JavaScript Programming

Text: EN

Software: EN

Version: 1.2

REALDOLMEN
a Gfi Group company

Table of contents

1. JavaScript Programming

1.1. Goal

Learn the #1 programming language that powers the internet and become an advanced JavaScript developer

1.2. What you will learn

- JavaScript fundamentals
- Concurrency model and the event loop
- Complex features
- What's new since ES6
- Asynchronous programming
- JavaScript modules
- Behind the scenes
- Modern development workflow

- JavaScript fundamentals: variables, dynamic typing, if/else, loops, operators and statements, objects, functions, arrays, scoping, etc.
- The JavaScript concurrency model and the JavaScript event loop
- Complex features like the this keyword, function constructors, prototypal inheritance, first-class functions, closures, IIFE
- What's new since ES6: let, const, arrow functions, classes, template literals, object and array destructuring, lexical this, etc.
- Asynchronous programming with callbacks, promises, async/await
- Organizing and structuring your code using JavaScript modules
- A true understanding of how JavaScript works behind the scenes
- Setting up a modern development workflow with NPM, Webpack, Babel

1.3. Agenda

- Introduction
- JavaScript Language Basics
- Variables and Data Types
- Operators and Statements
- Objects and Functions
- How JavaScript Works Behind The Scenes
- JavaScript in the Browser
- Advanced JavaScript
- Next Generation JavaScript
- Asynchronous JavaScript
- A Modern Development Workflow

- JavaScript in the Browser: DOM Manipulation and Event Handling
- Advanced JavaScript: Inheritance and the Prototype Chain
- Next Generation JavaScript: ES6 and beyond
- Asynchronous JavaScript: callbacks, promises, async/await
- A Modern Development Workflow with NPM, Babel, Webpack

2. Introduction

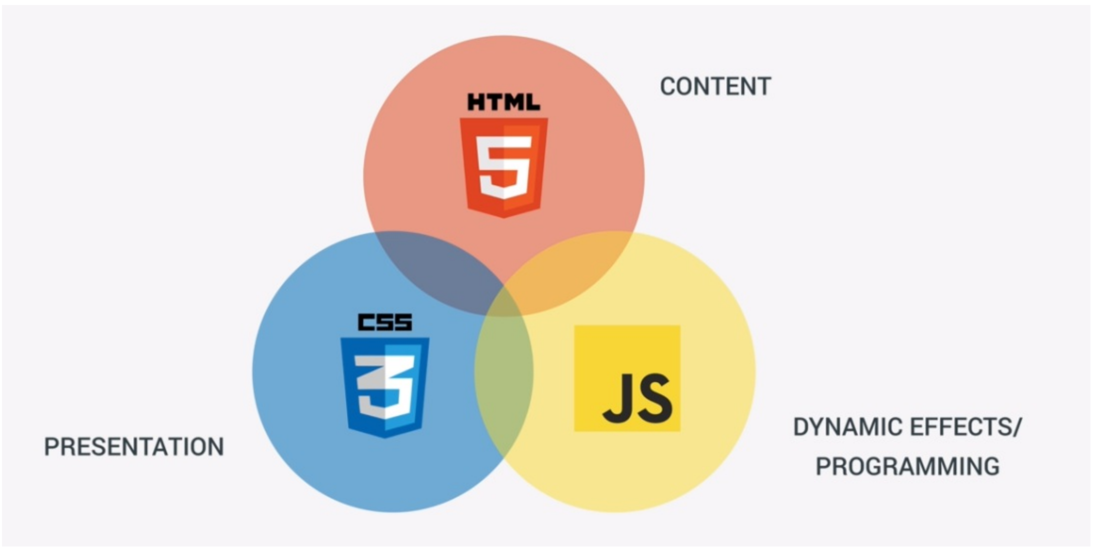
2.1. What is JavaScript?

- A lightweight, cross-platform, object-oriented programming language
- JavaScript is one of the three core technologies of web development
  - HTML
  - CSS
  - JavaScript
- Built into web browsers
  - DOM manipulation
  - Event handling
- JavaScript is what made modern web development possible
  - Dynamic effects and interactivity
  - Single page applications using AJAX

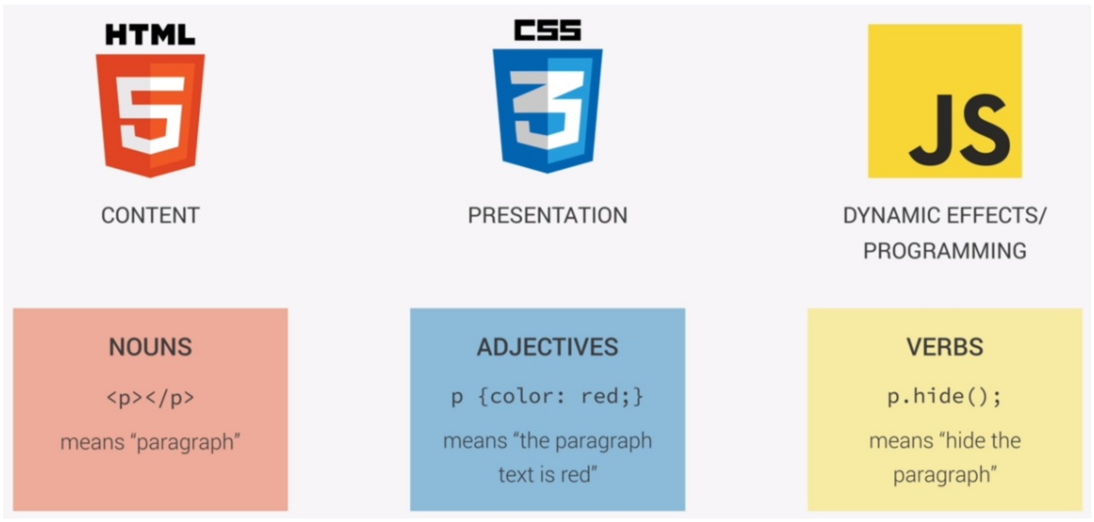
2.2. One language to rule them all

- Today JavaScript can be used everywhere
  - Client-side
  - Server-side
  - Desktop
- Frameworks like Angular and React are based on JavaScript
- You need to master JavaScript in order to use them
- Client-side: JavaScript was traditionally only used in the browser
- Server-side: Thanks to Node.js we can run JavaScript on the server as well
- Desktop: Frameworks like Electron allow to build desktop apps with JavaScript

2.3. The Role of JavaScript in Web Development



2.4. Nouns, adjectives and verbs



2.5. What can you do with JavaScript?

- JavaScript is very popular in web applications
- It allows web pages to become much more dynamic
    - Validate form fields
    - Set and retrieve cookies
    - Dynamically alter the appearance of a page element
    - Hide and show elements
    - Move elements about the page
    - Capture user events and adjust the page accordingly
    - Interface with a server-side application without refreshing
    - And much, much more…

2.6. Performance

- JavaScript used to be an interpreted language
    - This caused it to be relatively slow (100s of times slower than C/C++)
- Modern JavaScript engines are implemented as Virtual Machines
    - They now use advanced performance improvement techniques such as statement reordering, JIT compilation and dynamic optimization
- Performance has improved several orders of magnitude since then
    - Current performance is about 5 times slower than C/C++
    - That's actually very very fast (and does not even include case-specific optimizations)
    - This has enabled application domains such as games, scientific analysis, enterprise applications and even software development tools e.g. Visual Studio Code

2.7. JavaScript History

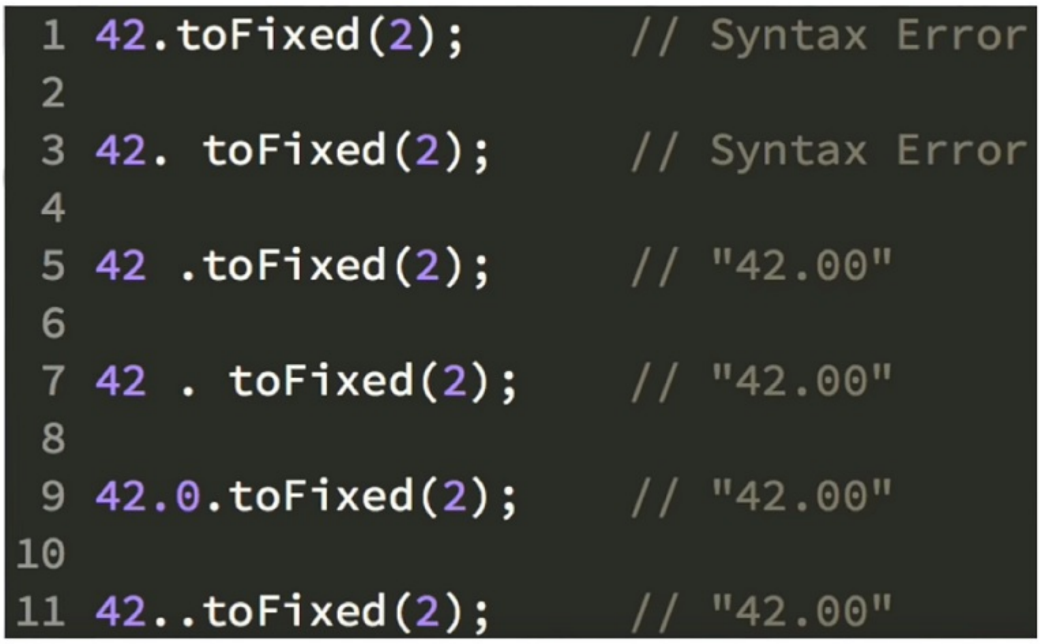
- Netscape's LiveScript in 1995
- After a partnership with Sun, the name was changed to JavaScript
    - To attract Java developers
    - JavaScript has almost nothing to do with Java
- Microsoft countered by releasing VBScript and later JScript

- Confusion and cross-browser incompatibilities followed
- In 1997 JavaScript was standardized as ECMAScript
    - ECMAScript is the standard specification of the language
    - JavaScript is one of the most popular implementations of ECMAScript
- ES1 / ECMAScript 1 became the first version of the standard
- JavaScript has grown due to Ajax popularity

2.8. JavaScript Issues

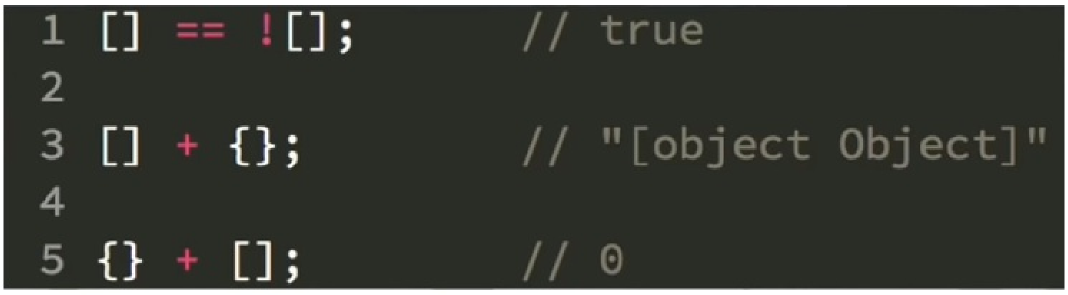
- Although very powerful, JavaScript has quite a lot of issues
- Legend goes that Brendan Eich only spent 10 days designing the language
- Originally intended as a small scripting language for browsers
- Thus it lacked features to manage larger code bases (packages,...)
- There are some notorious big mistakes in its type system
    - Confusing difference between primitive types, objects and functions
        - typeof(Number(1507)) = number
        - typeof(new Number(1507)) = object
        - typeof(Number) = function
    - Unintuitive type conversions
        - typeof(NaN) = number
- [What the... JavaScript?]()

2.9. toFixed



```
42.toFixed(2);          // Syntax Error

42. toFixed(2);         // Syntax Error

42 .toFixed(2);         // "42.00"

42 . toFixed(2);        // "42.00"

42.0.toFixed(2);        // "42.00"

42..toFixed(2);         // "42.00"
```

2.10. Coercion



```
[] == ![];          // true

[] + {};            // "[object Object]"

{} + [];            // 0
```

2.11. Don't Break The Web

- It is clear that JavaScript has some serious historical flaws
- These flaws can never be removed from the language
- It would break the web
- To improve the language, a different strategy had to be taken
- Alternatives are being added to the language
  - e.g. now use let / const instead of the older var (since ES6)
- In other words; JavaScript will evolve slowly over time

2.12. JavaScript Versions

- 1997: ES1 (ECMAScript 1) first release of the JavaScript language standard
- 2009: ES5 (ECMAScript 5) was released with lots of new features
- 2015: ES6/ES2015 (ECMAScript 2015) biggest update to the language ever!
- 2015: Changed to an annual release cycle
- 2016: Release of ES7/ES2016
- 2017: Release of ES8/ES2017
- 2018: Release of ES9/ES2018
- 2019: Release of ES10/ES2019
- ...

2.13. ES6 Harmony

- ES6 or ES2015 is an important revision of the ECMAScript language
- It is the start of a new era of changes to improve the language
- ES6 Harmony is the biggest update to the language ever
- It is the intention of ECMA to release a new update every year
- It will take ECMAScript engines a couple of years before they support all the features of ES6 let alone ES7, ES8, ES9, ES10 and beyond...
- This indeed means that "older" browsers in the field are not yet able to run JavaScript applications using the new coding syntax
- When can we then start developing with these new and cool features?

2.14. Why ES5 is still important

- You still need to understand ES5 today and in the future
- JavaScript fundamentals

- How the language works
- Advanced language features
- Many tutorials and code you find online are still in ES5
- When working on older codebases, these will be written in ES5

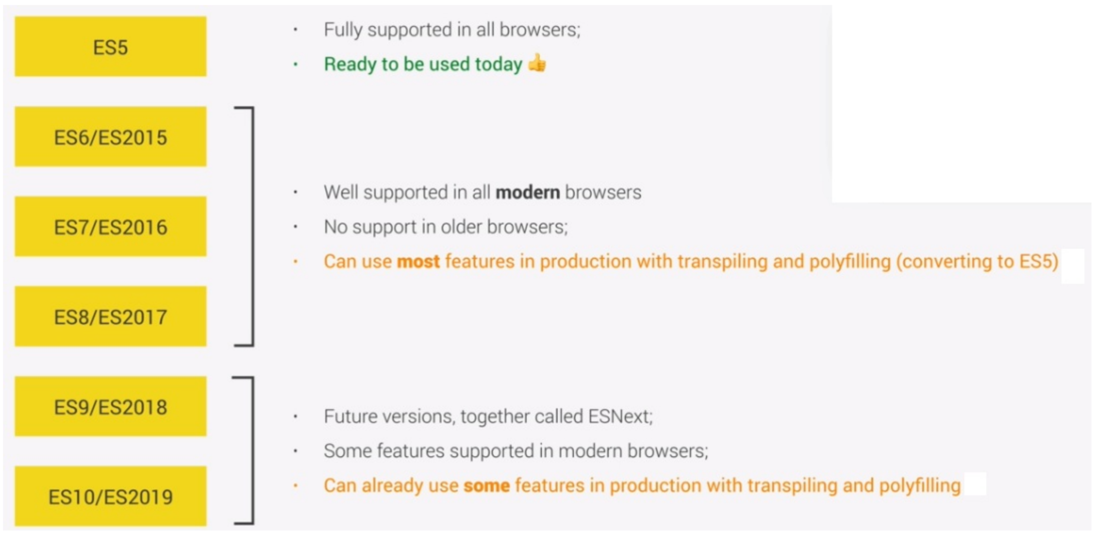
2.15. JavaScript Engines

- Overview of the most popular (modern) ECMAScript engines

Engine	By	Usage
V8	Google	Chrome, Node.js, Edge
Chakra	Microsoft	Internet Explorer, (Edge)
Spidermonkey	Mozilla	Firefox
JavaScriptCore	Apple	Safari
Carakan	Opera	Opera

- Not all browsers support all the newest JavaScript features
  - ES6 compatibility table http://kangax.github.io/compat-table/es6/
  - But we can solve that problem today !
- Microsoft is rebuilding its Edge browser on V8

2.16. JavaScript Today



ES5
- Fully supported in all browsers;
- Ready to be used today 👍

ES6/ES2015, ES7/ES2016, ES8/ES2017
- Well supported in all **modern** browsers
- No support in older browsers;
- Can use **most** features in production with transpiling and polyfilling (converting to ES5)

ES9/ES2018, ES10/ES2019
- Future versions, together called ESNext;
- Some features supported in modern browsers;
- Can already use **some** features in production with transpiling and polyfilling

2.17. Exercise

- Start the Chrome browser
- On Windows use Ctrl + Shift + J to open the Developer Tools
- Open the JavaScript Console
- Hello World
  - console.log('Hello world')
  - var x = 10
  - var y = x + 15
  - console.log(x, y)

- Let's try to manipulate the DOM
    - document.body.innerText = 'Hello world'

3. JavaScript Language Basics

3.1. Adding JavaScript to a page

- You will need to add HTML script tags to a page

```
<script>
... some JavaScript
</script>
```

- They instruct the browser to turn control over to the scripting engine
- Traditionally, these tags were added in the `<head>` of the page but could also be added to the `<body>` or both
- If the code does not run, it is possible JavaScript is disabled, or even rarer, JavaScript is not supported

You should always foresee the possibility that JavaScript is disabled, thus have an alternative ready in your page. If you want to test this yourself, you can disable JS in your browser and check what happens.

