Desarrollo de Aplicaciones Web Empresariales

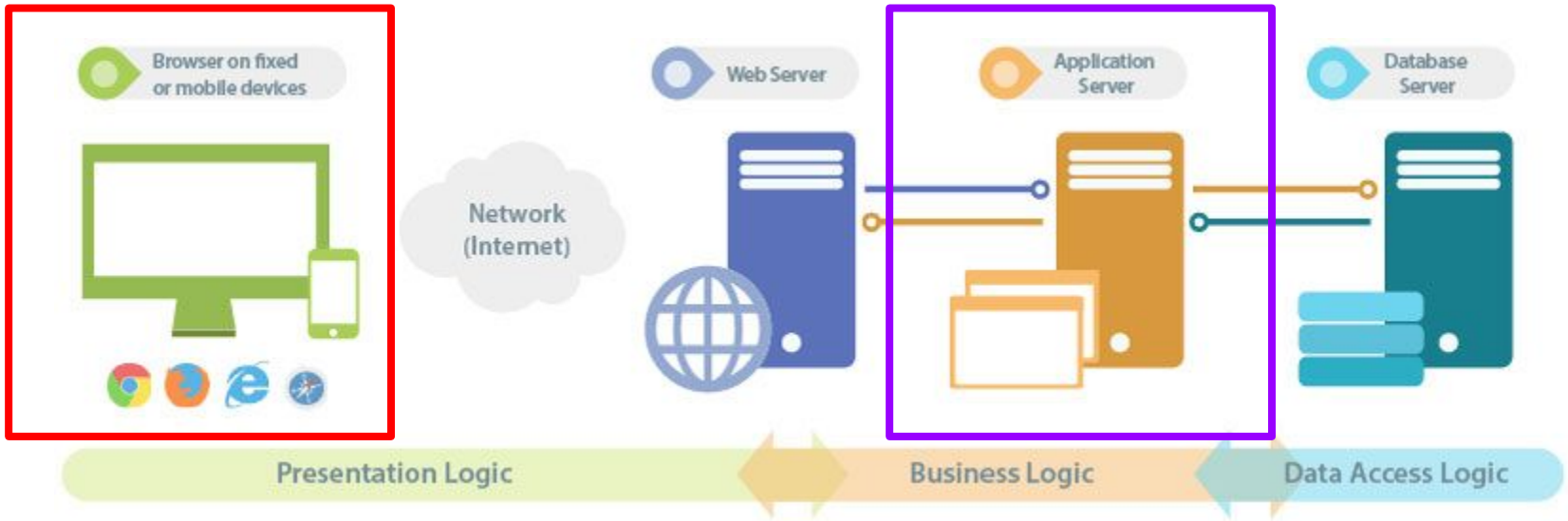
Exequiel Fuentes Lettura
exequiel.fuentes@ucn.cl

Información de contacto

- Exequiel Fuentes Lettura
  - Email: exequiel.fuentes@ucn.cl
  - Horario de Atención: Jueves y Viernes, bloque C

- Departamento de Ingeniería de Sistemas y Computación
  - Oficina: Y1 - 329
  - http://www.disc.ucn.cl

Arquitectura de aplicaciones Web



Material: https://developer.mozilla.org/en-US/docs/Web/JavaScript

Cómo se programa en Javascript

- Soporta:
  - Programación orientada a objetos.
  - Programación imperativa.
  - Programación funcional.
- El estándar actual fue lanzado recientemente: ECMAScript 2015 (Edición 6).
- Las nuevas características son soportadas por Chrome y Firefox.

Programación OO: Métodos

- Una propiedad de un objeto puede ser función:

```
var o = {count: 0};
o.increment = function (amount) {
    if (amount == undefined) {
        amount = 1;
    }
    this.count += amount;
    return this.count;
}
```

- Invocación:

```
o.increment();  // retorna 1
o.increment(3);  // retorna 4
```

Keyword: this

- En los métodos, ***this*** liga los objetos.

```
var o = {oldProp: 'vieja propiedad'};
o.aMethod = function() {
    this.newProp = "nueva propiedad";
    return Object.keys(this); // contiene 'newProp'
}
o.aMethod(); // retorna ['oldProp','aMethod','newProp']
```

- En funciones que no son métodos:
  - ***this*** será el objeto global.
  - Pero si se utiliza “use strict” (característica agregada en v5), será ***undefined***

```
"use strict";
x = 3.14;       // Error, x no está definida
```

Funciones con propiedades

```
function plus1(value) {
    if (plus1.invocations == undefined) {
        plus1.invocations = 0;
    }
    plus1.invocations++;
    return value + 1;
}
```

- En este caso, ***plus1.invocations*** será el número de veces que la función se llama.
- Es como una propiedad de clase en Orientación a Objetos.

