1. Подготовить текс почему я охрененный тестер\QA специалист
   1. Последние 2 проекта (общие черты)?
   2. Какие виды тестировования использовались?
   3. Что именно делал?
   4. Как выглядела команда?
   5. My teamwork with Team Lead (I helped him with **Testing Result reports – или что-то близкое к этому**)

*My penultimate project was associated with the company, which is a leader of logistics services in Ukraine. It was more GUI testing company’s web-site. Now my project is related with Regulatory reporting in big Swiss multinational investment bank.*

*Here I mostly perform front-to-back testing. I book trades in different front office system and check correct flow through all the necessary nodes to the regulator and check the response from it. I also control the correct flow of these trades in Data warehouse and how correctly they display in different report systems, such us SAP Bussiness Object. I do both functional, UI and backend testing, retesting, regression testing.*

* ***Testing methods: static/dynamic testing, the box approach (black, white, grey)***

***• Testing levels: unit, integration, system***

***• Testing types: functional (smoke, sanity, regression, positive, negative, acceptance, exploratory, boundary), non-functional (security, usability, localization, GUI, cross browser, cross platform), automated testing***

*We have a fairly large team - about 30 people, some of them are in India, some in Wroclaw and some in Krakow. The team had 3 Leads and one PM, who controlled the work of the whole team and coordinated the actions of the leads. Every morning we have the Daily Kanban Stand-up, we discuss the progress of the previous day, the blockers that stop us with further work and what each of us plans to do today.*

*In one of the previous companies I had quite a long experience in managing the team, and to be honest – I like it and I'm good at it so now in the process I try to interact as much as possible with our leader, who oversees our part of the team. I help with the actual formation of the table of test results.*

*=== Strenght and weakness ===*

*Я очень хорошо понимаю разницу между недостатками о которых я знаю и недостатками о которых не знаю. С первыми – намного легче. Мы всю жизнь работаем над своими слабыми сторонами, которые мешают нам в жизни и в работе. Гораздо хуже когда есть какие-то недостатки, о которых мы не знаем, т.е. мы теряем очки в глазах других людей и не понимаем почему. Поэтому я очень положительно отношусь к такой ситуации, когда кто-то со стороны говорит мне о моих недостатках. Я считаю это конструктивно. Соглашусь я с этим человеком или нет – это уже второй вопрос, но по крайней мере это конструктивно. Из последних слабых сторон...*

1. Расскажите о своих достижениях на прошлой работе?

- Подробно описать процесс перевода работыт на Джиру (

-перевод их Excel на Jira X-Ray,

? - скрипты,

? - API,

? - консольное приложение на Джаве,

- я работал с требованиями и описывал их QA Autation team + I have actively looking into their code)

* Погуглить как читать **X-Ray Jira API**
* Application reads excel data, call X-Ray plugin for Jira API, so I’ve prepared **REST API** queries in **Postman** to cover all requirement of this application.
  + List of reqiurement was created on **Gherkin language**

!! EXAMPLE NEED !!

1. Management + Planning /**difference between Scrum vs Canbun**/
   1. **Planing**
   2. **Grooming**
   3. **Daily standup**
   4. **Capacity vs Velocity**
2. Working with bugs
   1. How to create bug?
   2. What attributes bug has?
   3. Tracibillity matrix
3. **REST**
   1. What it is?

Пару месяцев назад при беглом изучении вопроса я понял о REST примерно следующее:  
Все является ресурсами с уникальным идентификатором (URL)

1. Все операции клиента с сервером stateless, т.е. сервер не должен хранить вообще никакой информации о клиенте – никакой сессии
2. Все запросы можно поделить на 4 типа в соответствии с [CRUD](http://en.wikipedia.org/wiki/Create,_read,_update_and_delete), причем каждому типу сопоставляется HTTP метод – Post, Get, Put и Delete
3. Вся логика крутится вокруг ресурсов, а не операций

