

## **Unidad 5.**

### **JavaScript**

Objetos Definidos por el Desarrollador.

## Programación orientada a Objetos JavaScript:

La programación orientada a objetos es un paradigma de programación que utiliza la abstracción para crear modelos basados en el mundo real. Utiliza diversas técnicas de paradigmas previamente establecidas, incluyendo la **modularidad**, **polimorfismo** y **encapsulamiento**.

Hoy en día, muchos lenguajes de programación (como Java, JavaScript, C#, C++, Python, PHP, Ruby y Objective-C) soportan programación orientada a objetos (POO).

La programación orientada a objetos puede considerarse como el diseño de software a través de un conjunto de objetos que cooperan, a diferencia de un punto de vista tradicional en el que un programa puede considerarse como un conjunto de funciones, o simplemente como una lista de instrucciones para la computadora. En la programación orientada a objetos, cada objeto es capaz de recibir mensajes, procesar datos y enviar mensajes a otros objetos. Cada objeto puede verse como una pequeña máquina independiente con un papel o responsabilidad definida.

POO pretende promover una mayor flexibilidad y facilidad de mantenimiento en la programación y es muy popular en la ingeniería de software a gran escala. Gracias a su fuerte énfasis en la modularidad, el código orientado al objetos está concebido para ser más fácil de desarrollar y más fácil de entender posteriormente, prestándose a un análisis más directo, a una mayor codificación y comprensión de situaciones y procedimientos complejos que otros métodos de programación menos modulares. <sup>2</sup>

### Terminología

**Clase:** Define las características del Objeto.

**Objeto:** Una instancia de una Clase.

**Propiedad:** Una característica del Objeto, como el color.

**Método:** Una capacidad del Objeto, como caminar.

**Constructor:** Es un método llamado en el momento de la creación de instancias.

**Herencia:** Una Clase puede heredar características de otra Clase.

**Encapsulamiento:** Una Clase sólo define las características del Objeto, un Método sólo define cómo se ejecuta el Método.

**Abstracción:** La conjunción de herencia compleja, métodos, propiedades que un objeto debe ser capaz de simular en un modelo de la realidad.

**Polimorfismo:** Diferentes Clases podrían definir el mismo método o propiedad.

### Programación basada en prototipos

La programación basada en prototipos es un estilo de programación orientada a objetos en la que las clases no están presentes y la reutilización de comportamiento (conocido como herencia en lenguajes basados en clases) se lleva a cabo a través de un proceso de decoración de objetos existentes que sirven de prototipos.

Este modelo también se conoce como programación sin clases, orientada a prototipos o basada en ejemplos.

El ejemplo original (y más canónico) de un lenguaje basado en prototipos es el lenguaje de programación autodesarrollado por David Ungar y Randall Smith. Sin embargo, el estilo de programación sin clases se ha hecho cada vez más popular y ha sido adoptado para lenguajes de programación como JavaScript, Cecil, NewtonScript, Io, MOO, REBOL, Kevo, Squeak (cuando se utiliza el Viewer framework para manipular los componentes Morphic) y varios otros. <sup>2</sup>

# Programación orientada a objetos de JavaScript

## Namespace

Un **espacio de nombres** es un contenedor que permite asociar toda la funcionalidad de un determinado objeto con un nombre único. En JavaScript un espacio de nombres es un objeto que permite a métodos, propiedades y objetos asociarse. La idea de crear espacios de nombres en JavaScript es simple: Crear un único objeto global para las variables, métodos, funciones convirtiéndolos en propiedades de ese objeto. El uso de los namespace permite minimizar el conflicto de nombres con otros objetos haciéndolos únicos dentro de nuestra aplicación.

Un **espacio de nombres** es un objeto:

Vamos a crear un **objeto global** llamado **MIAPLICACION**

```
// namespace global  
  
var MIAPLICACION = MIAPLICACION || {};
```

Nota: Para continuar con las mejores prácticas vamos a utilizar mayúsculas para los namespace.

En el código de ejemplo anterior comprobamos si MIAPLICACION ya se encuentra definida. Si es así utilizamos el objeto global MIAPLICACION que existe, si este no existe creamos un objeto vacío llamado MIAPLICACION que encapsulará métodos, funciones, variables y otros objetos que vayamos a crear.

