

Include JS from a file (recommended):

```
<html><head> <script type="text/javascript" src="myjavascripts.js"></script> </head></html>
```

Inside HTML:

```
<html><body> <script type="text/javascript">alert('Javascript here');</script> </body></html>
```

Duck Typing: var a = 1, b = "1"; a + b; // returns string "11"

Check for equality: == (var a = 1, b = "1"; a == b // true, a gets casted to string)

Check for identity: === (var a = 1, b = "1"; a === 1 // true a === b // false)

Function: function nameOfFunction(argument1, argument2) { return argument1 + argument2; }

Arrow style: nameOfFunction = (argument1, argument2) => { return argument1 + argument2; }

```
var functionOne = function() {...}; // Gets evaluated at runtime
function functionTwo() {...} // Gets evaluated at parse-time

functionOne();
var functionOne = function() {} // error

functionTwo();
function functionTwo() {} // works
```

Self-executing anonymous function: (x => console.log('Hello world'))() (avoid polluting global namespace)

Callbacks: setTimeout(x => alert('Hi there!'), 5000)

Strict mode: "use strict"; (silent error → throw error, fix mistakes for JS Engines difficult to optimize)

Primitive Types: numbers, strings, booleans, null, undefined, symbol (other: Objects)

Objects (References): Mutable keyed collections, container of properties (property has name and value), objects are class-free

var empty_object = {};

var stoooge = { "first-name": "Jerome", last-name: "Howard" };

Retrieval: stoooge["first-name"] or stoooge.first-name

Retrieve non-existent member (e.g. stoooge["FIRST-NAME"]) → undefined

Default values: var middle = stoooge["middle-name"] || "(none)";

Attempting to retrieve value from undefined → throw TypeError exception. Guard:

```
flight.equipment // undefined
flight.equipment.model // throw "TypeError"
flight.equipment && flight.equipment.model // undefined
Update: stoooge['first-name'] = 'Jerome';
```

this: Object which "owns" the method (can be different each time the function is called)

Manually set this in function: apply and call

theFunction.apply(valueForThis, arrayOfArgs)

theFunction.call(valueForThis, arg1, arg2, ...), e.g.

```
var car1 = { name: "Car1", age: 23 };
var sayHello = function() { alert("Hi, my name is "+this.name); };
sayHello.call(car1); // Hi, my name is Car1
sayHello.apply(car1); // Hi, my name is Car1
```

New: bind (create new function where this is bound to an object)

```
var car1SaysHello = sayHello.bind(car1); // creates a new function with 'car1' as 'this'
car1SaysHello(); // Hi, my name is Car1
```

Constructor: var Quo = function (string) { this.status = string; };

Public method: Quo.prototype.getStatus = function () { return this.status; };

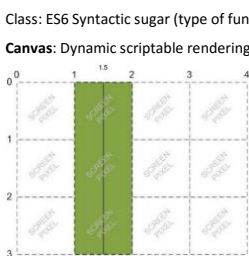
Make instance: var myQuo = new Quo("confused"); console.log(myQuo.getStatus()); // confused

Prototype inheritance:

```
var person = { canTalk : true,
  greet : function() {
    if (this.canTalk) {
      console.log("Hi, I'm "+this.name)
    } } }
var Customer = function(name) { this.name = name; } // "subclass"
Customer.prototype = person; // Inherit customer from person
var joe = new Customer('Joe');
joe.greet();
```

Class: ES6 Syntactic sugar (type of function, prototype based, create objects with new, super class with super, ...)

Canvas: Dynamic scriptable rendering of 2D shapes (updates a bitmap → raster graphics, not vectors) ((0, 0) is top left)



Markup: <canvas width="300" height="225"></canvas> (all drawings are programmed with JS)

Get context: \$('canvas#a').click(function() { var context = this.getContext("2d"); }); (context has to be '2D')

fillStyle / strokeStyle: CSS color / pattern / gradient (default: solid black)

fillRect(x, y, width, height): draw rectangle with current fill style

strokeRect(x, y, width, height): draw rectangle with current stroke style

clearRect(x, y, width, height): clear pixels in specified rectangle

Paths: moveTo(x, y): move pencil to starting point, lineTo(x, y): draw line to end point, stroke(): actually draw lines on canvas

Draw on 0.5 instead of 0: Most screens can't display half a pixel, expand line to cover total of 2 pixels

beginPath(): begin path or reset current path (resets most information on context but not fillStyle and strokeStyle).