Вот с такими воспоминаниями я начал бороздить просторы интернета. Первой мыслью было, а **почему выбрано название REST?** Representational State Transfer, в переводе википедии «передача состояния представления»… Никакой картинки в голове не вырисовывается даже при четвертом вчитывании. [Здесь](http://www.xfront.com/REST-Web-Services.html) пытаются ответить на мой вопрос и даже приводят то, как Рой Филдинг (человек, сформулировавший принципы REST) сам объяснял происхождение названия. Мысль сводится к тому, что запрос ресурса с сервера переводит клиентское приложение в определенное состояние (state), а запрос следующего ресурса меняет состояние приложения (transfer). А “Representational” означает то, что ресурс возвращается не просто так, а в каком-то представлении, например в представлении для машины или в представлении для человека. Сложно, как по мне, и сбивает с толку, т.к. состояние – это как раз то, что отсутвует в отношениях клиент-сервер в архитектуре REST. Я бы назвал как-то вроде*«Стандартизированное оперирование данными»*, вот только сначала надо что-то придумать, а потом уже яркое название выбирать. А Филдинг в своей диссертации признается, что название придумано не для того, чтобы было понятно, о чем речь, а «is intended to evoke an image of how a well-designed Web application behaves». Но это ничего, не будем обижаться на уважаемого человека, мы тоже в дипломных работах часто формулировали все так, чтобы было как можно непонятнее и нельзя было придраться. [Нашлась](http://habrahabr.ru/blogs/webdev/108993/) и неплохая формулировка идеи по-русски –*«представление данных в удобном для клиента формате»*. Справедливости ради надо отметить, что пока я формулировал свои доводы о нелогичности названия, я увидел в нем некоторую логику, по крайней мере в английском варианте.  
  
Важно понимать, что **REST – это не протокол и не стандарт, а архитектурный стиль**. У этого стиля есть свои принципы. Позволю себе скопировать их с понравившегося [источника](http://www.infoq.com/articles/rest-introduction) и прокомментировать:

1. **Give every “thing” an ID.**  
   Очччень желательно.
2. **Link things together.**  
   Например, в страницу (представление) о Mercedes C218 хорошо бы добавить ссылку на страницу конкретно о двигателе данной модели, чтобы желающие могли сразу туда перейти, а не тратить время на поиск этой самой страницы.
3. **Use standard methods.**  
   Имеется в виду, экономьте свои силы и деньги заказчика, используйте стандартные методы HTTP, например GET

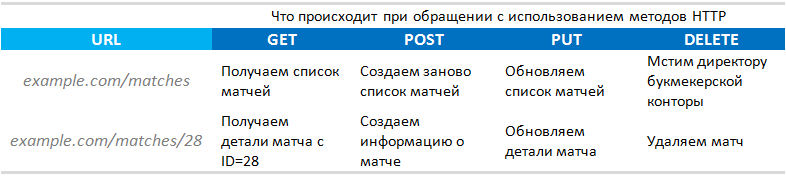
http://www.example.com/cars/00345

для получения данных вместо определения собственных методов вроде getCar?id=00345.

1. **Resources can have multiple representations.**  
   Одни и те же данные можно вернуть в XML или JSON для программной обработки или обернутыми в красивый дизайн для просмотра человеком.
2. **Communicate statelessly.**  
   Да, RESTful сервис должен быть как идеальный суд – его не должно интересовать ни прошлое подсудимого (клиента), ни будущее – он просто выносит приговор (отвечает на запрос).

Только что употребленный термин RESTful (веб-)сервис всего лишь означает сервис, реализованный с использованием принципов REST. **Так что же нам дает следование этим самым принципам REST?**Для начала я бы назвал простоту основным преимуществом архитектуры REST. Простоту идеи, простоту разработки и добавления функциональности к RESTful приложениям. Идея настолько проста и универсальна, что ее даже сложно сначала уловить. Мы не добавляем никакого нового слоя в наш и без того многослойный программерский пирог, а просто используем уже давно признанные стандарты. Поэтому чтобы ответить на вопрос о преимуществах и недостатках и чтобы анализ имел больше смысла, предлагаю перейти к сравнению подходов SOAP и REST.

