Programación por Prototipos

Junio, 2017 - Semana 1

Agenda

- Que és?
- Prototipos vs Extensión de clases
- Herencia vs delegación
- Funciones como componente de primera clase
- Alcances globales y localizados

Qué es?

OOP: es un paradigma de programación que usa objetos y sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, abstracción, polimorfismo y encapsulamiento.

Qué es?

OOP: es un paradigma de programación que usa objetos y sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, abstracción, polimorfismo y encapsulamiento.
Pero no necesariamente tiene que implementarse mediante clases

La programación por prototipos es una forma de programación orientada a objetos en el que el rehuso del comportamiento es a través de objetos ya existentes.

Javascript es un lenguaje de programación que usualmente es clasificado como un lenguaje con un modelo de objetos basado en prototipos (prototype-based object model).

Modelo de clases vs Modelo de prototipos

- Estático vs dinámico: la clase es estática mientras que el prototipo es dinámico, lo que permite tener múltiples ventajas:
  - Agregar comportamiento y estado durante la ejecución
  - Modificar todas las instancias del mismo prototipo a la vez
  - Crear nuevos prototipos de forma dinámica
  - Y más...

Algunos Lenguajes Prototipados

Lenguaje **Self**: desarrollado por David Ungar y Randall Smith

Y lo podemos ver implementado en lenguajes como:

Javascript

Pascual

Cecil

NewtonScript, Io, MOO, REBOL, Squeak, y otros

Javascript - Un poco de historia..

Javascript surge ante la necesidad de hacer que el HTML tuviera una interacción con el usuario más completa y de disminuir la cantidad de pedidos al servidor.

En 1995,surgen dos opciones: Java Applets y LiveScript, creado por Brendan Eich que luego fue incluido en Netscape 2.0 bajo el nombre de 'JavaScript'. Los applets no tuvieron mucho éxito pero sí lo tuvo JavaScript.

¿Cómo hacemos que un objeto herede de otro objeto?

Cuando vemos que cierto comportamiento entre objetos se repite, que hacemos?

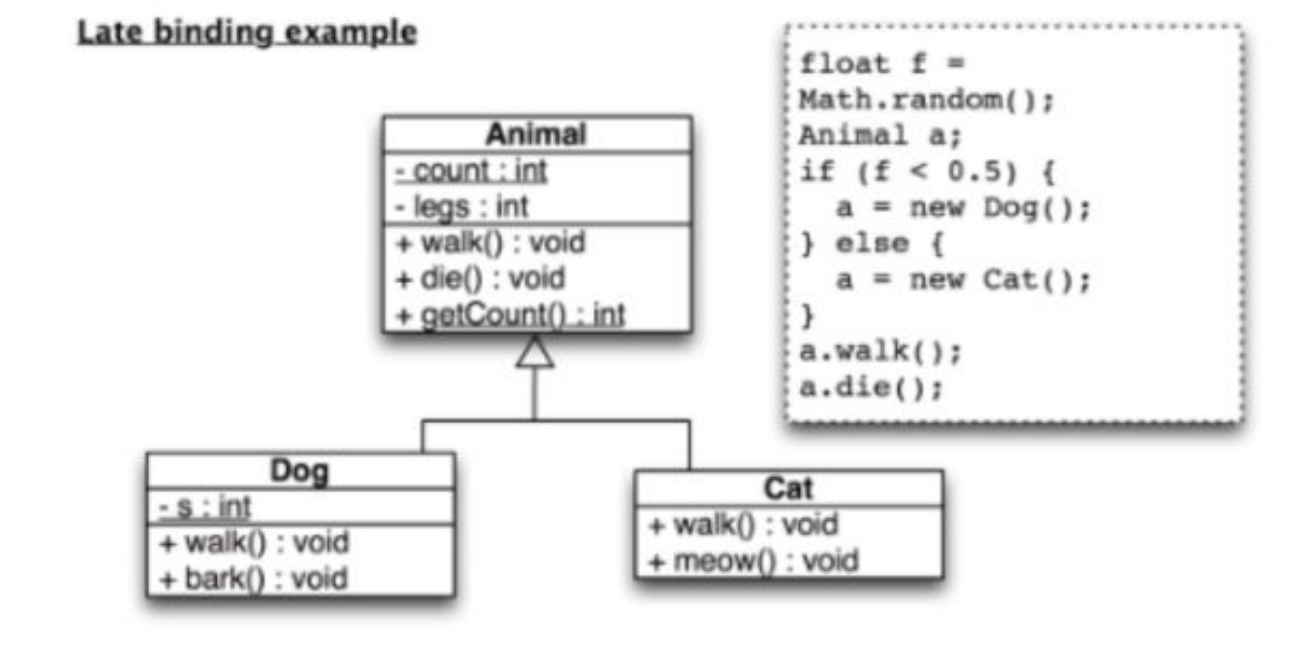
Aquí es donde entran los conceptos de clase y de prototipo,

