JavaScript and Java

JavaScript is a Self inspired scripting language that

- *Brendan Eich* developed for Netscape in 1995
- supports *object-oriented* and *imperative* programming styles
- was standardised as ECMAScript 1 in 1997, 2 in 1998, 3 in 1999

Comparison between JavaScript and Java

	JavaScript	**Java**
code model	scripting	programming
syntax style	C-like	C-like
source reader	parser / JIT compiler	compiler
runtime engine	interpreter / CPU	Java Virtual Machine
runtime code	source / native code	JVM bytecode
procedures	functions	methods (of objects)
objecthood	prototype-based typeless, extensible objects	class-based static, typed objects
typing	weak	strong
object creation	standard functions	constructor methods
binding	dynamic	dynamic and static
type safety	check references at runtime	check references at compile-time
variable scope	function (var), block (let)	block
standardisation	ECMAScript	Java Community Process
IO/OS support	host context APIs	standard APIs

Latest version is ECMAScript 9 released in June 2018.

JavaScript Syntax

JavaScript supports 6 primitive types

undefined	sole value possessed by unassigned variables
null	sole value denoting intentional lack of a value
Boolean	type having value of *true* or *false*
Number	type of double precision 64 bit IEEE format number
String	sequence of 16 bit UTF-16 characters
Symbol	ES 6+ - symbols have unique values and no literal syntax

Objects are members of the type *Object* that have

- *prototype* object
- extensible enumerable list of *properties* (including *functions*)
- include *keyed*, *indexed collection*, *function* types

Variables are case-sensitive identifiers that

- start with alphabetic letter, underscore or dollar sign
- contain only alphanumerics, dollar signs and underscore
- have no type - type belongs to their value

Keyword *var* declares variables with *scope* of function they're in.

In ES 6+ *let* and *const* declare *block-scoped* variables.

let variables can't be redeclared in scope. *var* variables can.

const variables can index loops but can't be reassigned values.

```
var i, number_of_hits = 0, done = true, fruit = "yoghurt";
let ID = Symbol();
const pi = Math.PI;
```

Undeclared variable has *global* scope unless declared after use.

JavaScript Statements

JavaScript statement ends in ";". Newline forces their interpolation.

JavaScript statements are composed of

comments	single start with "//", multiline inside "/*" and "*/"
expressions	compounds of values, variables and operators
keywords	reserved words like "function", "var", "new" etc.
operators	see below
values	primitive values and objects
variables	identifiers

Expressions can contain various types of operators such as

Type	**Operators**	**Example**
arithmetic	+, -, *, /, %	3 + 4 - 6 / 7 * 8
assignment	=, =+, =*, =/, =-	E = M * C * C
bitwise	&, ˆ, |, <<, >>	15 ˆ 9
comparison	>, >=, <, ==, !=, ===	x > y + z
logical	&&, ||, !	age >= 0 && age < 120
string	+	"Let there" + " be light!"
conditional	cond ? val1 : val2	age > 17 ? "adult" : "minor"

JavaScript has a few special identifiers including

NaN	Not a Number - special IEEE format value
Infinity	positive infinity number value

JavaScript has built-in objects incl. *Array*, *Date*, *Math*, *Regexp*, *String*

Its engines may have built-ins for GUI, IO, OS - *Navigator*, *Window*

JavaScript Control Structures

JavaScript supports various programming language control structures:

conditionals	if, if else, switch
loops	for, for of, for in, do while, while
functions	procedures with arguments that return a value

Examples of conditional expressions:

```
if ( s.length > 10 ) {
  s = s.substring(0, 10);
} else {
  s = s.trim();
}

switch ( Math.floor(Math.random()*3) ) {
  case 1: col = "red";
          break;
  case 2: col = "blue";
          break;
  default: col = "green";
}
```

Examples of loops

```
let sum = 0;
let arr = [1, 2, 3, 5, 7, 11, 13];
for (let i=0; i<arr.length; i++) { // classic loop
  sum += arr[i];
}

for (let item of arr) { sum += item; } // loop over collection

var car = { make: "Ford", model: "Focus", year: 2017 };
for (var p in car) {       // enumerable properties loop
   if ( isNaN(car[p]) ) car[p] = car[p].toUpperCase();
}

do {
  var j = page.indexOf("=", j);
  if ( j != -1 )
    s += page.substring(j, j + 2);
} while ( j != -1 && j < len );

var count = 0;
while ( (i = page.indexOf("href=", i)) != -1 ) {
  count++;
}
```

Standalone JavaScript

JavaScript can be run by standalone engines or within browsers.