1. **SOAP – это целое семейство протоколов и стандартов**, откуда напрямую вытекает, что это более тяжеловесный и сложный вариант с точки зрения машинной обработки. Поэтому REST работает быстрее.
2. SOAP используют HTTP как транспортный протокол, в то время как REST базируется на нем. Это означает, что все существующие наработки на базе протокола HTTP, такие как кеширование на уровне сервера, масштабирование, продолжают так же работать в REST архитектуре, а для SOAP необходимо искать другие средства. Взамен этого SOAP сервисы получают такое мифическое свойство, как возможность работать с любым протоколом транспортного уровня вместо HTTP, однако практической пользы от него зачастую не больше, чем сотрудникам [Челябинского трубопрокатного завода](http://ru.wikipedia.org/wiki/%D0%A7%D0%B5%D0%BB%D1%8F%D0%B1%D0%B8%D0%BD%D1%81%D0%BA%D0%B8%D0%B9_%D1%82%D1%80%D1%83%D0%B1%D0%BE%D0%BF%D1%80%D0%BE%D0%BA%D0%B0%D1%82%D0%BD%D1%8B%D0%B9_%D0%B7%D0%B0%D0%B2%D0%BE%D0%B4) от большого количесва статей в википедиях на [мертвых языках](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%80%D1%82%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA).
3. Есть [мнение](http://ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest), что разработка RESTful сервисов намного проще. Наверное, это правда, если использовать Notepad в качестве основной среды разработки, но вот с использованием наших чудесных средств разработки, я позволю себе усомниться в верности этого утверждения.
4. В первом гугловском [результате](http://spf13.com/post/soap-vs-rest) по запросу «REST vs SOAP» акцентируется внимание на том, что ответ REST может быть представлен в различных форматах, а SOAP привязан к XML. Это действительно важный фактор, достаточно представить себе вызов сервиса из javascript, ответ на который мы определенно хотим получать в JSON.
5. **«REST vs SOAP» можно перефразировать в «Простота vs Стандарты»**, что проявляется в том, что для SOAP мы имеем протокол WSDL для исчерпывающего описания веб-сервиса, который с использованием все тех же чудесных средств разработки прото-таки волшебным образом делает почти всю работу за нас. Со стороны REST мы имеем загадочный и неиспользуемый протокол WADL, который, в принципе, и не нужен – он мешает простоте.
6. Второй аспект предыдущего пункта – обработка ошибок. В SOAP она полностью стандартизована, а REST может использовать давно известные коды ошибок HTTP (если здесь Вас посетила мысль, что это же очевидно и зачем я это пишу, то значит Вы внимательно читаете статью).
7. То, с чего можно было бы начать, но я припас напоследок. Это одна из ключевых мыслей. **SOAP работает с операциями, а REST – с ресурсами**. Этот факт в совокупности с отсутствием клиентского состояния у RESTful сервисов приводит нас к тому, что такие вещи как транзакции или другая сложная логика должна реализовываться «SOAP-но».

Приведу пару примеров на понимание разницы между подходами. Букмекерская контора заказала сервис для работы с футбольной статистикой. Пользовательский функционал – получить список матчей, получить детали о матче. Для редакторов – редактировать (Create, Edit, Delete) список матчей, редактировать детали матча. Для такой задачи однозначно надо выбирать подход REST и получать бенефиты от его простоты и естественности во взаимодействии с HTTP. Не нужны нам здесь SOAP-конверты, SOAP-главпочтамты и SOAP-авиапочта, которая может использовать любую марку самолета. Нам всего лишь надо реализовать следующее:  
  
  
  
Все очень просто! Теперь пример посложнее. Та же букмекерская контора захотела API для ставок на live матчи. Эта процедура включает в себя многочисленные проверки, например, продолжает ли ставка быть актуальной, не изменился ли коэффициент, не превышена ли максимальная сумма ставки для маркета. После этого происходит денежная транзакция, результаты которой записываются в основную и в резервные базы данных. Лишь после этого клиенту приходит ответ об успешности операции. Здесь явно прослеживается ориентация на операции, имеются повышенные требования к безопасности и устойчивости приложения, поэтому целесообразно использовать SOAP.   
  
