**Тестовая стратегия** определяет типы тестов, которые нужно выполнять для данного функционала системы, в каком окружении, время на реализацию тестов, при каких условиях билд можно считать рабочим, польза продукта перекрывает дефект или нет?

**Тест-план** - главный документ, которым руководствуется QA инженер. В идеале тест-план должен описывать:

объект тестирования (тестируемый продукт)

* цели тестирования и скоупы работ
* стратегию тестирования (как будет производится тестирование, какие инструменты будут применяться)
* приоритеты проекта и роль тестирования в процессе
* процесс тестирования (когда можно начинать тестирование, как следует интерпретировать результаты и т.д.)
* тестовую среду
* собственно тестовые сценарии (test cases)

**Верификация и валидация**.

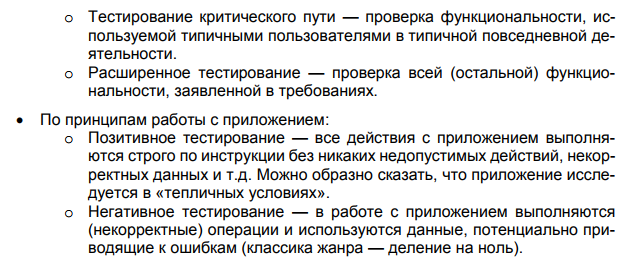
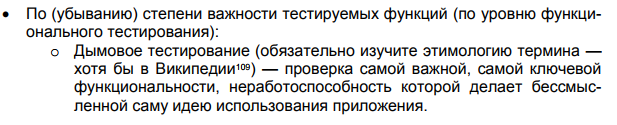
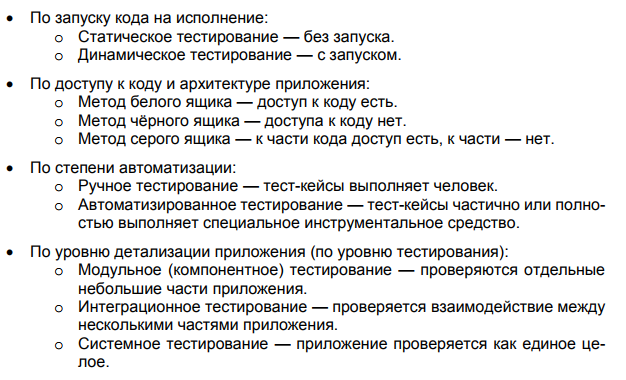
Верификация - подтверждение того, что определенные требования были выполнены.  
Валидация - проверка того, насколько программный продукт отвечает ожиданиям и потребностям пользователей.

**Требование (requirement) —** описание того, какие функции и с соблюдением каких условий должно выполнять приложение в процессе решения полезной для пользователя задачи.

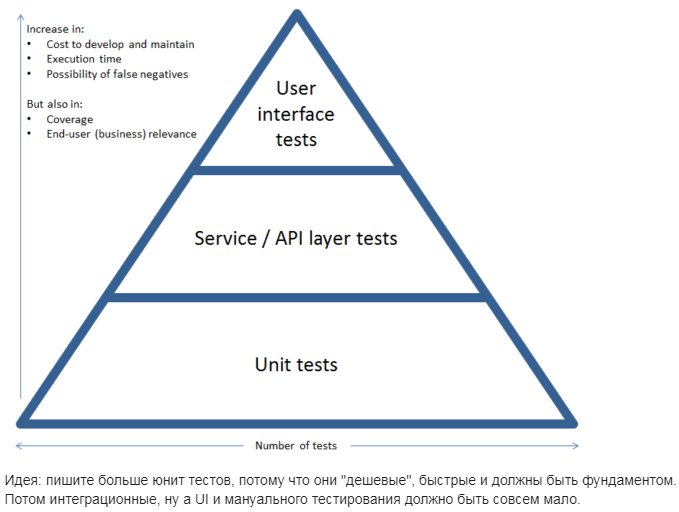
**3 уровня требований:**  
1. Бизнес-требования - то, что хотят видеть о продукта  
2.Требования пользователя - то, как можно пользоваться продуктом  
3.Функциональные требования - то, как ведет себя продукт при его эксплуатации  
существуют нефункциональные требования и системные требования.

