Chp2- JavaScript/ECMAScript

Core Notions - Usage

JavaScript/ECMAScript

INTRODUCTION TO JAVASCRIPT

A bit of History…

Introduction to JavaScript

- Netscape Communications, the company that released Netscape (the most popular web browser in the 90s) needed a « glue language » for HTML to make the web more dynamic

- This language must be:
  - Easy to use by web designers and programmers
  - Capable of assembling components (images, plugins,..) where the code could be written directly in the Web page markup

- Brendan Eich, recruited in 1995, developed a prototype in 10 days

Naming History

Introduction to JavaScript

- The language was first developed under the name **Mocha**
- When first shipped in Beta releases of Netscape Navigator 2.0, its official name was **LiveScript**
- It was renamed to **JavaScript** when deployed to the Netscape Navigator 2.0 Beta 3
  - It has no link whatsoever with the Java language, in syntax, paradigm or behaviour
  - Just a marketing choice, as Java was very famous at the time
- In 1996, Netscape submitted JavaScript to *Ecma International* to carve out a standard specification, leading to the official standard language specification name: **ECMAScript**

What is JavaScript?

Introduction to JavaScript

- JavaScript is a programming language used to make web pages interactive
- It runs on your computer and doesn't require constant downloads from your website
- One of the must-learn web languages:
  - HTML for specifying the content
  - CSS for specifying the presentation
  - JS for specifying the behaviour
- JavaScript is:
  - High-level
  - Dynamic
  - Untyped
  - Interpreted
  - Object-oriented and functional

Compiled vs Interpreted Language (?)

Introduction to JavaScript

- The title is not relevant: any language can be either compiled or interpreted
  - Compilation and Interpretation are implementation techniques, not attributes of a language
- Compilation (ex: C, C++, Assembler)
  - A compiler will translate the program directly into code that is *specific to the target machine* (executable), which is also known as machine code – basically code that is specific to a given processor and operating system
  - The computer runs then the machine code on its own.
  - Steps:
    - Source code is analyzed and parsed: typos and spelling errors are detected at this point
    - An in-memory representation is generated: detection of semantic mistakes
    - A code generator produces executable code
    - The code is then executed
    - An optimization of the code can happen at any stage of the compilation

Compiled vs Interpreted Language (?)

Introduction to JavaScript

- Interpretation (ex: Perl, Python, PHP, Posix)
  - The source code is not directly run by the target machine
  - Another program (*interpreter*) reads and then executes the original source code
  - The interpreter is usually written specifically for the native machine
  - In interpretation, the original source code is also typically converted into some **intermediate code** which is then processed by an interpreter that converts the intermediate code into machine specific code « as it goes » .
  - As the interpreter parses the code, it will look up the corresponding function and **execute it**
- Some implementations can be compiled and interpreted at once
  - Some programs are compiled into an intermediate language that is then interpreted (*ex. Java, translated to byte-code, which is interpreted*)
  - Some other programs are directly translated into executable language that is run immediatly

JavaScript/ECMAScript

CORE JAVASCRIPT

Lexical Structure

Core JavaScript

- Character Set
  - JS programs are written using the unicode character set
    - Supports virtually the characters of all the written language in the planet
  - JS is case sensitive, contrary to HTML for example
    - HTML tags and attributes, called from client-side JS, must be represented in lowercase, even though they can use any case in HTML
  - JS ignores spaces and line breaks between tokens in programs
    - Formatting is recommanded
  - Tabulation, vertical tab, form feed, non-breaking space,.. are considered as whitespaces
- Comments
  - // : comments all the text until the end of the line
  - /* text */ : comments the text in-between

Lexical Structure

Core JavaScript

- Identifiers
  - Must begin with a letter, an underscore(_) or a dollar sign ($)
  - Subsequent characters can be letters, digits, underscores or dollar signs
- Reserved words
  - Certain identifiers are reserved for the language as keywords

```
break       delete      function    return      typeof
case        do          if          switch      var
catch       else        in          this        void
continue    false       instanceof  throw       while
debugger    finally     new         true        with
default     for         null        try
```

  - Some reserved words are not used currently, but could be in future versions

```
class    const    enum     export   extends  import   super
```

Lexical Structure

Core JavaScript

- Semi-colon
  - Semi-colons are important to make the meaning of your code clear
  - They can be omitted
    - Between two statements if they are in two different lines
    - At the end of the program if the next token is a }
  - JavaScript interprets line breaks as semicolons when it cannot parse the second line as a continuation of the statement on the first line, except:
    - With *return, break* and *continue* statements (a line-break always means a semi-colon)
    - With ++ and -- operators (because they can be postfix or prefix)

Numbers

Core JavaScript

- JS does not make a distinction between integer values and floating-point values
  - All numbers are floating-point values
- A number appearing directly in JS is called a *numeric literal*
- Syntax for a numeric literal

```
[digits][.digits][(E|e)[(+|-)]digits]
```

- Arithmetic operations
  - `+, -, *, /, %`
  - Math object => `pow, round, abs, max, min, random, log,…`
  - `Infinity, NaN,…`