Nashorn is standalone JavaScript engine (ES5) released in Java 1.8

- based on the Da Vinci Machine
- implementing JSR-223 using new code base

Nashorn engine is launched with "jjs" or "jrunscript" command.

```
unix% java -v
java version "1.8.0_45"
unix% /usr/java/default/bin/jjs
jjs> president = "Donald Trump"
jjs> saying = "Make America great again!"
jjs> print(president + " says \"" + saying + "\"")
Donald Trump says "Make America great again!"
jjs> quit()
```

Java 1.6+ supports scripting and in 1.8 defaults to *nashorn*

```
unix% jrunscript
nashorn>
```

Nashorn supports some useful built-in functions

exit()	causes script process to terminate
load()	loads and runs a script from path, URL or script object
print()	prints argument(s) as string(s) to stdout with newline
readLine()	reads line of input from stdin (in scripting mode)

quit() does same thing as *exit()*. *echo()* does same thing as *print()*.

Examples of their use:

```
unix% jjs -scripting
jjs> fruit = readLine("What fruit do you like? ")
What fruit do you like? pineapple
jjs> print("You like to eat", fruit + "s")
You like to eat pineapples
jjs> load("script.js")
jjs> load("http://www2.macs.hw.ac.uk/~bozo/script.js")
```

JavaScript Functions

JavaScript functions are *first class* objects where

- *function* keyword has optional name
- arguments are enclosed in (...) and body in { ... }
- arguments are *passed by value*
- arguments may pass objects (e.g. functions) by reference

Suppose *isFruit()* is declared in *fruit.js*:

```
function isFruit(x) {
  if ( x == "apple" || x == "fig" || x == "pear" ) { return true; }
  return false;
}
```

Function can be used by loading its definition and invoking it:

```
jjs> load("fruit.js")
jjs> isFruit("fig")
true
```

Anonymous functions are defined *inline* and may be

- assigned to a name
- passed as argument in a function call
- self-invoked i.e. applied inline to arguments

Examples would be

```
jjs> anon = function() { print('I am anonymous'); }
jjs> anon()
I am anonymous
jjs> invoke = function(fn, arg) { print(fn(arg)); }
jjs> invoke(function(i){ return i + 1; }, 5)
6
jjs> (function(s) { print('Sue says', s); })("Paul lies")
Sue says Paul lies
```

Builtin objects like Math support standard functions like

```
jjs> Math.random()
0.6452514843444341
```

JavaScript String Handling

Methods and properties for handling strings include

charAt(N)	returns character at index *N*
indexOf(S)	returns index of *S* in string
lastIndexOf(S)	returns last index of *S* in string
length	property storing length of string
split(S)	returns array of substrings separated by *S*
substring(M)	returns substring from index to end
substring(M, N)	returns substring between given indices
toLowerCase()	maps all letters to lower case
toUpperCase()	maps all letters to upper case
trim()	removes whitespace from both ends of string

Function in *url.js* shows some of their use to normalise a URL:

```
function normaliseURL(url) {
  url = url.trim();
  if ( url.length == 0 ) return "";
  var i = url.indexOf("://");
  var j = url.indexOf("/", i + 3);
  var access_method = "";
  var hostname = "";
  if ( i != -1 ) {
    access_method = url.substring(0, i).toLowerCase() + "://";
    if ( j != -1 )
      hostname = url.substring(i + 3, j).toLowerCase();
  } else {
    j = 0;
  }
  return access_method + hostname + url.substring(j);
}
```

It can be exercised as follows:

```
unix% jjs -scripting
jjs> load("url.js")
jjs> normaliseURL(readLine("URL? "))
URL? Http://WWW.NEXT.COM/HomePage.asp
http://www.next.com/HomePage.asp
```

JavaScript Arrays

JavaScript Array is created using *Array* builtin

```
cars = new Array();
cars[0] = "Honda Civic";
cars[1] = "Ford Focus";
cars[2] = "Fiat Punto";
```