**Какими должны быть требования?**

Завершенными, выполнимыми, атомарными, актуальными, корректными, проверяемыми, прослеживаемыми, недвусмысленными, непротиворечивыми, модифицируемыми, проранжированы (важность, стабильность, срочность)



**Регрессионное** тестирование (regression testing) — тестирование, направленное на проверку того факта, что в ранее работоспособной функциональности не появились ошибки, вызванные изменениями в приложении или среде его функционирования.



**АТ -** это набор техник, подходов и инструментальных средств, позволяющий исключить человека из выполнения некоторых задач в процессе тестирования.

**Задачи автоматизированного тестирования**.

* Выполнение тест-кейсов, непосильных человеку.
* Решение рутинных задач.
* Ускорение выполнения тестирования.
* Высвобождение человеческих ресурсов для интеллектуальной работы.
* Увеличение тестового покрытия.
* Улучшение кода за счёт увеличения тестового покрытия и применения специальных техник автоматизации.

**Compatibility testing (тестирование совместимости) -** Это тестирование на совместимость с объявленным в спецификации оборудованием, операционными системами и программными продуктами третьих фирм.

**Нагрузочное** тестирование (load testing , capacity testing) — исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором превышении

**Класс эквивалентности** (Equivalence Classes) - это набор входных (иногда и выходных) данных, которые обрабатываются приложением одинаково или обработка которых приводит к одному и тому же результату.

**Fuzz Testing** - метод тестирования программного обеспечения, часто автоматизированый или полуавтоматизирований, который включает в себя недопустимые, неожиданные или случайные данные на вход компьютерной программы. Фаззинг обычно используется для проверки проблем безопасности в программных х системма.

**Тест-кейс** — набор входных данных, условий выполнения и ожидаемых результатов, разработанный с целью проверки того или иного свойства или поведения программного средства.

ID - у каждого тест-кейса есть свой уникальный номер(название), Кто создал - тестировщик 1 , Когда создан – дата, Название тест-кейса - краткое описание сути, что тестируем, Предусловия начала тестирования конкретной области ПО, Шаги тест-плана - описание ваших действий, Ожидаемый результат

Баг репорт содержит  
● Заголовок (Summary), ● Описание (Description), ● Номер сборки (build), ● Environment (где был найден баг)  
● Severity, ● Priority, ● Кем дефект занесён в систему (это учитывается автоматически), ● На кого назначен (assigned) дефект

В общем случае приемочным тестированием называют тестирование, целью которого является определение пригодности продукта. Обычно производится с помощью набора тестов (автоматически или вручную), которые заранее подготовлены и ожидают определенного результата.

В большинстве случаев приемочное тестирование производится с целью проверить соответствие продукта функциональным требованиям.Основная задача гарантировать, что заказчик получит продукт, обладающий всеми необходимыми функциональными свойствами.

Синонимы: release acceptance, final testing, validation testing

Граничное тестирование применяется для проверки поведения продукта на крайних значениях входных данных, таких как, например максимумы и минимумы значений. Набор тестов для стресс-тестирования может включать граничные тесты.

Граничное тестирование может также включать тесты, проверяющие поведение системы на входных данных, выходящих за допустимый диапазон значений. При этом система должна определенным (заранее оговоренным) способом обрабатывать такие ситуации. Например, с помощью исключительной ситуации или сообщения об ошибке.

**CI –** инженерная практика, которая подразумевает частую интеграцию моделей системы воедино. Jenkins.

**JIRA** – инструмент для управления agile-проектами. Atlassian. **Scrum** – agile-методика, в которой создание продукта происходит в виде серий итераций фиксированной продолжительности (спринт). Scrum-board – для визуализации всех этапов работы в рамках спринта. Спринт состоит из планирования, выполнения, релиза, ретроспективы.

GIT. **Git fetch -** Для получения данных из удалённых проектов на локальный репозиторий. Git merge – выполняет сливание.

**git chery-pick** - ты забираешь комиты из одной ветки в другую, это бывает полезно когда изменения сделанные другим разработчиком в его ветке, прямо сейчас нужны тебе в твоей ветке, и что бы не писать этот код заново, ты забираешь его комит себе в ветку  
  
**git rebase master** - ты синхронизируешься с главной веткой в которую коммитят все разработчики проекта, это полезно когда кто-то изменил участок кода с которым ты сейчас работаешь в своей ветке, дабы через неделю ты смог без проблем смержиться с master веткой. Обычно делается каждое утро перед началом рабочего дня и в конце когда фича готова.  
  
