Javascript

Marco Torchiano

Version 2.5.0 - April 2020

License 

Introduction

JavaScript

The programming language of the Web

- Developed in 1995 in Netscape (by Brendan Eich)
- Standardized in 1997 by ECMA as *ECMAScript*
  - Also ISO/IEC 16262
- Used both for
  - Client side: inside browsers
  - Server side: e.g. Node.js

JS main features

- Structured language
- Dynamic typing
  - type of variables are not declared
  - a variable can change its type upon assignment
- Run-time evaluation
  - compilation takes place when code is executed
- Object-based
  - support for creating objects
  - no explicit support for classes (until ES6)

History

- 1997, ES1 - First version of ECMAScript
- 1999, ES3 - Language improvements (RE,strings,try-catch)
- 2009, ES5 - Resolved ambiguities
- 2011, ES5.1 - Alignment with ISO/IEC 16262:2011
  - widely supported
- 2015, ES6/ES2015 - largely though not fully supported
  - rest/spread operators for variables
  - arrow functions (`=>`)
  - classes

History

- 2016, ES7/ES2016 - Minor changes (e.g. `**`)
- 2017, ES8/ES2017 - Several changes
  - features for concurrency and atomics
  - syntactic integration with promises (async/await)
- 2018, ES9/ES2018 - Several changes
  - asyncronous iteratiosn (`for await`)
  - regexp improvements
- 2019, ES10/ES2019 - Several changes
  - Arrays improvements

Integration with HTML

- Via the `<script>` element
  - Inline: code written inside the element
  - External: using the `src` attribute, conventionally to `.js` resource

    Note: when the `src` attribute is present all code included in the element is ignored!
- Inside values for event-related attributes
  e.g. `onload` attribute of the `<body>` element
- As *scriptlet*: bookmark a URL starting with `javascript:`

Programming environment

- Interactive (REPL)
  - Web browser console (Ctrl-Shift-I or ⌘ ⌥ I)
  - Node.js console (`node`)
- On-line:
  - Online editors, e.g. Code Pen or JS Fiddle
- Off-line:
  - Any editor or IDE (text editor, Web Storm, etc.)
    & Web server to serve the contents
    & Any browser to render and execute the contents

Fundamental concepts

Types

- `number`, e.g. `23` or `3.14`
  - usual arithmetic operations
  - stored as 64 bits floating point
- `boolean`, i.e. `true` or `false`
  - logical operators `&&`, `||`, and `!`
- `string`, e.g. `"PI"`, or `'PI'`
  - delimiters `"` and `'` are equivalent
  - quotation with backslash `\`
  - contatenation with plus `+`

Variables

- Variables are declared with
  - `var` as in `var three;` (scope = function)
  - `let` as in `let i;` (scope = block)
  - `const` as in `const PI=3.14`
- Type of variable defined upon each assignment
  - Function/operator `typeof` returns the type
- Special values:
  - `undefined`: variable not initialized
  - `null`: no value/object defined
  - `NaN`: invalid numerical value

`undefined` vs. unresolved

- variables declared but not initialized are assigned the special value `undefined`

```
var x; // = undefined
console.log(x);
```

↪ *undefined*

- variables never declared trigger errors

```
var x = undeclared_var + 1; // unresolved
```

⚠ `ReferenceError: undeclared_var is not defined`

Automatic type conversion

When applying operators to values / vars of differrent types an automatic conversion is applied

```
var s = "10";
s+1; ~"101"
```

```
(+s)+1; ~11
```

```
s*2+s; ~"2010"
```

```
!s; ~false
```

```
(+s)+!!s; ~11
```

Control

- Conditional:
  - `if`
- Loops:
  - `for`
  - `while`

  Both with `break` and `continue`
- Selection:
  - `switch`

Conditions

- Boolean expressions using logical connectors `!`, `&&`, `||`
- Comparison operators `==`, `!=`, `<`, `>`, `<=`, `>=`
- When a condition is required (e.g. in `if()`), any expression is converted to `boolean`
  - `""`, `0`, `undefined`, `null`, `NaN` is `false`
  - anything else converts to `true`

