.

Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных.

В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий.

Что такое инкапсуляция

Инкапсуляция - это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса), это свойство которое позволяет закрыть доступ к полям и методам класса другим классам, а предоставлять им доступ только через интерфейс(метод). При использовании объектно-ориентированного подхода не принято применять прямой доступ к свойствам

какого-либо класса из методов других классов. Для доступа к свойствам класса принято задействовать специальные методы этого класса для получения и изменения его свойств.

Что такое абстракция

Абстракция - это выделение общих характеристик объекта,исключая набор незначительных.

С помощью принципа абстракции данных, данные преобразуются в объекты. Данные обрабатываются в виде цепочки сообщений между отдельными объектами. Все объекты проявляют свои уникальные признаки поведения. Огромный плюс абстракции в том, что она отделяет реализацию объектов от их деталей, что в свою очередь позволяет управлять функциями высокого уровня через функции низкого уровня.

В чем преимущества объектно-ориентированных языков программирования

Они представляют реальные объекты в жизни, например, Машина, Джип, Счет в банке и так далее. Инкапсуляция, наследование и полиморфизм делает его еще мощнее.

Как использование объектно-ориентерованного подхода улучшает разработку программного обеспечения

Основные преимущества:

- повторное использование кода(наследование)
- реальное отображение предметной области. Объекты соответствуют реальности

Имеется выражение "является" и "имеет". Что они подразумевают в плане принципов ООП? В чем разница между композицией и агрегацией

- является - наследование
- имеет - композиция

В качестве примера предположим что у нас есть классы Строение, Дом и Ванная комната. Так вот Дом является строением, что нельзя сказать про Ванну, которая не является домом. А вот Дом имеет\включает в себя Ванну. Если вы хотите использовать повторно код,то не обязательно использовать наследование. Если нет отношения "является", то лучше тогда использовать композицию для повторного использования кода.

Не используйте наследование для получение полиморфизма, если нет ключевой зависимости "является". Используйте интерфейсы для полиморфизма.

Из спецификации можно узнать, что:

- Ассоциация обозначает связь между объектами
- Агрегация и композиция это частные случаи ассоциации

Агрегация предполагает, что объекты связаны взаимоотношением "part-of" (часть). **Композиция** более строгий вариант агрегации. Дополнительно к требованию part-of накладывается условие, что "часть" не может одновременно принадлежать разным "хозяевам",и заканчивает своё существование вместе с владельцем.

Например:

- мотоцикл -> сумка с багажём - ассоциация. *Отношение: имеет*.
- мотоцикл -> колесо - композиция. *Отношение: имеет*
- группа по интересам -> человек - агрегация. человек часть группы, но может принадлежать нескольким разным группам.

Что вы подразумеваете под полиморфизмом, инкапсуляцией и динамическим связыванием

Полиморфизм означает способность переменной данного типа, которая ссылается на объекты разных типов, при этом вызывается метод, характерный для конкретного типа ссылки на объект.

В чем преимущество полиморфизма? Он позволяет добавлять новые классы производных объектов, не нарушая при этом код вызова. Также использование полиморфизма называют динамическим связыванием объектов.

Рассмотрим пример полиморфизма:

Имеется классы: Figure, Circle и Triangle. Circle и Triangle наследуется от Figure соответственно. Каждый класс имеет метод draw(). В Circle и Triangle этот метод переопределен.

Так вот, создаем объект с типом Figure и присваиваем ей ссылку на объект типа Circle и вызываем на этом объекте метод draw(). В итоге вызывается метод класса Circle, а не класса Figure как ожидалось.

```
Figure  f = new Circle();
f.draw();
```

Также вместо класса родителя Figure к примеру можно использовать интерфейс Figure, определив там метод рисовать. Этот интерфейс мы имплементируем в классах Circle, Triangle. Далее на интерфейсе создаем объект и присваиваем ему ссылку на объект какого-то из реализующих этот интерфейс классов.

Это удобно например если у нас есть некий метод:

```
public void drawShape(Figure f){
f.draw();
}
```

Обратите внимание что в метод мы передаем параметр с типом интерфейса, т.е. мы не знаем какой именно тип объекта будет, но реализация будет такая же. Далее мы можем просто создать еще класс, к примеру Trapezoid,и имплементировать интерфейс Figure и просто передать экземпляр класса в метод, ничего не меняя в реализации и вызове.

Наследование это включение поведения(методы) и состояния(поля) базового класса в производный от него. В результате этого мы избегаем дублирования кода и процесс исправления ошибок в коде также упрощается.