**git merge** - обычно используется когда у вас 2 и более master ветки (к примеру master и prototype) в этих ветках очень много комитов (и rebase здесь не подходит) и обчно через пару недель, maintainer репозитория наработки из prototype ветки "сливает" в master ветку по средствам этого самого git merge.

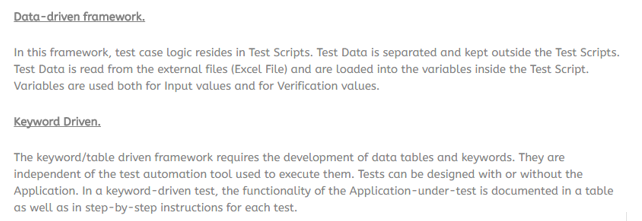
Git reset – переводит текущую ветку на другой коммит

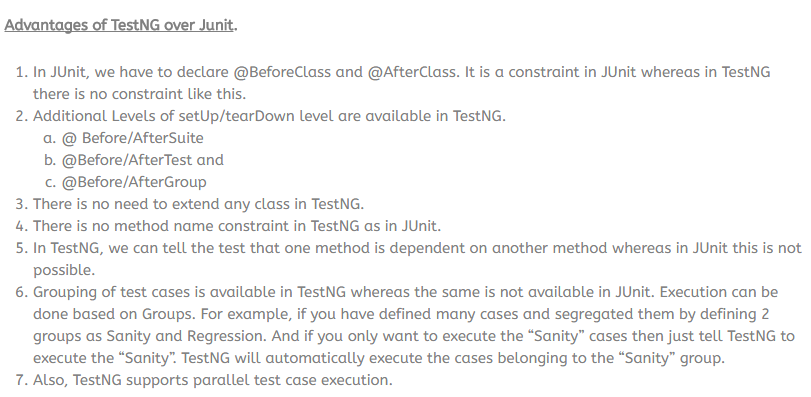
Git merge - В разных ветках могут храниться разные изменения файлов. Чтобы все привести к одному виду, нужно слить ветки (merge branches).

rebase делает так, что меняется вся ветка. Коммиты одной ветки переносятся в другую так, чтобы вторая ветка стала частью первой

**Test-driven development** (разработка под управлением тестированием) - подход к разработке и тестированию, при котором сначала создаются тесты, которым должен удовлетворять код и только потом его реализация. A way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases. Общая идея TDD выражается принципом Red-Green-Refactor.

**BDD (или behavior-driven development)** — расширение подхода TDD к разработке и тестированию, при котором особое внимание уделяется поведению системы/модуля в терминах бизнеса(заказчика). Как правило, такие тесты иллюстрируют и тестируют различные сценарии, которые интересны непосредственно клиенту системы. В связи с этим при составлении таких тестов часто используется фреймворки, обладающие синтаксисом, обеспечивающим читаемость тестов не только программистом, но и представителями заказчика. Behavior-driven development (BDD) is a software development methodology in which an application is specified and designed by describing how its behavior should appear to an outside observer.





**Node** или **Node.js** — программная платформа, основанная на движке **V8** (транслирующем **JavaScript** в машинный код), превращающая **JavaScript** из узкоспециализированного языка в язык общего назначения.

**Node.js** добавляет возможность **JavaScript** взаимодействовать с устройствами ввода-вывода через свой API (написанный на C++), подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из **JavaScript**-кода. **Node.js** применяется преимущественно на сервере. В основе **Node.js** лежит событийно-ориентированное и асинхронное (или реактивное) программирование с неблокирующим вводом/выводом.

Node.js® is a JavaScript runtime built on [Chrome's V8 JavaScript engine](https://developers.google.com/v8/). Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

NPM is the package manager used by Node.js applications.

The best way to manage locally installed npm packages is to create a package.json file.

A package.json file affords you a lot of great things:

* It serves as documentation for what packages your project depends on.
* It allows you to specify the versions of a package that your project can use using semantic versioning rules.
* Makes your build reproducible which means that its way easier to share with other developers.

As a bare minimum, a package.json must have:

**"name" -** all lowercase , one word, no spaces, dashes and underscores allowed

**"version"** in the form of x.x.x, follows semver spec.

Development dependencies are called development dependencies for a reason - you don't have to install them in production. It makes your deployment artifacts smaller and more secure, as you will have less modules in production which can have security problems.