Dates and Times

Core JavaScript

- Date()

```
var then = new Date(2010, 0, 1);          // The 1st day of the 1st month
                                             of 2010
var later = new Date(2010, 0, 1,          // Same day, at 5:10:30pm, local
              17, 10, 30);                   time
var now = new Date();                     // The current date and time
var elapsed = now - then;                 // Date substraction: interval in
                                             milliseconds
later.getFullYear()                       // => 2010
later.getMonth()                          // => 0: zero-based months
later.getDate()                           // => 1: one-based days
later.getDay()                            // => 5: day of week.
                                             0 is Sunday 5 is Friday.
later.getHours()                          // => 17: 5pm, local time
later.getUTCHours()                       // hours in UTC time;
                                             depends on timezone
```

Text
Core JavaScript

- Strings can be enclosed either between ' or "
- In ECMAScript 5, strings can be represented across multiple lines by ending each line (but the last) with \
- Escape sequences: special characters preceded by \
  - `\n, \', \t, \\,…`
- String concatenation with `+`
- To determine the number of characters in a string s: `s.length`
- Other possible methods:

```
var s = "hello, world"          // Start with some text.
s.charAt(0)                     // => "h": the first character.
s.charAt(s.length-1)            // => "d": the last character.
s.substring(1,4)                // => "ell": the 2nd, 3rd and 4th characters.
s.slice(1,4)                    // => "ell": same thing
s.slice(-3)                     // => "rld": last 3 characters
s.indexOf("l")                  // => 2: position of first letter l.
s.lastIndexOf("l")              // => 10: position of last letter l.
s.indexOf("l", 3)               // => 3: position of first "l" at or after 3
s[0]                            // => "h"
s[s.length-1]                   // => "d"
```

Boolean Values

Core JavaScript

- Only two possible values: true or false
- Generally the result of comparisons (ex. `a==4`)
- Commonly used in control structures: `if (a==4){ //do}`
- A JS value can be converted to a boolean value, for example
  - Values evaluated to false: `undefined, null, 0, -0, NaN, ""`
  - All other values are converted to true
- Boolean values can be converted to strings using the `toString()` method
- Operations on booleans
  - && : AND operation
  - || : OR operation
  - ! : NOT operator

null and undefined

Core JavaScript

- null
  - Evaluates to the absence of a value
  - Special object value that indicates « no object »
  - `typeof(null)` `// => « object »`
  - Considered to be the sole member of its type
- undefined
  - Value of
    - the variables that have not been initialized,
    - an object's attribute or an array that doesn't exist
    - functions that have no return values
    - function parameters of which no argument is supplied
- Difference
  - undefined: represents a system-level, unexpected or error-like absence of value
  - null: program-level, normal or expected absence of value

Type Conversion
Core JavaScript

- Implicit conversion
  - When JS expects a value of a certain type (like boolean or string), it will convert whatever value is given to the needed type
  - For numbers, if the supplied value cannot be converted, the variable is converted to a NaN

```
10 + " objects"           // => "10 objects". Number 10 converts to a string
"7" * "4"                 // => 28: both strings convert to numbers
var n = 1 - "x";          // => NaN: string "x" can't convert to a number
n + " objects"            // => "NaN objects": NaN converts to string "NaN"
```

- Explicit conversion
  - Usage of : `Boolean(), Number(), String(), Object()`
  - Usage of `+ or !!`

```
Number("3")               // => 3
String(false)             // => "false" Or use false.toString()
Boolean([])               // => true
Object(3)                 // => new Number(3)
x + ""                    // Same as String(x)
+x                        // Same as Number(x). You may also see x-0
!!x                       // Same as Boolean(x). Note double !
```

Type Conversion
Core JavaScript

- Other conversions

```
var n = 17;
binary_string = n.toString(2);          // Evaluates to "10001"
octal_string = "0" + n.toString(8);     // Evaluates to "021"
hex_string = "0x" + n.toString(16);     // Evaluates to "0x11 »

var n = 123456.789;
n.toFixed(0);                           // "123457"
n.toFixed(2);                           // "123456.79"
n.toExponential(1);                     // "1.2e+5"
n.toExponential(3);                     // "1.235e+5”

parseInt("3 blind mice")                // => 3
parseFloat(" 3.14 meters")              // => 3.14
parseInt("-12.34")                      // => -12
parseInt(".1")                          // => NaN: integers can't start with "."
parseFloat("$72.47");                   // => NaN: numbers can't start with "$”

parseInt("11", 2);                      // => 3 (1*2 + 1), converts to base 2
parseInt("ff", 16);                     // => 255 (15*16 + 15)
```

Variable Declaration

Core JavaScript

- Before using a variable in JS, you should declare it
  - Usage of the *var* keyword
- Variable declaration can be combined with its initialization
  - If not, the initial value is *undefined*
- The declaration statement can appear in the *for* or *for/in* loops
  - The counter is then declared as being part of the loop syntax itself
- No type is associated with the variable, and a variable can change the type of value it holds at runtime
- It is legal and harmless to declare a variable more than once
- If a variable is undeclared
  - Reading it generates an error
  - Assigning a value to it makes it a **global value** (if not in strict mode)

Variable Scope

Core JavaScript

- The **scope** of a variable is the region of your program source code in which it is defined.
- Global variables
  - Have a global scope: defined everywhere in your JS code
- Local variables
  - Declared in the body of the function
  - Defined throughout the function, and also within any functions nested in it
  - You should always use *var* for local variables
- JS has no *block scope*!

```
function test(o) {
      var i = 0;                              // i is defined throughout function
      if (typeof o == "object") {
            var j = 0;                        // j is defined everywhere, not just block
            for (var k=0; k < 10; k++) {      // k is defined everywhere, not just  loop
                  console.log(k);             // print numbers 0 through 9
            }
            console.log(k);                   // k is still defined: prints 10
      }
      console.log(j);                         // j is defined, but may not be initialized
}
```