И еще пару задач для того, чтобы почувствовать тему: 

* Футбольный клуб заказывает CMS для подробных сведений об игроках команды-неприятеля. Нужен функционал добавления характеристик игрока простыми пользователями прямо во время матча с последующей интеграцией с табло стадиона, на котором необходимо в реальном времени отображать комментарии.
* Мексиканский наркобарон Педро Гонсалес заказывает API для учета продаж героина в Юго-Западных штатах США. Он особо просит мобильное приложение под эту задачу, т.к. его бизнес часто проходит на открытом воздухе, где нету других вариантов выхода в сеть.
* Анонимный миллиардер очень хочет такую программу, которая бы ему показывала всех его любовниц в городе, в котором он сейчас находится и то, какой текущий статус отношений. Он хочет интегрировать эту программу с уже существующим его личным десктопным приложением для подбора мест для отдыха, он очень хочет большую красную надпись о возможных неприятностях в окошке, где предлагаются варианты авиаперелета.

Какие подходы Вы бы использовали в данных задачах?  
  
Хотел я еще написать про то, что это все дает .NET разработчику и как это использовать в своих целях, однако вижу, что индекс нудности статьи приближается к критическому, поэтому буду закругляться. С целью понижения все того же показателя я намеренно избегал аспектов безопасности и, например, ответа на вопрос ”А как вообще возможна аутентификация в архитектуре REST, если читателю на протяжении всей этой статьи внушалось, что RESTful сервис должен быть stateless?”.  
  
А выводы статьи будут следующими:

1. **Филдинг со своими принципами REST ничего не изобрел**, а просто собрал в одну диссертацию то, что уже существовало в каком-то виде и изложил то, как можно получать максимальную выгоду из уже сформировавшейся архитектуры сети.
2. **SOAP и REST – не конкуренты**. Они представляют разные весовые категории и вряд ли найдется задача, для которой будет сложно сказать, какой подход рациональнее использовать – SOAP или REST. Поэтому «религиозные» убеждения в вопросах выбора архитектуры для веб-сервиса вряд ли будут полезны. Для тех, кто не знает, с чего начать анализ задачи, могу порекомендовать [эту](http://www.slideshare.net/cesare.pautasso/rest-vs-soap-making-the-right-architectural-decision-1st-international-soa-symposium-amsterdam-october-2008-presentation)презентацию. У них чаще побеждает REST.
   1. **Difference REST vs SOAP** (с SOAP не работал, но читал про него и знаю....)

SOAP and REST can't be compared directly, since the first is a protocol (or at least tries to be) and the second is an architectural style. This is probably one of the sources of confusion around it, since people tend to call REST any HTTP API that isn't SOAP.

Pushing things a little and trying to establish a comparison, the main difference between SOAP and REST is the degree of coupling between client and server implementations. A SOAP client works like a custom desktop application, tightly coupled to the server. There's a rigid contract between client and server, and everything is expected to break if either side changes anything. You need constant updates following any change, but it's easier to ascertain if the contract is being followed.

A REST client is more like a browser. It's a generic client that knows how to use a protocol and standardized methods, and an application has to fit inside that. You don't violate the protocol standards by creating extra methods, you leverage on the standard methods and create the actions with them on your media type. If done right, there's less coupling, and changes can be dealt with more gracefully. A client is supposed to enter a REST service with zero knowledge of the API, except for the entry point and the media type. In SOAP, the client needs previous knowledge on everything it will be using, or it won't even begin the interaction. Additionally, a REST client can be extended by code-on-demand supplied by the server itself, the classical example being JavaScript code used to drive the interaction with another service on the client-side.