Text: fillText(): No box model (margin, padding, word wrapping, ...), font can be anything in CSS font rule, e.g.: context.font = "bold 12px sans-serif"; context.fillText("abc", 760, 580);

Other: drawImage: draw image on canvas, arc(): Takes center point x,y, radius, start&end angle (radians), direction flag (f: clockwise, t: counter-clockwise)

Draw a circle: context.beginPath(); context.arc(x, y, radius, 0, Math.PI * 2, false); context.closePath(); context.stroke();

LocalStorage: up to 64KB user data, 10x more for intranet sites (XML based structure). Up to 100KB "Flash Cookies". Google Gears SQLite DB.

Cookies disadvantages: Included in every HTTP request (more traffic), sending data unencrypted (unless whole web-app over SSL), max 4KB.

Session storage (intended for short-lived data): Only shared with pages from same domain, doesn't persist after window/tab closed (new session storage for each window/tab).

Local Storage: Store data for more than a session (same as sessionStorage but persistent), shared across windows/tabs, 5MB space by default (using too much throws QUOTA_EXCEEDED_ERR), based on key/value pairs. Set value:

localStorage.setItem("key", value); or localStorage["key"] = value; get value: var value = localStorage.getItem("key"); or var value = localStorage["key"]; (if not retrieving strings: parseInt() / parseFloat() / ...), delete value: localStorage.removeItem("key"); (if doesn't exist nothing happens), clear all values: localStorage.clear();

Save Object: localStorage.setItem('test', myObject); saves string representation["object Object"] → localStorage.setItem('test', JSON.stringify(myObject)); ; JSON.parse(localStorage.getItem('test'));

Websockets: Persistente Verbindung Client/Server mit einfachem API. Verbindungsaufbau über HTTP (Connection: Upgrade in Header). Minimale ws:// oder wss:// Header. Bidirektionale Übertragung.

REST: Collection URI (e.g. http://example.com/products): GET (List URIs & details of collection's members), PUT/PATCH (Replace entire collection), POST (new entry in collection (new URI as return value)), DELETE (del entire collection)

Member URI (e.g. http://example.com/products/17): GET (retrieve member in Internet Media Type), POST/PUT/PATCH (Replace member of collection or create if doesn't exist), DELETE (del member)

jQuery GET: \$.getJSON('http://zhaw.herokuapp.com/task_lists/demo', function(data) { console.log(data); }); (getJSON returns a jqXHR Object)

jQuery POST: \$.post('http://zhaw.herokuapp.com/task_lists/', '{"tasks":[{"title": "Do homework"}]}', function(data) { console.log(data); });

Stateless HTTP: HTTP is stateless, every request in isolation → each request contains all information to fulfill request

Advantages: easier to distribute across servers (scaling application: add servers), easy to cache, URIs work when re-visited/shared/bookmarked

Node.js: Application server written in JS

Ember.js: Client side application framework

Events: JS is single-threaded → blocking in JS blocks whole page (Blocking: sleep(5000); alert('Hi there!');, asynchronous: setTimeout(function() { alert('Hi there!') }, 5000);)

Bind event: .on(events, handler(eventObject)), e.g. \$('img').on('click', function(event) { alert('Hi there!'); });

A few shortcut commands (\$('img').click(function(event) { alert('Hi there!'); })); change, dblclick, error, focus, keypress, keyup, load, mousemove, mousedown, resize, scroll, select, ...

Execute event handle only once (once executed handler removed): .one(eventType, data, listener), data optional (caller-supplied data that's attached to event instance as property named data available to handler functions)

Remove event handler: .off(events, listener) (events optional, not supplied → remove all events; listener optional, not supplied → remove all listeners for event)

this in event handler: reference element where event is being delivered (create jQuery object: \$(this))

Event instance (first parameter to function): altKey (t/f), ctrlKey, data, keyCode, metaKey, pageX/Y, screenX/Y, shiftKey, target, type (eg. click, for one event handler multiple events), which (MB), ...

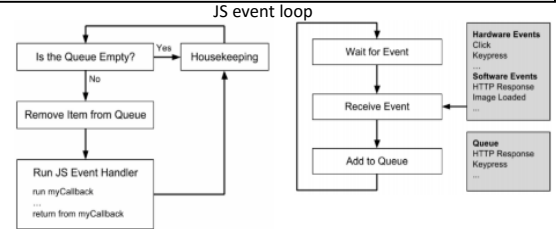
Prevent Default: preventDefault() (block links for a-Elements, form submission, toggle state of checkbox, ...)

New HTML injected to page: Handler not added automatically → **Delegated** events (apply event to handler).