Variable Scope

Core JavaScript

- Local variables are visible throughout the function, even before their declaration!
- This feature is called *hoisting*
  - JS behaves as if all variable declarations in a function (but not any associated assignments) are "hoisted" to the top of the function

```
var scope = "global";
function f() {
     console.log(scope);                  // Prints "undefined", not "global"
     var scope = "local";                 // Variable initialized here, but defined
                                             everywhere
     console.log(scope);                  // Prints "local"
}
```

Expressions
Core JavaScript

- Expression
  - A phrase of JS that the interpreter can evaluate to produce a value
  - Complex expressions are built from simple expressions, thanks to operators for example

- Primary expressions
  - Those that stand alone
    - Constants or *literal* values
      - Numbers (1.23), strings (« hello »), regular expressions (/[1-9][0-9]*/), …
    - Certain language keywords
      - true, false, null, this…
    - Variable references
      - i, sum, undefined…

Expressions : Array Literals

Core JavaScript

- Not primary literals, as they include subexpressions that specify property and element values
- Array initializer
  - Comma separated list of expressions contained in square brackets
  - The value of an array initializer is a newly created array.
  - The elements of this new array are initialized to the values of the comma-separated expressions
  - The element expressions in an array initializer can themselves be array initializers

```
[]                                          // An empty array: no expressions inside
                                               brackets means no elements
[1+2,3+4]                                   // A 2-element array. First element is 3,
                                               second is 7

var matrix = [[1,2,3], [4,5,6], [7,8,9]];
var sparseArray = [1,,,,5];                 // sparseArray[1]==undefined
```

Expressions : Object Literals

Core JavaScript

- Object initializer
  - Like initializer expressions, but the square brackets are replaced by curly brackets
  - Each subexpression is prefixed with a property name and a colon
  - Object literals can be nested
  - The property names in object literals may be strings rather than identifiers
    - Useful to specify property names that are reserved words or are otherwise not legal identifiers

```
var p = { x:2.3, y:-1.2 };                          // An object with 2 properties
var q = {};                                         // An empty object with no properties
q.x = 2.3; q.y = -1.2;                              // Now q has the same properties as p

var rectangle = {                                   // Nested object literals
     upperLeft: { x: 2, y: 2 },
     lowerRight: { x: 4, y: 5 } };

var side = 1;
var square = { "upperLeft": { x: p.x, y: p.y },
               'lowerRight': { x: p.x + side, y: p.y + side}};
```

Other Expressions
Core JavaScript

- Function Literals
  - Keyword function, followed by a comma-separated list of zero or more identifiers (parameters) between parentheses, and a block of JS code
  - Can include the function name

```
// This function returns the square of the value passed to it.
var square = function (x) { return x * x; }
```

- Property Access Expressions
  - The value of an object property
    - `expression.identifier`
  - The value of an array element
    - `expression[expression]`
- Invocation Expressions
  - Calling a function or method

```
var o = {x:1,y:{z:3}};   // Object
var a = [o,4,[5,6]];     // Array
o.x                      // => 1
o.y.z                    // => 3
o["x"]                   // => 1
a[1]                     // => 4
a[2]["1"]                // => 6
a[0].x                   // => 1
```

```
f(0)
Math.max(x,y,z)
a.sort()
```

Relational Expressions

Core JavaScript

- Operators that test for a relationship between two values and return true or false
- Equality
  - == : equality operator (relaxed)
  - === : strict equality operator
- Inequality
  - != : relaxed inequality
  - !== : strict inequality

```
undefined == null          // true
undefined === null         // false
2 == "2"                   // true
2 === "2"                  // false
"1"  != false              // true
```

in Operator

Core JavaScript

- in operator
  - Expects a left-side operand that is or can be converted to a string
  - Expects a right-side operand that is an object
  - Evaluates to true if the left-side value is a property of the object on the right side

```
var point = { x:1, y:1 };          // Define an object
"x" in point                       // => true: object has property named "x"
"z" in point                       // => false: object has no "z" property.
"toString" in point                // => true: object inherits toString method
var data = [7,8,9];                // An array with elements 0, 1, and 2
"0" in data                        // => true: array has an element "0"
1 in data                          // => true: numbers are converted to strings
3 in data                          // => false: no element 3
```

instanceof Operator

Core JavaScript

- instanceof operator
  - Expects a left-side operand that is an object
  - Expects a right-side operand that identifies a class of objects
    - A class is defined by the constructor function that initializes its objects
  - Evaluates to true if the left-side object is an instance of the right-side class

```
var d = new Date();       // Create a new object with the Date() constructor
d instanceof Date;        // Evaluates to true; d was created with Date()
d instanceof Object;      // Evaluates to true; all objects are instances of Object
d instanceof Number;      // Evaluates to false; d is not a Number object

var a = [1, 2, 3];        // Create an array with array literal syntax
a instanceof Array;       // Evaluates to true; a is an array
a instanceof Object;      // Evaluates to true; all arrays are objects
a instanceof RegExp;      // Evaluates to false; arrays are not regular expressions
```