Conversion and comparison

- Comparison involves a conversion, JS selects the one with the least loss of precision (often `string`)

```
      "10" == 10   ; ~true
       "" == 0     ; ~true
 5   +   5  == 10   ; ~true
"1" +   0  == "10" ; ~true
"5" +  "5" == 55   ; ~true
 5  +  "5" == 55   ; ~true
 5  + +"5" == 10   ; ~true
```


Comparison and conversion

- The usual operators first convert, then compare

```
"200" > 33; ~true
"200" < "33"; ~true
```

- Use `===` and `!==` to avoid automatic type conversion

```
var s = "10";
var n = 10;
s == n; ~true
s === n; ~false
```

Quirks of logic connectors

- `||`
  - convert *lhs* operand to boolean, then
  - if `true` returns *lhs* operand (*unconverted!*)
  - if `false` returns *rhs* operand (*unconverted!*)
- `&&`
  - convert *lhs* operand to boolean, then
  - if `false` returns *lhs* operand (*unconverted!*)
  - if `true` returns *rhs* operand (*unconverted!*)

Logic or

```
      "a" || "b" ; ⇝ "a"
       "" || "b" ; ⇝ "b"
undefined || "b" ; ⇝ "b"
```

The behavior of `||` is like

```
function or( lhs, rhs ){
  if( lhs===false || lhs ==="" || lhs===0
      || lhs===undefined || lhs===null) return rhs;
  else return lhs;
}
```

Logic and

```
       "a" && "b" ; ~"b"
        "" && "b" ; ~""
undefined && "b" ;
```

The behavior of `&&` is like

```
function and( lhs, rhs ){
  if( lhs===false || lhs ==="" || lhs===0
      || lhs===undefined || lhs===null) return lhs;
  else return rhs;
}
```


Common Methods

Objects and methods

- A value in JS is not just a few bits stored in a memory location
- It is a complex structures called *Object*
- An *Object*
  - contains data (the bits)
  - provide methods
- A *Method* represent an operation performed on the object / value

Method invocation

- The operation represented by a method can be performed by *invoking* (calling) the method
- The sintax is similar to a function call:

  `object.method( arguments ... )`

  where `object` can be:
  - a simple value (e.g. `42`)
  - a variable (that refers to a value/object)

Number methods

- `toString()` converts it to string
  - the base can be passed as optional parameter
- `toFixed()` uses a fixed notation
  - number of decimal digits can be passed as optional parameter
- `toExponential()` uses the exponential notation
  - same parameter as `toFixed()`
- `toPrecision()` uses the given precision
  - optional parameter define the numer of digits

Number conversion examples

```
var PI = 3.1415;
PI.toString(); ↝"3.1415"
```

```
3.14.toString(); ↝"3.14"
(10).toString(2); ↝"1010"
```

```
PI.toFixed(2); ↝"3.14"
```

```
PI.toExponential(); ↝"3.1415e+0"
```

```
PI.toPrecision(2); ↝"3.1"
```

Strings methods

- `length` property containing the string length
- `charAt()` returns character at given position
- `slice()` extracts substring from pos `a` to pos `b` (excl.)
  - similar to `substring()`
- `indexOf()` position of substring
  - also `search()` that uses regular expressions
- `endsWith()` check if the string ends with argument
- `split()` divides a string at given separators

String examples

```
var asos = "A saucerful of secrets";
asos.length; ~22
```

```
asos.charAt(0); ~"A"
```

```
asos.slice(2,5); ~"sau"
```

```
asos.slice(15 /*,til end*/); ~"secrets"
```

```
asos.indexOf('u'); ~4
```

```
asos.endsWith('ets'); ~true
```

`Math` object

Methods:

- `min()`, `max()`
- `ceil()`, `floor()`, `round()`, `abs()`
- `pow()`, `sqrt()`
- `exp()`, `log()`, `log2()`, `log10()`
- `sin()`, `cos()`, `tan()`, `atan()` ...
- `random()`

Constants: `E`, `PI`

Dates

A `Date` object can be created with `new`

```
var d = new Date();
```

arguments can be:

- none: current date
- `ms`: date for the given ms afte Jan 1, 1970
- `string`: convert string into date
- `y, m, d, h, m, s, ms` or a subsequence

Dates

Methods are available to get components of the date

- `getDay()`, `getMonth()`, `getYear()`
- `getHours()`, `getMinutes()`, `getSeconds()`
- `toISOString()`, `toDateString()`, `toTimeString()`,
- `getTime()` (UTC milliseconds elapsed between 1 January 1970)
- `Date.now()`

