



## JAVASCRIPT AVANZADO

© 2017, ACTIBYTI PROJECT SLU, Barcelona  
Autor: Ricardo Ahumada



MINISTERIO  
DE ENERGÍA, TURISMO  
Y AGENDA DIGITAL

red.es



ESTRATEGIA DE  
EMPRENDIMIENTO Y  
EMPLEO JUVENIL  
*garantía juvenil*



UNIÓN EUROPEA

Fondo Social Europeo  
“El FSE invierte en tu futuro”

# ÍNDICE DE CONTENIDOS

1. Caso práctico
2. Funciones & funciones puras
3. OOP en JS
4. Closures
5. Promises

1

# CASO PRÁCTICO

# Caso Práctico: BananaTube Avanzado



“BananaTube” es el proyecto estrella de Banana Apps.

**BananaTube** será el próximo boom! de las redes sociales; permitirá a sus usuarios gestionar videos, exponerlos en su muro, comentar videos propios y de sus amigos, calificarlos y compartirlos en varios canales.

En la interfaz de la aplicación existen un conjunto de funcionalidades que requieren mecanismos sofisticados, como por ejemplo la gestión de los videos como una unidad conceptual, la interrelación entre distintos componentes, separar las funcionalidades en capas, etc.

En esta etapa queremos usar los mecanismos de los dispone Javascript para generar una versión mejor de nuestra aplicación y poder compartimentar y reusar los módulos que se generen.



# Discutamos

- Qué problemas estructurales tendremos que abordar en la implementación de las funcionalidades de la aplicación?
- Qué es un módulo? Cómo podemos implementarlos?
- Qué opciones me ofrece Javascript para solicitudes asíncronas, cuyo resultado recibiré en el futuro?
- Cómo puedo hacer que varios elementos de la interface se actualicen en función de ellas?
- Que significa ser reactivo?

2

# Funciones & funciones puras

# Qué es una función

- Una función es un proceso que toma algunas entradas, llamadas argumentos, y produce una salida llamada valor de retorno. Las funciones pueden servir para los siguientes fines:
- **Mapeo:** Produce una salida basada en entradas dadas. Una función asigna los valores de entrada a los valores de salida.
- **Procedimientos:** Una función puede ser llamada para realizar una secuencia de pasos. La secuencia se conoce como un procedimiento, y la programación en este estilo se conoce como programación procedural.
- **E/S:** Algunas funciones existen para comunicarse con otras partes del sistema, como la pantalla, el almacenamiento, los registros del sistema o la red.

# Funciones puras

- Una función pura es una función donde el valor de retorno sólo está determinado por sus valores de entrada, sin efectos secundarios observables.
- **Las funciones puras tienen que ver con el mapeo.** Las funciones asignan argumentos de entrada a valores de retorno, lo que significa que para cada conjunto de entradas, existe una salida. Una función tomará las entradas y devolverá la salida correspondiente.
- Así es como funciona en matemáticas: *Math.max(x)*, para un mismo valor de x, siempre devuelven el mismo resultado. La programación no cambia x. No escribe en los archivos de registro, realiza solicitudes de red, pregunta por la entrada del usuario o cambia el estado del programa.

```
Math.max(2, 8, 5); // 8
```

- Cuando una función realiza cualquier otra "acción", aparte de calcular su valor de retorno, la función es impura.

# Ejemplos

## ➤ Función impura

```
var values = { a: 1 };

function impureFunction ( items ) {
  var b = 1;

  items.a = items.a * b + 2;

  return items.a;
}

var c = impureFunction( values );
// Now `values.a` is 3, the impure function
modifies it.
```

## ➤ Función pura

```
var values = { a: 1 };

function pureFunction ( a ) {
  var b = 1;

  a = a * b + 2;

  return a;
}

var c = pureFunction( values.a );
// `values.a` has not been modified, it's
still 1
```

# Características y beneficios

Una **función es pura** si:

- Dada la misma entrada, siempre devolverá la misma salida.
- No produce efectos secundarios.
- No depende de ningún estado mutable externo.

Un pista para identificar si una función es impura es si tiene sentido llamarla sin usar su valor de retorno (return).

**Beneficios:**

- Las funciones puras tienen muchas propiedades beneficiosas, y forman la base de la programación funcional.
- Son completamente independientes del estado exterior, y como tal, son inmunes a errores que tienen que ver con el estado mutable compartido.
- Su naturaleza independiente los convierte en excelentes candidatos para procesamiento paralelo a través de muchas CPU, lo que los hace imprescindibles para muchos tipos de tareas científicas y de uso intensivo de recursos informáticos.
- Son fáciles de mover, refactorizar y reorganizar, haciendo que los programas sean más flexibles y adaptables a los cambios futuros.