Only place JavaScript in the `<body>` when the script generates dynamic content when the page loads. This can be avoided entirely when using DOM to generate new content.

The old way to declare a JavaScript block:

```
<script type="text/javascript">
... some JavaScript
</script>
```

- "Hello World!" in the `<head>`

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello World! In the head...</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <script>
            function hello( ) {
                var dt = new Date( );
                var msg = 'Hello, World! Today is ' + dt;
                window.alert(msg); // alert(msg); is also correct
            }
        </script>
    </head>

    <body onload="hello( );">
    </body>
</html>
```

- "Hello World!" in the `<body>`

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello World! In the body...</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    </head>
    <body>
        <script>
            var dt = new Date( );
            var msg ='<h3>Hello, World! Today is ' + dt + '</h3>';
            document.writeln(msg); // uses DOM to add the content
        </script>
    </body>
</html>
```

- What you see in the examples:
    - Use of the `<script>` tag
    - Declaration of the user-defined hello() function

    - Use of statements ending with a ";"
    - Use of variables and scope
    - Use of the "=" assignment operator
    - Use of the built-in Date object
    - Use of a string
    - Use of the '+' operator
    - Use of the '.' property operator
    - Use of the alert() function from the window object
    - A line comment
    - Use of the onload event handler
- This syntax will be explained throughout this course

3.2. JavaScript Files

- Makes your JavaScript reusable in more than one page
- Easier to maintain:
    - If you need to make changes, you only have to do them once

```
<script src="somejavascript.js"></script>
```

- Best practice:
    - Place all blocks of JavaScript code within external JavaScript files
        - This keeps your web pages uncluttered
        - It also prevents problems with validation and incorrect interpretation of text

3.3. Comments

- Tells the programmer what a block of code is doing
- Makes the code more readable and more maintainable

```
// This line is commented out in the code
/* This is a multiline comment that extends through 2 lines.
 Multiline comments are particularly useful commenting on a function */
```

3.4. Troubleshooting and Debugging

- For a quick check of data without a debugger

```
alert(value);
```

- More sophisticated debugging tools are specific for each browser
- Use Ctrl+Shift+J to open the Chrome Developer tools on Windows
    - Tabbed interface
    - Developer console
    - DOM element inspection
    - Supports breakpoints
    - One click validation and source viewing

```
console.log("Here is the result %s", variable);
```

- Create an index.html file

- Add a JavaScript file called script.js
- Write some logging in the Developer console
- Add a button to dynamically add text to the div with id "contents"

```
<div id="contents">add contents here when clicking the button</div>

<button onclick="myFunction()">Click me</button>
```

- You will need to use document.querySelector() or document.getElementById()
- Check the Mozilla Developer Network (MDN) for more information
- Debug your JavaScript code with the browser's developer tools

4. Variables and Data Types

4.1. Variables and Data Types: what are they?

- In order to be able to start programming, we need to know what Data Types and variables are
- Variables are at the core of (any) program. They are a way of storing a value in memory for later use.
- A variable needs to be declared and you can assign a value to it
- Variables can have different Data Types
  - A Data Type is used to let the program know what to expect: a text, a number, ...

4.2. Variables

- Declaration:
  - Giving a name to the variable, this is called an identifier

```
var a;
var message;
var a, b, c;
```

- Assignment:
  - To assign a value to a variable, we use the '=' operator. This example shows how we would set the variable "name" to point to the string literal "Walter":

```
var firstName; // declare the variable first
firstName = "Walter"; // assign the variable to a string
```

- A shorter way, is a combination of declaring and assigning:

```
var FirstName = "Walter";
```

4.3. Variables : Scope

- Variables can have a global and local scope
  - Local scope (or function scope): a variable defined, initialized and used within a function; when the function terminates, the variable ceases to exist
  - Global scope: can be accessed anywhere within any JavaScript function

- Variables can be declared with and without the "var" keyword, but using "var" is strongly recommended
  - It helps prevent collisions between local and global variables having the same name
  - If you do not use "var", the variable is treated as global
  - Best practice: use "var" regardless of scope

- Prefer local over global variables
  - Global variables may be used in other external JavaScript files
  - They add to the memory burden of the whole application

4.4. Variables : Naming conventions

- Identifier: is the name you give to a variable
- Identifiers can use any combination of letters, digits, underscores and dollar signs, but should not begin with a digit nor a capital
- Valid identifiers:

```
_variableidentifier
```

```
variableIdentifier
$variable_identifier
```

- JavaScript is case sensitive!
- You cannot use any of the JavaScript keywords as a variable identifier

- Reserved Words : abstract, boolean, break, byte, case, catch, char, class, const, continue, debugger, default, delete, do, double, else, enum, export, extends, false, final, finally, float, for, function, goto, if, implements, import, in instanceof, int, inteface, long, native, new, null, package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, typeof, var, volatile, void, while, with
- Not reserved, but not to be used either: undefined, NaN, Infinity

- Guidelines
  - Put words together, using CamelCase capitalization
  - Prefer using full words instead of abbreviations
  - You can add a data type clue as part of the name
  - Use plurals for variables containing collections
  - Start variables and functions with lowercase letters
    - (Objects are capitalized to identify them from variables)

```
var interestRate = .75;
var strName = "Shelley";              // Hungarian notation
var customerNames = new Array(  );
var firstName = String("Shelley");
function validateName(firstName,lastName){ ... }
```

4.5. Data Types

- JavaScript has three primitive types
  - string
  - number
  - boolean

- A primitive is data that is not an object and has no methods.
- All primitives are immutable (cannot be changed).

4.6. Data Type: String

- String literals are sequences of characters delimited by single or double quotes
- Strings can include quotes, but be consistent

```
var strString = "This is a string";
var anotherString = 'But this is also a String';
var stringAsNumber = "543";
var string_value = "This is a 'string' with a quote";
var stringValue = 'This is a "string" with a quote';
var emptyString1 = "";
var emptyString2 = '';

var s = String(1507)

!!Never ever write: new String() // This is Java!
```

4.7. Data Type: String: Escape Sequences

- An escape sequence is a pattern to include specific characters in a string

```
var newLine = "This is first \n This is second";
var quotes = "This is \"quoted\" in the string";
var backslash = "This adds \\backslashes\\ in the string";
```

4.8. Data Type: String: URL Encoding

- Sometimes, you will need to encode complete strings for safe HTML processing
- Use the following methods
  - encodeURI
  - decodeURI
  - encodeURIComponent
  - decodeURIComponent

```
function encodeStrings() {
    var sOne = encodeURIComponent(
        "http://burningbird.net/index.php?pagename=$1&page=$2");
    var sTwo = encodeURI(
        "http://someapplication.com/?cat_name=Zoë&URL=");
    var sOutput = "<p>Link is " + sTwo + sOne + "</p>";
    document.write(sOutput);
    var sOneDecoded = decodeURI(sTwo);
    var sTwoDecoded = decodeURIComponent(sOne);
    var sOutputDecoded = "<p>" + sOneDecoded + "</p><p>" + sTwoDecoded + "</p>";
    document.write(sOutputDecoded);
    }
```

4.9. Data Type: Number

- Floating-point numbers, with or without fractions

```
var negativeNumber = -1000;
var zero = 0;
var fourDigits = 2534;
var someFloat = 0.3555;
var anotherNumber = 144.006;
var negDecimal = -2.3;
var lastNum = 19.5e-2; // which is equivalent to .195
var zeroDecimal = 12.0;
var firstHex = -0xCCFF; // hexadecimal notation
var firstOct = 0526; // octal notation
```

- Special numbers exist: Infinity, -Infinity
  - When a math overflow occurs, or when the number is smaller than the minimum

4.10. Data Type: Boolean

- Has two possible values: true and false
- Do not use quotes
  - "true" != true
- You can convert values explicitly using Boolean or !!

```
var isMarried = true;
var hasChildren = false;

var someValue = 0;
var someBool = Boolean(someValue); // evaluates to false
var strValue = "1";
var boolValue = !!strValue; // converts "1" to a true

var numValue = 0;
boolValue = !!numValue; // converts 0 to false
```

4.11. Data Type: null

- A null variable is defined & has been assigned null as value

```
var nullString = null;
```

4.12. Data Type: undefined

- If the variable is declared but not initialized it is considered undefined

```
var undefString;
```

- When a variable has an initial value, it is not null and is not undefined

```
var sValue = "";
```

- You can test if a variable is NaN with the isNaN function

4.13. Constants

- You use the keyword `const` to create a constant
  - It gets an initial value once, but is never reassigned
  - Best practice: make them global and use uppercase names

```
const CURRENT_MONTH = 3;
```

4.14. Converting Data Types

- You can convert other types to a string
- Typically, this conversion is done automatically, based on the context

```
var num_value = 35.00;
alert(num_value); // expects a string
var strValue = "4" + 3 + 1; // becomes "431"
var strValueTwo = 4 + 3 + "1"; // becomes 71
var firstResult = "35" - 3; // subtraction is applied, resulting in 32
```

- You can also convert explicitly using String

```
function convertToString() {
   var newNumber = 34.56;
   var strNumber = String(newNumber); // "34.56"
   var newBoolean = true;
   var strBoolean = String(newBoolean); // "true"
   var nothing;
   var strUndefined = String(nothing); // "undefined"
   var newNull = null;
   var strNull = String(newNull); // "null"
}
```

4.15. Converting Strings to Numbers

- Use parseInt and parseFloat or Number
  - Conversion is done from left to right until wrong characters are encountered

```
function convertToNumber() {
   var sNum = "1.23e-2";
   document.writeln("<p>" + parseFloat(sNum) + "</p>"); // 0.0123
   document.writeln("<p>" + parseInt(sNum) + "</p>"); // 1
   var fValue = parseFloat("1.45 inch");
   document.writeln("<p>" + fValue + "</p>"); // 1.45
   var iValue = parseInt("-33.50");
   document.writeln("<p>" + iValue + "</p>"); // -33

   X = Number("1507");
}
```

- You can test for Infinity using isFinite
  - If the value is Infinity or NaN, it returns false
  - Else it returns true

4.16. JavaScript Data Typing

- JavaScript is forgiving with respect to data typing
  - If you have a string containing a number, you may use it as a number
  - If you want to use it as a string again, that is also possible
- You could also get wrong results
  - If the string you want to use as number, does contain something else like an email
- It all depends on context, a.k.a. variable scope
- Let's get a good understanding of function scope
- What's the output of this script?

```
    var foo = 'initial value';

    function myFunction() {
        var foo = 'outside';
        if (true) {
            var foo = 'inside'
            console.log(foo);            //print inside
        }
        console.log(foo);
    }

    myFunction();

    console.log(foo);
```

5. Operators and Statements

5.1. Operators

- Binary operators
  - `+ - * / %`
- Concatenating strings
  - `+`
- Unary operators
  - Pre- and post-increment
  - `++ -- -`
- Operator precedence
  - Same as mathematical precedence
  - Use "(" and ")" to control precedence
- Shortcut assignment
  - `+= -= *= /= %=`
- Bitwise operators
  - `& | ^ ~ << >> >>>`

5.2. JavaScript Statements

- JavaScript has different statement types
  - Assignment, function call, conditional, loop
- Statements usually end with a semicolon ";"
  - This is not always required, but is considered best practice
- Whitespace has little impact on the processing, but improves readability
  - Indent code for readability
  - Add a space surrounding the "=" and "+" operators
  - Add comments for maintainability
- Remove whitespace if you want to reduce the size of your JavaScript

5.3. The Assignment Statement

- Is an expression with a variable on the left side, the assignment operator ("=") and an expression on the right

```
nValue = 35.00;
nValue = nValue + 35.00;
nValue = someFunction();
var firstName = 'Shelley'; var lastName = 'Powers';
var firstName = lastName = middleName = "";
var nValue1,nValue2 = 3; // nValue1 is undefined
var nValue1=3, nValue2=4, nValue3=5;
```

5.4. Conditional Statements

- Influence the program flow

```
if (tstValue === 3) { // tstValue really equals 3?
   alert("value is 3");
}
```

- Best practice: Use curly braces around blocks of code
  - You never know if additional code is added later
  - Also add indentation for each following condition in a block

```
if (stateCode === 'MA') {
   taxPercentage = 3.5;
} else {
   taxPercentage = 4.5;
}
```

- It is possible to chain conditional statements

```
if (stateCode === 'OR') {
      taxPercentage = 3.5;
   } else if (stateCode === 'CA') {
      taxPercentage = 5.0;
   } else if (stateCode === 'MO') {
      taxPercentage = 1.0;
   } else {
      taxPercentage = 2.0;
   }
```

- This can become clumsy, hard to read, and inefficient so prefer the switch statement

- The switch statement is useful when several outcomes can result of an expression

```
function choices() {
   var stateCode = 'MO';
   var statePercentage = 0.0;
   var taxPercentage = 0.0;
   switch (stateCode) {
      case 'OR':
      case 'MA':
      case 'WI' :
         statePercentage = 0.5;
         taxPercentage = 3.5;
         break;
      case 'MO' :
         taxPercentage = 1.0;
         break;
      default :
         taxPercentage = 2.0;
         statePercentage = 2.3;
   }
   alert("tax is " + taxPercentage + " and state is "
         + statePercentage);
}
```

5.5. The Conditional Operators

- Equality (uses implicit conversion)
  - `== !=`

- Strict equality (does not use implicit conversion)
  - `=== !==`

- Relational operators
  - `> < >= <=`

- Ternary operator
  - `? :`

```
var nValue = 1.0;
var sResult = (nValue > 0.5) ? "value over 0.5" : "value not over 0.5";
```