dependiendo de las herramientas que tengamos disponibles.

Prototipos Vs Extensión de Clases

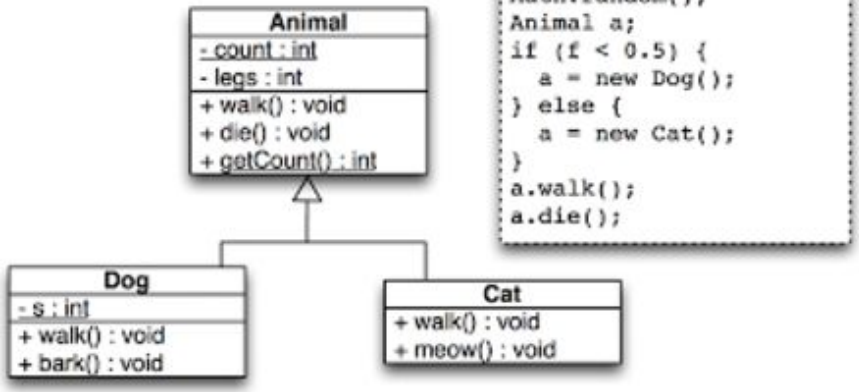
¿Cómo hacemos que un objeto herede de otro objeto?

- Clases

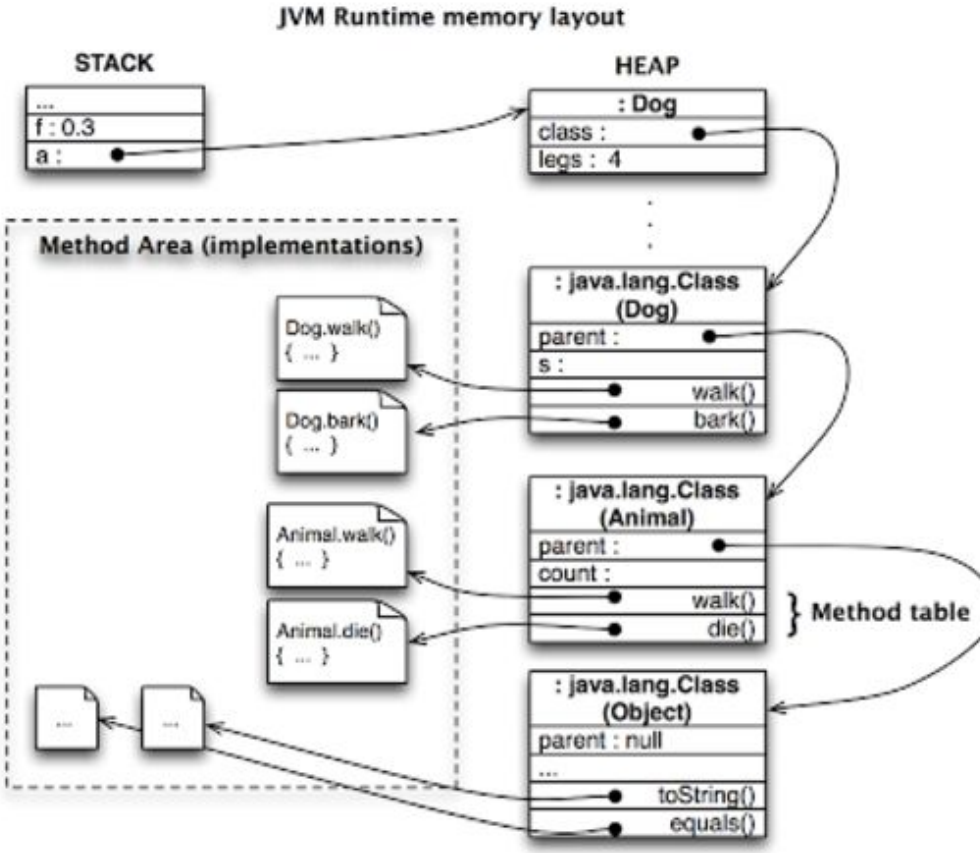


Prototipos Vs Extensión de Clases

Late binding example



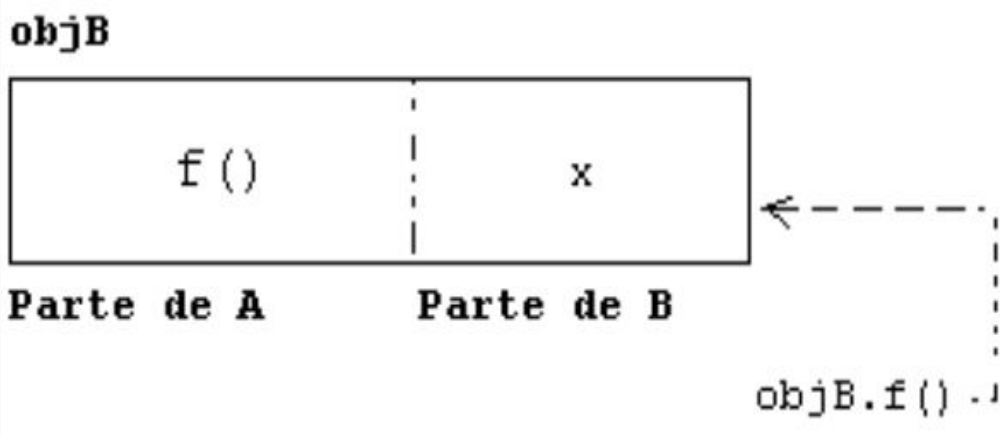

```
float f =
Math.random();