Basic output

The common ways to generate output are:

- Using teh `window` to open a dialog
- Using the `console` support object
- `document.write()` insert content into the HTML page

`window` object

The `window` object in browsers offers a few basic methods:

- `alert()` opens an alert dialog
  - e.g. `alert('This is a message!')`
- `prompt()` opens an input dialog
  - e.g. `prompt('Enter a number:','42')`
- `confirm()` opens a confirmation dialog
  - e.g. `confirm('Are you sure?')`

`console` object

The `console` object offeres a few methods for output log messages:

- `log()`, `warn()`, `error()`

```
console.log("Log message");
console.warn("Warning message");
console.error("Error message");
```

`<script>` tag and `document.write()`

Add content inside document where it is executed

```
<p>Odd numbers:
<script>
for(let i=1; i<30; i+=2)
  document.write((i==1?"":", ") + i);
</script></p>
<p>Generated by a script!</p>
```


`document.write()`

- Can used within a `<script>` tag
- Content in placed right after the tag that executes it
- Must be executed **while** document is loading
    - once doc is loaded it erases the entire document!

```
<p>Before:<script>
here=function(){document.write("<b>HERE</b>");}
</script></p>
<p>After:
<script>here();</script></p>
```

Regular Expressions

- defined as `/expr/flags`
  - e.g. integer number `/0|[1-9][0-9]*/`
  - flags can be: `i` ignore case, `g` global, `m` multiline
- `exec()` find first match in a string
  - advances on each invocation if *global*
- `test()` check if a match is found in string

See Regular Expressions for details

Functions

Definition

- Introduced by the `function` keyword
- Return value with `return` keyword
- Invocation with `()`

```
function square(x){
  return x*x;
}
console.log(square(7));
```

↪ `49`

Functions as values

- Functions are yet another kind of object
- Functions can be declared *anonymously*
  - i.e. without a name
- Functions can be assigned as values to variables

```
var double = function(x){
    return 2*x;
};
double(21); ~ 42
```

- Functions can be returned by other functions

Function usage

- Invocation:

```
var four = double(2);
console.log(four)
```

↪
```
4
```

- Reference:

```
var twice = double;
console.log(twice)
```

↪
```
function(x){
    return 2*x;
}
```


Call-back functions

A functions passed as argument to a function that will be called-back when needed by the main function.

```
function pickIntegers(n, test_callback){
  let res="";
  for(let i=0; i<=n; ++i){
    if( test_callback(i) ) res += (i?", ":"") + i;
  }
  return res;
}
var three = function(x){ return x % 3 == 0;}
console.log( pickIntegers(12,three) );
```

↪
```
0, 3, 6, 9, 12
```

Arrow functions

`params => body`

- Body can be an expression or a code block `{}`
- Params can be a single name `x` or a list within `()`

```
var inv =  x => 1/x;
console.log(inv(5));
```

```
↪ 0.2
```

```
var add = (a,b) => { return a+b; };
console.log(add(12,30));
```

```
↪ 42
```


Lexical Scoping

- Functions define a scope
  - variables and functions belong to a scope
  - scopes (functions) can be nested
  - code blocks do not define scopes (as far as ES6)
- *Global* variables are defined at top level:
  - Can be accessed and modified by any function
  - Assignment to undeclared var creates a global one
- *Local* variables are defined inside a function (scope)
  - They can be accessed from within the function only

Lexical scoping example

```
function fg(){
  var x = 3;
  y = 2;
  return x+y;
}
console.log("y: " + y);
console.log("result: " + fg());
console.log("y: " + y);
```

⚠ `ReferenceError: y is not defined`

↪
```
result: 5
y: 2
```

Variable `y` is added at the *global* level

Strict mode

- To avoid declaring global var *by mistake* a code block can be declared as *strict mode*

```
function fg_sm(){
  "use strict";
  var x1 = 3;
  y1 = 2;
  return x1+y1;
}
fg_sm();
```

⚠ `ReferenceError: assignment to undeclared variable y1`

Scoping functions

- Anonymous functions that are immediately executed and then lost: `(function(){ ... })()`
  - avoid cluttering *global* environment
  - reduce risk of name clashes with other scripts

```
(function(){
  var x;
  x = 3;
  console.log("x="+x);
})()
```

```
↪ x=3
```