Normal: \$("dataTable tbody tr").on("click", function(event) { alert(\$(this).text()); });, delegate: \$("dataTable tbody").on("click", "tr", function(event) { alert(\$(this).text()); });

event.delegateTarget: DOM-Element that handler was attached to, event.target: DOM-Element that triggered event

Trigger event handler: .trigger(eventType)



Promises: "Easier" / more readable than callbacks, errors handled outside primary application logic. 3 states: pending, fulfilled, rejected (fulfilled & rejected are immutable)

Listeners are always called (run script line for line and when script finished check if EventListener returned anything) →

Priority1: run script, Priority2: process events

The jqXHR returned after Ajax request supports Promises. var promise =

\$.getJSON('http://zhaw.herokuapp.com/task_lists/demo'); promise.done(function(data) { console.log(data); });

promise.error(...);

Multiple consumers possible (called in order of being registered) (promise.then(...); promise.then(...);)

Create a resolved promise: new Promise(function (resolve, reject) { resolve('the long way') }); OR

Promise.resolve('the short way'); (same with reject)

When one promise is rejected all subsequent promises in chain are rejected

Utility function with promises: function loadImage(url) {
var promise = new Promise(
function resolver(resolve, reject) {
var img = new Image(); img.src = url;
img.onload = function () { resolve(img); };
img.onerror = function (e) { reject(e); };
});
return promise; }

Usage with promises: var promise = loadImage('zhaw.png');
promise.then(function (img) {
document.body.appendChild(img);
});
promise.catch(function (e) {
console.log('Error occurred while loading image');
console.log(e);
});

Jasmine: Let a single test sound like a sentence in a specification

```
Example: Code: function Player() { }
Player.prototype.play = function(song) {
  this.currentlyPlayingSong = song;
  this.isPlaying = true;
};

Tests: describe("Player", function() {
  var player;
  var song;
  beforeEach(function() {
    player = new Player();
    song = new Song();
  });
  it("should be able to play a Song", function() {
    player.play(song);
    expect(player.currentlyPlayingSong).toEqual(song);
  });
});
```

Situations to use **Mocks**: breaks isolation (e.g. network requests), supplies non-deterministic replies (e.g. current time/temperature), has states that are difficult to reproduce (e.g. network error), is slow (e.g. complete database), does not yet exist or may change behaviour.

toEqual	Checks if equal (not necessarily same object)	expect({}).toEqual({});
toBeTruthy toBeFalsy	Checks if something evaluates to true Falsy examples: false, 0, "", undefined, null, NaN	expect(true).toBeTruthy(); expect(null).toBeFalsy();
not	Reverse Matchers	expect(foo).not().toEqual(bar);
toContain	Element is member of array (also works with strings)	expect([1, 2, 3, 4]).toContain(3);
toBeNull	Checks if something is null	expect(null).toBeNull();
toBeGreaterThan toBeLessThan	Check greater / smaller	expect(8).toBeGreaterThan(5); expect("a").toBeLessThan("z");
toBeCloseTo	Check number close given decimal precision	expect(12.34).toBeCloseTo(12.3, 1);
toMatch	Regex, argument as regex or string	expect("foo bar").toMatch(/bar/); expect("jasmine@example.com").toMatch(/^\w+@\w+\.\w+\$/);

Spies example: describe("Person", function() {
it('uses the dictionary to say "hello world"', function() {
var dictionary = new Dictionary;
var person = new Person;
spyOn(dictionary, "hello"); // replace hello function with a spy
spyOn(dictionary, "world"); // replace world function with another spy
person.sayHelloWorld(dictionary);
expect(dictionary.hello).toHaveBeenCalled();
expect(dictionary.world).toHaveBeenCalled();
});
});
Make dictionary in other language → still works because method invocation checked not output.
Check output → specify return value of spy: spyOn(dictionary, "hello").and.returnValue("bonjour");
Create a spy for a function that doesn't yet exist: person.getName = jasmine.createSpy("Name spy");
person.getName();
expect(person.getName).toHaveBeenCalled();
spyOn "eats" an existing function, jasmine.createSpy doesn't have to.
Spy Object: var tape = jasmine.createSpyObj('tape', ['play', 'pause', 'stop', 'rewind']);
tape.play();
tape.rewind(10);

Mock Ajax GET:

```
// mock the ajax call to the server loading the tasklist
spyOn($, "getJSON").and.callFake(function(url, callback) {
  callback({ title: 'the list', tasks: [
    { title: 'first task', done: true },
    { title: '2nd task', done: false },
  ]});
});
// execute a mocked ajax call and populate tasklist into result
var result;
TaskList.load('testlist', function(taskList) {
  result = taskList;
});
expect(result.title).toEqual('the list');
```

JS: var songName = document.getElementById("songTextInput").value;

jQuery: var songName = \$("#songTextInput").value;

Der \$-Funktion eine Funktion übergeben → wird ausgeführt wenn DOM fertig geladen ist: \$(function() { \$("#table tr:nth-child(even)").addClass("even"); });

Chaining: \$(".div.notLongForThisWorld").fadeOut().addClass("removed"); (der Class removed wird erst ausgeführt, wenn fadeOut fertig)