I think these are the crucial points to understand what REST is about, and how it differs from SOAP:

* REST is protocol independent. It's not coupled to HTTP. Pretty much like you can follow an ftp link on a website, a REST application can use any protocol for which there is a standardized URI scheme.
* REST is not a mapping of CRUD to HTTP methods. Read [this](https://stackoverflow.com/questions/19843480/s3-rest-api-and-post-method/19844272#19844272) answer for a detailed explanation on that.
* REST is as standardized as the parts you're using. Security and authentication in HTTP are standardized, so that's what you use when doing REST over HTTP.
* REST is not REST without [hypermedia](https://stackoverflow.com/a/29586455/1202421) and [HATEOAS](http://en.wikipedia.org/wiki/HATEOAS). This means that a client only knows the entry point URI and the resources are supposed to return links the client should follow. Those fancy documentation generators that give URI patterns for everything you can do in a REST API miss the point completely. They are not only documenting something that's supposed to be following the standard, but when you do that, you're coupling the client to one particular moment in the evolution of the API, and any changes on the API have to be documented and applied, or it will break.
* REST is the architectural style of the web itself. When you enter Stack Overflow, you know what a User, a Question and an Answer are, you know the media types, and the website provides you with the links to them. A REST API has to do the same. If we designed the web the way people think REST should be done, instead of having a home page with links to Questions and Answers, we'd have a static documentation explaining that in order to view a question, you have to take the URI stackoverflow.com/questions/<id>, replace id with the Question.id and paste that on your browser. That's nonsense, but that's what many people think REST is.

This last point can't be emphasized enough. If your clients are building URIs from templates in documentation and not getting links in the resource representations, that's not REST. Roy Fielding, the author of REST, made it clear on this blog post: [REST APIs must be hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven).

With the above in mind, you'll realize that while REST might not be restricted to XML, to do it correctly with any other format you'll have to design and standardize some format for your links. Hyperlinks are standard in XML, but not in JSON. There are draft standards for JSON, like [HAL](http://stateless.co/hal_specification.html).

Finally, REST isn't for everyone, and a proof of that is how most people solve their problems very well with the HTTP APIs they mistakenly called REST and never venture beyond that. REST is hard to do sometimes, especially in the beginning, but it pays over time with easier evolution on the server side, and client's resilience to changes. If you need something done quickly and easily, don't bother about getting REST right. It's probably not what you're looking for. If you need something that will have to stay online for years or even decades, then REST is for you.

* + SOAP активно использует XML для кодирования запросов и ответов, а также строгую типизацию данных, гарантирующую их целостность при передаче между клиентом и сервером. С другой стороны, запросы и ответы в REST могут передаваться в ASCII, XML, JSON или любых других форматах, распознаваемых одновременно и клиентом, и сервером. Кроме того, в модели REST отсутствуют встроенные требования к типизации данных. В результате пакеты запросов и ответов в REST имеют намного меньшие размеры, чем соответствующие им пакеты SOAP.
  + В модели SOAP уровень передачи данных протокола HTTP является «пассивным наблюдателем», и его роль ограничивается передачей запросов SOAP от клиента серверу с использованием метода POST. Детали сервисного запроса, такие как имя удаленной процедуры и входные аргументы, кодируются в теле запроса. Архитектура REST, напротив, рассматривает уровень передачи данных HTTP как активного участника взаимодействия, используя существующие методы НТТР, такие как GET, POST, PUT и DELETE, для обозначения типа запрашиваемого сервиса. Следовательно, с точки зрения разработчика, запросы REST в общем случае более просты для формулирования и понимания, так как они используют существующие и хорошо понятные интерфейсы HTTP.
  + Модель SOAP поддерживает определенную степень интроспекции, позволяя разработчикам сервиса описывать его API в файле формата Web Service Description Language (WSDL, язык описания веб-сервисов). Создавать эти файлы довольно сложно, однако это стоит затраченных усилий, поскольку клиенты SOAP могут автоматически получать из этих файлов подробную информацию об именах и сигнатурах методов, типах входных и выходных данных и возвращаемых значениях. С другой стороны, модель REST избегает сложностей WSDL в угоду более интуитивному интерфейсу, основанному на стандартных методах HTTP, описанных выше.
  + В основе REST лежит концепция ресурсов, в то время как SOAP использует интерфейсы, основанные на объектах и методах. Интерфейс SOAP может содержать практически неограниченное количество методов; интерфейс REST, напротив, ограничен четырьмя возможными операциями, соответствующими четырем методам HTTP.
* **Rest queries**
* Exaple where POST in scenario and where use PUT

1. SQL (I was not doing DB administration activities, only queries)
   * What type of DB (Oracle)
   * **Difference MySQL vs Oracle**
   * DISTINCT
   * Different types of JOIN
2. Other
   * OFF LIMIT PAGINATION
   * Pagination in Oracle, in SQL (na Stackoverflow)
   * How to check is paginatoin working?
     + !!!! найти и написать !!!!