Arrays

Arrays

Creation:

- with `Array( len )`
- enumerated with the `[..]` syntax

```
var a = [ 1, 2, 3, 5, 10, 12];
```

  - Note: `[]` is the same as `Array(0)`

Access:

- Elements can be accessed by index through `[]`

Arrays iteration

- Lenght is available through property `length`

```
var sum = 0;
for(let i=0; i<a.length; ++i){
  sum += a[i];
} ~33
console.log( sum / a.length);
```

↪ `5.5`

Array methods

- `push()`: add at the end
- `pop()` : remove and return from end
- `join()`: concatenates elements with given separator
- `fill()`: set all elements to the given value

```
var names = ["Mario","Giuseppe","Joe"];
names.push("Jane"); ⇝4
names.join("; "); ⇝"Mario; Giuseppe; Joe; Jane"
```


Example arrays

Sieve of Eratosthenes

```
function primesES(n) {
  let primes = [1];
  let isPrime = Array(n+1).fill(true);
  for(let m = 2; m <= n; m++)
    if(isPrime[m]) {
      primes.push(m);
      for(let k = m * m; k <= n; k += m)
        isPrime[k] = false;
    }
  return primes;
}
```

Sorting arrays

Method `sort()` sorts arrays

```
var sary = "The quick brown fox".split(" ");
console.log(sary.sort())
```

↪ `["The", "brown", "fox", "quick"]`

Values are always converted to string before comparison

```
var nary = [ 12, 2, 3, 4, 10, 1];
console.log(nary.sort())
```

↪ `[1, 10, 12, 2, 3, 4]`

Sorting numeric arrays

A function can be provided to specify how to compare elements

```
nary.sort(function(a,b){ return a-b; });
console.log(nary);
```

↪ `[1, 2, 3, 4, 10, 12]`

or (using arrow function)

```
nary.sort( (a,b) => b-a );
console.log(nary);
```

↪ `[12, 10, 4, 3, 2, 1]`

Operations on arrays

- `slice(begin,end)` creates a copy containing elements from `begin` up to excluding `end`
  - `end` is optional, default is array `length`
  - `begin` is optional, default is `0`
    - `a.slice()` duplicates array `a`
- `splice(begin,count,items...)` removes and replaces `count` elements of the array
  - if `count` is omitted, all elements from `begin` on are removed
  - if `items` are missing, elements are just removed

Operations example

```
var oneToTen = [1,2,3,4,5,6,7,8,9,10];
console.log(oneToTen.slice(1,4));
oneToTen.splice(2,1,"three")
console.log(oneToTen);
var copy = oneToTen.slice();
copy.splice(3,1,"four");
console.log(oneToTen);
console.log(copy);
```

```
↪ [2, 3, 4]
[1, 2, "three", 4, 5, 6, 7, 8, 9, 10]
[1, 2, "three", 4, 5, 6, 7, 8, 9, 10]
[1, 2, "three", "four", 5, 6, 7, 8, 9, 10]
```

Mapping arrays

- `map(mapping)` creates an array where each element is computed from the original one by appling the `mapping` function
- `filter(predicate)` creates an copy of the original array that contains only the elements for which the `predicate` function returns `true`
- `join(separator)` creates a string by joining all the elements of the array using the `separator` string

Mapping example

```
var oneToTen = [1,2,3,4,5,6,7,8,9,10];
console.log(oneToTen.map( x => x*x ));
console.log(oneToTen.filter( x => x%2==0 ));
console.log(oneToTen.join("-:-"));
```

```
↪ [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
  [2, 4, 6, 8, 10]
  1-:-2-:-3-:-4-:-5-:-6-:-7-:-8-:-9-:-10
```

Objects

Properties

- Everything in JS is an object.
- Objects possess *properties* that can be accessed using
  - dotted notation (`.`)
  - subscript notation (`[]`)

```
var text = "Hello";
text.length; ↝ 5
text["length"]; ↝ 5
```

Objects manipulation

- Objects can be created by enumeration

```
var car = {
  brand:"Fiat",
  power:"120HP"
};
```

  - Note: enumeration requires `:`

- Properties can be added after creation:

```
car.model="500L"; ⇝"500L"
car.color="Silver"; ⇝"Silver"
```