- Logical operators
  - `&& || !( )`

5.6. The while Loop

- Tests a condition at the start of each loop and continues if the expression evaluates to true
- Some event in the code should force the expression to false

```
function loops() {
   var strValue = "";
   var nValue = 1;
   while (nValue <= 10) {
      strValue+=nValue;
      nValue++;
   }
   alert(strValue);
}
```

5.7. The do...while Loop

- When you need the loop to be processed at least once

```
function loops() {
   var strValue = "";
   var nValue = 1;
   do {
      strValue+=nValue;
      nValue++;
   }  while (nValue <= 10);
   alert(strValue);
}
```

5.8. The for Loop

- For loops that have to be processed a set number of times

```
for (var i = 0; i < 10; i++) {
   document.writeln("hello");
}
```

- You can also count backwards

```
for (var i = 10; i > 0; i--) {
   document.writeln("hello");
}
```

5.9. The for...in Loop

- Use the for...in loop to iterate over an object's properties

```
function doFor() {
  var MyText = {
     one : "A",
     two : "B",
     three : "C"
  };
  for (var prop in MyText) {
    document.writeln(prop + "<br />");
      }
}
```

- Prefer using the traditional for loop for array processing

5.10. Break and continue

- Both appear inside a block of a ‘while’ or ‘for’ statement
- break
  - Stops entire loop
  - Gives control to the statement following a ‘while’ or a ‘for’-loop
  - Can also be used in a 'switch' to end a specific case
- continue
  - Stops the current repetition
  - Proceeds with next repetition

6. Functions

6.1. JavaScript Functions

- Are a key part of the language
- Are objects!
  - Can be defined
  - Can be created
  - Can be printed
  - Can be put in a variable
  - Can be put in an array
  - Can be passed as an argument to another function (callback)
- Three approaches to creating functions
  - Declarative / static
  - Dynamic / anonymous
  - Literal

6.2. Declarative Functions

- Most common type of functions

```
function sayHi(toWhom) {
   alert("Hi " + toWhom);
}
sayHi("World!");
```

- Naming conventions
  - Use a name that summarizes the activity of the function
  - Begin with the name, following nouns capitalized
    - runQuote()
    - printDate()
    - processName()
    - addNumbers()
- Try to keep your functions small and specific to a task

6.3. Function Returns and Arguments

- Variables are passed to a function "by value"
- Objects are passed to a function "by reference"
- Functions may or may not return a value
  - Use the return statement
    - This stops the processing of the function and returns control
  - You can have more than one return
  - Are not required, but may be helpful in error handling
    - Return false if the method is not successful

```
function testValues(numValue) {
   if (isNaN(numValue)) {
      return "error -- not a number";
   }
   ...
   return ...
}
```

6.4. Anonymous Functions

- Functions are objects
  - As such, they can be created using a function constructor
  - Since those are not explicitly named, they are called anonymous

```
var variable = new Function("param1", "param2", ... , "paramn", "function body");
```

```
var func = new Function("x", "y", "return x * y");

// is equivalent to

function func(x, y) {
   return x * y;
}
```

6.5. Function Literals

- You do not have to use a function constructor to assign it to a variable...

```
var func = function(params) {
   // statements;
}
```

```
var func = function(x, y) {
   return x * y;
}

alert(func(3, 3));
```

- This means you can pass a function as a parameter to another function

```
function funcObject(x,y,z) {
   alert(z(x,y));
}

function testFunction() {
  // third parameter is function
  funcObject(3, 4, function(x,y) { return x * y });
}
```

- Passing a non anonymous function as a parameter

```
var func = function multiply(x,y) {
   return x * y;
}

funcObject(3, 4, multiply);
```

6.6. Recursive functions

- Are functions that invoke themselves
- When a process must be performed more than once and each iteration is performed on the previously processed result
- Can be useful when dealing with tree-structures (DOM)

6.7. Nested functions

- Functions defined inside other functions
- They have access to the outer function's variables and arguments

```
var foo = 'initial value';

function myFunction() {
    var foo = 'outside';

    function nestedFunction() {
        console.log('inside the nested function', foo);
        foo = 'xyz';
    }

    console.log(foo);
    nestedFunction();
    console.log(foo);
}

myFunction();

console.log(foo);
```

6.8. Callback functions

- Functions that are invoked automatically based on some event
- Examples are Array methods that take a callback as argument
  - `filter(), foreach(), every(), map(), some()`

code example:

```
function doHomework(subject, callback) {
  alert(`Starting my ${subject} homework.`);
  callback();
}

doHomework('math', function() {
  alert('Finished my homework');
});
```

6.9. Function Scope

- Variables can be in a local or global scope
  - If you use var inside a function, it will be local, and undefined outside of the function
  - If you use var outside a function, it will be global, as it is not contained in the function
  - If you forget to put the var keyword inside a function, it is considered global!
  - Global variables are not freed up until the application terminates
- Be careful with scope, as variables with the same name might override important information
- Always use the var keyword, avoid using global variables

```
function test() {
   myTest = "test";
   ...
   alert(myTest); // prints out "test"
}
test(); // creates the myTest global variable
alert(myTest); // prints out "test"
```

6.10. Function Object

- If you create a Function object, you also have access to some interesting properties and methods

```
    - length            // returns the number of arguments
    - toString()        // prints the Function object
    - call()            // calls the function by giving an object scope for "this"
    - apply()           // calls the function by giving an object scope for "this"
    - bind()            // bind the "this" scope inside the fuction to an object
```

- apply() and call() differ by the argument list given
  - call() lets you pass the arguments individually
  - apply() lets you pass an array of arguments
- You also have access to the scope of "this" inside a function
  - This scope is used as the first argument of apply() and call(), so you can apply or call the function on a different "this" object

6.11. This

- What's the output of this script?

```
var john = {
    lastName: 'Smith',
    firstName: 'John'
};

function doSomethingWithObject(prefix, obj, postfix) {
    console.log(prefix, obj.lastName, postfix);
    console.log(this);
    console.log(this.lastName);
}

doSomethingWithObject('the-prefix', john, 'the-postfix');
```

6.12. Call

- What's the output of this script?

```
var john = {
    lastName: 'Smith',
    firstName: 'John'
};

function doSomethingWithObject(prefix, obj, postfix) {
    console.log(prefix, obj.lastName, postfix);
    console.log(this);
    console.log(this.lastName);
}

doSomethingWithObject.call(john, 'the-prefix', john, 'the-postfix');
```

6.13. Apply

- What's the output of this script?

```
var john = {
    lastName: 'Smith',
    firstName: 'John'
};

function doSomethingWithObject(prefix, obj, postfix) {
    console.log(prefix, obj.lastName, postfix);
    console.log(this);
    console.log(this.lastName);
}

doSomethingWithObject.apply(john, ['the-prefix', john, 'the-postfix']);
```

6.14. Bind

- What's the output of this script?

```
var john = {
    lastName: 'Smith',
    firstName: 'John'
};

function doSomethingWithObject(prefix, obj, postfix) {
    console.log(prefix, obj.lastName, postfix);
    console.log(this);
    console.log(this.lastName);
}

var doSomethingWithJohn = doSomethingWithObject.bind(john);

doSomethingWithJohn('the-prefix', john, 'the-postfix');
```

7. Objects

7.1. The JavaScript Objects

- Are inherent components of the language itself
- Some are parallel to data types
    - String for strings, Boolean for booleans, Number for numbers
    - They encapsulate primitives and add functionality
- JavaScript also has additional built-in objects
    - Math, Date, RegExp, Array

7.2. Primitive Data Types as Objects

- Objects have methods and other properties
- You can access them using the object property operator
    - A period "."

```
var myName = "Shelley";
alert(myName.length);
alert(myName.strike()); // returns <strike>Shelley</strike>
```

- You can also explicitly create the object

```
var myName = String("Shelley");
alert(myName.valueOf());
alert(myName);
```

- If you want to use the string as primitive, creating an object is not required

7.3. The Boolean Object

- Creating a Boolean object

```
var boolFlag1 = Boolean(); // false
var boolFlag2 = Boolean(0); // false
var boolFlag3 = Boolean(1); // true
var boolFlag4 = Boolean(false); // false
var boolFlag5 = Boolean(true); // true
var boolFlag6 = Boolean(""); // false
var boolFlag7 = Boolean("false"); // true
```

- All objects have the instance methods valueOf() and toString()

```
alert(boolFlag3.valueOf()); // the value true
var strFlag = boolFlag3.toString(); // string containing "true"
```

7.4. The Number Object

- Also has some static properties and methods
    - These are only accessible from the Number object itself
    - They are called directly on the object class
- Static properties used for testing overflow conditions

```
Number.MAX_VALUE // The maximum number representation
Number.MIN_VALUE // The smallest positive number representation
Number.NaN // Represents Not-a-Number
Number.NEGATIVE_INFINITY // Represents negative infinity
Number.POSITIVE_INFINITY // Represents infinity
```

- Additional instance methods

```
toExponential()    // exponential notation
toFixed()        // fixed-point notation
toPrecision()    // uses a specific precision
toLocaleString()    // convert local date/time to a string
```

- To be used on a primitive variable
  - toString(value) // value represents a base for conversion

7.5. The String Object

- Has several instance properties and methods

```
valueOf()        // returns the string literal
length        // contains the length of the string
anchor()        // creates an HTML anchor
charAt()        // returns a character at a given position (0-based)
indexOf()        // starting point of first occurrence of a substring
lastIndexOf()    // starting point of last occurrence of a substring
link()        // returns a HTML link
concat()        // concatenates strings together
split()        // splits strings in tokens based on a separator
slice()        // returns a slice from the string
substring(), substr()    // returns a substring
big(), blink(), bold(), italics(), small(), strike(), sub(), sup()
match(), replace(), search()     // use regular expressions
toLowerCase(), toUpperCase()     // convert case
```

7.6. Some Remarks on Strings

- Strings are immutable
  - You cannot change a string once you created it
  - Minimize the amount of concatenation, as this impacts performance
- substring(), substr() and slice()
  - substring() takes two indices and extracts the characters
    - If the stop index is less than the start, the values are swapped
  - substr() takes an index and a length of the substring
  - slice()
    - If the stop index is less than the start, the empty string is returned
- Static method
  - fromCharCode()
  - Takes Unicode values separated by commas and returns a string

7.7. Regular Expressions and RegExp

- Regular expressions re arrangements of characters that form a pattern to
  - Find matches
  - Make replacements
  - Locate specific substrings

- You can create regular expressions using the new keyword and the RegExp object, or using a literal

```
var searchPattern1 = new RegExp('s+');
var searchPattern2 = /s+/;
// are there one or more occurrences of s in the string?
```

Expression	Description
[abc]	Find any character between the brackets
[^abc]	Find any character NOT between the brackets
[0-9]	Find any character between the brackets (any digit)
[^0-9]	Find any character NOT between the brackets (any non-digit)
(x|y)	Find any of the alternatives specified

7.8. RegExp Methods

- Instance methods
  - test() // determines whether the string matches
  - exec() // searches a string for a specified value
- The exec() method returns the matched text if a match is found, and null if a match is not found
- Regular expressions are more commonly used with the String object replace(), match() and search()

7.9. The Date Object

- To create instances of dates
- Creating a Date

```
var dtNow = new Date();
var dtMilliseconds = new Date(5999000920);
// Wed, 11 Mar 1970 10:23:20 GMT
var newDate1 = new Date("March 12, 1980 12:20:25");
var newDate2 = new Date('March 12, 2008'); // zeros for the times
var newDt1 = new Date(1977,11,23); // 11 = DECEMBER
var newDt2 = new Date(1977,11,24,19,30,30,30);
```

7.10. Date Methods

- Several get and set instance methods for retrieving or setting specific components
- According to local times
- getFullYear()
  - getHours()
  - getMilliseconds()
  - getMinutes()
  - getMonth() // number between 0 and 11 inclusive
  - getSeconds()
  - getDay() // number between 0 and 6 from Sunday
  - getDate() // the day of the month
- According to Coordinated Universal Time (UTC)
  - getUTCFullYear()
  - getUTCHours()

  - getUTCMilliseconds()
  - getUTCMinutes()
  - getUTCMonth()
  - getUTCSeconds()
  - getUTCDay()
  - getUTCDate()

- All these get methods have equivalent set methods

- Converting the date to a formatted string
  - toString() // in local time
  - toGMTString() // in GMT standards
  - toLocaleDateString() // using the locale
  - toLocaleString() // using the current locale
  - toUTCString() // using UTC standards

- Static methods
  - now() // the current date and time
  - parse() // the number of milliseconds since Jan 1, 1970
  - UTC() // the number of milliseconds based on numbers

```
var numMs = Date.UTC(1977,11,24,19,30,30,30);
```

7.11. The Math Object

- Static properties
  - E // the value of e
  - LN10 // natural logarithm of 10
  - LN2 // natural logarithm of 2
  - LOG2E // base-2 logarithm of e
  - LOG10E // base-10 logarithm of e
  - PI // the value of pi
  - SQRT1_2 // square root of ½
  - SQRT2 // square root of 2

7.12. Math Methods

- Static methods
  - abs() // returns the absolute value of a number
  - sin(), cos(), tan(), acos(), asin(), atan(), atan2()
  - ceil() // rounds to the next highest whole number
  - floor() // rounds down to the next lowest whole number
  - round() // rounds to the nearest integer
  - exp() // e to the power of the value
  - pow() // raises a number to a given power
  - min(), max() // compare two or more numbers and return ...
  - random() // a number with range [0, 1[

7.13. Arrays

- A JavaScript Array is an object

- Creating an Array

```
var newArray1 = new Array('one','two');
var newArray2 = ['one','two']; // use this a lot
```

- Accessing array elements by index value (0-based)

```
alert(newArray1[0]);
```

- Arrays can have multiple dimensions, and grow as needed

```
var threedPoints = new Array();
threedPoints[0] = new Array(1.2,3.33,2.0);
threedPoints[1] = new Array(5.3,5.5,5.5);
threedPoints[2] = new Array(6.4,2.2,1.9);
var newZPoint = threedPoints[2][2]; // remember, arrays start with 0
var testArray = new Array();
testArray[99] = 'some value'; // testArray now has 100 elements
alert(testArray.length); // outputs 100
```

7.14. Array Methods

- Instance methods
  - splice() // to insert and/or remove elements
  - concat() // concatenates an array to the end of the other
  - join() // generates a string with a join character
  - reverse() // reverses the order of the elements
- FIFO Queue methods
  - push() // adds an element to the end of the array
  - unshift() // adds an element tot the beginning
  - pop() // removes the last element of the array
  - shift() // removes the first element of the array
- Traversing arrays is done using for-loops

```
for (var i = 0; i < someArray.length; i++) {
    alert(someArray[i]);
}
```

7.15. Using the for...in Loop

- What's the output?

```
var programLanguages = new Array('C++','Pascal','FORTRAN','BASIC','C#','Java','Perl','JavaScript');

for (var item in programLanguages) {
   document.writeln(item + "<br>");
}
```

7.16. Using the for...of Loop

- What's the output?

```
var programLanguages = new Array('C++','Pascal','FORTRAN','BASIC','C#','Java','Perl','JavaScript');
```

```
for (var item of programLanguages) {
   document.writeln(item + "<br>");
}
```

7.17. A Comparison

	`for..in`	`for..of`
Applies to	Enumerable Properties	Iterable Collections
Use with Objects?	Yes	No
Use with Arrays?	Yes, but not advised	Yes
Use with Strings?	Yes, but not advised	Yes