TASK:

Есть список промокодов...

Вопросы

* Какие требования?
* Как аппликация отвечает на правильный\неправильный промокод?
* Как аппликация должна реагировать на дубликат промо-кода?
* Как UI реагирует на промокод?
* Validation on UI or on Backend?
* Требования к первофмансу
  + - Пример: Акция на 15 мин – аппликация работала, но почтовый сервер перестал отсылать подтверждения и ...
* !!! Придумать тесты 🡪 **DBB Scenario outline, DBB Scenario with parameters**
  + Secutiry in format XSS attack
  + Performance
  + UI test
  + BackEnd tesing
  + UI+BackEnd

**1. Java**

**Разница между абстрактным классом и интерфейсом**

**Ver 1.**

***Абстрактный класс*** *— это класс, у которого не реализован один или больше методов (некоторые языки требуют такие методы помечать специальными ключевыми словами).*

***Интерфейс*** *— это абстрактный класс, у которого ни один метод не реализован, все они публичные и нет переменных класса.*

*Интерфейс нужен обычно когда описывается только интерфейс (тавтология). Например, один класс хочет дать другому возможность доступа к некоторым своим методам, но не хочет себя «раскрывать». Поэтому он просто реализует интерфейс.*

*Абстрактный класс нужен, когда нужно семейство классов, у которых есть много общего. Конечно, можно применить и интерфейс, но тогда нужно будет писать много идентичного кода.*

**Ver 2.**

***Абстрактный класс*** *— это «заготовка» класса: реализовано большинство методов (включая внутренние), кроме нескольких. Эти несколько нереализованных методов вполне могут быть внутренними методами класса, они лишь уточняют детали имплементации. Абстрактный класс — средство для повторного использования кода, средство, чтобы указать, какой метод обязан быть перекрыт для завершения написания класса.*

***Интерфейс*** *же — это своего рода контракт: интерфейсы используются в определениях чтобы указать, что объект, который будет использован на самом деле, должен реализовывать (для входных параметров) или будет гарантированно реализовывать (для выходных параметров) набор методов и (что намного важнее!) иметь определённую семантику. Интерфейс вполне может быть и пустым, тем не менее, имплементировать интерфейс означает поддерживать данную семантику.*

**Ver 3.**

***Абстрактный(Abstract) класс*** *- класс, который имеет хотя б 1 абстрактный (не определенный) метод; обозначается как abstract.*

***Интерфейс*** *- такой же абстрактный класс,только в нем не может быть свойств и не определены тела у методов.*

*Так же стоит заметить, что абстрактный класс наследуется(etxends), а интерфейс реализуется (implements). Вот и возникает разница между ними, что наследовать мы можем только 1 класс, а реализовать сколько угодно.*

*ВАЖНО! При реализации интерфейса, необходимо реализовать все его методы, иначе будет Fatal error, так же это можно избежать, присвоив слово abstract.*

Пример:

interface I {

public function F();

public function say();

}

abstract class A implements I {

function say() {

echo 'Hello';

}

// function F() - не реализована

}

**Ver 4.**

***В абстрактных классах*** *можно использовать все, что и в классах, + можно добавлять abstract-методы (без реализации) и нельзя создать экземпляр абстрактного класса.*

***В интерфейсах*** *нельзя использовать поля (с java8, если не ошибаюсь, есть возможность определять поля, но они будут статическими) и определять поведение функций по умолчанию (с java8 появились default-методы).*