*Given a version number MAJOR.MINOR.PATCH, increment the****MAJOR****version when you make incompatible API changes,****MINOR****version when you add functionality in a backwards-compatible manner, and****PATCH****version when you make backwards-compatible bug fixes.*

Instead of specifying in package.json the exact version to be installed, npm allows you to widen the range of accepted versions. You can allow newer patch level version with tilde (~) and newer minor or patch level version with caret (^). The default when using --save is to use caret (^).

start: pretty straightforward, it just describes the starting point of your application, it can be invoked with npm start

test: the purpose of this one is to run your tests - one gotcha here is that in this case mocha doesn't need to be installed globally, as npm will look for it in the node\_modules/.bin folder, and mocha will be placed there as well. It can be invoked with: npm test.

your-custom-script: anything that you want, you can pick any name. It can be invoked with npm run your-custom-script - don't forget the run part!

While npm2 installs all dependencies in a nested way, npm3 tries to mitigate the deep trees and redundancy that such nesting causes.

Promise (обычно их так и называют «промисы») – предоставляют удобный способ организации асинхронного кода.

Promise – это специальный объект, который содержит своё состояние. Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).

Способ использования, в общих чертах, такой:

Код, которому надо сделать что-то асинхронно, создаёт объект promise и возвращает его.

Внешний код, получив promise, навешивает на него обработчики.

По завершении процесса асинхронный код переводит promise в состояние fulfilled (с результатом) или rejected (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде.

A promise represents the eventual result of an asynchronous operation. It is a placeholder into which the successful result value or reason for failure will materialize.

A promise can be in one of 3 states:

Pending - the promise’s outcome hasn’t yet been determined, because the asynchronous operation that will produce its result hasn’t completed yet.

Fulfilled - the asynchronous operation has completed, and the promise has a value.

Rejected - the asynchronous operation failed, and the promise will never be fulfilled. In the rejected state, a promise has a *reason* that indicates why the operation failed.

**Синтаксис создания Promise:**

var promise = new Promise(function(resolve, reject) {

// Эта функция будет вызвана автоматически

// В ней можно делать любые асинхронные операции,

// А когда они завершатся — нужно вызвать одно из:

// resolve(результат) при успешном выполнении

// reject(ошибка) при ошибке

})

Универсальный метод для навешивания обработчиков:

promise.then(onFulfilled, onRejected)

Promise.all() takes an array of promises as an argument and creates a promise that is resolved when all of promises have successfully been resolved.

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application. http, url, fs. In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below. var module = require('module\_name');

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The module.exports is a special object which is included in every JS file in the Node.js application by default. Use module.exports or exports to expose a function, object or variable as a module in Node.js.

РО – шаблон проектирования, который широко используется в автоматизированном тестрования. Позволяет разделять логику выполнения тестов от их реализации. Моделирует страницы тестируемого приложения в качетсве объектов в коде, мы получаем отдельные классы работы со страницами. В случае изменения пользовательского интерфейса можно выполнить изменения только в одном месте. Инкапсулирует работу с отдельными элементами.

WebDriver – библиотека для управления браузером.

Selenium – серия программных продуктов с open source.

RC – для удаленного управления deprecated

IDE – плагин к браузеру firefox, который может записывать действия и воспроизводить

Server + GRID – server позволяет управлять браузером с удаленной машины, grid – кластер состоящий из нескольких selenium servers.

Различия между RC и Webdriver: архитектура(WD напрямую посылает команды в браузер, используя нативный интерфейс, RC – использует прослойку), скорость, реалистичное поведение пользователя, api проще WD.

WD можно разделить на 3 части

* Клиентские библиотеки – для различных языков программирования, которые реализуют доступ к WD
* WD API
* Драйвера браузера

Процесс работы: автоматизатор создает тест, используя высокоуровневый язык программирования. Далее api WD принимает команды от клиентской библиотеки, интерпретирует их и отсылает драйверу браузера. Браузер выполняет манипуляции, описанные в сценарии.

+

Бесплатный, open source, поддерживает все ЯП, большой количество ОС и браузеров, можно итегрировать с CI tools build tools. Быстрее выполняется, не требует для тестов сервера.

-

Для того чтобы начать писать на WD нужны знания. Нет встроенной поддержки для генерации репортов, поддержки для загрузки файлов, создание тестов занимает время, нет поддержки для работы с диалоговыми окнами.