Programación OO: Clases

- Javascript como lenguaje dinámico no tiene implementación de **class** como en Java o C++. En ES6 se introduce el keyword **class**, pero es equivalente a usar **prototype**.
- Las funciones son clases en Javascript:

```
function Rectangle(width, height) {
    this.width = width;
    this.height = height;
    this.area = function() { return this.width*this.height; }
}
var r = new Rectangle(26, 14); // {width: 26, height: 14}
```

No hacerlo de esta forma!, hay uso extra de memoria

- Las funciones utilizadas de esta forma son llamadas constructores:
  - r.constructor.name == 'Rectangle'
  - console.log(r): Rectangle { width: 26, height: 14, area: [Function] }

Programación OO: Herencia

- Javascript permite crear objetos **prototype** para cada instancia.
  - Objetos **prototype** pueden tener objetos **prototype**
- Una propiedad de un object se busca en esta cadena de **prototype** hasta que es encontrada.
  - Esto se llama herencia basada en prototype.
- Actualizar una propiedad es diferente: siempre se crea una propiedad en el objeto si no existe.
- Métodos creados usando **prototype** son creados una vez y heredados a sus objetos.

Usando prototype

```
function Rectangle(width, height) {
    this.width = width;
    this.height = height;
}

Rectangle.prototype.area = function() {
    return this.width*this.height;
}

var r = new Rectangle(26, 14);   // {width: 26, height: 14}
var v = r.area();   // v == 26*14
Object.keys(r) == [ 'width', 'height' ]   // propiedades
```

Prototype vs instancia de objectos

var r = new Rectangle(26, 14);

- Entender la diferencia entre:

r.newM = function() { console.log('New Method called'); }

- y

r.prototype.newM = function() { console.log('New Method called'); }

- Entender esto, hace la diferencia en el tiempo de ejecución.

Herencia

Rectangle.prototype = Object.create(Shape.prototype);

- Si esta propiedad no está en Rectangle.prototype, entonces Javascript no encontrará la herencia que entrega Shape. Otro ejemplo:

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype.dist = function () {
    return Math.sqrt((this.x*this.x)+(this.y*this.y));
};
Point.prototype.toString = function () {
    return "("+this.x+", "+this.y+")";
};
```

```
ColorPoint.prototype = Object.create(Point.prototype);
ColorPoint.prototype.constructor = ColorPoint;
ColorPoint.prototype.toString = function () {
    return this.color+" "+Point.prototype.toString.call(this);
};

const cp = new ColorPoint(5, 3, "red");
console.log(cp.toString()); // red (5, 3)
console.log(cp instanceof Point);  // true
console.log(cp instanceof ColorPoint); // true
```

Programación funcional

- Imperativo:

```
for (var i = 0; i < anArr.length; i++) {
    newArr[i] = anArr[i]*i;
}
```

- Funcional:

```
const newArr = anArr.map(function (val, ind) {
    return val*ind;
});
```

- Se puede escribir todo un programa de esta forma:

```
anArr.filter(filterFunc).map(mapFunc).reduce(reduceFunc);
```

Closures o cerraduras

- Las cerraduras son funciones que hacen referencia a las variables definidas en el ámbito que las contiene.
- Estas funciones “recuerdan” el entorno en el que se crearon.

```
var globalVar = 1;
function localFunc(argVar) {
    var localVar = 0;
    function embedFunc() { return ++localVar + argVar + globalVar; }
    return embedFunc;
}

var myFunc = localFunc(10);
console.log(myFunc());    // 12
console.log(myFunc());    // 13
```

Usando alcance y cerraduras

- Una cerradura es un tipo especial de objeto que combina:
  - Una función.
  - El ambiente en que la función es creada.
- Considera el efecto de los siguientes alcances:

var i = 1;	(function () {
	  var i = 1;
	})();

Mantener propiedades privadas

```
var myObj = (function() {
  var privateProp1 = 1; var privateProp2 = "test";
  var setPrivate1 = function(val1) { privateProp1 = val1;  }
  var compute = function() {return privateProp1 + privateProp2;}
  return {compute: compute, setPrivate1: setPrivate1};
})();

console.log(myObj.compute()); // 1test
myObj.setPrivate1(5)
console.log(myObj.compute()); // 5test

console.log(typeof myObj); // 'object'
console.log(Object.keys(myObj)); // [ 'compute', 'setPrivate1' ]
```

Complicaciones para programación imperativa

```
// Read files './file0' and './file1' and return their length
for (var fileNo = 0; fileNo < 2; fileNo++) {
    fs.readFile('./file' + fileNo, function (err, data) {
        if (!err) {
            console.log('file', fileNo, 'has length', data.length);
        }
    });
}
```

- Al final imprime:
  - file 2 has length
  - Por qué?

Veamos que pasa