Objects manipulation

- Properties can be deleted:

```
delete car.power; ⇝true
```

- Properties can be probed and accessed

```
"model" in car; ⇝true
car.model; ⇝"500L"
car.power===undefined; ⇝true
```

Objects as associative containers

An object allows associating keys to values.

```
var people = {
"RSSMRI95B23X987Y" : "Rossi Mario",
"GPPVRD99E47U876X" : "Verdi Giuseppina",
"FRCNRI97F52W765Z" : "Neri Federica"
};
var ssn = "FRCNRI97F52W765Z";
console.log(people[ssn]);
```

```
↪ Neri Federica
```


Iterating on properties

- *for-in* construct

```
for (let x in people) {
  console.log( x + " -> " + people[x]);
}
```

```
↪ RSSMRI95B23X987Y -> Rossi Mario
  GPPVRD99E47U876X -> Verdi Giuseppina
  FRCNRI97F52W765Z -> Neri Federica
```

Object properties

Properties values can be on their turn objects.

```
var people = {
"RSSMRI95B23X987Y" : {last:"Rossi", first:"Mario"},
"GPPVRD99E47U876X" : {last:"Verdi", first:"Giusy"},
"FRCNRI97F52W765Z" : {last:"Neri", first:"Federica"}
};
var ssn = "GPPVRD99E47U876X";
console.log(people[ssn]);
console.log(people[ssn].last+", "+people[ssn].first);
```

```
↪ {last:"Verdi", first:"Giusy"}
Verdi, Giusy
```


Methods

- Properties of type `function` are are called *methods*
- The special var `this` is defined within methods, it refers to the object upon which method was invoked.
  - `this` allows access to current object's properties
- *Methods* are invoked with the dotted notation

```
car.show = function(){
  return "This is a "+this.brand+" "+this.model;
}
console.log("Method -> "+car.show());
```

```
↪ Method -> This is a Fiat 500L
```

Creating objects

- Using an object literal

```
var car = {
  brand:"Fiat",
  model:"500L",
  color:"Silver"
};
```

- Using `new Object()` and adding properties

```
var car = new Object();
car.brand="Fiat";
car.model="500L";
car.color="Silver";
```


Constructors

- Definition of a constructor

```
function Car(brand,model,color){
  this.brand = brand;
  this.model = model;
  this.color = color;
}
```

- Usage of constructor

```
var car = new Car("Fiat","500L","Silver");
```

Classes

Since ES5 using `class` keyword

```
class Auto {
  constructor(brand,model,color){
    this.brand = brand;
    this.model = model;
    this.color = color;
  }
  show() {
    return "This is a "+this.brand+" "+this.model;
  }
}
```


Class usage

- Object can be created in the usual way

```
var c1 = new Auto("Tesla","Model 3","Red");
console.log(c1.show());
```

↪ `This is a Tesla Model 3`

- `class` allows a more compact notation without introducing new constructs

Properties

- Access to properties (as opposed to methods) can be defined using `set` and `get`

```
class Item {
  constructor(c) {
    this.validCols = ['Red','White','Black'];
    this.color=c; }
  get color() { return this._color; }
  set color(c) {
      if(this.validCols.includes(c)){
            this._color=c;
      }else{
        console.error("Invalid color: "+c)
} }   }
```


Properties

```
var m = new Item("Red");
console.log("Initial color: " + m.color);
m.color = "White"; → "White"
console.log("Now color is: " + m.color);
m.color = "Purple"; → "Purple"
console.log("Color is still: " + m.color);
var m1 = new Item("Purple");
console.log(m1.color);
```

```
Initial color: Red
Now color is: White
🛑 Invalid color: Purple
Color is still: White
🛑 Invalid color: Purple
undefined
```

Built-in constructors

- `Object()` same as `{}`
- `Array()` same as `[]`
- `RegExp()` same as `/()/`
- `Function()` same as `function (){}`
- `Date()`
- `String()` object version of primitive `""`
- `Number()` object version of primitive `0`
- `Boolean()` object version of primitive `true`

Generally primitives are better performing

Programming in the large

Exceptions

Mechanism to handle anomalies in a decoupled way

- Error handling code is kept separate from normal (nominal case) code
- Avoid using special return codes
- Anomaly are signled by rising exceptions

```
throw "Anomaly detected";
```