Miscelleneous Operators

Core JavaScript

- Conditional Operator
  - `x > 0 ? x : -x`
  - Stands for: `if (x>0) {return x} else {return –x}`

- typeof
  - Returns the type of the operand placed after it
  - Example: `typeof 3  // => number`

- delete
  - Deletes the object property or array element specified as its operand
  - Example:
    - `delete o.x  // deletes the property x of the object o`
    - `delete a[2]  // deletes the second element of the array a`

Conditionals

Core JavaScript

- Decision points of your code
- Also known as branches
- if / else / else if

```
if (n == 1) {
     // Execute code block #1
}
else if (n == 2) {
     // Execute code block #2
}
else if (n == 3) {
     // Execute code block #3
}
else {
     // If all else fails, execute block #4
}
```

- Switch

```
switch (n) {
     case 1:                // Start here if n == 1
          // Execute code block #1.
          break;            // Stop here
     case 2:                // Start here if n == 2
          // Execute code block #2.
          break;            // Stop here
     case 3:                // Start here if n == 3
          // Execute code block #3.
          break;            // Stop here
     default:               // If all else fails...
          // Execute code block #4.
          break;            // Stop here
}
```

Loops
Core JavaScript

- while

```
var count = 0;
while (count < 10) {
     console.log(count);
     count++;
}
```

- do/while

```
function printArray(a) {
     var len = a.length, i = 0;
     if (len == 0)
          console.log("Empty Array");
     else {
          do {
               console.log(a[i]);
          } while (++i < len);
     }
}
```

for

```
var i,j;
for(i = 0, j = 10 ; i < 10 ; i++, j--)
     sum += i * j;
```

for/in

```
var o=[0,1,2,3];
for (var p in o)
     console.log(o[p]);
```

Jumps
Core JavaScript

- Any statement may be labeled by preceding it with an identifier and a colon
- By labeling a statement, you give it a name that you can use to refer to it elsewhere in your program
- By giving a loop a name, you can use:
  - break : inside the body of the loop to exit the loop
  - continue: to jump directly to the top of the loop to begin the next iteration

```
mainloop: while (token != null) {
      // Code omitted...
      continue mainloop;       // Jump to the next iteration of mainloop
      // More code omitted...
}
```

Exceptions
Core JavaScript

- Exception:
  - Signal that indicates that some sort of exceptional condition or error has occurred
  - To throw an exception is to signal such an error or exceptional condition
  - To catch an exception is to handle it—to take whatever actions are necessary or appropriate to recover from the exception

```
function factorial (x) {
      // If the x is invalid, throw an exception!
      if (x < 0)
            throw new Error(
                  "x must not be negative");
      // Otherwise, compute and return normally
      for (var f = 1; x > 1; f *= x, x--);
      return f;
}
```

```
try {
      var n = Number(prompt("Please
            enter a positive integer"));
      var f = factorial(n);
      alert(n + "! = " + f);
}
catch (ex) {
      alert(ex);
}
```

Strict Mode

Core JavaScript

- Added in ECMAScript 5
  - Not supported for all browsers
- Uses a restricted variant of JS
  - Has different semantics from normal code
  - Replaces some silent errors with throw errors
  - Fixes mistakes in order to optimize the code
- Can be defined for entire scripts or individual functions

```
function strict(){
      // Function-level strict mode syntax
      'use strict';
      x = 3;                        //returns an error (x is not defined)
      function nested() { return "And so am I!"; }
      return "Hi! I'm a strict mode function! " + nested();
}

function notStrict() {
      y = 2;                        // y is created as a global variable
      return "I'm not strict.";
}
```

JavaScript/ECMAScript

OBJECTS IN JAVASCRIPT

Primitive Values vs Objects

Objects in JavaScript

- Primitives:
  - undefined, null, boolean, numbers, and strings
  - **Immutable** : their initial allocated memory location never changes
  - Are compared by **value**: two primitives are the same only if they have the same value

- Objects :
  - Anything else: objects, arrays and functions
  - **Mutable**: their memory location can change dynamically
  - Are compared by **reference**:
    - Two objects are equal if and only if they refer to the same underlying object
    - Two different objects having the same properties are not equal

Objects

Objects in JavaScript

- An object is a composite value
- It is an unordered collection of properties, each with a name and a value, that can be:
  - Attributes
  - Methods
- It inherits the properties of another object, its prototype
- Properties can be added and deleted dynamically
- Three types of objects:
  - Native Object: defined by ECMAScript specification (Arrays, functions, dates and regular expressions, for ex)
  - Host Object: defined by the host environment (such as web browser) where the interpreter is embedded (ex. HTMLElement in client-side JS)
  - User-defined Object: object created by the execution of JS code

Object Creation
Objects in JavaScript

- With Object Literals
  - Comma-separated list of colon-separated name:value pairs
  - Properties must be defined as string literals if:
    - They include spaces/hyphens
    - They are reserved words
- With the new operator
  - new must be followed by a function invocation (*a constructor*)
- With Object.create()
  - Creates a new object, using its first argument as the prototype of that object
  - To create an object that inherits no properties or methods (at all), pass null as an argument
  - To create an object that inherits the properties and methods of Object, use `Object.create(Object.prototype)`

```
// With object literals
var book = {
      "main title": "JavaScript",
      'sub-title': "The Definitive Guide",
      "for": "all audiences",
      author: {
            firstname: "David",
            surname: "Flanagan"
      }
};


// With new operator
var o = new Object();
var a = new Array();
var d = new Date();
var r = new RegExp("js");


// With Object.create()
var o1 = Object.create({x:1, y:2});
      // o1 inherits properties x and y.
var o2 = Object.create(null);
      // o2 inherits no props or methods.
var o3 = Object.create(Object.prototype);
      // o3 is like {} or new Object().
var o4 = Object.create(o1);
      // o4 inherits the prototype of o1
```