Source: https://bitsofco.de/for-in-vs-for-of/

A custom object

- What's the output?

```
var john = {
    lastName: "Smith",
    firstName: "John",
    age: 32,
    log: function() {
        console.log(this.firstName, this.lastName, "is", this.age, "years old");
    }
};

john.log();

var mark = {
    firstName: "Mark"
};

john.log.call(mark);
```

- You are working for a company as a financial analyst. They ask you to calculate the profit for the entire year and also per month. The average margin on sales is 30 %. These are the sales figures they give you:
  - January: 150k
  - February: 250k
  - March: 145k
  - April: 90k
  - May: 60k
  - June: 200k
  - July: 195k
  - August: 56k
  - September: 85k
  - October: 86k
  - November: 45k
  - December: 275k

8. Handling Browser Events

8.1. Events

- Are fired when a certain activity occurs within a page
- Can be captured using event handlers
    - Will execute a function or other JavaScript when the event fires

- Some events are associated to page elements
- Some events are associated to the page itself
- For each event, there is an event handler
    - load event → onload event handler

- Events have three categories:
    - User interface (mouse, keyboard)
    - Logical (result of a process)
    - Mutation (action that modifies a document)

- A List of Events
    - abort // when an image is prevented from loading
    - blur, focus // when an object looses or receives focus
    - change // when a selection or value changes
    - click, dblclick // clicking or double-clicking with the mouse
    - contextmenu // right-clicking with the mouse
    - error // when the page or image cannot load
    - load, unload // page or image finishes loading, page is closed
    - mousemove // when the mouse moves
    - reset // when a form is reset
    - resize // a window or frame is resized
    - select // selecting text
    - scroll // an object is scrolled
    - submit // a form is submitted
    - keydown, keyup, keypress // pressing, releasing a key
    - mousedown, mouseup // pressing, releasing the mouse
    - mouseover, mouseout // moving over, out with cursor

8.2. Basic Event Handling

- Associate page elements to events by adding attributes
- This is called the inline registration model

```
<body onload="var i = 23; i *= 3; alert(i);">
```

```
<body onload="calcNumber();">
```

- You can also access the event handler as property on the element
- This is called the traditional registration model

```
window.onload=calcNumber;
```

- If you want the event to stop default behavior, you can return false

```
function doSomething() {
   // does some code
   return false;
}
```

8.3. The Event Object

- Is associated with all the events
- Has properties that provides information about the event
  - Such as the location of a mouse click in the web page
- The use and access of the object differs from browser to browser

```
// in Firefox, the event is passed as an argument
function mouseDown(nsEvent) {
  // in IE, the event is available from the window object
  var theEvent = nsEvent ? nsEvent : window.event;
  var locString = "X = " + theEvent.screenX + " Y = " + theEvent.screenY;
  alert(locString);
}
```

NSEvent

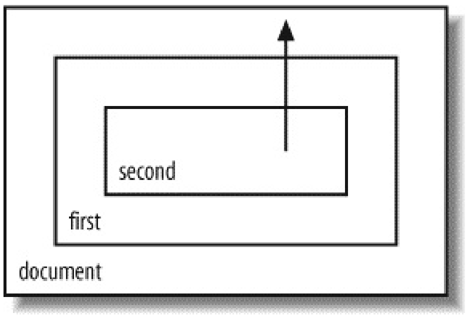
An object that contains information about an input action such as a mouse click or a key press.

8.4. Event Object Properties

- You can fetch interesting values from the Event Object
  - altkey // is the Alt key pressed at the time of the event
  - clientX // client x coordinate of the event
  - clientY // client y coordinate of the event
  - ctrlkey // is the Ctrl key pressed at the time of the event
  - keyCode // the code (number) of the key pressed
  - screenX // the screen x coordinate of the event
  - screenY // the screen y coordinate of the event
  - shiftkey // is the Shift key pressed at the time of the event
  - type // the type of the event
- Some other properties are not compatible across browsers

8.5. Event Bubbling

- In event bubbling, on nested elements, the innermost element would fire its event, followed by the next in the stack, until you reach the outermost element



8.6. "Pop The Bubble"

- To cancel an event (to stop it from bubbling), you will have to call a method on the event

```
function stopEvent(evnt) {
   if (evnt.stopPropagation) {
     evnt.stopPropagation();
   } else {
     evnt.cancelBubble = true;
   }
}
```

8.7. Accessing The Containing Element

- JavaScript uses the special keyword "this"
- It represents the owner of the function or method currently being processed
- For a global function, this object is window
- In an event handler, it represents the element that received the event

```
function setObjects() {
    document.getElementById("personData").firstName.onblur = testValue;
}
fuction testValue() {
    alert("Hi " + this.value); // this represents the firstName field
}
```

8.8. Cascading Down Events

- You can also capture events, called cascade down
- The events will fire from the outside in
- In W3C, both event handling types are supported

```
function cascadeDown(evnt) {
   alert("Capturing: " + this);
}
function bubbleUp(evnt) {
   alert("Bubbling: " + this);
}
window.onload=setup;
function setup(evnt) {
   // capturing
  document.addEventListener('click', cascadeDown, true);
  document.forms[0].addEventListener('click', cascadeDown, true);
  document.forms[0].elements[0].addEventListener('click', cascadeDown, true);
  // bubble up events
  document.addEventListener("click", bubbleUp, false);
  document.forms[0].addEventListener("click", bubbleUp, false);
  document.forms[0].elements[0].addEventListener("click", bubbleUp, false);
}
```

- You can use stopPropagation() to stop the events
- You can remove a listener as well

```
document.forms[0].elements[0].removeEventListener("click", cascadeDown, true);
```

8.9. Generating Events

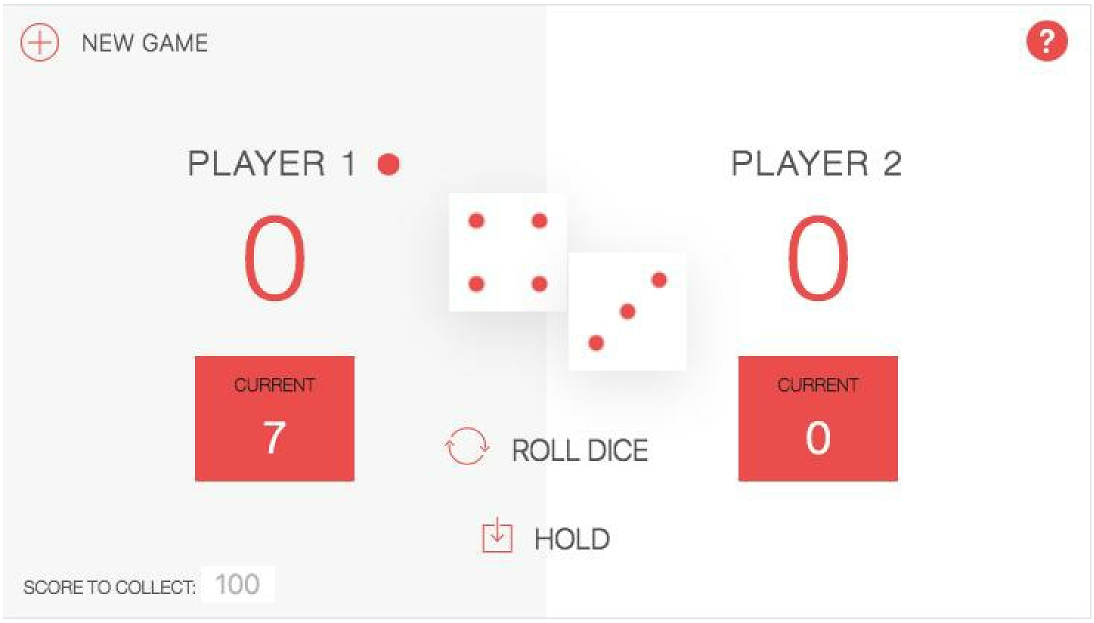
- Sometimes, you want to trigger an event through code
- Usually for putting the focus on an element

```
document.getElementById("someButton").click();
```

```
window.onload=setObjects;

function setObjects() {
   document.getElementById("personData").lastName.focus();
}
```

9. The Pig Game Exercise

9.1. Implement the Pig Game



9.2. Rules

- Each turn the player rolls two dice
- If the player gets 1 on any of the dice he earns nothing and the other player then gets his turn to roll the dice
- On getting 2 to 6 on the dice, player's sum is being added to his round score, until he decides to stash his current score
- If the player stashed his score, the other player gets his turn
- If a player gets 6 on any dice twice in a row, the player loses all the points from his current and overall scores
- The player who got 100 or more points wins the game

10. Dynamic Web Pages (DHTML)

10.1. DHTML

- The key element is the introduction of CSS to the DOM
  - Can be used to change the appearance of elements without relying on external applications, plug-ins or excessive use of images
  - Separates the presentation from the content
- In combination with DOM, we can dynamically change individual element properties even after page loading
- This gives us the possibility to create an even more interactive user experience

10.2. The style Property

- All nodes of the DOM have a style property
  - First, you need to access the element
  - Then you change the style attribute using assignment

```
function changeCSS() {
  var div = document.getElementById("div1");
  div.style.backgroundColor="#f00"; // CSS background-color
  div.style.width="500px";
  div.style.color="#fff";
  div.style.height="200px";
  div.style.paddingLeft="50px"; // CSS padding-left
  div.style.paddingTop="50px"; // CSS padding-top
  div.style.fontFamily="Verdana"; // CSS font-family
  div.style.borderColor="#000"; // CSS border-color
}
```

- This works for any CSS2 property, on any valid X/HTML object, as long as the attribute has meaning for the element
- Remarks:
  - If you do not set the style using JavaScript or inline, this property stays blank or undefined, even if set by the CSS stylesheet
  - To access the current style, you will need other to use other properties, but those do not work consistently across browsers
  - JavaScript libraries such as Prototype, Dojo and jQuery take care of the browser differences
  - Conclusion: use style only to set attributes, not to read them!

10.3. Fonts and Text

- Font properties have something to do with the characters themselves
  - Their family, size, type and appearance
- Text attributes have more to do with decoration attached to text
  - Text decoration (underline), alignment, ...
- Typical usage:
  - Expanding the size of a block of text to make it legible
  - Highlight the data
  - Changing the font color by "greying out" text that does not apply

10.4. Position and Movement

- Controlling the page layout
  - Consider a page as a graph, with both x and y coordinates
  - With CSS, you can set an element's position
  - With JavaScript you can move elements around in the page
- Important properties
  - top, left, bottom, right, z-index
- Typical usage:
  - Set elements outside the visible range
  - Move elements based on user clicks
  - To create fly-ins (e.g. help tutorials)

10.5. Size and Clipping

- Size can be controlled using six CSS attributes
  - width, height, min-height, min-width, max-height, max-width
  - The first two are most common
  - The other four are not supported consistently among browsers
- If the content is too large for an element, you can use the overflow attribute
  - Can be set to visible, hidden, scroll and auto
- Best practice:
  - Always set the overflow, as the content of an element might drastically change with the use of Ajax
- You can control the visible part of the element using the clipping rectangle
- Clipping constrains how much of the actual element content is displayed

```
clip: rect(topval, rightval, bottomval, leftval);
```

- Typical usage:
  - To create a scrolling effect, paired or not to the element
  - To create an accordion effect (click on a header, seemingly unscrolls the content underneath)
  - Typically combined with a timer for a small animation

10.6. Display, Visibility and Opacity

- Visibility
  - Use the visibility property
    - hidden or visible
  - The element keeps its position within the page flow
- Opacity
  - Can be altered to make an element transparent
  - For fading effects or deemphasis
- Display
  - If set to none, it hides an element, but also removes it from the page layout
  - Can be displayed as block or inline
  - Pushes down other content when it is displayed
- Typical usage:
  - Tooltips

- Help balloons or hints

11. Forms and Validation

11.1. Accessing Forms From JavaScript

- Forms are accessed through the DOM using the document object
- There are two different ways
    - Using the forms property of the document object
    - Using the method getElementById

```
var theForm = document.forms[0];
// or
var theForm = document.getElementById("someform");
```

- It is also important to intercept the form submission, as it has to be cancelled if the form is invalid

11.2. Attaching Events to Forms

- The primary event of a form is submit, with the event handler onsubmit

```
document.getElementById("someform").onsubmit=formHandler;
// or
<form name="someForm" onsubmit="return formHandler();">
```

- To cancel the event, just return false from the event handler function
- This is a difficult approach to maintain, as you will have to write a lot of similar code for each event you want to capture
- Validation procedure:
    - Capture submit event, check form elements and show a message to the user

11.3. Cross-Browser Canceling of an Event

- To stop the propagation, you will also need a cross-browser function to make sure the event is really cancelled

```
function formFunction(evnt) {
    var event = evnt ? evnt : window.event;
    ...
}
```

```
function cancelEvent(event) {
   if (event.preventDefault) {
       event.preventDefault();
       event.stopPropagation();
   } else {
       event.returnValue = false;
       event.cancelBubble = true;
   }
}
```

11.4. Validating a Selection

- An example select element

```
<select name="theSelection" multiple="multiple">
<option value="Opt1">Option 1</option>
<option value="Opt2">Option 2</option>
...
<option value="Optn">Option n</option>
```

```
</select>
```

- Properties
  - disabled // is the element disabled?
  - form // the containing form
  - length // the number of options
  - options // the array of options
  - selectedIndex // the number of the selected item (or first)
  - type // the type of element
- Option properties
  - selected // true if the option was selected
  - value // the option value
  - text // the option text visible to the user
- Fetching the selected options:
  - For single selection, use selectedIndex
  - For multiple selection, you will need to iterate

```
var slIdx= document.getElementById("formname").theSelection.selectedIndex;
var opt = document.getElementById("formname").theSelection.options[slIdx];
```

```
var opts = document.getElementById("someForm").selectOpts.options;
for (var i = 0; i < opts.length; i++) {
   if (opts[i].selected) {
      alert(opts[i].text + " " + opts[i].value);
   }
}
```

11.5. Dynamically Modifying a Selection

- You can add and remove options on the fly

```
// adding an option
opts[opts.length] = new Option("Option Four", "Opt4");
// removing an option
opts[2] = null;
// removing all options
opts.length = 0;
```

- Remark:
  - Do not forget to cancel a form submit if you want to react to user input, as a page reload will remove all the changes made by the user...
- More advanced uses
  - Auto-selection of options (see book)

11.6. Validating Radio Buttons and Checkboxes

- An example of radio buttons

```
<form id="someForm" action="">
<p>
<input type="radio" value="Opt 1" name="radiogroup" />Option 1<br />
<input type="radio" value="Opt 2" name="radiogroup" />Option 2<br />
</p>
</form>
```

- The name is the same, to group the buttons
- Accessing the radio button group

```
var buttons = document.getElementById("someform").radiogroup;
for (var i = 0; i < buttons.length; i++) {
   if (buttons[i].checked) {
      alert(buttons[i].value);
   }
}
```

- An example of checkboxes

```
<form id="someForm" action="">
   <p>Option 1: <input type="checkbox" name="checkbox1" value="Opt1" /> <br />
      Option 2: <input type="checkbox" name="checkbox2" value="Opt2" /> <br /></p>
</form>
```

- You can select more than one checkbox, you will have to iterate over all of them to see which one has been selected

- Iterating over checkboxes

```
function checkColors(evnt) {
   var theEvent = evnt ? evnt : window.event;
   var colorOpts =
     document.getElementById("someForm").getElementsByTagName("input");
   // check through input elements for checkbox and checked
   var isChecked = false;
   for (var i = 0; i < colorOpts.length; i++) {
     if ((colorOpts[i].type == "checkbox") && (colorOpts[i].checked)) {
       isChecked=true;
       break;
     }
   }
   // none were checked
   if (!isChecked) {
      alert("You must check one of the four color checkboxes");
      cancelEvent(theEvent);
   }
}
```

- You can also disable buttons, or handle their specific click event with an onclick handler

11.7. Validating Freeform Text Fields

- Freeform text fields include text, textarea, password and hidden input elements
- Example freeform fields

```
<input type="text|hidden|password" name="fieldName" value="Some value" />
```

```
<textarea name="fieldName" rows="10" cols="10">Initial text</textarea>
```

You can access their contents using the value property

```
var strResults = "";
var textInputs =
  document.getElementById("someForm").getElementsByTagName("input");
for (var i = 0; i < textInputs.length; i++) {
  if (textInputs[i].type != "submit") {
    strResults += textInputs[i].value;
  }
}
document.getElementById("text4").value=strResults;
```

- For text validation, events of interest are change, focus and blur
- Checking a required field

```
catchEvent(document.getElementById("text2"), "blur",checkRequired);
function checkRequired (theEvent ) {
var target = theEvent.target ? theEvent.target : theEvent.srcElement;
  var txtInput = target.value;
  if (txtInput == null || txtInput == "") {
    alert("value is required in field");
  }
}
```

- Best practice:
  - Do not enforce required fields using focus as this can be irritating!