В джава есть два вида наследования:

- наследование классов. Каждый наследник может иметь толко одного родителя
- наследование интерфейсов. Интерфейс может иметь сколько угодно родителей

Некоторые тонкие ньюансы по поводу наследования интрефейсов и классов. Мы имеем два интерфейса с одинаковыми по имени полями. Имплементируем эти интерфейсы на каком-то классе. Как нам вызвать поля этих интерфейсов?

У нас неоднозначность. Необходимо объект класса привести к нужному интерфейсу.

```
Class c = new Class();
((InterfaceOne) c).field;
```

Хорошо, что будет если мы имеем метод с одинаковой сигнатурой в интерфейсах и реализуем эти интерфейсы на классе. Как нам в классе реализовать два метода с одинаковой сигнатурой?

Никак, мы просто реализовываем один общий метод в классе. Это является недостатком, так как нам может потребоваться разная реализация.

И третий случай: У нас есть класс и интерфейс с одинаковым по сигнатуре методом. Мы наследуемся от этого класса и имплементируем этот интерфейс. Что нам нужно делать? ведь необходимо реализовать метод интерфейса по всем правилам.

И вот тут интересно, компилятор не выдает ошибок, так как метод уже у нас реализован в классе родителе.

DRY

Don’t repeat yourself или DRY — это принцип разработки программного обеспечения, нацеленный на снижение повторения информации различного рода, особенно в системах со множеством слоёв абстрагирования.

Нарушения принципа DRY называют WET — «Write Everything Twice» или «We enjoy typing»

Принцип DRY формулируется как: «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы». Когда принцип DRY применяется успешно, изменение единственного элемента системы не требует внесения изменений в другие, логически не связанные элементы. Те элементы, которые логически связаны, изменяются предсказуемо и единообразно. Помимо использования методов и функций в коде, Томас и Хант считают необходимым использование генераторов кода, автоматических систем компиляции.

GRASP

`GRASP` General Responsibility Assignment Software Patterns — шаблоны, используемые в объектно-ориентированном проектировании для решения общих задач по назначению ответственностей классам и объектам.

Краткая характеристика девяти шаблонов

Информационный эксперт (Information Expert)

Шаблон определяет базовый принцип распределения ответственностей:

> Ответственность должна быть назначена тому, кто владеет максимумом необходимой информации для исполнения — информационному эксперту.

Этот шаблон — самый очевидный и важный из девяти. Если его не учесть — получится спагетти-код, в котором трудно разобраться. Локализация же ответственностей, проводимая согласно шаблону:

Повышает:

- Инкапсуляцию
- Простоту восприятия
- Готовность компонентов к повторному использованию

Создатель (Creator)

Класс должен создавать экземпляры тех классов, которые он может:

- Содержать или агрегировать
- Записывать
- Использовать
- Инициализировать, имея нужные данные

Контроллер (Controller)

Отвечает за операции, запросы на которые приходят от пользователя, и может выполнять сценарии одного или нескольких вариантов использования (например, создание и удаление) Не выполняет работу самостоятельно, а делегирует компетентным исполнителям.

Слабая сцепка (Low Coupling)

«Степень сцепки — мера зависимости элемента от других элементов. «Слабое» зацепление является оценочной моделью, которая диктует, как распределить обязанности, которые необходимо поддерживать.

Высокая связность (High Cohesion)

Высокая связность класса — это оценочная модель, направленная на удержание объектов должным образом сфокусированными, управляемыми и понятными. Высокая связность обычно используется для поддержания низкого зацепления. Высокая связность означает, что обязанности данного элемента тесно связаны и сфокусированы. Разбиение программ на классы и подсистемы является примером деятельности, которая увеличивает связность системы. И наоборот, низкая связность — это ситуация, при которой данный элемент имеет слишком много несвязанных обязанностей. Элементы с низкой связностью часто страдают от того, что их трудно понять, трудно использовать, трудно поддерживать.

Полиморфизм (Polymorphism)

См. ООП - Полиморфизм

Устройство и поведение системы:

- Определяется данными
- Задано полиморфными операциями её интерфейса

Чистая выдумка (Pure Fabrication)

«Pure Fabrication» отражает концепцию сервисов в модели предметно-ориентированного проектирования. *Пример* задачи: Не используя средства класса «А», внести его объекты в базу данных. *Решение*: Создать класс «Б» для записи объектов класса «А» см. ORM -> « `Data Access Object` ».