Manipulating Properties
Objects in JavaScript

- Querying and setting properties
  - Use the . or []

```
// Querying and Setting properties
var author = book.author;
var name = author.surname;
var title = book["main title"];
```

- Deleting properties
  - Removes a property from an object
  - Delete returns false if the property has its *configurable* attribute to *false*

```
// Deleting properties
delete book.author;
delete book["main title"];
delete Object.prototype;    //can't delete
```

- Testing and enumerating properties
  - Use the *in* operator to test if a property exists in an object, and to iterate over the properties
    - To access the value of the operator, use o[index], not o.index!

```
//Testing and enumerating properties
var o = { x: 10}
"x" in o                    // true
"y" in o                    // false

for (p in o) {
     console.log(p);        //=> x
     console.log(o[p]);     //=> 10
     console.log(o.p);      //=> undefined
}
```

Accessor Properties (Getters & Setters)

Objects in JavaScript

- Defined as one or two functions whose name is the same as the property name, with the *function* keyword replaced with get and/or set
- Is used, when called, like a data attribute

```
// This object generates strictly increasing serial numbers
var serialnum = {
      // This data property holds the next serial number.
      // The $ in the property name hints that it is a private property.
      $n: 0,
      // Return the current value and increment it
      get next() {   return ++this.$n;  },
      // Set a new value of n, but only if it is larger than current
      set next(n) {
            if (n >= this.$n) this.$n = n;
            else throw "serial number can only be set to a larger value";
      }
};
console.log(serialnum.next);                  // => 1
serialnum.next = 5;
console.log(serialnum.next);                  // => 6
```

Serializing Objects
Objects in JavaScript

- Object Serialization
  - Process of converting an object's state to a string from which it can later be restored
  - 2 functions:
    - *JSON.stringify(object)*: returns a string, representing the enumerable own properties of the object
    - *JSON.parse(object)*: restores the object from the JSON string

```
o = {x:1, y:{z:[false,null,""]}};        // Defines a test object
s = JSON.stringify(o);                   // s is '{"x":1,"y":{"z":[false,null,""]}}'
p = JSON.parse(s);                       // p is a deep copy of o
```

JavaScript/ECMAScript

ARRAYS IN JAVASCRIPT

An Array…

Arrays in JavaScript

- Is an ordered collection of values
  - Each value is called an element
  - Each element has a numeric value in the array, called index
- Is untyped
  - An element may be of any type
  - Different elements of the same array may be of different types
- Is dynamic
  - Grows or shrinks as needed
  - No need to declare a fixed size
- May be sparse
  - There may be gaps (undefined elements between defined ones)
- Has a length property
- Is a special type of objects, where indexes represent integer property names
- Inherits properties from Array.prototype

Array Creation

Arrays in JavaScript

- With an array literal

```
var empty = [];                          // An array with no elements
var primes = [2, 3, 5, 7, 11];           // An array with 5 numeric elements
var misc = [ 1.1, true, "a", ];          // 3 elements of various types +
                                              trailing comma

var base = 1024;
var table = [base, base+1, base+2, base+3];

var b = [[1,{x:1, y:2}], [2, {x:3, y:4}]];

var count = [1,,3];   // An array with 3 elements, the middle one undefined.
var undefs = [,,];    // An array with 2 elements, both undefined.
```

- With the Array() constructor

```
var a = new Array();         // creates an empty array with no elements
var a = new Array(10);       // preallocates space for a 10 elements array
var a = new Array(5, 4, 3, 2, 1, "testing, testing");  // pre-filled array
```

Manipulating Elements in an Array

Arrays in JavaScript

- Reading and writing elements
  - With the [] operator
- An array can be truncated by changing the value of the length property
- Adding and deleting elements
  - push(): adds one or more values to the end of an array
  - unshift(): adds a value at the beginning of the array
  - delete: deletes an element
- Iterating over an array
  - Use for loop or for/in

```
// Reading and Writing
var a = ["world"];
var value = a[0];
a[1] = 3.14;
i = 2;
a[i] = 3;
a[i + 1] = "hello";
a[a[i]] = a[0];

a.length          // => 4
a.length = 2      // a is now["world",3.14]

// Adding
a.push("zero");       //a[2]=="zero"
a.push(1,3);          //a[3]==1, a[4]==3

//Deleting
a = [1,2,3];
delete a[1];      // a now has no element
                     at index 1
1 in a            // => false: no array
                     index 1 is defined
a.length          // => 3: delete does not
                     affect array length

// Iterating
for (var index in sparseArray) {
     var value = sparseArray[index];
     // Do something with index and value
}
```

Predefined Array Methods

Arrays in JavaScript

join()	converts all elements to strings and concatenates them
reverse()	reverses the order of the elements
sort()	sorts the elements of the array (alpha. by default)
concat()	adds new elements at the end of the array
slice(i,j)	returns a subarray from the index i to j
splice(i,nb)	removes nb elements from index i, and returns them
push() & pop()	work with arrays as if they were stacks
shift() & unshift()	like pop and push, but at the beginning of the array
forEach()	iterates through array and executes a func for each elem
filter()	passes a func. determining which elements to extract
indexOf()	returns the index of the first element found

...