Как WD работает внутри

Есть API WD (команды в тесте) –> потом команды WD преобразовываеются в команды WD CPI (процедурный код), чтобы передавать дальше информацию в браузер, используется json wire protocol (механизм). Возвращается респонз от браузера о выполнении команды. JSON maps CPI commands to RESTish HTTP resources.

Для JS все операции полностью асинхронны и результатом выполнения команды является промис (объект, который содержит свое состояние: pending fullfield rejected).

WD API делится на несколько групп:

- для работы с окном браузера (close, quit, get, navigate, sleep).

- для работы с элементами (clear, click, getText, isSelected).

- методы для работы с pop-up, alert

- отвечает за синхронизацию (wait)

Класс By описывает механизм нахождения элемента на страничке.

Function createDriver(){…} создает объект драйвера

Protractor is an end-to-end test framework for Angular and AngularJS applications. Protractor runs tests against your application running in a real browser, interacting with it as a user would.

Протрактор – js тестовый фреймворк , построенный на основе webdriverjs. Позволяет выполнять тесты в реальном браузере. Задача – обеспечить синхронизацию между тестовыми скриптами и приложением, написанным на ангуляр. Протрактор включает в себя фреймворк дл запусков тестов, интеграцию с популярными тестовыми фреймворками (jasmine cucumber), поддерживает обработку промисов, дополнительный апи с работой элементами, обеспечивает синхронизацию с ангуляром.

**Waiting for Page to Load -** When navigating to a new page using **browser.get**, Protractor waits for the page to be loaded and the new URL to appear before continuing.   
**Waiting for Angular** - Protractor only works with Angular applications, so it waits for the **angular** variable to be present when it is loading a new page.   
**Waiting for Page Synchronization** - Before performing any action, Protractor asks Angular to wait until the page is synchronized. This means that all timeouts and http requests are finished.

Прострактор использует nodejs. Protractor needs two files to run, the test or spec file, and the configuration file. Protractor tests are written using the syntax of your test framework, for example Jasmine, and the Protractor API. The configuration file tells Protractor how to set up the Selenium Server, which tests to run, how to set up the browsers, and which test framework to use. The configuration file can also include one or more global settings. The Protractor install includes the following: protractor command line tool, webdriver-manager command line tool, Protractor API (library).

Слои протрактора: test runner (command line interface чтобы запускать тесты), configuration (сет опций в conf.js: где находится environment, который мы будем тестировать, какой селениум используем. в каких браузерах), synchronization (синхронизация загрузки страницы, загрузки ангуляра, ожидание таймаутов), api, controlflow. control flow is the order function calls, instructions, and statements are executed or evaluated when a program is running (механизм, который позволяет писать синхронный код и избавляет от использования промис-ченниг).

API протрактора можно разделить на 4 основных элемента:

**browser** - A wrapper around an instance of WebDriver, used for navigation and page-wide information.

браузер, который представляет собой обверту над вебдрайвером, он может делать все тоже самое что и вебдрайвер и предназначен для управления страницей.

**element** - A helper for finding and interacting with DOM elements on the page you are testing that provides two classes:

**ElementFinder** - a wrapper around webdriver.WebElement, you may perform actions (i.e. click, getText) on them as you would a WebElement. Unlike a WebElement, an ElementFinder will wait for angular to settle before performing findings or actions. **ElementArrayFinder** - is used for operations on an **array of** **ElementFinder** (as opposed to a single element).

Элемент – хелпер для нахождения дом-элементов.

Класс by (model, buttontext, options, repeater, binding).

Protractor – предоставляет методы работы с клавиатурой, для ожиданий загрузки элементов.



Specs - какие тесты мы хотим запускать. Capability – браузер, где мы будем ранить тесты. onPrepare – преднастройки, при старте браузера будет выполняться.

*browser*.ignoreSynchronization = **true**;

+ поддержка, встроенные паттерны, конфиг файлы, поддерживает тестовые фреймворки, промисы, интеграцию с CI.

-иногда тесты могут быть медленными (от окружения зависит).

**DOM** (Document Object Model) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document. ДОМ – интерфейс, позволяет получить доступ к элементам HTML. Структура древовидная.

Локаторы.

Ошибки: Использование абсолютного пути, длинные локаторы, игнорирование фреймов.

Jasmine is a behavior-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests.

