Java Script

- What is JavaScript?
- Why learn JS?
- History
- Working Environment

What is JavaScript?

JavaScript® (often shortened to **JS**) is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages, but it's used in many non-browser environments as well. It is a prototype-based, multi-paradigm scripting language that is dynamic, and supports object-oriented, imperative, and functional programming styles.

Mozzilla development network (MDN)

- First class functions
  - ❖ passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures
- Ptototype-based
  - ❖ Instead of declaring a class structure, you can create objects of the same type, and add to their definition any time you like using the object's prototype.
- Imperative
  - ❖ Your code focuses on creating statements that change program states by creating algorithms that tell the computer how to do things.

    vs declarative (like HTML)
- Functional programming ("purely functional programming“)
  - ❖ Writing programs that solve problems by composing expressions

    i.e. Linq in .NET , Stream API in Java, etc.

- Dynamic language
  - A **dynamic programming language** is a programming language in which operations otherwise done at compile-time can be done at run-time. For example, in JavaScript it is possible to change the type of a variable or add new properties or methods to an object while the program is running.

- Lightweight
  - easy to run

- Interpreted
  - no compilation, executed directly

- Object oriented

- Scripting
  - instruction are written for runtime environment, like batch file, shellscript

More on JS

- ❑ Created by Brendan Eich 1995 in 10 days
- ❑ Under netscape + Sun-microsystem (which designed Java)
- ❑ Based on Scheme & self languages
- ❑ While developed called Mocha
- ❑ First released was called Livescript
- ❑ Netscape browser runned it
- ❑ Renamed to Javascript as a joke to Sun-microsystem
  - confused between JS and Java they are very different

More on JS

- Integrated in 1996 to ie3 as JScript
- We had many vendors giving the same language different names
- Need a Standard body to make sure the standards are kept between the browse vendors
  - Web- develop once and run anywhere
  - Adopted by ECMA: European Computer Manufacturer Association
- Since ES1 there has been many revision, most famous: ES5 2009 ES6 2015
- Visit ECMA web site to understand how JS language works under the hood

How JS works?

All languages starts with human readable syntax

Computer understands machine-code (11010101)

DOM parser - converts HTML code to a visual page

CSS parser - rendering engine which renders CSS code into the HTML

JavaScript engine - different between browsers:

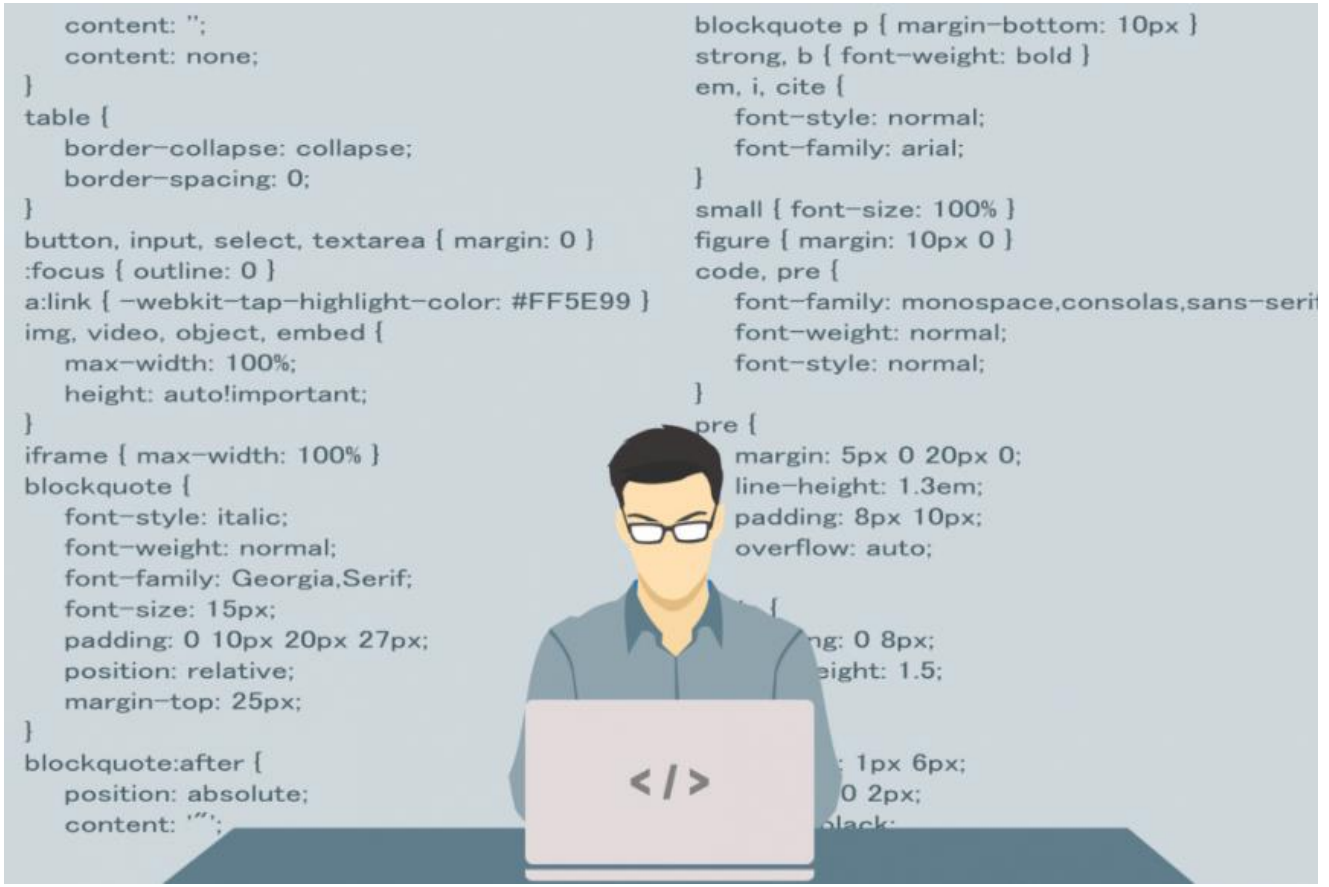
- Firefox: spidermonkey,
- Chrome: V8
- Safari: nitro
- IE: Chackra

⇒ take javascript syntax and convert into machine code (10101010)

why it is called client side?

since it is downloaded from the site (can look at the source code) and converted and run at the client browser (JIT compiler, convert only a command which it needs to run in the specific time)

Let’s Code!


content: '';
content: none;
}
table {
border-collapse: collapse;
border-spacing: 0;
}
button, input, select, textarea { margin: 0 }
:focus { outline: 0 }
a:link { -webkit-tap-highlight-color: #FF5E99 }
img, video, object, embed {
max-width: 100%;
height: auto!important;
}
iframe { max-width: 100% }
blockquote {
font-style: italic;
font-weight: normal;
font-family: Georgia,Serif;
font-size: 15px;
padding: 0 10px 20px 27px;
position: relative;
margin-top: 25px;
}
blockquote:after {
position: absolute;
content: '"';
blockquote p { margin-bottom: 10px }
strong, b { font-weight: bold }
em, i, cite {
font-style: normal;
font-family: arial;
}
small { font-size: 100% }
figure { margin: 10px 0 }
code, pre {
font-family: monospace,consolas,sans-seri
font-weight: normal;
font-style: normal;
}
pre {
margin: 5px 0 20px 0;
line-height: 1.3em;
padding: 8px 10px;
overflow: auto;
ng: 0 8px;
eight: 1.5;
1px 6px;
0 2px;
</>

- Types:
  - Number, String, Boolean, Object, null, undefined
  - Typeof + creation using type(value)
- var
- No modifiers
- Type coercion == , ===
- console.log, info, error
- if, for, while
- switch
- Object creation: { }, method creation, constructor
- Object.member object[“member”] – good for reflection
- undefined vs null