```
// Read files './file0' and './file1' and return their length
for (var fileNo = 0; fileNo < 2; fileNo++) {
    fs.readFile('./file' + fileNo, function (err, data) {
        if (!err) {
            console.log('file', fileNo, 'has length', data.length);
        }
    });
}
```

La ejecución comienza con fileNo = 0

Veamos que pasa

```
// Read files './file0' and './file1' and return their length
for (var fileNo = 0; fileNo < 2; fileNo++) {
    fs.readFile('./file' + fileNo, function (err, data) {
        if (!err) {
            console.log('file', fileNo, 'has length', data.length);
        }
    });
}
```

Llama a la función fs.readFile, antes se deben evaluar los argumentos. El primer argumento es una concatenación de string “./file0”. El segundo argumento es una función. En este caso, fileNo es accedido por la cerradura y vale 0.

Veamos que pasa

```
// Read files './file0' and './file1' and return their length
for (var fileNo = 0; fileNo < 2; fileNo++) {
    fs.readFile('./file' + fileNo, function (err, data) {
        if (!err) {
            console.log('file', fileNo, 'has length', data.length);
        }
    });
}
```

fs.readFile retorna después de que ha comenzado a leer el archivo, pero antes de llamar a la función del argumento. La ejecución incrementa fileNo y llama a fs.readFile con el argumento “./file1” y una función. La cerradura tiene solamente fileNo (actualmente 1)

Veamos que pasa

```
// Read files './file0' and './file1' and return their length
for (var fileNo = 0; fileNo < 2; fileNo++) {
    fs.readFile('./file' + fileNo, function (err, data) {
        if (!err) {
            console.log('file', fileNo, 'has length', data.length);
        }
    });
}
```

Después de crear 2 funciones con cerraduras y llamar a fs.readFile dos veces. En algún momento en la ejecución el archivo se leerá y fs.readFile llamará a las funciones pasadas. Recuerda que fileNo es ahora 2.

Veamos que pasa

```
// Read files './file0' and './file1' and return their length
for (var fileNo = 0; fileNo < 2; fileNo++) {
    fs.readFile('./file' + fileNo, function (err, data) {
        if (!err) {
            console.log('file', fileNo, 'has length', data.length);
        }
    });
}
```

Ahora, se ejecuta la función, no hay error, entonces entra al siguiente bloque.

Veamos que pasa

```
// Read files './file0' and './file1' and return their length
for (var fileNo = 0; fileNo < 2; fileNo++) {
    fs.readFile('./file' + fileNo, function (err, data) {
        if (!err) {
            console.log('file', fileNo, 'has length', data.length);
        }
    });
}
```

Cuando evalúa los argumentos, se va a la cerradura y se obtiene el valor de fileNo. Se encuentra en 2. Se imprime el correcto largo de los datos, pero no el número del archivo. Lo mismo sucede en la siguiente llamada.

Arreglo: hacer fileNo un argumento

```
function printFileLength(fileNo) {
    fs.readFile('./file' + fileNo, function (err, data) {
        if (!err) {
            console.log('file', fileNo, 'has length', data.length);
        }
    });
}

for (var fileNo = 0; fileNo < 2; fileNo++) {
    printFileLength(fileNo);
}
```

- Probar el ejemplo anterior usando el keyword **let** (ES6).

JavaScript Object Notation (JSON)

- JSON es el estándar para enviar datos desde y hacia el browser.

```
var obj = { ps: 'str', pn: 1, pa: [1,'two',3,4], po: { sop: 1}};

var s = JSON.stringify(obj) = '{"ps":"str","pn":1,"pa":[1,"two",3,4],"po":{"sop":1}}'

typeof s == 'string'

JSON.parse(s) // retorna un objeto con las mismas propiedades.
```

Algunas otras cosas de Javascript

- Asignar un valor por defecto:
  - hostname = hostname || "localhost";
    port = port || 80;
- Acceder a una propiedad **undefined**
  - const obj = {'prop1': 1};
    const prop = obj && obj.prop2;
- Manejar múltiples **this**:
  - fs.readFile(this.fileName + fileNo, function (err, data) {
      console.log(this.fileName, fileNo); // Malo!!, este **this** corresponde a este alcance.
    });

Algunas otras cosas de Javascript

- Asignar un valor por defecto:
  - hostname = hostname || "localhost";
    port = port || 80;
- Acceder a una propiedad **undefined**
  - const obj = {'prop1': 1};
    const prop = obj && obj.prop2;
- Manejar múltiples **this**:
  - var self = this;

    fs.readFile(self.fileName + fileNo, function (err, data) {

    console.log(self.fileName,fileNo);

    });

¿Preguntas?