También podemos crear **Sub-espacios de nombres**:

```
// Sub-namespace  
  
MIAPLICACION.event = {} ;
```

A continuación se muestra el código para la creación de un **espacio de nombre** y como agregar variables, funciones y métodos:

```
// Creación del contenedor llamado MIAPLICACION.metodoComun de método y propiedades comunes.
MIAPLICACION.metodoComun = {
  regexParaNombre: "", // define regex para la validación del nombre
  regexParaTelefono: "", // define regex para validación del teléfono
  validaNombre: function(nombre){
    // Hace algo con el nombre que usted ingresa a la variable reExParaNombre
    // usando "this.regexParaNombre"
  },

  validaNroTelefono: function (numTelefono){
    // Hace algo con el número de teléfono
  }
}

// Objeto junto a la declaración del método
MIAPLICACION.event = {
  addListener: function(el, type, fn){
    // código de relleno
  },
  removeListener: function(el, type, fn){
    // código de relleno
  },
  getEvent: function(e) {
    // código de relleno
  }

  // Puedes agregar otras propiedades y métodos
}

// Sintaxis de utilización del método addListener:
MIAPLICACION.event.addListener("turel", "tipo", callback);
```

## Objetos básicos

JavaScript tiene varios objetos incluidos en su núcleo, como Math, Object, Array y String. El siguiente ejemplo muestra cómo utilizar el objeto Math para obtener un número al azar mediante el uso de su método random().

```
alert (Math.random ());
```

**Nota:** este y todos los demás ejemplos suponen que una función llamada alert (como el que se incluye en los navegadores web) se define de forma global. La función alert no es realmente parte de JavaScript.

Consulta [Core JavaScript 1.5 Reference:Global Objects](#) para obtener una lista de los objetos básicos en JavaScript.

Cada objeto en JavaScript es una instancia del objeto Object, por lo tanto, hereda todas sus propiedades y métodos.

## Objetos personalizados

### La clase

JavaScript es un lenguaje basado en prototipos que no contiene ninguna declaración de clase, como se encuentra, por ejemplo, en C++ o Java. Esto es a veces confuso para los programadores acostumbrados a los lenguajes con una declaración de clase. En su lugar, JavaScript utiliza funciones como clases. Definir una clase es tan fácil como definir una función. En el ejemplo siguiente se define una nueva clase llamada **Persona**.

```
function Persona() { }
```

### El objeto (ejemplo de clase)

Para crear un nuevo ejemplo de un objeto *obj* utilizamos la declaración `new obj`, asignando el resultado (que es de tipo *obj*) a una variable para tener acceso más tarde.

En el siguiente ejemplo se define una clase llamada Persona y creamos dos instancias ( *persona1* y *persona2* ).

```
function Persona() {  
}
```

```
var persona1 = new Persona();  
var persona2 = new Persona();
```

Por favor, consulta también [Object.create](#) para ver un método nuevo y alternativo de creación de ejemplos.

### El constructor

El constructor es llamado en el momento de la creación de la instancia (el momento en que se crea la instancia del objeto). El constructor es un método de la clase. En JavaScript, la función sirve como el constructor del objeto, por lo tanto, no hay necesidad de definir explícitamente un método constructor. Cada acción declarada en la clase es ejecutada en el momento de la creación de la instancia.

El constructor se usa para establecer las propiedades del objeto o para llamar a los métodos para preparar el objeto para su uso. Más adelante describiremos como agregar métodos a clase y sus definiciones ya que se realiza utilizando una sintaxis diferente.

En el siguiente ejemplo, el constructor de la clase Persona muestra un alerta que dice (**Una instancia de persona**) cuando se crea la instancia de la clase Persona.

```
function Persona() {  
    alert('Una instancia de Persona');  
}
```

```
var persona1 = new Persona();  
var persona2 = new Persona();
```

## La propiedad (atributo del objeto)

Las propiedades son variables contenidas en la clase, cada instancia del objeto tiene dichas propiedades. Las propiedades deben establecerse a la propiedad prototipo de la clase (función), para que la herencia funcione correctamente.

Para trabajar con propiedades dentro de la clase se utiliza la palabra reservada **this**, que se refiere al objeto actual. El acceso (lectura o escritura) a una propiedad desde fuera de la clase se hace con la sintaxis: **NombreDeLaInstancia.Propiedad**. Es la misma sintaxis utilizada por C++, Java y algunos lenguajes más. (Desde dentro de la clase la sintaxis es **this.Propiedad** que se utiliza para obtener o establecer el valor de la propiedad).