- Arrays
  - Definition
  - Out-of-bound?
  - Push, pop, shift, unshift, slice, forEach [don’t use for], indexOf, map, reduce, reduce – array builder, filter, find, every, some
  - Number, String, Boolean, Object, null, undefined
    - ❖ Objects vs primitives
- Methods
  - No overloading
  - arguments array
  - Running method from console – undefined return
  - Function expressions [call without ()]
  - Method as an argument
  - Parameters are value types (unless member inside object)
  - Functions inside an object
- Objects again
  - Using this
  - Call (benny), apply, bind…
  - Nested object

- Scopes and closure
  - global var
  - Functions scoping
  - Scope – global, into function, into inner function
  - IIFE – no scope!
  - Read vs write
  - Windows object
  - What happens without strict mode?
  - Hoisting (and compilation)
  - Closure – getters and setters
- Prototypes
  - __proto__
  - Function.prototype
  - object.constructor (get type question)
  - Inheritance
  - Add for all objects

ES6 refers to version 6 of the ECMA Script programming language. ECMA Script is the standardized name for **JavaScript**, and version 6 is the next version after version 5, which was released in 2011. It is a major enhancement to the JavaScript language, and adds many more features intended to make large-scale software development easier.

ECMAScript, or ES6, was published in June 2015. It was subsequently renamed to ECMAScript 2015. Web browser support for the full language is not yet complete, though major portions are supported. Major web browsers support some features of ES6. However, it is possible to use software known as a **transpiler** to convert ES6 code into ES5, which is better supported on most browsers.

- Why use let/ const?
  - Improved scope
  - More readable – it clarify the code very much!! [easier to understand]
    - var does not give any information about the variable
    - See next page

- const
  - Usually use const (if it is not going to change)
  - Error if try to modify
  - If points to object/ array – the object properties could be modified
    - Use Object.freeze to avoid
- let
  - If it’s not a const use let instead of var
  - Expect this variable will be changed

```
function buildExpressions(code) {
  const transformedCode = JSXTransformer.transform(code).code;
  const codeByLine = transformedCode.split('\n');
  const tokenized = esprima.tokenize(transformedCode, { loc: true });
  const parens = { '(': 0, '{': 0, '[': 0 };
  let wasOpen = false;

  var exp = _.reduce(tokenized, (expressions, { value, loc: { end } }, index) => {
    const lineNumber = end.line;
    const lineContents = codeByLine[lineNumber - 1];
    const lineHasMoreDelimiters = this.lineHasMoreDelimiters(end, lineContents);
    const endOfLine = end.column === lineContents.length;

    if (expressions[lineNumber]) { return expressions; }

    if (OPEN_DELIMITERS.includes(value)) {
      parens[value] += 1;
      wasOpen = true;
    }
  }
```

- String (template string)

  - Back tick :
    - ${ }
    - line gap

  - More:
    - "hello".startsWith("ello", 1) *// true*
    - "hello".endsWith("hell", 4) *// true*
    - "hello".includes("ell") *// true*
    - "hello".includes("ell", 1) *// true*
    - "hello".includes("ell", 2) *// false*

- Fat Arrow
  - Shorter syntax:
    - function add(x, y) { return x + y }

      const add = (x, y) => x + y;
    - For one argument: const pow2 = x => x*x;
    - i.e. const numbers = [1, 2, 3]; numbers.map(x => x * 2);
  - **Lexical this**

```
const team = {
  members: ['Jane', 'Bill'],
  teamName: 'Super Squad',
  teamSummary: function() {
    return this.members.map(function(member) {
      return `${member} is on team ${this.teamName}`;
    }.bind(this))
  }
}
console.log( team.teamSummary() )
…
```

- Fat Arrow - continue
  - **Lexical this** – continue:
    - var self = this
  - Or just use =>

```
const team = {
  members: ['Jane', 'Bill'],
  teamName: 'Super Squad',
  teamSummary() {  // this == team
      return this.members.map(member => `${member} is on team ${this.teamName}`)}
 }
console.log( team.teamSummary() )
```

  - Lexical = replacement of this term depends on how its interpolated or evaluated.