Особенности: Test suits (describe, it, easy to manage, nested siuts), timeouts (в конфиг файле Default timeout interval, работает для всех тестов), expectation (не надо добавлять сторонние библиотеки для экспектейшн), setup and tear down(before/after), disabling suits (xit – выключает тест, fit -–только тот тест, который нужен), async support (промисы в before/after).

Хуки в describe (beforeEach, afterEach). Хуки родительского дескрайба будут работать в дочернем.

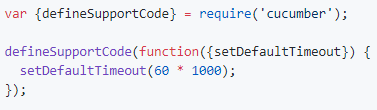
CUCUMBER – заточен под js. Cucumber JS is a JavaScript port of the popular BDD tool Cucumber. It allows you to define Feature Specs in a **Domain Specific Language** (DSL) - called **Gherkin** - and run your specs using a command line tool which will report the passing and/or failing of scenarios and the steps they are comprised of. Gherkin is the language that Cucumber understands. It is a [Business Readable, Domain Specific Language](http://martinfowler.com/bliki/BusinessReadableDSL.html) that provides you describe software’s behaviour without detailing how that behaviour is implemented. It serves two purposes - Documentation and Automated tests. The third is a bonus — when it yells in red it’s talking to you, telling you what code you should write.

Тест делится на фичу (часть которая нуждается в тестировании) и сценарий(шаги для тестирования). Бэкграунд (общие шаги перед каждым сценарием).

Given - When Then

Scenario Outline: examples с параметрами (таблица со значеняими). Data tables (чтобы данные не прописывать в шагах).

Hooks are used for setup and teardown the environment before and after each scenario. (before/after, beforeAll/afterAll). By default, asynchronous hooks and steps timeout after 5000 milliseconds. This can be modified globally with:

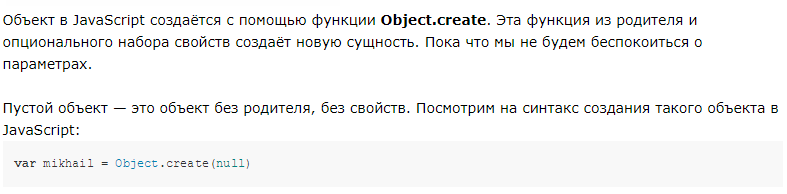


World -

Step\_definitions – шаги.

Tags – для структуризации фичи файлы по логическим параметрам, роль наследования.

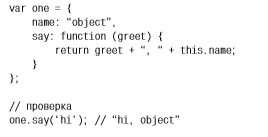
Одна из таких особенностей — это реализация прототипного наследования. Объект в JavaScript — это просто коллекция пар ключ-значение.