JavaScript/ECMAScript

FUNCTIONS IN JAVASCRIPT

A function …

Functions in JavaScript

- Is a special object
  - You can set properties to them, and invoke methods on them
- Is a block of JS code defined once but can be executed any number of times
- Is parametrized
  - Includes a set of identifiers, parameters, working as local variables
  - Function invocation provides arguments (values for these parameters)
- Often returns a computed value
- Has an invocation context
  - The value of the keyword this
- Can be assigned to the property of an object, thus being a method
  - When a function is invoked through an object, this object becomes its invocation context
- Can be designed to initialize a newly created object, thus being a constructor

Defining Functions
Functions in JavaScript

- Use the function keyword
- Function definition expression
  - A function literal which value is the newly defined function
  - The function name is not visible outside of its scope

```
var square = function(x) { return x*x; }
```

- Function declaration statement
  - Sibling of the variable declaration, but for functions.
  - (Officially) Prohibited into non-function blocks (in an **if** block, for example)

```
function square(x) { return x*x; }
```

- Be careful with hoisting!
  - All function declaration statements will be hoisted, but for function definition expressions, only the variable declaration is hoisted, not its init value (aka the function)

Defining Functions

Functions in JavaScript

```
function foo(){
    function bar() {
        return 3;
    }
    return bar();
    function bar() {
        return 8;
    }
}
alert(foo());

//returns 8

/************************************/

function foo(){
    var bar = function() {
        return 3;
    };
    return bar();
    var bar = function() {
        return 8;
    };
}
alert(foo());

//returns 3
```

```
alert(foo());
function foo(){
    var bar = function() {
        return 3;
    };
    return bar();
    var bar = function() {
        return 8;
    };
}

//returns 3

/************************************/

function foo(){
    return bar();
    var bar = function() {
        return 3;
    };
    var bar = function() {
        return 8;
    };
}
alert(foo());

//[Type Error: bar is not a function]
```

Defining Functions
Functions in JavaScript

- Nested functions
  - Functions may be nested within other functions
  - A nested function can access the parameters and variables of the function they are nested within

```
function hypotenuse(a, b) {
    function square(x) { return x*x; }
    return Math.sqrt(square(a) + square(b));
}
```

Invoking Functions

Functions in JavaScript

- The JavaScript code that makes up the body of a function is not executed when the function is defined but when it is invoked

- JavaScript functions can be invoked in four ways
  - As functions
  - As methods
  - As constructors
  - Indirectly through their **`call()`** and **`apply()`** methods

Invoking Functions

Functions in JavaScript

- Function invocation
  - Using an invocation expression
  - Each argument expression in the invocation expression is **evaluated**, and the resulting values become the arguments of the function
  - The return value of the function becomes the value of the invocation expression
    - If the function returns because the interpreter reaches the end, the return value is undefined
    - If the function returns because the interpreter executes a return, the return value is the value of the expression

```
printprops({x:1});
var total = distance(0,0,2,1) + distance(2,1,3,5);
var probability = factorial(5)/factorial(13);
```

Invoking Functions

Functions in JavaScript

- Method invocation
  - Method: JS function stored in a property of an object
  - Using an method invocation expression
  - Differ from function invocations in their invocation context
    - The object used to access the method becomes its invocation context
    - The function body can refer to that object using *this*
  - Other ways to use method invocation
    - `o["m"](x,y); // Another way to write o.m(x,y).`
    - `a[0](z) // assuming a[0] is a function.`
    - `f().m(); // Invoke method m() on return value of f()`

```
var calculator = {        // An object literal
     operand1: 1,
     operand2: 1,
     add: function() {
          // Note the use of the this keyword to refer to this object.
          this.result = this.operand1 + this.operand2;
     }
};
calculator.add();         // A method invocation to compute 1+1.
calculator.result         // => 2
```

Invoking Functions
Functions in JavaScript

- Constructor invocation
  - If the function or method invocation is preceded by the keyword new, it is a constructor invocation
  - It creates a new empty object that inherits from the prototype property of the constructor
  - If the constructor includes an argument list, those argument expressions are evaluated and passed to the construction function
  - If a constructor has no parameters, it is possible to omit entirely the argument list and the parentheses

```
var o = new Object();
var o = new Object;       // equivalent to the previous one
```

Invoking Functions
Functions in JavaScript

- Indirect invocation
  - JS functions are objects, and thus, can have methods
  - Two predefined methods invoke the function indirectly: call() and apply()
  - Both enable invoking a temporary method on an object, without adding it to its prototype
    - call(): uses its own argument list are arguments of the function
    - apply(): expects an array of values to be used as arguments

```
f.call(o,1,2);
f.apply(o,[1,2]);

// Both equivalent to
o.m = f;          // Make f a temporary method of o.
o.m(1,2);         // Invoke it, passing 2 arguments.
delete o.m;       // Remove the temporary method.
```