- Regular expressions can be useful to match certain patterns
- Validating a field with a regular expression

```
catchEvent(document.getElementById("text1"), "change", validateField);
function validateField(evnt) {
  var theEvent = evnt ? evnt : window.event;
  var target = theEvent.target ? theEvent.target : theEvent.srcElement;
  var rgEx = /^\d{3}[-]?\d{2}[-]?\d{4}$/g; // nnn-nn-nnnn
  var OK = rgEx.exec(target.value);
  if (!OK) {
    alert("not an ssn");
  }
}
```

- Best practice:
  - Using alerts for feedback can become irritating, better options exist (such as DHTML, see later)

- Common scenarios for regular expression validation
  - Warranty or purchase certificates
  - Email addresses
  - Phone numbers
  - SSN or other forms of ID
  - Dates
  - Abbreviations (states, ...)
  - Credit card numbers
  - Web page URLs or URIs

- Many regular expressions can be found on
  - http://regexlib.com

- Best practice:
  - On the server, there should also be some form of validation, to eliminate the threat of cross-site scripting (XSS) such as SQL injection attacks

12. JSON

12.1. What is JSON?

- JavaScript Object Notation
  - Lightweight data interchange format
  - Human-readable
  - Part of ECMAScript standard since 1999
- Two basic data types
  - Key/value pairs become a JavaScript Object
  - Ordered value list becomes a JavaScript Array
- Information and scripts on
  - http://www.json.org
- Syntax basics
  - An object is unordered list of key/value pairs
    - : is used to separate the key and value
    - , is used to separate the pairs
  - A value can be any one of
    - string, number, boolean, object or array
  - An object begins with { and ends with }
  - An array begins with [and ends with]
- o = JSON.parse(s)
- s = JSON.toString(o)

12.2. Example of JSON Notation

- An order object has
  - A table number
  - An array of order lines
- Every order line object has
  - A beer id
  - The ordered quantity

```
{ table: 10 ,
  lines: [ { id: "geuze", quantity: 1 }, { id: "leffe", quantity: 3 } ]
}
```

```
// using JSON directly in JavaScript code
var order = {
    table: 10,
    lines: [
        {id: "geuze", quantity: 1},
        {id: "leffe", quantity: 3} ]
};

alert("Table " + order.table + " has ordered "
               + order.lines.length + " distinct products");

// making a JSON string out of the order object
var jsonString = order.toJSONString();
alert(jsonString);

// recreating the order objects from a JSON string
var order2 = eval("(" + jsonString + ")");
var order3 = jsonString.parseJSON();
```

```
// both orders are exactly the same
alert(order2.lines[0].id + " -> " + order3.lines[0].quantity);
alert(order3.lines[1].id + " -> " + order2.lines[1].quantity);
```

13. Advanced JavaScript

13.1. This or That

- What is this thing called this in JavaScript?

```
function foo() {
    console.log(this.a);
}

var a=2;

foo(); // 2
```

13.2. This can be complicated

- A binding that is made when a function is invoked
- Referencing some object or some function
- Determined entirely by the call-site where the function is called
- It can be confusing :-)

13.3. Call-site of a function

```
function baz() {
    // call-stack is: `baz`
    // so, our call-site is in the global scope

    console.log( "baz" );
    bar(); // <-- call-site for `bar`
}

function bar() {
    // call-stack is: `baz` -> `bar`
    // so, our call-site is in `baz`

    console.log( "bar" );
    foo(); // <-- call-site for `foo`
}

function foo() {
    // call-stack is: `baz` -> `bar` -> `foo`
    // so, our call-site is in `bar`

    console.log( "foo" );
}

baz(); // <-- call-site for `baz`
```

13.4. Default binding

- Standalone function invocation
- The default catch-all rule when none of the other rules apply
- Variables declared in the global scope are synonymous with global-object properties of the same name
- In a browser the global object is called window

```
function foo() {
    console.log( this.a ); // this = the global object
}

var a = 2; // defines a property on the global object, so this is actually 'window.a'

foo(); // 2, the call-site is in the global scope so 'this' points to the global object
```

```
console.log(window.a); // 2
```

13.5. Default binding in Strict mode

- If in strict mode, the global object is not eligible for the default binding
- So this is instead set to undefined

```
function foo() {
    "use strict";

    console.log( this.a ); // this = undefined
}

var a = 2;

foo(); // TypeError: `this` is `undefined`
```

13.6. Implicit binding

- When there is a context object for a function reference, the implicit binding rule says that it's that object which should be used for the function call's this binding

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var a = 1;

foo(); // 1

obj.foo(); // 2
```

13.7. Implicitly lost

- Fallback to the default binding

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // function reference/alias!

var a = "oops, global"; // `a` also property on global object

bar(); // "oops, global"
```

13.8. Implicitly lost with a callback function

- The more subtle, more common and more unexpected way this occurs is when we consider passing a callback function

```
function foo() {
    console.log( this.a );
}

function doFoo(fn) {
    // `fn` is just another reference to `foo`

    fn(); // <-- call-site!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a` also property on global object

doFoo( obj.foo ); // "oops, global"
```

13.9. Implicitly lost with built-in callback functions

- setTimeout(fn, delay)

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a` also property on global object

setTimeout( obj.foo, 100 ); // "oops, global"
```

13.10. Explicit binding

- Force a function call to use a particular object for the this binding
- Without putting a property function reference on the object

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

13.11. Hard binding

- Unfortunately, explicit binding alone still doesn't offer any solution to the issue of a function "losing" its intended this binding

```
function foo() {
    console.log( this.a );
}
```

```
var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a` also property on global object

setTimeout( foo.bind(obj), 100 ); // 2
```

13.12. API call "Context"

- Many libraries and many new built-in functions in the JavaScript language provide an optional parameter, usually called context
- Designed as a work-around for you not having to use bind(...)
- Ensures that your callback function uses a particular this

```
function foo(el) {
    console.log( el, this.id );
}

var obj = {
    id: "awesome"
};

// use `obj` as `this` for `foo(..)` calls
[1, 2, 3].forEach( foo, obj ); // 1 awesome  2 awesome  3 awesome
```

13.13. New binding

- In JavaScript constructors are just functions that happen to be called with the new operator in front of them
- They are not attached to classes, nor are they instantiating a class
- They are not even special types of functions
- They're just regular functions that are, in essence, hijacked by the use of the new keyword in their invocation

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );

console.log( bar.a ); // 2
```

13.14. Function constructor

- These functions are called function constructors
- Invoked with the new keyword
- They have nothing to do with classes!

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

var john = new Person('John', 'Smith');

console.log(john.firstName); // John
```

13.15. Determining this

- Let's summarize the rules, in order of precedence, for determining this from a function call's call-site
- Is the function called with new binding?
  - var bar = new foo()
- Is the function called with explicit or hard binding?
  - var bar = foo.call(obj)
- Is the function called with implicit object binding?
  - var bar = obj.foo()
- Otherwise, use default binding...
  - var bar = foo()

13.16. Lexical this

- ES6 introduces a special kind of function that does not use these rules
- Arrow-functions are not defined by the function keyword but by =>
- This is the so called "fat arrow" operator
- These functions adopt the this binding from the enclosing scope
- The lexical binding of an arrow-function cannot be overridden!

```
function foo() { // returns an arrow function
    return () => {
        console.log(this.a); // `this` here is lexically adopted from `foo()`
    };
}

var obj1 = { a: 2 };
var obj2 = { a: 3 };

var bar = foo.call(obj1);

bar.call(obj2); // 2, not 3!
```

13.17. Arrow-functions as callbacks

- Arrow-functions created inside a function lexically capture whatever this is at call-time of the enclosing function
- The most common use-case will likely be in the use of callbacks
- Such as event handlers or timers

```
function foo() {
    setTimeout(() => {
        // `this` here is lexically adopted from `foo()`
        console.log(this.a);
    }, 100);
}

var obj = {
    a: 2
};

foo.call(obj); // 2
```

13.18. Pre-ES6 pattern to solve this problem

```
function foo() {
    var self = this; // lexical capture of `this`

    setTimeout(function() {
        console.log(self.a); // closure
    }, 100);
}

var obj = {
    a: 2
};

foo.call(obj); // 2
```

14. Next Generation JavaScript

14.1. What is ES6?

- ES6 or ES2015 is an important revision of the ECMAScript language
  - It is the start of a new era of changes to improve the language
- It is the intention of ECMA to release a new update every year
  - ES7 in 2016 (ES2016)
  - ES8 in 2017 (ES2017)
  - ...
  - ES-Next
- On its own ES6 was superseded quickly by ES7
  - The point is that ES6 is the start of a new era

Remember that the language which we commonly call "JavaScript" is actually an implementation of ECMAScript.

ECMAScript is standardized by ECMA (European Computer Manufacturers Association). ECMA intends to release a new version of ECMAScript yearly in the next couple of years, so ES7 and ES8 are already there to take over from ES6. The reason why ES6 is especially interesting is because it is the beginning of a new era of improvements of ECMAScript, that will help to get rid of it's bad legacy language design choices. Actual implementations will add support for these features on an individual level and much more gradually.

On wikipedia (https://en.wikipedia.org/wiki/ECMAScript#Versions) you can easily grasp the relatively long silent period in the language, and why this renewed energy is important.

- It is clear that JavaScript has some serious historical flaws
  - These flaws can never be removed from the language
    - It would break the web
- To make the platform better, a different strategy needs to be taken
  - Alternative mechanisms will be provided
    - These will in time replace the old (bad) parts
- In other words: JavaScript will evolve slowly over time
  - This has some disadvantages as well though
    - The status-quo will always be in the suboptimal middle

It is impossible to fix the problems of JavaScript (and HTML and CSS for that matter) by creating a new version of the specification, and replacing it entirely because:

- It would break millions of (old) pages around the web, or
- Developers would have to throw away existing expertise and relearn the new semantics.

A different strategy is therefor necessary. One that does not break the web and does not cause all know-how to become obsolete. For this reason, a much more gentle and phased approach is taken. Existing features will remain in place. In time, better alternatives will become available. These alternatives will become dominant, and the old (bad) features will be used less and less, until eventually only the replaced features remain.

A very good example for this is the new `let` keyword, which will - in time - eliminate the need for `var` entirely, and with it some very bad features of `var` (namely hoisting and the lack of block scope).

The downside of this is that browsers will gradually introduce new features of ES6, which will cause the adoption rate to be slower (since developers still need to support older browsers). Currently no JavaScript platform has full support for ES6 out-of-the-box.

14.2. Feature Overview

- These are the most important features of ECMAScript 6

Feature	Feature
Constants	Destructuring assignments
Let	Module system
Block scope	Classes
Arrow functions	Iterators
Lexically scoped this	Generators
Spread operator	Standardized Promises
Default parameters	Runtime library extensions
Binary and octal literals	Reflection
Variable interpolation	Proxies
Symbols	

Most of these features are examined throught the rest of the course.

14.3. Web Assembly and ES-Next: The future

- ES7 (aka ES2016), ES8 (aka ES2017), ...
  - ES6 has been designed partly to set the stage for these future versions
  - They will become the JavaScript of the future
- Web Assembly
  - Will support other languages / VMs than JavaScript
  - Based on the work by ASM.js (from Mozilla)
  - Expected to grow into a form of "byte code" for the web

An example of where the features introduced in ES6 are actually preparing for what comes next are generators and promises. These two features are designed from day one to eventually be combined in ES7's async/await feature, which will solve the infamous asynchronous callback hell.

Because JavaScript in its current form lacks support for large code bases (no module mechanism, etc...) the trend is to write programs in a different language (C, Java, TypeScript, Dart, Clojure, Coffeescript, ...) and compile this to ES5. In the future the platform will have a dedicated mechanism to support this through byte code. This byte code will start out as a subset of JavaScript syntax itself (as is the case in ASM.js), but is expected to evolve into a more binary form. This approach does not require browsers to implement an entirely new virtual machine engine. They can simply keep using their existing JavaScript engines.

14.4. Proposal Stages and TC39

- ECMAScript changes are managed by TC39 (Technical Committee 39) of ECMA
  - New features go through several stages of validation
- There are five stages
  - Stage 0: "Strawman"
    - Allows input into the specification
  - Stage 1: "Proposal"
    - Strawman proposals that are deemed "champions". Allows for better motivation
  - Stage 2: "Draft"

    - Stabilization of syntax
  - Stage 3: "Candidate"
    - Requests for comments by implementations
  - Stage 4: "Finished"
    - Ready for standardization

Each feature in ECMAScript is carefully evaluated. Proposals that "survive" up to stage 2 are likely to be accepted into the language. More information about the technical procedures involved in designing new language features can be found on https://tc39.github.io/process-document/

When working on the bleeding edge of web development, you have to be careful to steer away from experimental features. The ECMAScript strawman proposals for example are not always retained in the final version of the specification, but browsers sometimes already have a preliminary implementation. This is often the case when a browser vendor is a supporter of a given feature, they would allow these features to be used in order to appeal to the public.

Outside the strict boundary of ECMAScript this is also the case. For example, some HTML5 proposals that never made it in the final version of the specification are File API, Microdata and Web SQL. Many books exist though that boast these features as being new in the specification (they were written in a time where there was high probability that these features would end up in the specification).

14.5. Execution Environments

- Overview of the most popular (modern) ECMAScript engines

Engine	Used by
V8	Node.js, Chrome, Opera, Edge
Chakra	Internet Explorer (9+)
Spidermonkey	Firefox
Nitro	Safari

- To date, none of these support all ES-Next features
  - Engines are gradually adding support for all features in the near future.
    - http://kangax.github.io/compat-table/es6/
- We currently can not execute ES-Next code directly!
  - But we can solve that problem today!

Currently there are no ECMAScript engines that have full support for ES-Next. Some do have some features, and some have similar features that are not entirely specification compliant. Also, some engines have these features implemted but are disabled by default. This often requires strict mode to be enabled and/or specific configuration flags to be turned on.

14.6. Transpilers

- To be able to use ES-Next today, we can use a "transpiler"
  - Compile ES-Next code to ES5 or ES6 code
    - ES5 support is widely available!
- Debugging is possible using source maps
  - These map the compiled output back to the ES-Next source
  - Are widely supported by all browsers

        - Allows placing breakpoints
- When engines start supporting features out of the box, we can simply turn off this step

By using a transpiler we can use ES-Next features today that are not supported yet by browsers and other engines. In the future these features will become natively supported, we can disable transpilation for them.

This approach also offers backwards compatibility with older browsers that will never support ES-Next.

- These are two popular ES-Next to ES5 transpilers:
    - Traceur
        - https://github.com/google/traceur-compiler
    - Babel
        - https://babeljs.io/

15. New Features in ES6

- Constants make reassignment impossible
  - Use them instead of `let` or `var`