Animal a;
if (f < 0.5) {
  a = new Dog();
} else {
  a = new Cat();
}
a.walk();
a.die();
```



Prototipos Vs Extensión de Clases

¿Cómo hacemos que un objeto herede de otro objeto?

- Clases



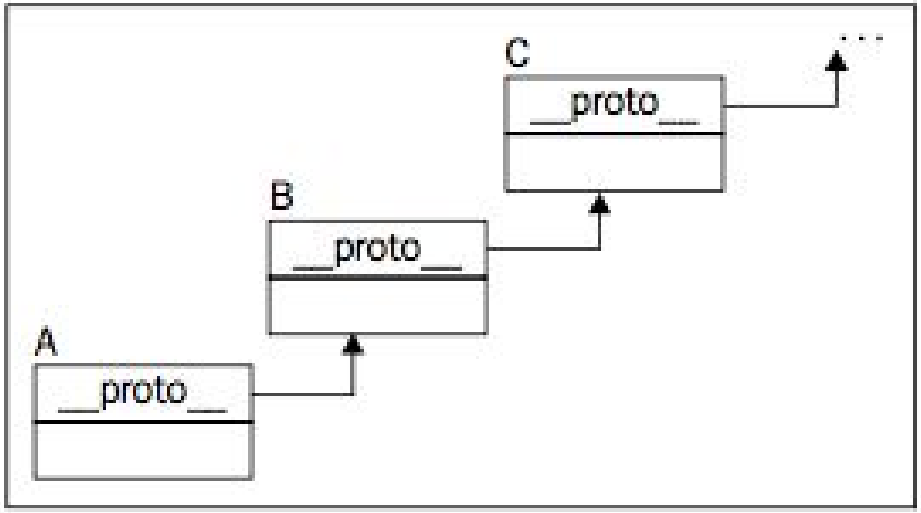
Todos los atributos y métodos heredados están disponibles en el mismo objeto (si bien se necesita la clase para poder interpretarlos)