En el siguiente ejemplo definimos la propiedad **primerNombre** de la clase **Persona** y la definimos en la creación de la instancia.

```
function Persona(primerNombre) {
  this.primerNombre = primerNombre;
  alert('Una instancia de Persona');
}

var persona1 = new Persona("Alicia");
var persona2 = new Persona("Sebastian");

// Muestra el primer nombre de persona1
alert ('persona1 es ' + persona1.primerNombre); // muestra "persona1 es Alicia"
alert ('persona2 es ' + persona2.primerNombre); // muestra "persona2 es Sebastian"
```

## Los métodos

Los métodos siguen la misma lógica que las propiedades, la diferencia es que son funciones y se definen como funciones. Llamar a un método es similar a acceder a una propiedad, pero se agrega ( ) al final del nombre del método, posiblemente con argumentos.

En el siguiente ejemplo se define y utiliza el método **diHola()** para la clase **Persona**.

```
function Persona(primerNombre) {
  this.primerNombre = primerNombre;
}

Persona.prototype.diHola = function() {
  alert ('Hola, Soy ' + this.primerNombre);
};

var persona1 = new Persona("Alicia");
var persona2 = new Persona("Sebastian");

// Llamadas al método diHola de la clase Persona.
persona1.diHola(); // muestra "Hola, Soy Alicia"
persona2.diHola(); // muestra "Hola, Soy Sebastian"
```

En JavaScript los métodos son objetos como lo es una función normal y se vinculan a un objeto como lo hace una propiedad, lo que significa que se pueden invocar desde "fuera de su contexto". Considera el siguiente código de ejemplo:

```
function Persona(primerNombre) {
  this.primerNombre = primerNombre;
}

Persona.prototype.diHola = function() {
  alert ("Hola, Soy " + this.primerNombre);
};

var persona1 = new Persona("Alicia");
var persona2 = new Persona("Sebastian");
var funcionSaludar = persona1.diHola;

persona1.diHola();           // muestra "Hola, Soy Alicia"
persona2.diHola();           // muestra "Hola, Soy Sebastian"
funcionSaludar();             // muestra "Hola, Soy undefined (ó da un error
con el                          // TypeError en modo estricto

alert(funcionSaludar === persona1.diHola);           // muestra true (verdadero)
alert(funcionSaludar === Persona.prototype.diHola);   // muestra true (verdadero)
funcionSaludar.call(persona1);                        // muestra "Hola, Soy Alicia"
```

En el ejemplo se muestran todas las referencias que tenemos de la función **diHola** — una de ellas es **persona1**, otra en **Persona.prototype**, en la variable **funcionSaludar**, etc. — todas se refieren a la misma función.

El valor durante una llamada a la función depende de como realizamos esa llamada.

En el común de los casos cuando la llamamos desde una expresión donde tenemos a la función desde la propiedad del objeto — **persona1.diHola()**.— Se establece en el objeto que tenemos en la función (persona1), razón por la cual **persona1.diHola()** utiliza el nombre "Alicia" y **persona2.diHola()** utiliza el nombre "Sebastian".

Pero si realizamos la llamada de otra manera, se establecerá de otra manera: Llamandola desde una variable —**funcionSaludar()**— Este establece al objeto global (windows, en los navegadores). Desde este objeto (probablemente) no tiene a la propiedad **primerNombre**, que finalizará con "Hola, Soy indefinido". (El cual se incluye en modo de código suelto, sino sería diferente [un error] en modo estricto, pero para evitar confusiones ahora no vamos a entrar en detalles.) O podemos establecerla de forma explícita utilizando **Function.call** (ó **Function.apply**), como se muestra al final del ejemplo **funcionSaludar.call(persona1)**.

Consulta más información al respecto en [Function.call](#) y [Function.apply](#)

## Herencia

La herencia es una manera de crear una clase como una versión especializada de una o más clases (JavaScript sólo permite herencia simple). La clase especializada comúnmente se llama hija o secundaria, y la otra clase se le llama padre o primaria. En JavaScript la herencia se logra mediante la asignación de una instancia de la clase primaria a la clase secundaria, y luego se hace la especialización.

JavaScript no detecta la clase hija `prototype.constructor`, vea las propiedades del [Core JavaScript 1.5 Reference:Global Objects:Object:prototype](#), así que debemos decírselo de forma manual.