```
const ANSWER = 42;
ANSWER = 1507;      // Not allowed (TypeError)!
```

- Note that they do not make references immutable

```
const person = {
    name: 'Jimi Hendrix'
};

person.name = 'Janis Joplin';    // Still ok
```

Constants are also called "immutable variables".

Many ES environments already support constants to some degree. You may be able to use it in your browser if you enable strict mode (`"use strict";`). Not all environments currently support const in all places, such as inside a for loop.

On Node.js this is enabled by default, where it is very often used to import libraries using require:

```
const fs = require('fs');
```

15.1. Let

- In ES6 you can use the `let` keyword instead of `var`
  - Let offers block scoping which var doesn't have

```
function f(x) {
    if(x == 5) {
        let v = 1507;
        console.log(v); // Will output 1507
    }

    console.log(v); // ReferenceError: undefined!
                    // With 'var' would output 1507
}

f(5);
```

- This fixes a time-old issue with `var` not behaving the way most developers think it would
  - Causing many bugs

The `var` keyword used in JavaScript has a history of inducing bugs. Not due to the fact that it is inherently wrong, but due to the fact that it does not behave like in most other languages. Thus developers write bugs because of this enexpected behaviour.

Together with `let`, JavaScript now supports a true `block scope`, meaning that variables are pushed and popped on the stack on the boundaries of a block statement: `{}` (like an `if`, `while`, …). `var` does not have this. It only has function and global scope.

Although they can never remove `var` from the specification in the near future, it is expected that in time `let` will replace `var` everywhere. You are this encouraged to start using `let` exclusively, and treat `var` as a legacy feature.

Note that this also works in a for statement:

```
let data = ['a', 'b', 'c'];
for(let i=0; i < data.length; i++) {
  console.log(data[i]);
}
console.log(i);    // Still exists with var, but not with let
```

15.2. Block Scoped Functions

- Functions can be declared block scope

```
function outer() {
    function inner() { console.log('Inner'); }
    inner();

    {
        function inner() { console.log('Nested inner'); }
        inner();
    }
    inner();
}

outer();
```

```
Inner
Nested inner
Inner
```

It is now possible to declare a function within a block, and have that function exist only inside that block. The example shown demonstrates the use of this through hiding if the `inner()` function.

15.3. Arrow Functions

- You can now use a shorter syntax to define a function
    - Also popularly known as "lambda expressions"

```
var f = (x) => {
    let y = x * 2;
    console.log('The answer is', y);
    return y;
};
var y = f(1507); // The answer is 3014
```

- Even shorter if the function is just one statement

```
var f = (x) => x * 2;
console.log(f(5)); // 10
```

- Very convenient as a callback

```
['a', 'b', 'c'].forEach((index, item) => {       // Functional iteration (for function)
    console.log('Item ' + index + ' = ' + item);
});
```

Adding support for labmda expressions is an extremely popular addition to many languages, and ECMAScript is not different. Actually, in ECMAScript it is even more convenient given the already highly functional nature of the language. The need for adding the `function` keyword everytime is very annoying, and is now solved in a very elegant way.

Arrow functions are actually originating from the mathematical domain of "lambda calculus", and the associated. The often associated way to pass a lambda expression as a parameter to another function (as in the example) is also inspired from mathematics, where a function that received another function as a parameter (Note: not the result of that function) is called a higher order function.

15.4. Lexically scoped 'this'

- Arrow functions have a lexically scoped `this`

```
let person = {
  name: 'Jimi Hendrix',
  print: function() {
    // Nested function using lambda syntax has this set to person object, rather than Global
    let getName = () => {
      return this.name;
    }
    console.log('Person:', getName());
  }
};

person.print();
```

- Prevents the need for subsitute this variables

```
var thiz = this;   // Also often seen: 'self', using this as a closure

function f() {
    thiz.doSomething();
}
```

Working with the `this` reference in JavaScript has always been tricky because of two reasons.

- It can be changed to whatever you want
- Functions sometimes receive a variable bound to `this` that is not convenient nor is expected

The first item is part of the expressive nature of the language, and is a good thing. The second item however is generally considered to be a design flaw. Inner declared functions have their `this` bound to the global scope, which is horribly bad. Now that ECMAScript 6 introduces better scoping rules, this evil can finally be resolved by binding the `this` of an inner function to the contextual object rather than the global scope.

The practical benefit that we have from this is that it will in many cases no longer be necessary to use a temporary `this` substitute like in this example:

```
function Animal(name) {
    this.name = name;

    var self = this;    // Self is awkward substitute for this

    function helper() {
        console.log(self.name);
    }

    helper();
}

var a = new Animal('Shere Khan');
```

Of course, this can only be in the new syntax (otherwise we would break the web), and thus you have to use an arrow function for this to become available:

```
function Animal(name) {
    this.name = name;

    var helper = () => {
        console.log(this.name);
    }

    helper();
}

var a = new Animal('Shere Khan');
```

15.5. Default parameters

- Functions can now have default parameters

```
let action = (name = 'Jimi', what = 'playing guitar') => {
  console.log(name, 'is', what);
};

action('Jim', 'eating a sandwich');    // Jim is eating a sandwich
action('Janis');                    // Janis is playing guitar
action();                            // Jimi is playing guitar
```

Very useful feature combined with JavaScript's existing feature of optional parameters.

This will eliminate the need for lazy comparisons like this:

```
let action = (name, what) => {
    let n = name || 'Jimi';
    let w = what || 'playing guitar';
};
```

15.6. Rest Parameters

- It is now possible to use vararg parameters to capture all remaining parameters.
  - This is different from the `arguments` array as this would contain all parameters.

```
let person = function(name, ...aliasses) {
  console.log('Person name:', name);
  for(let i in aliasses) {
    console.log('Alias:', aliasses[i]);
  }
};

person('Robbert', 'Robbie', 'Bob', 'Mr. Pink');
```

This is a feature often called "variable arity functions" or "variadic parameters". Also known from languages like C, C++, Java, etc...

15.7. Tail Call Optimization

- ECMAScript 6 requires all engines to optimize tail calls
  - For purely-functional languages a rather important feature

```
function factorial(n, product = 1) {
    if (n <= 1) {
        return product;
    }
    return factorial(n - 1, n * product);   // Tail call!
}
console.log(factorial(5)); // 120
```

- Reduces burden on the call stack
  - Function in tail position can execute on the last stack frame, rather than pushing a new one

In ES5 it was never a requirement that tail calls were optimized. Although it was left to the implementation to do this if they pleased, most engines did not. This was due to some complications the way JIT compilers treat the dynamic features of the language.

Starting from ES6 though, proper tail call optimization is mandatory. This will ensure much better performance of highly recursive algorithms. This particular feature can not be solved using a transpiler though, so until browsers actually support this you'll have to sit it out.

15.8. Spread operator

- The Spread operator is similar to Rest, but for iterables (see later) rather than functions
  - Iterables include `Array` and `String`

```
let extras = ['d', 'e', 'f'];
let values = ['a', 'b', 'c', ...extras];
console.log(values);    // a,b,c,d,e,f

let message = 'Hello World';
let characters = ['X', 'Y', ...message];
console.log(characters);    // X,Y,H,e,l,l,o, ,W,o,r,l,d
```

- It allows combining arrays more elegantly

This used to be done by using the `concat()` function.

15.9. Template String Interpolation

- Using backticks and `${...}` expressions you can now use variable interpolation in strings

```
let name = 'Jimi';
let message = `Hello ${name}, welcome to my temple!`;
console.log(message);
```

- Also works with objects

```
let person = {name: 'Jimi', age: 27};
let message = `Hi ${person.name}, you are ${person.age} years old.`;
console.log(message);
```

- And even calculations

```
let jimi = 27;
let richard = 63;
let message = `Your combined age is ${jimi + richard}`;
console.log(message);
```

Variable interpolation is a feature some other dynamic languages also have. One example of such a language is PHP.

- Backticks also allow multi-line strings to be created

```
console.log(`This is a very
very long string...`);
```

- Interpolation can even be customized with an arbitrary function
  - This is called a "tag function"

```
let name = 'Jimi';
let age = 27;

function tag(strings, ...values) {
  let res = '';
  res += strings[0];               // "name: "
  res += values[0].toUpperCase(); // "JIMI"
  res += strings[1];               // " age: "
  res += values[1];                // "27"
  return "Yippie " + res;
}

let message = tag`name: ${name} age: ${age}`;
console.log(message);                // Yippie name: JIMI age: 27
```

With the new backtick support, ECMAScript now effectively has three ways to declare strings, with the backtick-approach clearly being the most powerful.

Tag functions allow a template string to be processed in any way the function defines. This is extremely powerful, but somewhat confusing. By prefixing the backticked string with a tag, a function with the same name is invoked with parameters mapped to the literal pieces as an array, and then all following expression values. This last piece can be used as a rest parameter to capture a variable number of expressions into a single array as in the example. Additionally a tag function has a `raw` property (not shown) which allows the pieces of a tagged string literal to be exposed in unprocessed form (even the escape `\` characters will not be processed here yet). All in all, the backtick string processing allows fully customized string parsing.

The customization of interpolation by means of a tag function can be extremely useful in frameworks that offer template evaluation, such as Angular.js and Handlebars.js. It is expected that these frameworks will in time integrate with this new feature.

15.10. Radixed Literals

- It's now possible to define numbers not only as decimal and hexadecimal, but also binary and octal.

```
let b = 0b1101011010;
let o = 0o737;
```

- The parse functions also accept binary and octal radix

```
let b = parseInt('01011101010', 2);
let o = parseInt('76547254', 8);
```

15.11. Better Object Literals

- Object literals that have the same name-value can be written in more compact form

```
var firstName = 'Jimi', lastName = 'Hendrix';
let oldly = { firstName: firstName, lastName: lastName };
let newly = { firstName, lastName }; // Less repetition
```

- Object keys can now be computed in literal notation

```
let cap = (s) => s.charAt(0).toUpperCase() + s.slice(1);
let o = {
  ['first' + cap('name')]: 'Jimi'
};
console.log(o.firstName); // Object has property 'firstName'
```

Object are often created out of variables that have the same name as the keys of the object. This can now be defined in a much more compact way. Here is another example:

```
let point = (x, y) => {
    return {x, y};
}
```

Previously the keys of objects could be dynamically calculated, but not in an object literal. It had to be done in two statements like this:

```
var o = {};
o['first' + cap('name')] = 'Jimi Hendrix';
```

- Functions can be added more directly

```
let person = {
  firstName: 'Jimi',
  lastName: 'Hendrix',
  name() {  // No function keyword
    return this.firstName + ' ' + this.lastName;
  }
};
console.log(person.name());    // Jimi Hendrix
```

- No more need for `function` keyword

This promotes a less verbose style for object oriented applications. Previously this was as follows:

```
let person = {
  firstName: 'Jimi',
  lastName: 'Hendrix',
  name: function() {
    return this.firstName + ' ' + this.lastName;
  }
};
console.log(person.name());    // Jimi Hendrix
```

15.12. Classes

- Introduction of classes in JavaScript is a highly disputed subject
- Pro:
  - It adds familiarity and convenience for OO developers
- Contra:
  - Prototype-objects already provide a dynamic OO construct
- The implementation of classes in JavaScript accepts both arguments
  - Classes are implemented as nothing more than syntactic sugar over the already existing prototype-objects
- Using a very

When searching the internet for arguments on the development of classes for ECMAScript, there are some interesting discussions to be found. Some die-hard JavaScript developers claim that JavaScript does not need classes, because everything can already be done using the prototypical nature of JavaScript. Anyone who would not recognize this does not "know how to use JavaScript". On the other hand, proponents of classes argue that regardless of JavaScript's flexible OO implementation, classes are a construct that you see in virtually all OO programming languages. It is therefore a matter of practicality to also have it in JavaScript because it would greatly reduce porting effort (of both code and skills).

There are some design patterns used in JavaScript that allow to create classical inheritance structures using the existing JavaScript syntax. This subject is actually covered extensively in our course "JavaScript: The Good Parts" based on the equally named book by Douglas Crockford (see https://education.realdolmen.com/en/Course/JGP100). These patterns are arguably not very elegant:

```
function Animal(name, age) {
    this.name = name;
    this.age = age;
}
```

```
Animal.prototype.sound = function() {
    console.log('Animal ' + this.name + ' is making sound!');
};

function Cat(name, age, lives) {
    Animal.call(this, name, age);
    this.lives = lives;
};

Cat.prototype.scratch = function() {
    console.log('Cat ' + this.name + ' scratches you. Nasty!');
};

var cat = new Cat('Missie', 5, 8);
```

What classes will do is create the same structure as this, but in a much more syntactically friendly way, using methods, constructors, inheritance, etc...

- Simple classes can be made as follows:

```
class Animal {  // Or: let Animal = class { ... }
    constructor(name, age, type) {  // Constructor
        this.name = name;
        this.age = age;
        this.animalType = type;
    }

    get type() {                    // Getter for property.
        return this.animalType;
    }

    set type(t) {                   // Setter for property.
        this.animalType = t;
    }

    sound() {                       // Method
        console.log('Animal', this.name, 'is making sound.');
    }
}
```

This example shows all the typical simple features of classes:

- Constructors
- Methods
- getters and setters

Getters and setters are used in JavaScript in the same way as true properties are in C#. You invoke them using propery access, not by calling a function.

Although this is called a class, it is still a dynamic object based on prototypes, so do not confuse this with the classica static type inheritance you may know from languages such as Java, C++ or C#.

- Inheritance is also implemented in a recognizable way

```
class Cat extends Animal {
    constructor(name, age, lives) {
        super(name, age, 'cat');
        this.lives = lives;
    }

    sound() {   // Override!
        console.log('Cat', this.name, 'is making sound.');
    }
}
```

- Instantiation of classes

```
var a = new Animal('Shere Khan', 26, 'tiger');
console.log(a.name, a.age, a.type);
```

```
a.sound();

var c = new Cat('Missie', 5, 8);
console.log(c.name, c.age, c.lives);
c.sound();
```

Inheritance is done using the `extends` keyword. Super constructors can be called using `super()` and overriding of methods is by simply redefining them. In ECMAScript overloading is not applicable due to the dynamic nature of the language.

Instantiating is done using the familiar `new` keyword. This is noteworthy because some influential authors have made a case for not using the `new` operator. Some of these have actually influenced the design of ES5. (Douglas Crockford and the introcution of `Object.create()`).

- Static methods are supported using the `static` keyword

```
class Cat {
    constructor(name) {
        this.name = name;
    }

     static defaultLives() {
        return 9;
    }
}
```

- Can be used like this

```
console.log(Cat.defaultLives());    // Yes
console.log(c.defaultLives());      // No, TypeError!
```

Static methods are methods that belong to all instances together rather than belonging to a single instance. It is shared.

Note that is is not possible to call a static method from an instance. This is unlike some other OO languages.

JavaScript's classes are implemented as prototype objects, so multiple inheritance is not possible the 'standard way'. However support for mix-ins is possible, if the 'superclass' specifically prepares for this. This is a very typical way to 'simulate' a multiple-inheritance scheme.

- Multiple inheritance is not supported
  - But there are some advanced tricks you can do do implement mix-ins