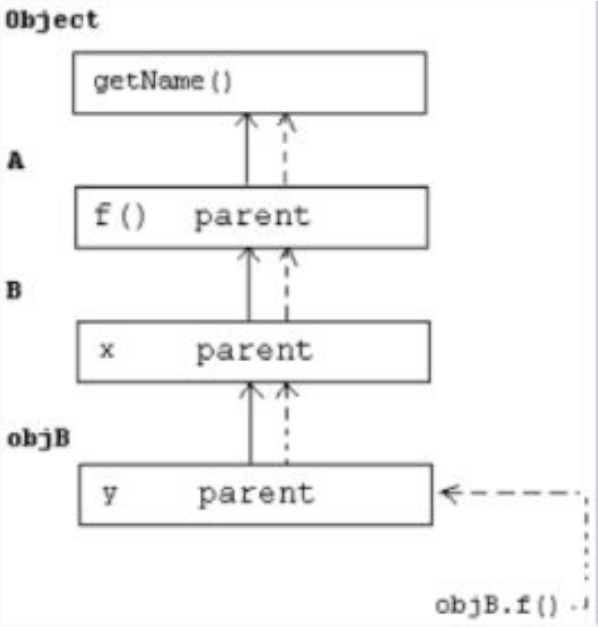
Prototipos Vs Extensión de Clases

¿Cómo hacemos que un objeto herede de otro objeto?

- Prototipos.

Cada objeto creado tiene una referencia a su prototipo.
El prototipo es otro objeto. Se genera una Prototype Chain





Prototipos Vs Extensión de Clases

¿Cómo hacemos que un objeto herede de otro objeto?

Usando prototipos. Cada objeto creado tiene una referencia a su prototipo.

```
var shape = {};
```

Prototipos Vs Extensión de Clases

¿Cómo hacemos que un objeto herede de otro objeto?

Usando prototipos. Cada objeto creado tiene una referencia a su prototipo.

```
var shape = {};

shape.constructor.prototype  // Object{}

shape.constructor.prototype === Object.prototype; // true

Object.getPrototypeOf(shape) === Object.prototype; // true
```

Prototype Chaining (Cadena de prototipos)

```
function Shape() { }

Shape.prototype = {
    name: "shape",
    toString: function() {
        return this.name;
    }
};

var instancia1 = new Shape();
var instancia2 = new Shape();
console.log(instancia1 == instancia2); // false
Object.getPrototypeOf(instancia1) ==
Object.getPrototypeOf(instancia2); // true
```

```
instancia1.name; // "shape"
Shape.prototype.a = 1;
instancia1.a; // 1
```

Prototype Chaining (Cadena de prototipos)

```
function TwoDShape() {
    this.name = "2D shape";
};

function Triangle(side, height) {
    this.name = "Triangle"; this.side = side; this.height = height;
    this.getArea = function() { return this.side * this.height / 2;}
};

TwoDShape.prototype = new Shape();
Triangle.prototype = new TwoDShape();
TwoDShape.prototype.constructor = TwoDShape;
Triangle.prototype.constructor = Triangle;
```

Prototype Chaining (Cadena de prototipos)

```
var t = new Triangle(5,10);

t.getArea();  // 25

t.toString(); // "Triangle"
```

Funciones como componentes de Primera Clase

Al igual que los lenguajes funcionales, las funciones se pueden tratar como valores de primera clase. Se puede hacer lo mismo que con otros tipos integrados.

- Se le pueden asignar un nombre
- Se pueden guardar en una variable
- Se pueden pasar por parámetro a otra función
- Se pueden retornar como respuesta de una función

```
var f = function(param) { return param; }
f(f);
f(f)(2);
f( function() { return "123"; } )();
```

Funciones como componentes de Primera Clase

ANONIMAS

```
function(){
    alert(“aaa”);
}
```

SELF INVOKING FUNCTION

```
(
    function(){
        alert("aaa");
    }
)();
```

CALLBACKS FUNCTIONS

```
function doSomethingAsync( inErrorCase, inSuccessCase ){
...
}
doSomethingAsync( function(err){ alert(err);}, function(result){ ….. } );
```

Funciones como componentes de Primera Clase

INNER (Private) FUNCTIONS

```
function a(param){
        function b(theinput){
                return theintput * 2;
        }
        return “result: “ + b(param);
}
```

b es una función privada, solo accesible desde a

Literal Notation:

```
var a = function (param){
        var b = function (theinput){
                return theintput * 2;
        }
        return “result: “ + b(param);
}
```

Beneficios de usar funciones privadas:

- El global space es menor, menos colisiones de nombres.
- Privacidad, se expone solo que que se quiere exponer

Funciones como componentes de Primera Clase

Funciones que retornan funciones:

```
function a(){
        alert (‘A!’);
                return function (){
                alert (‘b!’);
        }
}

var c = a();
c();
a()();
a= a();
a();
```