*Что касается выбора одного или другого:*

*Выбирайте* ***интерфейс****, когда вы просто дергаете одинаковые методы на совершенно разных объектах. Как пример - машина и самолёт. У них есть метод move(), который описан в соответствующем интерфейсе. И вам совершенно не нужно знать, что это за объект. Вы видите его как Movable и лишь двигаете.*

*Выбирайте абстрактный класс тогда, когда у объектов много общего, и есть такие методы, реализация которых идентична. Это могут быть легковая машина и грузовик. Метод rotateWheels() может иметь одну и ту же реализацию, и в целом в классах будут лишь малые различия. Тогда выбирайте абстрактный класс Car, в котором закладываете базовую реализацию, и дальше наследуетесь.*

*An interface is a* ***contract****: The person writing the interface says, "hey, I accept things looking that way", and the person using the interface says "OK, the class I write looks that way".*

*An interface is* ***an empty shell****. There are only the signatures of the methods, which implies that the methods do not have a body. The interface can't do anything. It's just a pattern.*

*For example (pseudo code):*

// I say all motor vehicles should look like this:

interface MotorVehicle

{

void run();

int getFuel();

}

// My team mate complies and writes vehicle looking that way

class Car implements MotorVehicle

{

int fuel;

void run()

{

print("Wrroooooooom");

}

int getFuel()

{

return this.fuel;

}

}

*Implementing an interface consumes very little CPU, because it's not a class, just a bunch of names, and therefore there isn't any expensive look-up to do. It's great when it matters, such as in embedded devices.*

***Abstract classes****, unlike interfaces, are classes. They are more expensive to use, because there is a look-up to do when you inherit from them.*

***Abstract classes*** *look a lot like interfaces, but they have something more: You can define a behavior for them. It's more about a person saying, "these classes should look like that, and they have that in common, so fill in the blanks!".*

*For example:*

// I say all motor vehicles should look like this:

abstract class MotorVehicle

{

int fuel;

// They ALL have fuel, so lets implement this for everybody.

int getFuel()

{

return this.fuel;

}

// That can be very different, force them to provide their

// own implementation.

abstract void run();

}

// My teammate complies and writes vehicle looking that way

class Car extends MotorVehicle

{

void run()

{

print("Wrroooooooom");

}

}

**базовое понятие generic**

**Принципы ООО**

**Collections, разница между set и list, hashmap и linkedlisthashmap**

**Методы сортировки**

**Как развернуть массив?**

**Убрать из массива дубликаты**

**2. TestNG**

**Аннотации.**

**Разница с Junit**

**Как запустить тесты в параллеле**

**3. Передача данных, взаимодействие с сервером**

**Rest Api - что такое, какие методы есть. Какие библиотеки, какие тулы.**

* **json vs xml. Библиотеки работы с json и xml в java**

**Как сделать с java запрос к ssh cерверу**

**Какие библиотеки для работы с csv, xls файлами**

**Как работает клиент-сервер**

**Интернет протоколы**

**Базовая WEB-Archicture**

**4. Design паттерны**

**какие использовал и для чего в автотестировании /выбрать и разобрать самые популярные/**

**5. SQL**

**Какие базы данных использовал?**

**Как работал с базой из java**

**Разница GROUP BY и ODRER BY**

**JOIN**

**Primary&Secondary key**

**6. Selenium**

**exceptions, как смоделировать иксепшен - описание поведение**

**Whai is the difference between Relatives and absolute Xpath?**

**How to traverse to sibling element using Xpath?**

**How to traverse back to Parens element from Child element using Xpath?**

**How to identify element Text based?**

**waiters - виды, какой лучше.**

**Класс Actions, как сделать drag and drop**

**Как пролистать ленту новостей на фейсбуке**

**Как сделать upload и download файла.**

**FindElementBy….**

**CSS /+ что такое классы в o CSS/**

**HTML**

**7. Базовые команды git /что такое Master Branch?/**

**8. Базовые команды linux**

* **GRAB**
* **PIPER**
  + **Как сделать файл execution (CHMOD)**
  + **Навигация**
  + **Удалить/создать**
  + **Права файла – что обозначают цифры**