Не относится к предметной области, но:

- Уменьшает зацепление
- Повышает связность
- Упрощает повторное использование

Посредник (Indirection)

Слабое зацепление между элементами системы (и возможность повторного использования) обеспечивается назначением промежуточного объекта их посредником.

Пример: В архитектуре MVC, контроллер ослабляет зацепление данных модели с их представлением.

Устойчивость к изменениям (Protected Variations)

Шаблон защищает элементы от изменения другими элементами (объектами или подсистемами) с помощью вынесения взаимодействия в фиксированный интерфейс, через который (и только через который) возможно взаимодействие между элементами. Поведение может варьироваться лишь через создание другой реализации интерфейса.

KISS

`KISS` акроним для «Keep it simple» — принцип проектирования, принятый в ВМС США в 1960.

Принцип KISS утверждает, что большинство систем работают лучше всего, если они остаются простыми, а не усложняются. Поэтому в области проектирования простота должна быть одной из ключевых целей, и следует избегать ненужной сложности. Вариации на фразу включают «англ. Keep it Simple, Silly», «Keep it short and simple», «keep it small and simple», «Keep it simple, stupid».

Этот принцип лучше всего иллюстрируется историей, когда Джонсон вручил команде инженеров-конструкторов несколько инструментов, бросив им вызов с тем, что реактивный самолёт, который они проектировали, должен быть ремонтируемым для среднего механика в поле в боевых условиях только с этими инструментами. Таким образом, «stupid» относится к отношению между тем, что всё ломается, и сложностью необходимого для этого ремонта.

- Разбивайте задачи на подзадачи которые не должны по вашему мнению длиться более 4-12 часов написания кода
- Разбивайте задачу на множество более маленьких задач, каждая задача должна решаться одним или парой классов
- Сохраняйте ваши методы маленькими. Каждый метод должен состоять не более чем из 30-40 строк. Каждый метод должен решать одну маленькую задачу, а не множество случаев. Если в вашем методе множество условий, разбейте его на несколько. Это повысит читаемость, позволит легче поддерживать код и быстрее находить ошибки в нём. Вы полюбите улучшать код.
- Сохраняйте ваши классы маленькими. Здесь применяется та же техника что и с методами.
- Не бойтесь избавляться от кода. Изменение старого кода и написание нового решения два очень важных момента. Если вы столкнулись с новыми требованиями, или не были оповещены о них ранее, тогда порой лучше придумать новое более изящное решение решающее и старые и новые задачи.

TMTOWTDI

Принцип `TMTOWTDI` произносится «Тим Тоуди», или «There's More Than One Way To Do It» — девиз языка Perl.

Этот принцип с самого начала имелся в виду при создании данного языка программирования. В соответствии с этой идеей синтаксис языка предоставляет программисту множество возможностей для записи одного и того же алгоритма, позволяя выбирать ту из них, которая кажется наиболее удобной и эффективной в данном конкретном случае.

> Perl спроектирован так, чтобы дать несколько способов сделать одно и то же, обдумайте и выберите наиболее читаемый.
>
> В Python Должен быть один — и желательно только один — очевидный способ сделать это.

YAGNI

`YAGNI` «You aren't gonna need it» или «Вам это не понадобится» — процесс и принцип проектирования ПО, при котором в качестве основной цели и/или ценности декларируется отказ от избыточной функциональности, — то есть отказ добавления функциональности, в которой нет непосредственной надобности.

Согласно адептам принципа `YAGNI`, желание писать код, который не нужен прямо сейчас, но может понадобиться в будущем, приводит к следующим нежелательным последствиям:

- Тратится время, которое было бы затрачено на добавление, тестирование и улучшение необходимой функциональности.
- Новые функции должны быть отлажены, документированы и сопровождаться.
- Новая функциональность ограничивает то, что может быть сделано в будущем, — ненужные новые функции могут впоследствии помешать добавить новые нужные.
- Пока новые функции действительно не нужны, трудно полностью предугадать, что они должны делать, и протестировать их. Если новые функции тщательно не протестированы, они могут неправильно работать, когда впоследствии понадобятся.
- Это приводит к тому, что программное обеспечение становится более сложным (подчас чрезмерно сложным).
- Если вся функциональность не документирована, она может так и остаться неизвестной пользователям, но может создать различные риски для безопасности пользовательской системы.
- Добавление новой функциональности может привести к желанию ещё более новой функциональности, приводя к эффекту «снежного кома».