- Default parameters

```
function f (x, y = 7, z = 42)
{ return x + y + z }
f(1) === 50
```

- Rest parameters

```
function f (x, y, ...a)
{ return (x + y) * a.length }
f(1, 2, "hello", true, 7) === 9
```

- Spread operator

```
var params = [ "hello", true, 7 ]
var other = [ 1, 2, ...params ] // [ 1, 2, "hello", true, 7 ]
function f (x, y, ...a)
{ return (x + y) * a.length }
f(1, 2, ...params) === 9
var str = "foo"
var chars = [ ...str ] // [ "f", "o", "o" ]
```

- Enhanced Object Properties – property shorthand

```
var x = 0, y = 0
var obj = { x, y }
```

- Enhanced Object Properties – computed property name

```
let obj = { foo: "bar",
            [ "baz" + quux() ]: 42 }
```

- Enhanced Object Properties – method properties

```
obj = {
    foo (a, b) { … },
    bar (x, y) { … }  }
```

- Destructuring Assignment

```
var list = [ 1, 2, 3 ]
var [ a, , b ] = list
[ b, a ] = [ a, b ]
```

- Destructuring Assignment – parameter context matching

```
function f ([ name, val ])
{ console.log(name, val) }

function g ({ name: n, val: v })
{ console.log(n, v) }

function h ({ name, val })
{ console.log(name, val) }

f([ "bar", 42 ])
g({ name: "foo", val: 7 })
h({ name: "bar", val: 42 })
```

- **Symbols**

```
Symbol("foo") !== Symbol("foo")
const foo = Symbol()
const bar = Symbol()
typeof foo === "symbol"
typeof bar === "symbol"
let obj = { }
obj[foo] = "foo"
obj[bar] = "bar"
JSON.stringify(obj) // {}
Object.keys(obj) // []
Object.getOwnPropertyNames(obj) // []
Object.getOwnPropertySymbols(obj) // [ foo, bar ]
```

- **Global Symbols**

```
Symbol.for("app.foo") === Symbol.for("app.foo")
const foo = Symbol.for("app.foo")
const bar = Symbol.for("app.bar")
Symbol.keyFor(foo) === "app.foo"
Symbol.keyFor(bar) === "app.bar"
typeof foo === "symbol"
typeof bar === "symbol"
let obj = { }
obj[foo] = "foo"
obj[bar] = "bar"
JSON.stringify(obj) // {}
Object.keys(obj) // []
Object.getOwnPropertyNames(obj) // []
Object.getOwnPropertySymbols(obj) // [ foo, bar ]
```

- **Generators**

```
function* genny() {
  yield 'a'
  yield 'b'
  yield 'c'
}
let iter = genny()
for(let f of iter) {
  c(f)
}
let iter1 = genny()
c(iter1.next()) // { value : 'a', done: false}
c(iter1.next()) // { value : 'b', done: false}
c(iter1.next()) // { value : 'c', done: false} !!
c(iter1.next()) // { value : undefined, done: true}
```

```
function* gennyReturn() {
  console.log("once") // printed only once
  yield 'a'
  yield 'b'
  return 'c'
  yield 'd' // unreachable code
}
let iter2 = gennyReturn()
for(let f of iter2) {
  c(f)
}
let iter3 = gennyReturn()
c(iter3.next()) // { value : 'a', done: false}
c(iter3.next()) // { value : 'b', done: false}
c(iter3.next()) // { value : 'c', done: true} !!
c(iter3.next()) // { value : undefined, done: true}
```

```
function* genny4() {
  var x = 1
  while (x < 10) {
    yield x
    x++
  }
  yield -1
}
let iter4 = genny4()
for(let f of iter4)
  {
    c(f)
  }
```

```
function* genny4() {
  var x = 1
  while (x < 10) {
    yield x
    x++
  }
  yield -1
}
let iter4 = genny4()
for(let f of iter4)
  {
    c(f)
  }
```