Same array can also be created by either of

```
cars = new Array("Honda Civic", "Ford Focus", "Fiat Punto");
cars = ["Honda Civic", "Ford Focus", "Fiat Punto"];
```

Useful properties for an Array include

length	number of elements
prototype	property allowing Array object to be extended

Use of them is illustrated by

```
jjs> var cars = [ "Honda Civic", "Ford Focus", "Fiat Punto" ]
jjs> cars.length
3
jjs> Array.prototype.make = function(i) {return this[i].split(" ")[0]}
jjs> cars.make(0)
Honda
```

Useful functions for an Array include

indexOf(element)	index of *element*
push(element)	add *element* to end and return new length
splice(i, n, a, ..., k)	replace *n* elements from index *i* with *a* ... *k*

Use of them is illustrated by

```
jjs> var cars = [ "Honda Civic", "Ford Focus", "Fiat Punto" ]
jjs> cars.indexOf("Fiat Punto")
2
jjs> ["a", "b", "c", "a"].lastIndexOf("a")
3
jjs> cars.push("Fiat Tipo")
4
jjs> cars.splice(1, 2, "Rolls Royce")
Ford Focus,Fiat Punto
jjs> cars
Honda Civic,Rolls Royce,Fiat Tipo
```

JavaScript Arrays

In 2011 3 builtin functions were added to simplify array handling.

Each applies a function to array elements to implement their effect.

filter()	selects out array elements meeting a condition
map()	applies function to each array element
reduce()	applies accumulator/concatenator across array elements

General form of *filter()*

```
array.filter(function(elem, index, array) { ...  }, thisArg);
```

elem is element in *array* with current *index*.

thisArg is optional object nameable by *this* in callback.

Duplicate elements in array can be removed by *filter()* thus:

```
jjs> var bag = [4, 1, 4, 7, 1, 2]
jjs> bag.filter(function(e, i, ar) { return i === ar.indexOf(e); })
4,1,7,2
```

map() has same general syntax as *filter()* operation.

Centigrade temperatures can be converted to Fahrenheit by

```
jjs> var cent = [-40, 0, 35, 100];
jjs> cent.map(function(c) { return Math.round(9 * c / 5) + 32; });
-40,32,95,212
```

2 argument general form of *reduce()*

```
array.reduce(function(prevVal, initial) { ...  }, initial);
```

prevValue is cumulative value returned by each callback.

initial is optional object used as *prevVal* by first callback.

Elements of list can be summed by

```
jjs> var squares = [1, 4, 9, 16];
jjs> squares.reduce(function(prev, i) { return prev + i; }, 0);
30
```

JavaScript Objects

JavaScript objects are property collections that

- are *non-primitive* values
- have *prototypes* that are also objects
- *inherit* their properties and methods from their prototype

All objects inherit from prototype called *Object.prototype*.

Create an object with *new* keyword and assign properties to it:

```
vehicle = new Object();
vehicle.make = "Ford";
vehicle.model = "Focus";
vehicle.year = 2012;
```

vehicle can also be declared and initialised directly

```
vehicle = { make: "Ford", model: "Focus", year: 2012 };
```

Java uses items within { } lists to initialise arrays not objects.

```
// Java array initializer
String[] vehicle = { "Ford", "Focus", "2012" };
```

Associative array syntax in JavaScript can also create *vehicle*

```
vehicle["make"] = "Ford";
vehicle["model"] = "Focus";
vehicle["year"] = 2012;
```

Object prototype is standardly created by a constructor function.

car.js defines a constructor for object instances

```
function car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

that can be used with *new* keyword in expressions such as

```
vehicle = new car("lamborghini", "gallardo", 2013);
```

JavaScript Objects

Object's properties can be listed by *getProps.js*

```
function getProps(obj) {
  var props = [];
  for (var p in obj) { props.push(p); }
  return props;
}
```

car prototype can be dynamically changed to set default values