```
let Bridge = Base => class extends Base {
    evasiveManouvres() { console.log("It's a trap! Begin evasive manouvres!"); }
};
let LasersCanons = Base => class extends Base {
    fireLasers() { console.log("Concentrate all fire on that Super Star Destroyer!"); }
};
let FighterBays = Base => class extends Base {
    launchFighters() { console.log('Launch the X-Wings!'); }
};
class Starship {
    constructor(captain) { this.captain = captain; }
}
class MonCalamariCruiser extends Bridge(LasersCanons(FighterBays(Starship))) {
    constructor(captain) { super(captain); this.side = 'rebels'; }
}
let c = new MonCalamariCruiser('Admiral Ackbar');
console.log(c.captain, c.side);
c.evasiveManouvres();
c.fireLasers();
c.launchFighters();
```

The creation of the MonCalamariCruiser class you should read as 'Define class MonCalamariCruiser which extends from Starship with additional Bridge, LaserCanons and FighterBay'.

For more information about how to use these mix-ins, check out Mozilla's developer pages: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes#Mix-ins

15.13. Destructuring Assignment

- The destructuring assignment is an expression that can extract data from arrays and objects into variables.
- Here are some examples for arrays

```
var array = ['Jim', 'John', 'Janis', 'Jonas'];

// As assignment
let a, b;
[a, b] = array;
console.log(a, b); // Jim John

// As declaration
let [c, d, e] = array;
console.log(c, d, e); // Jim John Janis

// Defaults for undefineds
let [f, g, h, i, j='George'] = array;
console.log(f, g, h, i, j); // Jim John Janis Jonas George

// Ignoring values
let [,k,,l] = array;
console.log(k, l);    // John Jonas
```

The best way to explore this piece of new syntax, is by looking at many examples. The examples here are only for arrays.

- It is also possible to use destructuring assignment with objects:

```
let person = {firstName: 'Jimi', lastName: 'Hendrix', age: 27};

// In declarations
let {firstName, lastName, age} = person;
console.log(firstName, lastName, age);    // Jimi Hendrix 27

// Renaming variables
let {firstName: a, lastName: b} = person;
console.log(a, b);    // Jimi Hendrix

// In assignments
let c, d;
({firstName: c, age: d} = person);    // Note () are mandatory!
console.log(c, d);    // Jimi 27

// Using defaults for undefineds
let {instrument: e = 'guitar'} = person;
console.log(e);

// Deep destructuring
let animal = { name: 'Fifi', type: 'rancor', owner: { name: 'Jabba The Hutt' } }
let {owner: {name: p}} = animal;
console.log(p);
```

More examples exclusively with objects. The mechanism is similar, but uses curly braces instead. Remember that arrays in JavaScript are really objects with indexes as the keys, so it is not surprising that this works similarly.

- Destructuring assignment is a very powerful feature, allowing all sorts of interesting applications:

```
// Interesting use to swap variables
[a, b] = [b, a];
console.log(a, b); // John Jim
```

```
// Interesting use to import features from modules
const {read, write} = require('fs');
read();

// Interesting use: function parameters
let drawBorder = function({size = '1px', style = 'solid', color = 'red'} = {}) {
    console.log(size, style, color);
};
drawBorder({color: 'green'});    // 1px solid green
```

- They will also be used in combination with ES6 modules

Destructuring assignments is one of the new gems of ECMAScript. It has been carefully crafted to go hand in hand with other new features, such as modules, new object literals and the spread operator. This will surely become one of the directly 'visible' changes of ES6.

15.14. Modules

- ES6 provides a mechanism for loading modules without global namespace pollution
  - This is a huge deal since the lack of a module system has been a critical weakness of JavaScript!
  - Problematic for large code bases
- The problem used to be solved somewhat clumsy by means of JavaScript's expressiveness
  - Immediately invoking functions
  - Closure

It can not be stressed enough that the introduction of a proper module system in ES6 is a huge improvement to the language, since it has been one of - or even the - most critical weakness of the language. The truth is that JavaScript was never desinged to be the language it has become. Brendan Eich never foresaw that it would become a language in which large applications were written. For this reason a module loading system was never created, and the global namespace was used. There are some who to this day think JavaScript is not a productive language for this reason; an argument that certainly has its merit.

Thankfully JavaScript is a very expressive language, and there are ways to circumvent this problem, using closure and immediately invoking functions. Here is an example of an artificially created module:

```
var realdolmen = (function() {
    var exports = {};

    var privateVariable = 1507;

    var privateFunction = function() {
        console.log('Goodbye');
    }

    exports.publicVariable = 42;

    exports.publicFunction = function() {
        privateFunction();
        console.log('Hello');
    }

    return exports;
})();
```

This example uses closure and an IIF to isolate variables and prevent global namespace pollution. It is expressive, but not at all syntactically beautiful. Often this is combined with a module loader such as require, AMD or common js. The module loaders could deal with removing some of the jitter such as the IIF.

```
var rd = require('realdolmen.js');
rd.publicFunction();
```

In the next section, we will explore how ES6 solves and standardizes this problem.

- Modules are implemented using two keywords:
  - export
    - Marks symbols that need to be made available to other modules
  - import
    - Selects what symbols are to be made available from another module
- A module is simply another `.js` file
  - There is no keyword named `module`
- Defining a module and its exported symbols

```
// realdolmen.js
export let sum = (a, b) => {                   // Export lambda
    return a + b;
};

export function product(a, b) {                // Export old-style function
    return a * b;
}

export let person = {                          // Export variable
    firstName: 'Jimi',
    lastName: 'Hendrix'
};

let a = 1, b = 2, c = 3;
export {a, c};                                 // Named exports

const ANSWER = 42;                             // Not exported!
export const PI = 3.14;                        // Export constant

export default function(firstName, lastName) {  // Only one default export can be defined
    return { firstName, lastName };
};
```

Using the `export` keyword, it becomes possible to determine what symbols (functions, variables, constants, ...) are made available to other modules. This offers module level privacy. Note here that the systax is much more elegant that using an IIF.

There is one special type of export: the default export. This can be used only once, and will be useful if there is a primary export that would in most use cases be the only symbol another module is interested in. This will allow the importing side to use a more compact syntax.

- Any module can be imported from a different script

```
// importer.js
import p from './realdolmen.js';               // Importing (the one and only) default export as 'p'
console.log(p('Jane', 'Doe'));

import * as rd from './realdolmen.js';         // Importing everything as an object called 'rd'
console.log(rd.PI);
console.log(rd.product(5, 8));

import pers, {c} from './realdolmen.js';       // Import only default export as 'pers' and c
console.log(pers('Jane', 'Doe'));

import {sum, product} from './realdolmen.js';    // Import only selected items
console.log(sum(2, 2), product(2, 2));

import {sum as s} from './realdolmen.js';      // Import as alias
console.log(s(3, 5));
```

- These imports are static
  - Not allowed to be used inside a function or if-statement

As you can see, imports are very flexible. You have fine-grained control over what symbols are imported, and how (as an object, using a specific name, ...).

The wild card approach allows importing all exported symbols as an object.

Note also that this looks very similar to a destructuring assignment, but is not entirely (see the `as` keyword).

One disadvantage of this style of imports is that it is static. It is not allowed to use this inside a function or a conditional statement. Thus you can not dynamically load on-demand. For this you will need a dynamic module loader. This is a deliberate design choice: it allows more rigorous compile time checking of imports, which is a typical requirement for large code bases.

15.15. Symbol

- Symbols are unique entities to be used as object keys
  - No two Symbols are the same, even if they have the same name

```
var s = Symbol("RealDolmen");
var t = Symbol("RealDolmen");
console.log(s === t); // False!

var o = {};
o[s] = () => {
    console.log('Unique feature');
};

o[s]();
console.log(o);
```

- How is this useful?
  - It allows you to add features that can never conflict with past and or future features that coincidentally have the same name
    - Currently a source of interop problems when combining libraries

From an abstract perspective, a Symbol is a guaranteed-to-be-unique entity that is to be used as the keys for objects. This allows objects to use these Symbols to add functions and state to existing objects, without resorting to a classic string-key.

Since no two Symbols are the same, you can never run into the type of conflict where two libraries are coincidentally using the same string to enhance a shared object (Array or Function for example). This is a typically occurring conflict with libraries today, and limits portability. Also it puts a limit on new additions to JavaScript, because sometimes a library had already added a key that has the same name as a new ECMAScript feature, this breaking interoperability with the library and ESx. Using Symbols, ECMAScript can freely add new features without this ever conflicting with anything anyone has already done.

- Let's take a more practical example

```
// Breaks if someone else decides to add a a method with the same name!
Array.prototype.find = function() { console.log('My version of find()'); };
[1, 2, 3].find();

// Less likely to break. Not very nice though.
Array.prototype["j16q5n31rk"] = function() { console.log('My version of find()'); };
[1, 2, 3].j16q5n31rk();

// Solution is to use a Symbol! Never breaks anything... ever!
let find = new Symbol('find');
Array.prototype[find] = function() { console.log('My version of find()'); };
[1, 2, 3][find]();
```

- An number of well-known Symbols are also provided out of the box for ES6 features

Symbol	Symbol	Symbol	Symbol	Symbol
Symbol.iterator	Symbol.replace	Symbol.split	Symbol.species	Symbol.for
Symbol.match	Symbol.search	Symbol.unscopables	Symbol.toPrimitive	Symbol.keyFor

This example shows how it would be a problem if ES6 adds a `Array.prototype.find()` method. By simply existing, it breaks any library that had previously added an (incompatible) version of that method themselves. You could of course resort to hackish ways to create a name for your feature that 'nobody else would', such as a randomly generated string. This not neither nice nor maintainable (imagine having to debug that sort of code). So, a better way needed to be designed. ES6 Symbols solves that problem.

Actually Symbol is the seventh type that ECMAScript now has (next to Number, Boolean, String, Object Null and Undefined).

ES6 makes use of these Symbols a lot for adding new features such as the for-of construct or adding new functions to the global objects. These Symbols are available on the Symbol object, and can be used by developers as well (to support iteration for example).

15.16. Iterators

- A convention for iteration (called a protocol) has been added
  - Allows arbitrary iteration over any object that is iterable
    - Iterable means: has a function with name 'Symbol.iterator'

```
let gems = ['Diamond', 'Ruby', 'Emerald'];
let it = gems[Symbol.iterator](); // Arrays are iterable: they have a function "Symbol.iterator"
for(let entry = it.next(); !entry.done; entry = it.next()) {    // Cursor-style iteration
    console.log(entry.value);
}
```

- There is also a new for-of statement specially for iterators

```
// Equivalent to previous code block!
let gems = ['Diamond', 'Ruby', 'Emerald'];
for(let gem of gems) {  // Note for-of (this one is actually SAFE to use, unlike for-in)
    console.log(gem);   // Gem is a value, not an index (unlike for-in)!
}
```

- Since iterators are a protocol you can build your own objects with custom iteration!

It is known that JavaScript has some issues with safe iteration over objects. The for-in loop was dangerous and iterates over the indices rather than the values as shown here:

```
Array.prototype.satan = 666;

var gems = ['Diamond', 'Ruby', 'Emerald'];

gems.lucifer = function() {
    console.log('It\'s the devil!');
};

for(var i in gems) { // Note: for-in
    // Needed to avoid satan and lucifer!
    if(gems.hasOwnProperty(i) && typeof gems[i] != 'function') {
        console.log(gems[i]);
    }
}
```

This style of code is no longer necessary with an iterator.

Note that an iterator uses a `Symbol` for this, so it can never conflict with anything from before! See the section about

Symbols for more details on this.

Imagine a protocol is sort of like an interface from Java, C++, C#, but without the contract enforcement.

15.17. Generators

- Generators are pause-resumable functions
    - Pause is done using `yield` keyword
- A simple generator looks like this:
    - Recognizable by the star (*)

```
function *g() {    // Generators are (rock)stars!
     yield 1;
     yield 'John';
    return true;
}

let it = g();  // It returns an iterator. Does not execute g!
do {
    var e = it.next();
    console.log(e.value);
} while(!e.done);
// Output: 1 John true
```

A generator is a method that automatically returns an iterator. This iterator can then be traversed to control the execution of the generator step-by-step. A generator feeds values into the iterator by `yielding`. Finally when the generator returns normally using `return` the iterator will stop.

- A generator can also accept feedback from the iterator
    - It is bidirectional

```
function *g() {
     let feedback = yield 5;
    console.log(feedback);  // Will print 15
}

let it = g();
var input = it.next();
console.log(input.value);   // Will print 5
it.next(input.value * 3);   // Feed back 5 * 3 to the generator
// Normally loops until done, but not in this example.
```

- You can see a generator as a form of program flow-control

Generators can not only provide data to the iterator, but also receive feedback from the iterator. This allows some very powerful constructs to be created. If you find this interesting, take a look at Task.js (http://taskjs.org/) for some mind blowing use cases of generators.

- Generators are perfect to use as the iterator function of an custom iterable object

```
class Counter {
    constructor(from, to) {
        this.from = from;
        this.to = to;
    }

    *[Symbol.iterator]() {     // The star means it's a generator!
        let current = this.from;
        while(current <= this.to) {
            yield current++;
        }
    }
}
```

```
for(let v of new Counter(10, 15)) { // Python flavour (range)!
    console.log(v);
}
// Output: 10 11 12 13 14 15
```

This example of an ES6 class uses a generator to allow itself to be iterated over by the for-of construct. Alternatively it is perfectly possible to manually traverse the iterator, but it is less convenient in this example.

15.18. Sets and Maps

- The Set object allows storing unique and ordered values

```
let s = new Set();
s.add('one');
s.add('two');
s.add('one');

console.log(s.size);        // 2 (not a method!)
console.log(s.has("two"));  // True
for(let v of s) {           // Iterable (note for-of)
    console.log(v);
}
```

- Interesting methods and properties

Method	Purpose
size (property)	The amount of values
add(v), delete(v), clear()	Add or remove values
entries(), values(), keys(), @@iterator() (Symbol)	Retrieve values as Iterators
forEach(cb)	Functional traversal
has(v)	Check if a value exists

Although JavaScript's object (and arrays) are very versatile data structures that can be used to support various abstract data types such as lists, maps, stacks queues, dequeues etc... some cases are not covered elegantly.

For example you can use an object as a Set (a data structure that only keeps unique values) by exploiting the keys of the object as values, but this has some limitations:

- You can not store arbitrary data types (object keys are strings)
- You can not retain insertion order when looping over them.