# Mantener el estado Local

- Una función pura sólo puede acceder a lo que se pasa como parámetro, por lo que es fácil ver sus dependencias.
- **Cuando una función accede a algún otro estado del programa, como una instancia o una variable global, ya no es pura.**
- Tomando las **variables globales** como un ejemplo. Estas son típicamente considerados una mala idea.
  - Cuando partes de un programa empiezan a interactuar a través de forma global lo que hace que su comunicación sea invisible.
  - Hay dependencias que, en la superficie, son difíciles de detectar. Causan confusión de mantenimiento.
  - El programador necesita hacer un seguimiento mental de cómo las cosas están relacionadas y orquestar todo lo justo.
  - Pequeños cambios en un lugar pueden causar que todo el código tienda a fallar.

# Ejemplos

## ➤ Función impura

```
var values = { a: 1 };
var b = 1;

function impureFunction ( a ) {
  a = a * b + 2;

  return a;
}

var c = impureFunction( values.a );
// Actually, the value of `c` will depend on
// the value of `b`.
// In a bigger codebase, you may forget
// about that, which may
// surprise you because the result can vary
// implicitly.
```

## ➤ Función pura

```
var values = { a: 1 };
var b = 1;

function pureFunction ( a, c ) {
  a = a * c + 2;

  return a;
}

var c = pureFunction( values.a, b );
// Here it's made clear that the value of
// `c` will depend on
// the value of `b`. No sneaky surprise
// behind your back.
```

# Inmutabilidad

- En JavaScript, los argumentos de tipo objeto son referencias, lo que significa que si una función debía mutar una propiedad en un objeto o un parámetro de matriz, esto mutaría su estado fuera de la función.
- Las funciones puras no deben alterar el estado externo.

# Ejemplo – Carrito impuro (ES6)

```
// impure addToCart mutates existing cart
const addToCart = (cart, item, quantity) => {
  cart.items.push({
    item,
    quantity
  });
  return cart;
};
```

- El problema de este código es que hemos mutado un estado compartido.
- Si otras funciones dependen del estado del objeto de carrito.
- Ahora que hemos mutado el estado compartido, tenemos que preocuparnos del impacto que tendrá en la lógica del programa si cambiamos la Orden en que se llaman las funciones.

```
test('addToCart()', assert => {
  const msg = 'addToCart() should add a new item to the cart.';
  const originalCart = {
    items: []
  };
  const cart = addToCart(
    originalCart,
    {
      name: "Digital SLR Camera",
      price: '1495'
    },
    1
  );
  const expected = 1; // num items in cart
  const actual = cart.items.length;

  assert.equal(actual, expected, msg);

  assert.deepEqual(originalCart, cart, 'mutates original cart.');
  assert.end();
});
```

# Ejemplo – Carrito puro (ES6)

```
const addToCart = (cart, item, quantity) => {
  const newCart = lodash.cloneDeep(cart);

  newCart.items.push({
    item,
    quantity
  });
  return newCart;

};
```

- En este caso, la función pura devuelve un nuevo carrito, no muta el original

```
test('addToCart()', assert => {
  const msg = 'addToCart() should add a new item to the cart.';
  const originalCart = {
    items: []
  };

  // deep-freeze on npm
  // throws an error if original is mutated
  deepFreeze(originalCart);

  const cart = addToCart(
    originalCart,
    {
      name: "Digital SLR Camera",
      price: '1495'
    },
    1
  );

  const expected = 1; // num items in cart
  const actual = cart.items.length;

  assert.equal(actual, expected, msg);

  assert.notDeepEqual(originalCart, cart,
    'should not mutate original cart.');
  assert.end();
});
```



## Pongámoslo en práctica

- Tomando como base el ejemplo de Eric Elliot, clónalo y convierte las funciones impuras en puras.
- Los test unitarios no se deben cambiar. Tu código debe pasarlos.
- <https://codepen.io/ericelliott/pen/MyojLq?editors=0010>

3

# OOP en JS

# Objetos, Clases y Herencia

- Hasta su versión 5.0, JavaScript es un lenguaje basado en objetos que en lugar de estar basado en clases, se basa en prototipos.
- A partir de la versión 6.0 (ECMAScript 2015), las clases son parte nativa del lenguaje, pero el soporte de navegadores es muy básico todavía.
- Debido a esta diferencia, puede resultar menos evidente que JavaScript te permite crear jerarquías de objetos y herencia de propiedades y de sus valores.
- Los lenguajes orientados a objetos basados en clases, como Java y C#, se basan en la existencia de dos entidades distintas: clases e instancias.