Ejemplo:

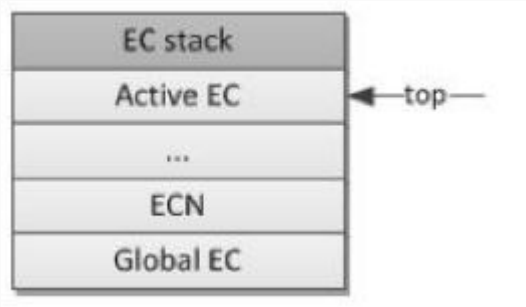
```
var a = function(){
        var setup = 'waiting..';
        function someSetup(){
                setup = 'done';
        }
        function actualWork(){
                alert('working ' + 'with setup: ' + setup);
        }
        someSetup();
        return actualWork;
}();

a();
```

Contexto de ejecución

Hay tres tipos de código con su contexto de ejecución asociado: global code, function code y eval code.

Stack del contexto de ejecución

El modelo es un stack con el global context como item inicial, a medida que se generan nuevos se apilan en el stack. A medida que el de más arriba (el activo) deja el control al que lo llamó (el que le sigue en el stack) es sacado del stack.



Alcances Globales y Localizados

Estructura del Contexto de ejecución

Cada contexto es representado por un objeto simple, con propiedades que permiten el seguimiento del código asociado.

Execution context	
Variable object	{ vars, function declarations, arguments... }
Scope chain	[Variable object + all parent scopes]
thisValue	Context object

thisValue:

1. Cuando un objeto es creado desde una función, this se refiere al objeto.
2. En el caso de una función, anidada o no, this se refiere al contexto global.

Global VO	
foo	10
bar	<function>
<built-ins>	

Variable Object

Representación de los datos que definen un alcance.

Alcances Globales y Localizados

Scope Chain:

No hay scope de llaves, sino scope de funciones

Lexical Scope:

El scope se crea cuando las funciones son definidas, no cuando son ejecutadas.

```
function f1(){
  var a = 1;
  return f2();
}
function f2(){
 return a;
}
f1();
// que retorna f1 ?
```

```
x = 0;
function BigTest() {
        this.x = 1;
}
BigTest();
console.log(x); //¿qué imprime y por qué?
```

```
if (true){
   var a = 1;
}
++a;
// que retorna? es valido?
```

Alcances Globales y Localizados

Scope Chain:

No hay scope de llaves, sino scope de funciones

Lexical Scope:

El scope se crea cuando las funciones son definidas, no cuando son ejecutadas.

```
function f1(){
  var a = 1;
  return f2();
}
function f2(){
 return a;
}
f1();
a is not defined
```

```
x = 0;
function BigTest() {
        this.x = 1;
}
var obj = new BigTest();
console.log(x); ¿qué imprime y por qué?
```

```
if (true){
   var a = 1;
}
++a;
2
```

Alcances Globales y Localizados

Breaking the chain with closure:

Closure #1:

```
function f(){
        var b = "b";
        return function(){
                return b;
        }
}
var n = f();
n();
b;
```

b solo existe dentro de f()
y es accesible solo por f

Closure #2:

```
var n;
function f(){
        var b = “b”;
        n = function(){
                return b;
        }
}
n();
b;
```

Closure #3:

```
function f(arg){
        var n = function(){
                return arg;
        }
        arg++;
        return n;
}
var arg = 123;
var m = f(arg);
m();  //124
m(); // ????
arg = 125;
m(); // ????
```

Alcances Globales y Localizados

Ejercicio:

```
function f(){
        var a = [];
        var i;
        for (i=0; i<3; i++){
                a[i] = function(){
                        return i;
                }
        }
        return a;
}

var a = f();
a[0]();  // ???
a[1]();  // ???
a[2]();  // ???
```

Alcances Globales y Localizados

Ejercicio:

```
function f(){
        var a = [];
        var i;
        for (i=0; i<3; i++){
                a[i] = function(){
                        return i;
                }
        }
        return a;
}

var a = f();
a[0]();  // 3
a[1]();  // 3
a[2]();  // 3
```

Como lo haríamos para que retorne 0, 1, 2 ?