- **Iterators**

```
let myIteratorObject = {
  count12 : 1,
  [Symbol.iterator] : function() {
    return {
      next: () => {
          while (this.count12 < 12) {
            this.count12++
            return { value : this.count12, done : false }
          }
          return { value : this.count12, done : true }
      }
    }
  }
}

for(let f of myIteratorObject)
  {
    c(f)
  }
```

- **Iterators**

Now try Fibonacci? Solution in next page…

- **Iterators - fibonacci**

```
let fibonacci = {
        [Symbol.iterator]()
        { let pre = 0, cur = 1
          return {
                    next ()
                    { [ pre, cur ] = [ cur, pre + cur ]
                      return { done: false, value: cur } }
           }
         }
 }
for (let n of fibonacci)
{
        if (n > 1000)
                    break
        console.log(n)
}
```

- **Classes**

```
class Shape
 {
       constructor (id, x, y)
       {
        this.id = id
        this.move(x, y)
       }
       move (x, y)
       { this.x = x this.y = y }
}
```

ES5:

```
var Shape = function (id, x, y)
{ this.id = id; this.move(x, y); };
Shape.prototype.move = function (x, y) { this.x = x; this.y = y; };
```

- **Classes inheritance**

```
class Rectangle extends Shape
{
        constructor (id, x, y, width, height)
        {
            super(id, x, y)  // mandatory, even if default super()
            this.width = width
            this.height = height
        }
}
class Circle extends Shape
 {
        constructor (id, x, y, radius)
        {
            super(id, x, y) // mandatory, even if default duper()
            this.radius = radius
        }
}
```

- **Async await**

```
<!DOCTYPE html>
<html lang="en">
<head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">
        <title>Async JS</title></head>
<body>
        <script src="callbacks.js"></script>
</body>
</html>
```

```
const posts = [
{ title: 'Post One', body: 'This is post one' },
{ title: 'Post Two', body: 'This is post two' }
];
function getPosts() {
    setTimeout(() => {
    let output = '';
    posts.forEach((post, index) => {
    output += `<li>${post.title}</li>`;
    });
    document.body.innerHTML = output;
    }, 1000);
}
function createPost(post, callback) {
    setTimeout(() => {
    posts.push(post);
    callback();
    }, 2000);
}
createPost({ title: 'Post Three', body: 'This is post three' }, getPosts);
```

Now with async await…

```
const posts = [
{ title: 'Post One', body: 'This is post one' },
{ title: 'Post Two', body: 'This is post two' }];
function getPosts() {
        setTimeout(() => {
        let output = '';
        posts.forEach((post, index) => {
        output += `<li>${post.title}</li>`;});
        document.body.innerHTML = output;
        }, 1000);
}
function createPost(post) {
                return new Promise((resolve, reject) => {
                                setTimeout(() => {
                                                posts.push(post);
                                                const error = false;
                                                if (!error) { resolve(); }
                                                else { reject('Error: Something went wrong’);}
                                                }, 2000);
                });}
```

```
createPost({ title: 'Post Three', body: 'This is post three' })
.then(getPosts)
.catch(err => console.log(err));

// Async / Await
async function init() {
    await createPost({ title: 'Post Three', body: 'This is post three' });
    getPosts();
}

init();

// Asunc / Await / Fetch
async function fetchUsers() {
    const res = await fetch('https://jsonplaceholder.typicode.com/users');
    const data = await res.json();
    console.log(data);
}
fetchUsers();
```

```
//Promise.all
const promise1 = Promise.resolve('Hello World');
const promise2 = 10;
const promise3 = new Promise((resolve, reject) =>
setTimeout(resolve, 2000, 'Goodbye')
);
const promise4 = fetch('https://jsonplaceholder.typicode.com/users').then(res =>
res.json()
);

Promise.all([promise1, promise2, promise3, promise4]).then(values =>
console.log(values)
);
```