Functions Arguments and Parameters

Functions in JavaScript

- No expected types for the parameters
- No type checking on the argument values at invocation
- No checking of the number of arguments at invocation
- If function invoked with fewer arguments than declared parameters
  - Additional parameters are set to undefined
  - Used when some parameters are optional, and a default value is assigned to them
  - Make sure tho declare the optional parameters at the end of the argument list

```
// Append the names of the enumerable properties of object o to the
// array a, and return a. If a is omitted, create and return a new array.

function getPropertyNames(o, /* optional */ a) {
     if (a === undefined) a = [];     // If undefined, use a new array
     for (var property in o) a.push(property);
     return a;
}
// This function can be invoked with 1 or 2 arguments:
var a = getPropertyNames(o);          // Get o's properties into a new array
getPropertyNames(p,a);                // append p's properties to that array
```

Functions Arguments and Parameters

Functions in JavaScript

- When a function is invoked with more argument values than there are parameter names, there is no way to directly refer to the unnamed values
- Use the `arguments` object
  - Array-like object that allows the argument values passed to the function to be retrieved by number rather than by name
  - Helps defining functions with any number of arguments

```
function max(/* ... */) {
     var max = Number.NEGATIVE_INFINITY;
     // Loop through the arguments, looking for, and remembering, the biggest.
     for (var i = 0; i < arguments.length; i++)
          if (arguments[i] > max) max = arguments[i];
     // Return the biggest
     return max;
}

var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6);       // => 10000
```

Functions as Values

Functions in JavaScript

```
// We define some simple functions here
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Here's a function that takes one of the
     above functions as an argument and
     invokes it on two operands
function operate(operator, operand1, operand2)
{
     return operator(operand1, operand2);
}
// We could invoke this function like this to
     compute the value (2+3) + (4*5):
var i = operate(add, operate(add, 2, 3),
          operate(multiply, 4, 5));
// We implement the simple functions again,
     this time using function literals within
     an object literal;
var operators = {
     add: function(x,y) { return x+y; },
     subtract: function(x,y) { return x-y; },
     multiply: function(x,y) { return x*y; },
     divide: function(x,y) { return x/y; },
     pow: Math.pow
};
```

```
// This function takes the name of an operator,
looks up that operator in the object, and then
invokes it on the supplied operands. Note the
syntax used to invoke the operator function.

function operate2(operation, operand1, operand2)
{
     if (typeof operators[operation] ===
          "function")
     return operators[operation](operand1,
          operand2);
     else throw "unknown operator";
}
// Compute the value ("hello" + " " + "world")
     like this:

var j = operate2("add", "hello", operate2("add",
     " ", "world"));

// Using the predefined Math.pow() function:
var k = operate2("pow", 10, 2);
```

Functions Properties, Methods and Constructor

Functions in JavaScript

- The *length* property
  - **`arguments.length`** : number of arguments passed to the function
  - **`function.length`**: returns the arity of the function : number of parameters declared in the parameter list

```
// This function uses arguments.callee, which is .
function check(args) {
    var actual = args.length;          // The actual number of arguments
    var expected = args.callee.length; // The expected number of arguments
    if (actual !== expected)           // Throw an exception if they differ.
        throw Error("Expected " + expected + "args; got " + actual);
}
function f(x, y, z) {
    check(arguments); // Check that the actual # of args matches expected #.
    return x + y + z; // Now do the rest of the function normally.
}
```

- The *prototype* property
  - Displays the prototype object of the function

Functions Properties, Methods and Constructor

Functions in JavaScript

- The *bind()* method
  - Binds a function to an object
  - Returns a new function which, when invoked, executes the original function as a method of the bound object.

```
function f(y) { return this.x + y; }  // This function needs to be bound
var o = { x : 1 };                    // An object we'll bind to
var g = f.bind(o);                    // Calling g(x) invokes o.f(x)
g(2)                                  // => 3
```

- The *toString()* property
  - Returns a string that follows the syntax of the function declaration statement
  - Complete source code for the function
    - Even the comments!

Functions as Namespaces

Functions in JavaScript

- A function can be used as a namespace for variables
  - As there is no block scope, all variables outside of a function become global
- Example : suppose there is a module that will be used by several JS codes
  - Variables declared in the module will be global to any code it is included in
  - Can cause conflicts with existing code
- Solution: Using functions as namespaces
  - Solution 1: A named function that must be invoked later
  - Solution 2: An anonymous function, defined and invoked in a single statement

Functions as Namespaces

Functions in JavaScript

```
// Solution 1:

function mymodule() {
     // Module code goes here.
     // Any variables used by the module are local to this
     // function instead of cluttering up the global namespace.
}
mymodule();  // But don't forget to
             // invoke the function!



// Solution 2:

(function() {
     // mymodule function rewritten as an unnamed expression
     // Module code goes here.
}()); // end the function literal and invoke it now.
```

Higher-Order Functions

Functions in JavaScript

- A function that operates on functions, taking one or more functions as arguments and returning a new function
  - The function passed as a parameter is called a callback function

```
// Return a new function that computes f(g(...)).
// The returned function h passes all of its arguments to g, and then passes
// the return value of g to f, and then returns the return value of f.
// Both f and g are invoked with the same this value as h was invoked with.

function compose(f,g) {
      return function() {
            // We use call for f because we're passing a single value and
            // apply for g because we're passing an array of values.
            return f.call(this, g.apply(this, arguments));
      };
}
var square = function(x) { return x*x; };
var sum = function(x,y) { return x+y; };
var squareofsum = compose(square, sum);
squareofsum(2,3)                                    // => 25
```

JavaScript/ECMAScript

CLASSES AND MODULES

Classes and Prototypes

Classes and Modules

- In JS, it is possible to define objects without a class, making them distinct even if they share the same properties and methods

- Classes are based on JS prototype-based inheritance mechanism
  - If two objects inherit properties from the same prototype object:
    - They are instances of the same class
    - They are created and initialized by the same constructor function

- Classes and prototype-based inheritance in JS are different from strongly-typed OO programming languages like Java
  - They are dynamically extensible, for ex.