```
// ES5
var s = {};
s["one"] = true;
s["three"] = true;
s["two"] = true;
s["one"] = true;

console.log(Object.keys(s).length); // 3
for(var v in s) {                   // Order not guaranteed
    console.log(v);
}
```

The Set datatype now solves that problem by providing a true Set datatype that only stores unique values and keeps it's order of insertion as well.

- Similarly the Map object allows cleaner operations of key-value pairs

```
let m = new Map();
m.set('Jeniffer', 40);          // Not put()!
```

```
m.set('Bob', 32);
m.set('Jaimie', 16);

console.log(m.get('Bob'));      // 32
for(let e of m) {               // Iterable
    console.log(e[0], e[1]);    // Values are [key, value]
}
```

- Some interesting methods and properties

Method	Purpose
size (property)	The amount of pairs
set(k, v), get(k), delete(k), clear()	Add or remove pairs
entries(), values(), keys(), @@iterator() (Symbol)	Retrieves values as Iterators
forEach(cb)	Functional traversal
has(k)	Check if a pair exists

The Map object also allows to work with key-value pairs in a much clearer way. Granted objects in JavaScript really are a type of associative map, but having a true `Map` datatype is a little more convenient.

- Both Set and Map have a Weak counterpart

```
var ws = new WeakSet();
var wm = new WeakMap();
```

- Weak datatypes have some limitations
  - Can not be iterated over
  - Values can disappear from the list (but that's what they are meant for)
- Use cases involve smart-references and data tracking

The difference between a normal datatype and a weak version is that a weak version does not hold its reference. If there are no other references anywhere in the application, the values from a weak datatype may be gargage collected.

It is thus a way to prevent memory leaks in some specialized circumstances. Due to the fact that their values may be garbage collected at any time, it is also impossible for them to iterate over their contents, so the only methods available are `add()`, `delete()` and `has()`.

Must use cases with weak datatypes are about keeping data about an object (such as how many times it was used), without forcing the object in question to be kept in memory. Also use cases with 'smart reference'-like characteristics are common.

15.19. Typed Arrays

- There is now an abstraction for memory buffers called Typed Arrays
  - A Buffer is a typeless blob of data
  - A View is a typed interpretation of that data

```
let buffer = new ArrayBuffer(32);       // 32 bytes of untyped memory
                                        // Usually other APIs 'give' you a buffer to play with.
let pixels = new Uint8Array(buffer);    // Could be a array of RGBA... pixel values perhaps
```

- They shape the way for interfacing with different types of I/O
  - Hardware, File access, 3D video buffers, ...

In a managed language like JavaScript, it is not usually possible to directly access machine I/O interfaces directly as you can do in an unmanaged language like C or C++. This provides a challenge. How can we enable high-performance applications such as 3D image processing without direct access to the hardware?

The answer is using a data abstraction that offers data from low-level interfaces (files, video memory or maybe even a webcam or other USB devices). This way the engine can offer us a data space which the application can interpret using views (such as an `UInt8Array` for individual bytes). All of this can happen without the need to directly access low-level memory structures (e.g. a pointer to a memory mapped I/O region).

How the data from an `ArrayBuffer` is interpreted depends on the used View

- An overview of available Views and their interpretation

View	Interpretation
Int8Array, Uint8Array, Uint8ClampedArray	Signed and unsigned 1 byte per entry
Int16Array, Uint16Array	Signed and unsigned 2 bytes per entry
Int32Array, Uint32Array	Signed and unsigned 4 bytes per entry
Float32Array, Float64Array	Floating point 4 and 8 bytes per entry

15.20. New Methods for Built-in Objects

- A lot of new methods were added to various existing objects
  - Most offer only convenience
- A list of some of the most interesting ones

Method	Purpose
Object.assign(t, ...s)	Copies a number of object properties into a target object
Array.find(cb), Array.from(), Array.of(), Array.fill(), Array.copyWithin()	Convenient array functions
String.repeat(n)	Repeats the string n times
String.startsWith(s), String.endsWith(s), String.includes(s)	Convenient string searching
Number.isNaN(), Number.isFinite(), Number.isSafeInteger()	Reliable int/float testing
Number.EPSILON	espilon for comparing floats
Math.trunc(n), Math.sign(n), Math.imul(a, b), Math.acosh(r), Math.hypot(a, b)	More mathematical functions

At first glance, there is nothing revolutionary about these new additions. Indeed, each of these could be implemented previously without ES6. The advantages of having them are clear though:

- Previously they needed to be reinvented for every project or library
- This "reinventing the wheel" caused a lot of square shaped wheels (wrong buggy implementations)

As an example to this point, let's take a very simple example: `Math.sign()`. This typically would be implemented in a naive way:

```
function sign(number) {
    return number < 0 ? -1 : 1;
}
```

This code does not handle NaN, 0, Infinite, etc. It is this type of bugs that can now be prevented by using the new built-in methods. It should be noted, that some of these methods were already available in some browsers for quite some time.

- Some examples of new method usage

```
let a = {firstName: 'Jimi'}, b = {lastName: 'Hendrix'}, c = {age: 27};
Object.assign(c, a, b);
console.log(c.firstName, c.lastName, c.age);

let animals = [
    {type: 'tiger', name: 'Shere Khan'},
    {type: 'panther', name: 'Bagheera'},
    {type: 'ape', name: 'King Louie'},
    {type: 'snake', name: 'Kaa'}
];
console.log(animals.find(a => a.type === 'panther'));

let s = "*".repeat(10); // **********
```

There are many more methods that have been added. For an overview, and how to use them, take a look at Mozilla's MDN pages (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference)

15.21. Proxies

- Proxy objects allow target objects to be trapped
  - A trap is an interception of an action on the target
  - Comparable with AOP around filter

```
let animal = {name: 'Bagheera', type: 'panther', color: 'black'};
let handler = {
    get: (target, name) => {        // get trap
        console.log('Intercepted get property with name', name);
        return target[name].toUpperCase();  // Changing the target
    }
}
let p = new Proxy(animal, handler);
console.log(p.name);    // BAGHEERA
```

- A proxy wraps itself around a target and adds a handler which contains any number of optional traps
  - Can intercept method calls, operators read/write assignments and more...

This example shows that a get-trap is wrapped around the animal object. This means that when retrieving a property of the animal (through the proxy), it will invoke the trap before handing over the result of the read. This way a trap can intercept getters and method calls and do something in between, including:

- Doing some extra side effects (such as the `console.log()` in the example)
- Changing the results (such as the `toUpperCase()` in the example)

With Proxies, ECMAScript thus has the possibility to add AOP (Aspect Oriented Programming) or decorator type behaviour in the application.

It is interesting to note that similar sort of behaviour in AOP is named differently:

- Trap is called Advice
- Handler is called Aspect

It's not entirely the same, but it may help your understanding if you are familiar with AOP concepts.

- Traps you can use include (subset):

Function in handler	Trap (interception)
handler.getPrototypeOf()	Trap for Object.getPrototypeOf
handler.defineProperty()	Trap for Object.defineProperty
handler.has()	Trap for the in operator

handler.get()	Trap for getters
handler.set()	Trap for setters
handler.deleteProperty()	Trap for delete
handler.ownKeys()	Trap for Object.getOwnPropertyNames
handler.apply()	Trap for calling a function
handler.construct()	Trap for invoking new

There are many different types of traps. Some of the most interesting have been hightlighted. Some others have been left out of the list for brevity. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy#Methods_of_the_handler_object for a complete reference.

15.22. Reflect

- All possible Proxy handler methods are grouped as static functions in Reflect
  - Allows a "third person" to trigger the corresponding actions on a subject object

```
function Animal(name, type) {
    this.name = name;
    this.type = type;
}

Animal.prototype.sound = () => {
    return 'Roar!!'
};

let a = Reflect.construct(Animal, ['Shere Khan', 'tiger']);
console.log(Reflect.get(a, 'name'));    // Shere Khan
console.log(Reflect.apply(Animal.prototype.sound, a, []));  // Roar
```

In case you are wondering: "Wait was all of this not already possible with JavaScript?": Yes. Absolutely. However, since JavaScript has a large number of "bad parts", doing these things the old way tends to be dangerous, quirky and especially not very readable.

Using a more explicit `Reflect` class shows much clearer intent, is always correct, and does not force you to resort to the "creative expressiveness" of the language (although some developers may find that last argument a bad thing).

15.23. Internationalization

- Much effort is invested in making ES6 support i18n and l10n better
- Language sensitive collation (sorting and comparison):

```
let g = new Intl.Collator('DE');    // In Germany ä combes before z
console.log(g.compare('ä', 'z'));   // 1
let s = new Intl.Collator('SV');    // In Sweden ä comes after z
console.log(s.compare('ä', 'z'));   // -1
```

- Language sensitive number formatting:

```
let e = new Intl.NumberFormat('en-US'); // In the US . is the decimal separator and , for grouping
console.log(e.format(1507.83));         // 1,1507.83
let b = new Intl.NumberFormat('nl-BE'); // In Belgium , is the decimal separator and . for grouping
console.log(b.format(1507.83));         // 1.507,83

let bc = new Intl.NumberFormat('nl-BE', {style: "currency", currency: "EUR"});
console.log(bc.format(1507.83));     // 1.507,83 €
let nc = new Intl.NumberFormat('nl-NL', {style: "currency", currency: "EUR"});
console.log(nc.format(1507.83));     // € 1.507,83
```

- Language sensitive date formatting:

```
var date = new Date("2016-04-17");

var en = new Intl.DateTimeFormat("en-US");
console.log(en.format(date)); // 4/17/2016 (month first!)

var be = new Intl.DateTimeFormat("nl-BE");
console.log(be.format(date)); // 17-4-2016 (day first!)
```

The `Intl` object allows creating `Collator`, `NumberFormat` and `DateFormat` instances that are localized to the desired language - country. This allows a number of built-in conventions to be used regarding to the way that region deals with sorting rules, number formats and dates.

For fine grained control, it is possible to pass - next to the locale string - also an objects object. Using this object you can control many more configurations such as:

- Case sensitivity
- Whether to ignore punctuations
- Numeric vs text bases sorting
- 24 or 12 based hours
- Timezones
- Currency conventions

15.24. Unicode Support

- ES6 has better support for unicode

```
let ch = '𠮷';

// Old way
console.log(ch == "\uD842\uDFB7");     // True. Old bad way (two code points: surrogate pair)
console.log(ch.charCodeAt(0))         // 0xD842
console.log(ch.charCodeAt(1))         // 0xDFB7

// New way
console.log(ch == "\u{20BB7}");         // True. New way (single code coint)
console.log(ch.codePointAt(0))         // New method to do a better job than 'charCodeAt()'
```

- Regular expressions also support them now using the `u` flag

```
console.log("𠮷".match(/./));    // ES5: Garbage (broken!)
console.log("𠮷".match(/./u));    // ES6: Actual character (use u to opt-in)
```

- Various other places are also working like they should now
    - Iterating over unicode text, etc...

Previous versions of JavaScript were known to be broken with regards to internationalization support and unicode. In fact, all code points higher than \uFFFF are represented as two code points rather than one (this is called a surrogate pair). This includes Asian languages.

Internally nothing has not changed, so the length of a unicode string may still produce multiple points for a single character.

```
console.log("𠮷".length); // 2
```

Regular expressions did not support matching on unicode characters. This is now solved if using the 'u' flag on the expression. (The opt-in is required because it would otherwise break the web.)

16. Asynchronous Programming

16.1. ES6: Promises

- Promises are objects for deferred any async processing
    - Represents a task that has not completed yet (but will be)
- A Promise has three distinct states:
    - Unresolved when the task is still pending
    - Fulfilled (or Resolved) when the task has completed successfully
    - Rejected when the task has completed with errors
        - In this last case the Promise was broken
- Async functions can return a value (promise) immediately
    - This value can be hooked into using callbacks
    - This way the async function itself can delegate callback handling to a seprate promise object

Promises are known in other languages as well. Java and C# know them as a `Future`, and JQuery has had it for a long time in the form of a `$.Deferred`. On other platforms such as Node.js there are special libraries such as promisify. It is clear that JavaScript has been flirting with the concept of Promises for some time now. ES6 finally standardizes the concept so that every JavaScript platform is able to use it in the same and portable way.

Promises in JavaScript have a very elegant terminology:

- Unresolvedness
- Fulfillment
- Rejection

Once you grasp the concepts of this, it allows for quite clear reasoning with promises.

- A (synchronous) promise in action
    - Not useful, but easier to explain

```
let task = () => {
    return new Promise((resolve, reject) => {
        let gamble = Boolean(Math.round(Math.random()));    // Random 50/50 chance for true or false
        if(gamble) {
            resolve(42);    // Success!
        } else {
            reject(24);       // Failure!
        }
    });
};

let p = task();
p.then((result) => {
    console.log('Promise successful with value: ' + result);
}).catch((result) => {
    console.log('Promise failed with value: ' + result);
});
```

This is an example of a synchronous Promise. In this case it's not really useful, but it is more simple to explain this way. As you can see, a promise takes in a function with two parameters `resolve` and `reject`. These two parameters are functions, that can be called with a result parameter when the promise succeeds or fails respectively.

The returned object `p` can then be taken separately and if so desired a `then()` and/or `catch()` function can be registered, to handle the success or failure case respectively. Doing this, the `task()` function does not have to deal with callbacks at all, so the code does not "grow" to the right so fast as it would in this example:

```
function f(function(val) {
    doSomething(function(val) {
        extraWork(function(val) {
            ...
        }
    });
});
```

- Promises really shine in async code

```
function query(send) {
    return new Promise((resolve, reject) => {
        let xhr = new XMLHttpRequest();
        xhr.open('get', 'http://www.realdolmen.com/data');
        xhr.onload = () => {
            if(this.status == 200) {
                resolve(xhr.responseText);
            } else {
                reject(this.status);
            }
        };
        xhr.send();
    });
}

// Thanks to promise, the (async) function can immediately return, and there is no callback!
query('some data').then((receive) => {
    console.log('Data received:', receive);
}).catch((code) => {
    console.log('Error status code:', code);
});
```

This example shows an XHR call being executed by means of a Promise. The entire point of this is that the `query()` function does not need to deal with the success and failure callbacks. Instead it returns a promise, who will take care of this. Of course the only thing the `query()` function must still do is the actual work involving the XHR call, and signalling when the data has arrived.

- Promises can be chained together
  - Reduces callback hell to a great extent (but not entirely)

```
let p = new Promise(function(resolve, reject) {
    resolve('1');
});

p.then(data => data + '5')                  // then() returns a new promise to build on
    .then(data => data + '0')
    .then(data => data + '7')
    .then(data => console.log(data));       // "1507"!
```

- There are also functions to group Promises together
  - Allows parallellization

```
Promise.all([p1, p2, p3]).then(data => {     // Executed when all promises are resolved!
    let [r1, r2, r3] = data;                 // Data received as an array
    console.log(r1, r2, r3);
});
```

Chaining is possible because each call to `then()` or `catch()` actually returns a new Promise. It is this chaining that eliminates callback hell.

The `Promise.all()` function returns a Promise that will be resolved after all passed Promises (`p1`, `p2`, `p3`) are resolved. This function is known in other libraries (like jQuery) as a `when()` (`$.when`).

16.2. ES7: Async Await

- Promises and Generators are combined in ES7 as Async/Await
  - This eliminates callback problems once and for all
  - Asynchronous applications now look sequential
  - It's like magic!

```
async function task() {     // Yes! No more callbacks!
    console.log('One');
    await sleep(1000);
    console.log('Two');
    await sleep(1000);
    console.log('Three');
    await sleep(1000);
}
```

With Generators as the engine, and Promises as the fuel, the Async/Await feature of ES7 will fully eliminate callback problems. ES7 will introduce two new keywords: `async` and `await`. These are nothing more than a generator that yields Promises. Since a generator is a pause-resumable method, this will allow us to write asynchronous programs that retain a synchronous algorithm structure.

17. In Closing

17.1. References

- Mozilla Developer Network (MDN)
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript
- ECMAScript 6 Feature Overview & Comparison
  - http://es6-features.org
- ECMAScript Compatibility Table
  - http://kangax.github.io/compat-table/es6/
- ES6 Fiddle
  - http://www.es6fiddle.net/