```
jjs> load("car.js")
jjs> load("getProps.js")
jjs> v = new car("porsche", 911, 2016)
[object Object]
jjs> getProps(v)
make,model,year
jjs> car.prototype.colour = "blue"
blue
jjs> getProps(v)
make,model,year,colour
jjs> getProps(v).map(function(e){return v[e];})
porsche,911,2016,blue
jjs> wheels = new car("Jaguar", "E-type", 1961)
[object Object]
jjs> wheels.colour
blue
jjs> delete wheels.year
true
jjs> getProps(wheels)
make,model,colour
```

Functions can be added to object prototype as follows

```
jjs> car.prototype.name = function() { return this.make + " " +
                                        this.model;}
jjs> wheels.name()
Jaguar E-type
```

Keyword *this* refers to object that owns JavaScript it occurs in

- in function *this* refers to object that owns function
- in object *this* refers to object itself

Global object has some useful meta-programming functions

eval()	evaluates string argument and executes it as code
isNaN()	determines whether argument is not a number

JavaScript Objects

ES 6 introduced class notation to JavaScript where

- classes are syntactic sugar over prototype-based objects
- no new object-oriented model is introduced

Class declaration *circle.js* can define a class function

```
class Circle {
  constructor(radius, x, y) { this.radius = radius;
                              this.x = x; this.y = y; }

  area() { return Math.PI * Math.pow(this.radius, 2); }

  static info() { console.log("I know about circles"); }
}
```

A class expression (with/without a name) can also define it

```
var ring = class Circle { ... }
```

First version of class can be exercised with Node REPL

```
unix% node
> .load circle.js
> Circle.info()
I know about circles
> let ring = new Circle(10, 0, 0)
> ring.area()
314.1592653589793
```

nicecircle.js shows how ES 6+ classes can be subclassed

```
class NiceCircle extends Circle {
  constructor(radius, colour) {
    super(radius, 0, 0);
    this.colour = colour;
  }
}
```

In derived classes *super()* must be called before *this*.

```
unix% node
> .load circle.js
> .load nicecircle.js
> var hoop = new NiceCircle(12, "red")
> hoop.area()
452.3893421169302
> hoop.colour
```

Nashorn and Java

Nashorn supports Java APIs via their fully qualified names through

- calls to static Java methods of standard classes
- creation of Java objects and then calls on their instance methods

Current working directory can be got from Java System property

```
jjs> java.lang.System.getProperty("user.dir")
/home/msc/bozo
```

JavaScript to list OS variable values can be put in file *env.js*:

```
var keys = java.lang.System.getenv().keySet().toArray();
for (i in keys)
  print(keys[i] + " = " + java.lang.System.getenv(keys[i]));
```

File *env.js* can then be executed by JavaScript shell

```
unix% jjs env.js
```

env.js can also be invoked via web.

listFile.js lists a file using Java IO APIs

```
function listFile(file) {
  try {
    var fis = new java.io.FileInputStream(file);
    var isr = new java.io.InputStreamReader(fis);
    var br = new java.io.BufferedReader(isr);
    while ( (s = br.readLine()) != null )
      print(s);
  } catch (exc) {
    print(exc);
  }
}
```

Note its use of exception handling. It can be invoked by

```
unix% jjs
jjs> load("listFile.js")
 ...
jjs> listFile("go")
java.io.FileNotFoundException: go (No such file or directory)
jjs> listFile("/etc/passwd")
root:x:0:0:root:/root:/bin/bash
 ...
```

Things To Do

Try Out REPL JavaScript Engine
http://www2.macs.hw.ac.uk/~hamish/js.html

Browse Nashorn JavaScript Tutorial
https://winterbe.com/posts/2014/04/05/java8-nashorn-tutorial/

Browse *ECMAScriptÂ® 2018 Language Specification*
http://www.ecma-international.org/ecma-262/9.0/

Try *Quiz on Lecture*

Do *Exercise 5*

Key Points

- JavaScript is a weakly typed, object-based (prototypes) language, standardised as ECMAScript. It was developed in 1995 for web client programming. It is nowadays also used for I/O bound standalone purposes especially web services.

- JavaScript has a C-like syntax, is multi-threaded, supports arrays, functions, objects and typical procedural control structures. Its primitive types are booleans, numbers, strings, symbols, null and undefined. Everything else is an object.

- While JavaScript was designed to be interpreted, modern JavaScript engines like Nashorn and V8 support just-in-time compilation as well.