Classes and Prototypes

Classes and Modules

- Example of a function that creates a new object inheriting from the prototype of another object, with a lot of type checking to avoid errors

```
function inherit(p) {
      if (p == null) throw TypeError();      // p must be a non-null object
      if (Object.create)                     // If Object.create() is defined...
            return Object.create(p);         // then just use it.
      var t = typeof p;                      // Otherwise do some more type checking
      if (t !== "object" && t !== "function")
            throw TypeError();
      function f() {};                       // Define a dummy constructor function.
      f.prototype = p;                       // Set its prototype property to p.
      return new f();                        // Use f() to create an "heir" of p.
}
```

Classes and Prototypes

Classes and Modules

```
// range.js:
// This is a factory function that returns a
// new range object.
function range(from, to) {
      // Use the inherit() function to create
      // an object that inherits from the
      // prototype object defined below. The
      // prototype object is stored as
      // a property of this function, and
      //defines the shared methods (behavior)
      // for all range objects.

      var r = inherit(range.methods);
      // Store the start and end points (state)
      // of this new range object.
      // These are noninherited properties that
      // are unique to this object.

      r.from = from;
      r.to = to;

      // Finally return the new object
      return r;
}
```

```
// This prototype object defines methods
// inherited by all range objects.

range.methods = {
      // Return true if x is in the range, false
      // otherwise.
      // This method works for textual and Date
      // ranges as well as numeric.

      includes: function(x) {
            return this.from <= x && x <= this.to;
      },

      // Invokes f once for each integer in the
      // range.
      foreach: function(f) {
            for(var x = this.from; x <= this.to; x++)
                  f(x);
      }
};

// Here are example uses of a range object.
var r = range(1,3);      // Create a range object
r.includes(2);           // true: 2 is in the range
r.foreach(console.log);  // Prints 1 2 3
```

Classes and Constructors

Classes and Modules

- The preceding example is not the typical way to create a class, because it does not use a constructor

- Constructor
  - Function designed for the initialization of a newly created object
  - Invoked using the new keyword
  - The **`prototype`** property of the constructor is used as the prototype of the new object
    - All objects created with the same constructor inherit from the same object

Classes and Constructors

Classes and Modules

```
//range2.js:
// Another class representing a range of values
// This is a constructor function that
// initializes new Range objects.
// Note that it does not create or return the
// object. It just initializes this.

function Range(from, to) {
      // Store the start and end points (state)
      // of this new range object.
      // These are noninherited properties that
      // are unique to this object.

      this.from = from;
      this.to = to;
}
```

```
// All Range objects inherit from this object.
// Note that the property name must be "prototype"
// for this to work.

Range.prototype = {
      // Return true if x is in the range, false
      // otherwise. This method works for textual
      // and Date ranges as well as numeric.

      includes: function(x) {
            return this.from <= x && x <= this.to;
      },
      // Invoke f once for each int in the range.
      // This method works only for numeric ranges.

      foreach: function(f) {
            for(var x = this.from; x <= this.to; x++)
                  f(x);
      },
};

// Here are example uses of a range object
var r = new Range(1,3); // Create a range object
r.includes(2);          // true: 2 is in the range
r.foreach(console.log); // Prints 1 2 3
```

JavaScript/ECMAScript

RUNNING EXAMPLE

Todo App
Running Example

- Now that you are more familiar with JavaScript, we are going to create a whole application, starting from this chapter.
- Our goal is to create a classical ToDo Application, step by step.
- First, we are going to create the necessary JS functions and objects that enable us:
  - Displaying todos
  - Adding a todo
  - Removing a todo
  - Updating a todo.
- In order to do this, we will follow, together, the steps defined in the next slide.

Todo App
Running Example

- Step 1
  - Create a first version of the application, containing an array of strings (called *todos*) and the necessary four functions manipulating it
  - Make sure that after every modification of the array, the list of todos is displayed.
- Step 2
  - Create an object *todoList* whose attribute is the array *todos* and whose methods are the four functions defined above
- Step 3
  - Now, the array will not contain only strings, but also a boolean value *completed* that tells if the todo task is completed or not
  - Create a new *toggleCompleted* method, that changes the value of the *completed* attribute of a given task
- Step 4
  - Add a method called *toggleAll* that sets all *completed* attributes to *true* if at least one task is completed, and false otherwise
  - Add any utilitary method you use useful. Arrange your code and optimize it as possible.

References

- M. Haverbeke, *Eloquent JavaScript: A modern introduction to programming*, 2014

- Textbook
  - D. Flanagan, *JavaScript: The Definitive Guide*, 6th edition, O'reilly, 2011