⚠ `Anomaly detected`

Exceptions

Exception catching and handling using `try{ }catch{ }`

```
function functionWithPotentialAnomalies(){
  if(true) throw "Easy to foresee..."
}
```

```
try{
  functionWithPotentialAnomalies();
}catch(error){
  console.error("Something happened: " + error);
}
```

↪ 🛑 `Something happened: Easy to foresee...`

Arguments

- The number of actual arguments is not enforced
- Fewer arguments than specified are accepted
  - all arguments are *de facto* optional
- Undefined arguments are replaced by `undefined`
  - operations on `undefined` evaluate to `NaN`

```
var x = double();
console.log(x)
```

```
↪ NaN
```


`arguments` object

- Extra arguments are allowed
  - they cannot be accessed through a named argument
  - the special object `arguments` can be used
- Number of arguments: `arguments.length`
- Access to `i`-th argument: `arguments[i]`

Missing arguments

- a missing argument is identified with `arg === undefined`
- This can be used to provide a default value

```
function increment(a, b) {
    if (b === undefined) {
        b = 1;
    }
    return a+b;
}
```


Default arguments

- A value can be specified to any argument to be computed when the argument is missing

```
function increment(a=0, b=1) {
    return a+b;
}
console.log(increment())
console.log(increment(10))
console.log(increment(2,2))
```

```
↪ 1
  11
  4
```

Coding conventions

- Use camelCase for identifier names
- All names start with a letter
- Put spaces around operators (= + - * /) and after commas
- Use 4 spaces for indentation of code blocks
- Use proper file extensions:
  - `.html`
  - `.css`
  - `.js`

Documentation

Functions can be documented using the *JSDoc* syntax

Documentation can be generated from special comment blocks starting with `/**`

```
/**
 * A function doubling its argument
 *
 * @param {number} x - value to be doubled
 */
function twoX(x){
  return 2 * x;
}
```

Parameter documentation

- `@param {type} name - description`

  describes a parameter, where `type` can be:

  - a primitive type, e.g. `string`, `number`, `boolean`
  - any type, e.g. `*`
  - an array, e.g. `Object[]`

- optional parameter is indicated as `[name]`
  - default value `[name=default]`

Documenting callbacks

When an argument is a callback function, its type must be defined separately

```
/**
 * @callback requestCallback
 * @param {string} content
 */
 /**
 * Async and call back on completion
 * @param {requestCallback} cb - The callback
 */
function doSomethingAsynchronously(cb) {
    // code
};
```

Advanced Functions and Closures

Functions

Every function is an object with a `Function` prototype.

- `call(obj, ...)` invoke a method on the object and passes the arguments
  - if `obj==null` it is a plain function call
- `apply(obj, args)` same as `call` but the arguments are the elements of the array `args`.

Apply functions example

- Enumerating arguments of a function:

```
Math.max(1, 5, 13, 8, 2, 3); ⇝13
```

- Using `apply()` to pass an array

```
var numbers = [1, 5, 13, 8, 2, 3];
Math.max.apply(null,numbers); ⇝13
```

- same as

```
Math.max(numbers[0],numbers[1],numbers[2],
         numbers[3],numbers[4],numbers[5]); ⇝13
```


Nested functions

Functions declarations can be nested

```
function dist(x0,y0,x1,y1){
  function squareDiff(a,b){
    return (a-b)*(a-b);
  }
  var sqx,sqy,res;
  sqx = squareDiff(x0,x1);
  sqy = squareDiff(y0,y1);
  res = Math.sqrt(sqy+sqx);
  return res;
}
```

Nested functions

A function can access variables defined in all of its enclosing scopes

```
function multiply(a,b){
  function by(x){ return x * a; }
  return by(b)
}
multiply(42,3); ~126
```


Nested functions (hoisted)

A function can access variables defined in all of its enclosing scopes

```
function multiply(a,b){
  return by(b)
  function by(x){ return x * a; }
}
multiply(42,3); ~126
```