En el siguiente ejemplo definimos la clase **Estudiante** como una clase secundaria de **Persona**. Luego redefinimos el método **diHola()** y agregamos el método **diAdios()**.

```
// Definimos el constructor Persona
function Persona(primerNombre) {
    this.primerNombre = primerNombre;
}

// Agregamos un par de métodos a Persona.prototype
Persona.prototype.caminar = function() {
    alert("Estoy caminando!");
};
Persona.prototype.diHola = function(){
    alert("Hola, Soy" + this.primerNombre);
};

// Definimos el constructor Estudiante
function Estudiante(primerNombre, asunto) {
    // Llamamos al constructor padre, nos aseguramos (utilizando Function#call) que "this" se
    // ha establecido correctamente durante la llamada
    Persona.call(this, primerNombre);

    //Inicializamos las propiedades específicas de Estudiante
    this.asunto = asunto;
};

// Creamos el objeto Estudiante.prototype que hereda desde Persona.prototype
// Nota: Un error común es utilizar "new Persona()" para crear Estudiante.prototype
// Esto es incorrecto por varias razones, y no menos importante, nosotros no le estamos pasando
// nada
// a Persona desde el argumento "primerNombre". El lugar correcto para llamar a Persona
// es arriba, donde nosotros llamamos a Estudiante.
Estudiante.prototype = Object.create(Persona.prototype);    // Vea las siguientes notas

// Establecer la propiedad "constructor" para referencias a Estudiante
Estudiante.prototype.constructor = Estudiante;

// Reemplazar el método "diHola"
Estudiante.prototype.diHola = function(){
    alert("Hola, Soy " + this.primerNombre + ". Yo estoy estudiando " + this.asunto + ".");
};

// Agregamos el método "diAdios"
Estudiante.prototype.diAdios = function() {
    alert("¡ Adios !");
};
```



```
// Ejemplos de uso
var estudiante1 = new Estudiante("Carolina", "Física Aplicada");
estudiante1.diHola(); // muestra "Hola, Soy Carolina. Yo estoy estudiando Física Aplicada."
estudiante1.caminar(); // muestra "Yo estoy caminando !"
estudiante1.diAdios(); // muestra "¡ Adios !"

// Comprobamos que las instancias funcionan correctamente
alert(estudiante1 instanceof Persona); // devuelve true
alert(estudiante1 instanceof Estudiante); // devuelve true
```

Con respecto a la línea **Estudiante.prototype = Object.create(Persona.prototype);** : Sobre los motores antiguos de JavaScript sin [Object.create](#), se puede utilizar un "polyfill" (aka "shim", vea el enlace del artículo), o se puede utilizar una función que obtiene el mismo resultado, como por ejemplo:

```
function crearObjeto(proto) {
  function ctor() { }
  ctor.prototype = proto;
  return new ctor();
}
```

// uso:

```
Estudiante.prototype = crearObjeto(Persona.prototype);
```

Ver [Object.create](#) para conocer más sobre lo que hace, y complementar con los motores antiguos.

## Encapsulación

En el ejemplo anterior, Student no tiene que saber cómo se aplica el método walk() de la clase Person, pero, sin embargo, puede utilizar ese método. La clase Student no tiene que definir explícitamente ese método, a menos que queramos cambiarlo. Esto se denomina **la encapsulación**, por medio de la cual cada clase hereda los métodos de su elemento primario y sólo tiene que definir las cosas que desea cambiar.

### Abstracción

Un mecanismo que permite modelar la parte actual del problema de trabajo. Esto se puede lograr por herencia (especialización) o por composición. JavaScript logra la especialización por herencia y por composición al permitir que las instancias de clases sean los valores de los atributos de otros objetos.

La clase Function de JavaScript hereda de la clase de Object (esto demuestra la especialización del modelo) y la propiedad Function.prototype es un ejemplo de Objeto (esto demuestra la composición)

```
var foo = function() {};
alert( 'foo es una Función: ' + (foo instanceof Function) );
alert( 'foo.prototype es un Objeto: ' + (foo.prototype instanceof Object) );
```

## Polimorfismo

Al igual que todos los métodos y propiedades están definidas dentro de la propiedad prototipo, las diferentes clases pueden definir métodos con el mismo nombre. Los métodos están en el ámbito de la clase en que están definidos. Esto sólo es verdadero cuando las dos clases no tienen una relación primario-secundario (cuando uno no hereda del otro en una cadena de herencia).