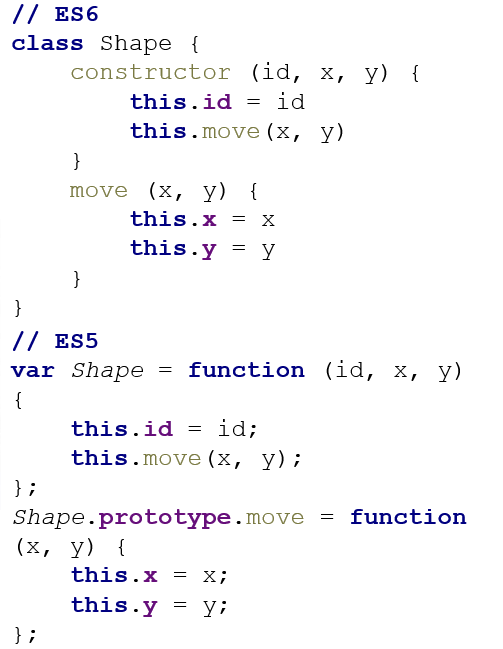


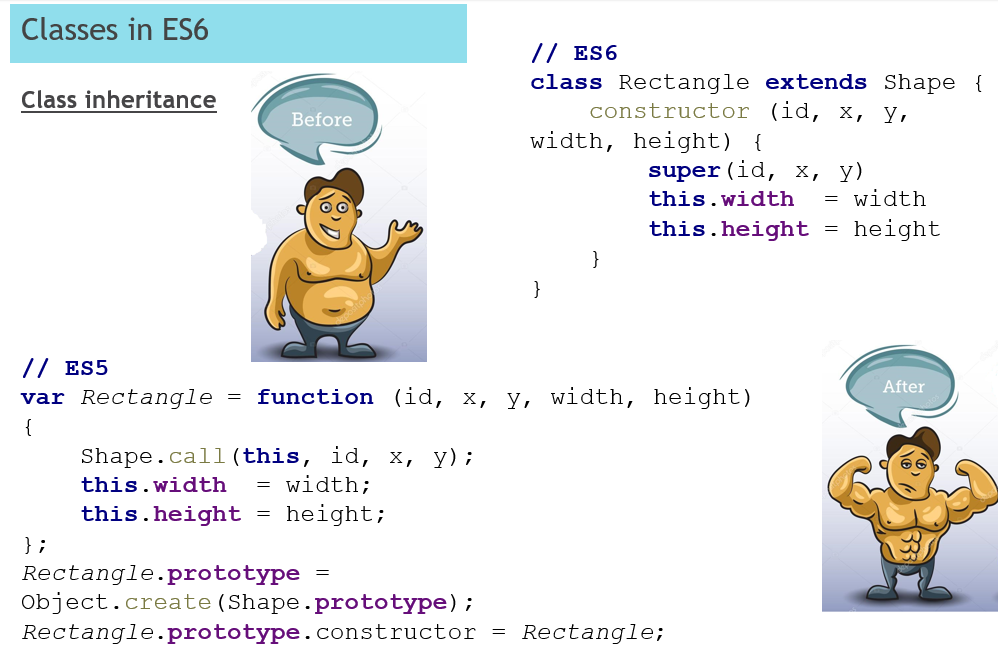
Свойства в JavaScript являются динамическими. Это означает, что мы их можем создавать или удалять в любое время. Свойства уникальны в том смысле, что ключ свойства внутри объекта соответствует ровно одному значению.

Создадим новые свойства через функцию **Object.defineProperty**, которая в качестве аргументов использует объект, имя свойства для создания и дескриптор, описывающий семантику свойства.

Пример объекта:

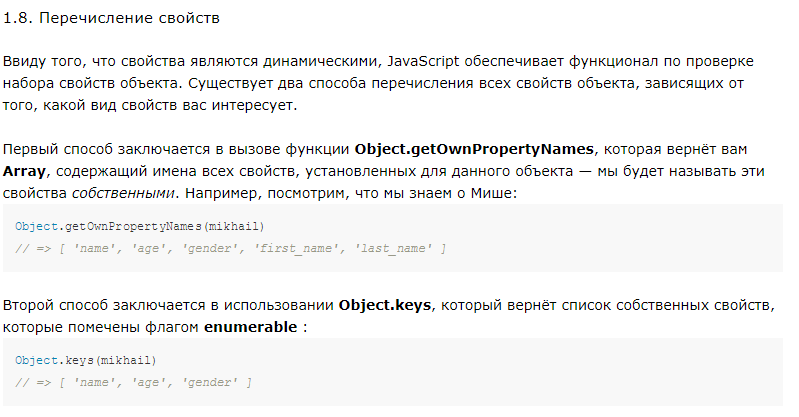
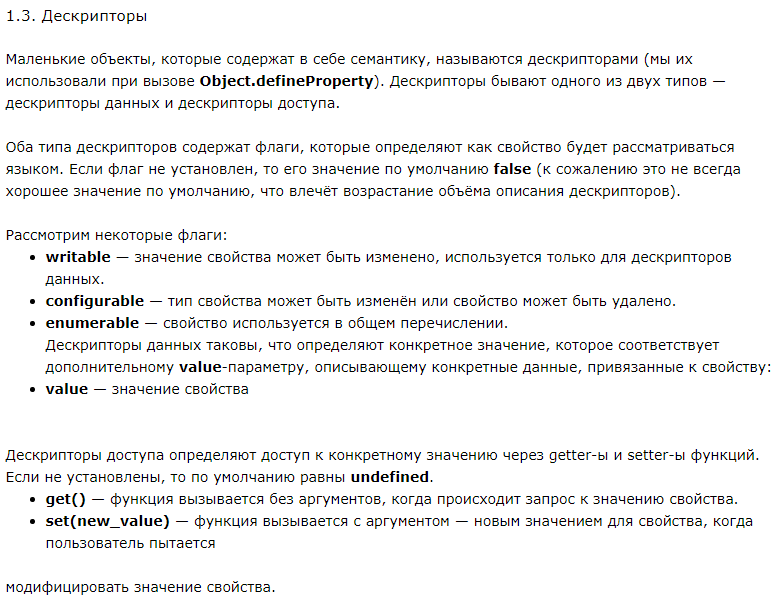






Page object

Design Pattern – готовые решения, которые используются для конкретной задачи. Преимущества – помогают определить архитектуру решения, применяются для любого языка.