Nested functions

Functions can access vars in all enclosing scopes

```
function quote(x,ql,qr){
  var res = "", i=0;
  if(Array.isArray(x))
    for(; i<x.length;++i) enclose(x[i]);
  else enclose(x);
  return res;
  function enclose(e){
      res += (i?", ":"") + ql + e + qr;
} }
quote("to be quoted","{","}"); ⇝"{to be quoted}"
quote(["a","b","c"],"{","}"); ⇝"{a}, {b}, {c}"
```


Return a function

A function can return a nested function

```
power = function(exponent){
  return function(x){
    return Math.pow(x,exponent);}
}
cube = power(3);
```

```
console.log( cube(2) );
```

↪ 8

Call stack

- Any time a function is invoked a new copy of its local variables and arguments is created on a stack
  - necessary for recursive functions
  - references are copied, not values
- When a function terminates the execution (i.e. it returns) the local variables are removed from the stack
  - as a consequence local variables are not persistent

Funarg problem

What happens if a nested function referring to a local variable or argument is returned by the outer function?

Funarg problem example

```
function buildGreeter(name) {
  var msg = "Hello " + name + "!";
  return function(){ return msg; };
}
var greeter = buildGreeter("Jon");
console.log(greeter());
```

↪ `Hello Jon!`

- once the `buildGreeter()` is terminated the variable `msg` *is* lost
- when the returned function `greeter()` is invoked, how can it retrieve `msg`?

Closures

Javascript makes the above code correct and working as epected using *closures*.

- a *closure* contains all the variables defined in the surrounding scope that are referenced by the nested function
- when a nested function is created its *closure* is created
- the function is linked to the *closure*
- the term *closure* is sometimes used to indicate a function that uses data from its closure

Returning a closure

```
function quoteFactory(qo,qc){
  return function qf(x){
      var res = "";
      if(Array.isArray(x))
        for(let i=0; i<x.length;++i)
          res+=(i?",":"")+ qf(x[i]);
      else res = qo + x + qc;
      return res;
  };
}
braces = quoteFactory("{","}");
braces("to be quoted") ~"{to be quoted}"
braces(["a","b","c"]) ~"{a},{b},{c}"
```

Returning a closure

Compact version:

```
function quoteFactory(qo,qc){
    return function qf(x){
        return Array.isArray(x)?
            x.reduce((a,i) => (a&&a+", ")+qf(i),"")
            : qo + x + qc;
    };
}
braces = quoteFactory("{","}");
braces("to be quoted") ⇝"{to be quoted}"
braces(["a","b","c"]) ⇝"{a}, {b}, {c}"
```


Closures vs. Objects

Closures vs. Objects

Closures are often used in place of objects

- when a single method is required
- because they are more compact
- sometimes harder to read

Example: an iterator on a sequence of numbers;

Object sequence

```
function Sequence(from,to){
    this.current=from;
    this.to=to;
    this.next = function() {
        if(this.current>this.to) return undefined;
        else return this.current++;
    }
}

var seq = new Sequence(1,10),res="",n;
while( (n = seq.next()) ){ res+=n + ","; }
console.log(res);
```

↪ `1,2,3,4,5,6,7,8,9,10,`

Closure sequence

```
function sequencer(from, to){
  return function() {
    if(from>to) return undefined;
    else return from++;
  };
}

var seq = sequencer(1,10),res="",n;
while( (n = seq()) ){ res += n+","; }
console.log(res);
```

The returned function uses the `from` argument as the current index. And returns `undefined` when the sequence has run out.

```
1,2,3,4,5,6,7,8,9,10,
```


Object iterator

```
function Iterator(from,to){
  this.current= from;
  this.to = to;
  this.next = function() {
      if(this.current>this.to) return undefined;
      else return this.current++;
    };
  this.hasNext = function() {
      return this.current<=to;
    };
}
```

This version adds the `hasNext` to build a standard iterator according to the Iterator Pattern