Alcances Globales y Localizados

```
function f(){
    var a = [];
    var i;
    for (i=0; i<3; i++){
        a[i] = (function(x){
            return function(){
                return x;
            }
        })(i);
    }
    return a;
}
```

```
function f(){
    function makeClosure(x){
        return function(){
            return x;
        }
    }
    var a = [];
    var i;
    for (i=0; i<3; i++){
        a[i] = makeClosure(i);
    }
    return a;
}
```

Alcances Globales y Localizados

Getter / Setter

```
var getValue, setValue;
(function(){
        var secret = 0;
        getValue = function(){
                return secret;
        };
        setValue = function(v){
                secret = v;
        };
})();
```

```
function Foob(){
    var privateData = {};
    this.get = function(key){
        return privateData[key];
    }
    this.set = function(key, value){
        privateData[key] = value;
        return this;
    }
}
```

Ejercicio:

Hacer una función que reciba un array de items, y retorne una función que la itere

Ejemplo:

var next = setupIterator([‘a’, 1, new Object(), ‘b’, [1,2,3]])

Alcances Globales y Localizados

```
function setupIterator(x){
        var i =0;
        return function(){
                return x[i++];
        }
}
```

Ejercicio:
Hacer una función que reciba un array de items, y retorne una función que la itere

```
var next = setupIterator(['a', 1, new Object(), 'b', [1,2,3]])

next(); //  'a'
next(); //   1
next(); //   Object{}
```

Objetos

Javascript es un lenguaje dinámico
Nos permite modificar las propiedades y métodos de objetos existentes en cualquier momento. (También agregar y quitar)

```
var hero = {}
typeof hero.breed // undefined
hero.breed = 'turtle';
hero.name = 'Leonardo';
hero.sayName = function(){ return this.name; }
hero["name"]; hero["sayName"] // acceder como un mapa

hero.sayName();
hero["sayName"]();
delete hero.name;
```

Creación de Instancias

```
var hero = {
        name : "Spiderman",
        sayName : function(){ return this.name;}
}
hero.sayName();
```

Constructor Functions

```
function Hero(name){
        this.name  = name;
        this.sayName = function(){ return this.name;}
}

var spider = new Hero("Spiderman");
var superman = new Hero("Superman");
```

La convención es usar Primer letra mayúscula para los funciones constructores para diferenciarla de las funciones comunes

Que pasa si se ejecuta una constructor function sin el operador new?

```
var h = Hero("Batman")
typeof h;  // undefined
```

que valor tiene h?
quien es this en ese caso?

Global Object

El Host Environment provee un objeto global y todo lo global en realidad es parte o son propiedades de este objeto global

```
this;
Window{...}
h.name;
h.sayName();
```

Prototype Chaining

```
function Shape(){
    this.name = 'shape';
    this.toString = function() {return this.name;};
}
function Triangle(side, height) {
    this.name = 'Triangle';
    this.side = side;
    this.height = height;
    this.getArea = function(){return this.side * this.height / 2;};
}
TwoDShape.prototype = new Shape();
Triangle.prototype = new TwoDShape();
```

```
function TwoDShape(){
  this.name = '2D shape';
}
```

Que sucede con el constructor al asignar una instancia de Shape como prototipo? TwoDShape.prototype.constructor

Built In Objects

```
Array.prototype.inArray = function(needle) {
     for (var i = 0, len = this.length; i < len; i++) {
      if (this[i] === needle) {
         return true;
       }
      }
    return false;
   }
```

Now all arrays will have the new method. Let's test:

```
  >>> var a = ['red', 'green', 'blue'];
  >>> a.inArray('red');
true

>>> a.inArray('yellow');
false
```

Cuidado con extender los prototipos de los Built In objects, que pasa si futuras versiones nativas incluyen ahora una funcion llamada inArray?

Recursos

- Java Script Patterns, Stoyan Stefanov, O'Reilly, 2010
- Pro JavaScript Design Patterns, Ross Harmes and Dustin Diaz, Apress, 2008 3
- JavaScript: The Good Parts, Douglas Crockford, O'Reilly, 2008.
- Object-Oriented JavaScript, Stoyan Stefanov, Packt Publishing, 2008.
- Test-Driven JavaScript Development, Christian Johansen, Adisson Wesley, 2011
- Node, https://nodejs.org/dist/latest-v8.x/docs/api/
- ECMA Script 262
  https://www.ecma-international.org/publications/standards/Ecma-262.htm