This хранит в себе ссылку на объект, которому принадлежит исполняющаяся функция. Это не обязательно означает, что **this** всегда равно объекту, в котором функция *хранится*. Нет, JavaScript не настолько эгоистичен.  
  
Функции являются [generic](http://en.wikipedia.org/wiki/Generic_function)-ами. Т.е. в JavaScript переменная **this** определяет динамическую ссылку, которая разрешается в момент исполнения функции.

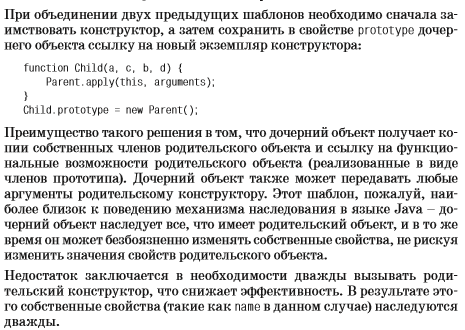
способ явно указать this при помощи методов call и apply.

Различие между двумя методами заключается в параметрах передаваемых в функцию и времени исполнения — **apply** работает примерно в 55 раз медленнее, чем непосредственный вызов, а вот **call**обычно не особо хуже.

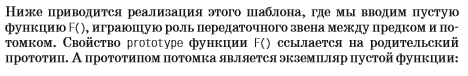
Прототип — это обычный объект, который делится своим поведением с другими объектами — в этом случае он выступает в качестве родителя. Концепт клонирования поведения не означает, что вы будете иметь две различные копии одной и той же функции или данных. На самом деле JavaScript реализует наследование через делегирование, т.е. все свойства хранятся в родителе, а доступ к ним расширен через ребёнка.

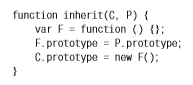
Классические паттерны наследования (повторного использования кода):

- заимствование и установка прототипа



- временный конструктор

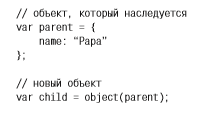


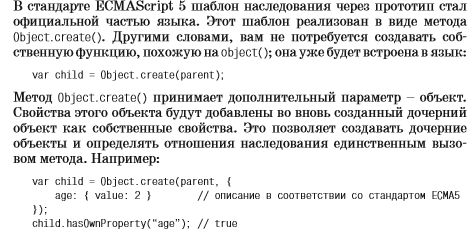


Соверменные методы наследования:

Наследование через прототип

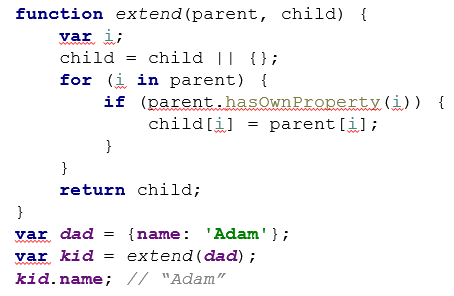
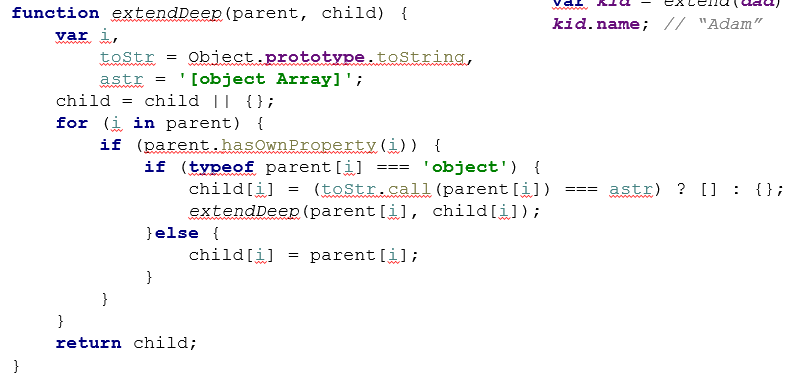
Представить такой тип наследования можно так: имеется некоторый объект, который можно было бы использовать повторно, и вам требуется создать второй объект, использующий функциональные возможности первого объекта.





Наследование копированием свойств

- поверхностное и глубокое

Смешивание