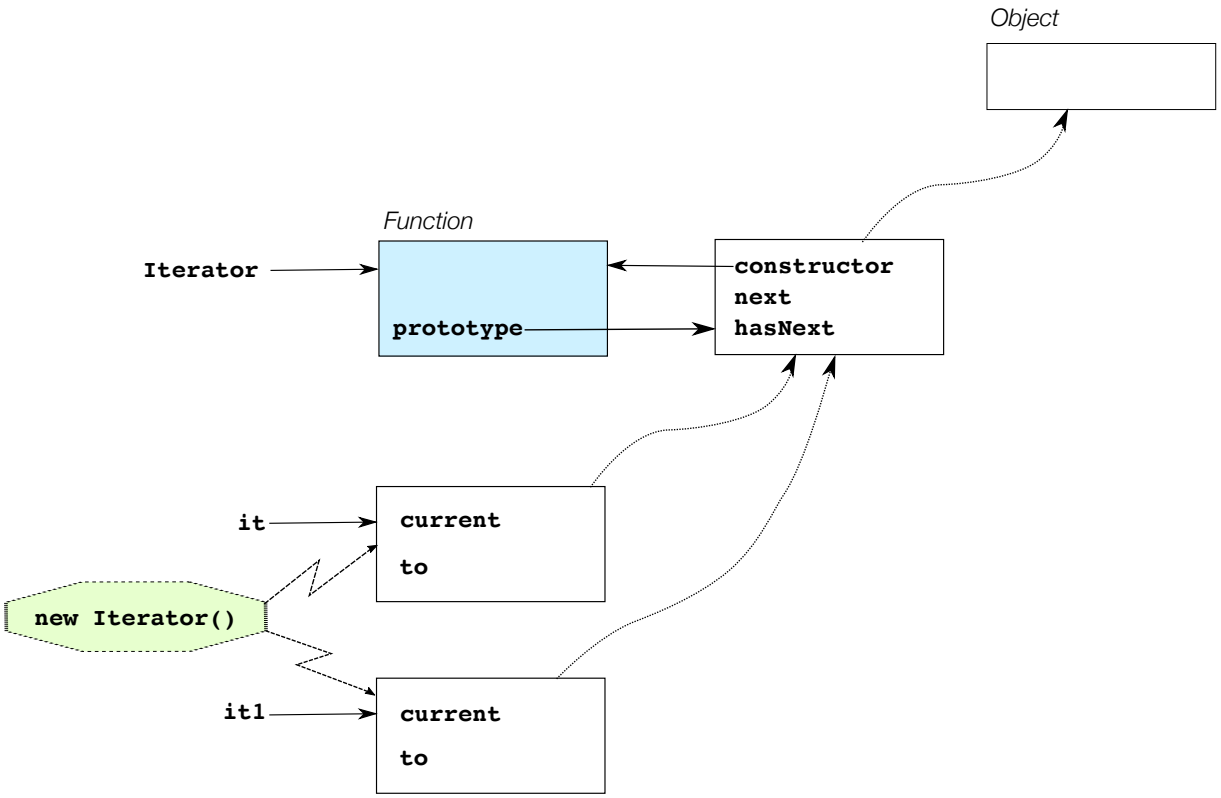
Object Prototypes

- Every JavaScript object has a prototype
- The prototype is also an object
- All JavaScript objects inherit their properties and methods from their prototype
- Prototypes can be used to host properties/methods shared by objects

Iterator with prototype

```
function Iterator(from,to){
  this.current = from;
  this.from = from;
  this.to = to;
}
Iterator.prototype.next = function() {
      if(this.current>this.to) return undefined;
      else return this.current++;
    };
Iterator.prototype.hasNext = function() {
      return this.current<=this.to;
    };
```

This version defines the methods inside the prototype.



Prototypes and dynamic binding

The prototype chain is used to resolve properties:

- `it.current`: the property is directly present in the object
- `it.next()`: the methods is not present in the object, it is found in the prototype
- `it.toString()`: the method is not present in the object, it is not present in the prototype, it can be found in the prototype's prototype, i.e. `Object`

`hasOwnProperty()` checks for a property in object only.

Override

- A property/method in a prototype can be overridden in an object

```
var it = new Iterator(1,100);
console.log("Before override: " + it.toString())
Iterator.prototype.toString = function(){
  return "[Iterator (" + this.from + "; " +
           this.to +  ") @ " + this.current + "]";
}
console.log("After override: " + it.toString())
```

```
Before override: [object Object]
After override: [Iterator (1; 100) @ 1]
```


References

- ECMA, 2015. Standart ECMA-262 - ECMAScript 2015 Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
- W3Schools. JavaScript Tutorial. http://www.w3schools.com/js/default.asp
- MDN. Javascript Documentation. https://developer.mozilla.org/en/docs/Web/JavaScript
- JSDoc Documentation. https://jsdoc.app