CSS Selectors:	\$("#a");	All link elements	\$("#specialID");	Elements with ID specialID
	\$(".specialClass");	Elements with class specialClass	\$("#div .specialClass")	Class specialClass within elements
	\$("#p:even");	All even elements	\$("#tr:nth-child(1)");	First row of each table
	\$("#body > div");	Direct children of	\$("#a[href\$=pdf]");	Links to PDF files
	\$("#body > div:has(a)")	Direct child of div containing links		

Custom jQuery Selectors:	:animated	Elements that are currently under animated control	:button	Any button (input[type=submit], input[type=reset], input[type=button])
	:checkbox	Checkboxes (input[type=checkbox])	:checked	Checkboxes or Radiobuttons that are checked
	:contains(foo)	Elements containing text foo	:disabled / :enabled	Form elements that are disabled / enabled
	:header	h1 to h6	:images	Images (input[type=image])
	:input	Form elements (input, select, textarea, button)	:not(filter)	Negates specified filter

Generate new HTML: \$("<div>Hello, world</div>"); (ready to be added to page)

Generate and append to DOM: \$("<p>Generated content.</p>").css("color", "red").appendTo(".row > .span4");

Wrapped Sets are like arrays: \$("#specialID").html('There are ' + \$(a).length + ' link(s) on this page.');

Get value of attribute of first element in set of matched elements: .attr(), e.g. \$("#myImage").attr("title")

Remove an attribute from each element in set of matched elements: .removeAttr()

Set value: \$("#myImage").attr("alt", "New alternative text"), set multiple attributes: \$("input").attr({ value: '', title: 'Please enter a value' }); (sets value of all input elements to empty and title to enter value)

Set attributes with a function: \$('*').attr('title', function(index) { 'I am element ' + index + ' and my name is ' + (this.id ? this.id : 'unset'); });

Class methods: .hasClass(), .addClass(), .removeClass(), .toggleClass()

Get/Set HTML content of every matched element: .html(), get text including descendants: .text()

.append: Insert content to end of each element, .appendTo(): Insert every matched element to end of target; .prepend(), .prependTo(): Insert beginning

Wrap HTML structure around each matched element: .wrap(); Wrap HTML structure around all matched elements: .wrapAll(); Wrap HTML structure around content of each matched element: .wrapInner()

Removing elements: Remove all child nodes of matched elements from DOM: .empty(); Remove set of matched elements from DOM, leaving matched elements in their place: .remove()

Get current value of first matched element or set value for all: .val(); Limitations: if first element not form element JS error is thrown, doesn't say checkbox/radiobutton checked (return value defined by value attribute) → radiobuttons with same name: \$('<name=radioGroup>:checked').val() returns value of checked rb or undefined.

Commands: .show(), .hide(), .toggle(), .fadeIn(), .fadeOut(), .fadeTo() (Adjust Opacity), .slideUp() (Hide), .slideDown() (Display), .slideToggle() (display or hide), .stop() (stops animation)

Protocols:

REST (Representational State Transfer) operations: GET, POST, PUT, PATCH, DELETE

XHR (XMLHttpRequest): Send HTTP(S) to server and load response data back to script. Can be used to alter current document in browser window without loading new web page → Responsive, dynamic web applications. Request Methods same verbs as HTTP responses.

JSON (JS object notation): Human readable data interchange. Represents simple data structures (objects). Language independent (parsers available in many languages). Used for serializing and transmitting data over network.

Ajax (Asynchronous JavaScript and XML): Web application can send and receive data to/from server asynchronously. Data retrieved using XHR object. Change state (e.g. data received) notify via events. Ajax is group of technologies (HTML/CSS for presentation, DOM for dynamic display&interaction with data, XML for interchange of data, XSLT for manipulation of data, XHR for asynchronous communication, JS to bring everything together).

Example command: \$('#someContainer').load('/serverResource'); (load whatever server returns into someContainer)

Load content: .load(url,parameters,callback) (url: server side resource, parameters optional (GET default, give parameter → object serialized and POST), callback optional)

GET request: \$.get(url,parameters,callback) (parameters: query string to add to url)

Get JSON data: \$.getJSON(url,parameters,callback) (arguments same as normal get)

POST request: \$.post(url,parameters,callback) (arguments same as get)

Ajax request with full control: \$.ajax(options) (type: GET/POST, data (in GET query string / POST data), dataType (expected type to be returned), timeout (in ms, else error callback called), global (t/f), contentType, success (function callback), error (function callback), complete (function callback success or error), beforeSend (function), async (synchronous/asynchronous), processData, ifModified (t/f))

Testing Ajax: Integration tests aren't isolated, servers need to be running, housekeeping, ...; Unit tests: no network connection, response of server mocked.