# Lenguajes basados en clases vs. basados en prototipos

- Una clase define todas las propiedades (considerando como propiedades los métodos y campos de Java/C#) que caracterizan un determinado conjunto de objetos.
- Una clase es una entidad abstracta, y no un ejemplar del conjunto de objetos que describe.
- Una instancia, sin embargo, es la instanciación (puesta en marcha, podríamos decir) de una clase; es decir, uno de sus miembros.
- Una instancia tiene exactamente las propiedades de su clase padre (ni más, ni menos).

# Lenguajes basados en clases vs. basados en prototipos

- Un lenguaje basado en prototipos, como JavaScript, no hace esta distinción: simplemente tiene objetos.
- Un lenguaje basado en prototipos tiene la noción del objeto prototípico, un objeto que se utiliza como una plantilla a partir del cual se obtiene el conjunto inicial de propiedades de un objeto.
- Cualquier objeto puede especificar sus propiedades, ya sea cuando es creado inicialmente o en tiempo de ejecución.
- Además, cualquier objeto puede ser utilizado como el prototipo de otro objeto, permitiendo al segundo objeto compartir las propiedades del primero.

# Lenguajes basados en clases vs. basados en prototipos

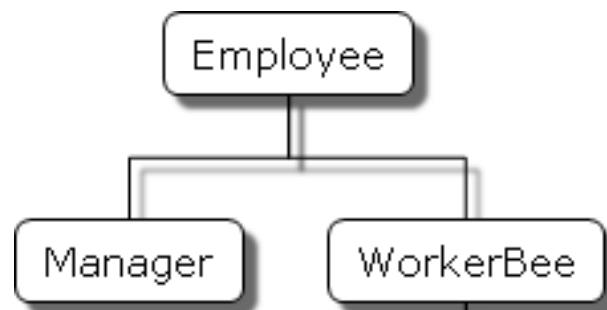
- JavaScript no tiene la definición de clase separada del constructor.
- En su lugar, se define una **función constructor** para crear objetos con un conjunto inicial de propiedades y valores.
- Cualquier función JavaScript puede utilizarse como constructor.
- Se utiliza el operador ***new*** con una función constructor para crear un nuevo objeto:
  - **var fecha = new Date();** // para *clases predefinidas*
  - **var employee = new Employee();** // para *clases de usuario previamente definidas mediante una función constructora.*

# Lenguajes basados en clases vs. basados en prototipos

Basado en clases (Java, C#)	Basado en prototipos (JavaScript)
La clase y la instancia son entidades distintas	Todos los objetos son instancias
Define una clase en la definición de clase; se instancia una clase con los métodos constructores.	Define y crea un conjunto de objetos con funciones constructoras.
Se crea un objeto con el operador new.	Igual.
Se construye una jerarquía de objetos utilizando la definición de las clases para definir subclases de clases existentes..	Se construye una jerarquía de objetos mediante la asignación de un objeto como el prototipo asociado a una función constructor.
Se heredan propiedades siguiendo la cadena de clases.	Se heredan propiedades siguiendo la cadena de prototipos.
La definición de una clase especifica todas las propiedades de todas las instnacias de esa clase. No se pueden añadir propiedades dinámicamente en tiempo de ejecución.	El conjunto inicial de propiedades lo determina la función constructor y el prototipo. Se pueden añadir y quitar propiedades dinámicamente a objetos específicos o a un conjunto de objetos.

# Lenguajes basados en clases vs. basados en prototipos

- Imaginemos que queremos crear una jerarquía de personas relacionadas con una empresa dada.
- Si partimos de la idea de ***Employee*** como base, podemos imaginar una jerarquía similar a la del gráfico siguiente:



# Lenguajes basados en clases vs. basados en prototipos

- A la hora de implementar la herencia, el programador deberá asociar un objeto prototípico con cualquier función constructor.
- De esta forma puedes crear una relación entre *Employee* y *Manager*, pero usando una terminología diferente, sin clases.
- En primer lugar, se define la función constructor *Employee*, especificando las propiedades ***nombreProp*** y ***dept***. La función constructora puede adoptar una forma similar a ésta:

```
function Employee () {  
    this.nombreProp = "";  
    this.dept = "general";  
}
```

- Una instancia como la declarada anteriormente, automáticamente "hereda" las propiedades ***nombreProp*** y ***dept***.

# Lenguajes basados en clases vs. basados en prototipos

- Después hay que definir la función constructor **Manager**, especificando la propiedad **reports**.
- Por último, hay que asignar un nuevo objeto **Employee** como el **prototype** de la función constructora de **Manager**.

```
function Manager () {  
    this.reports = [];  
}  
Manager.prototype = new Employee();
```

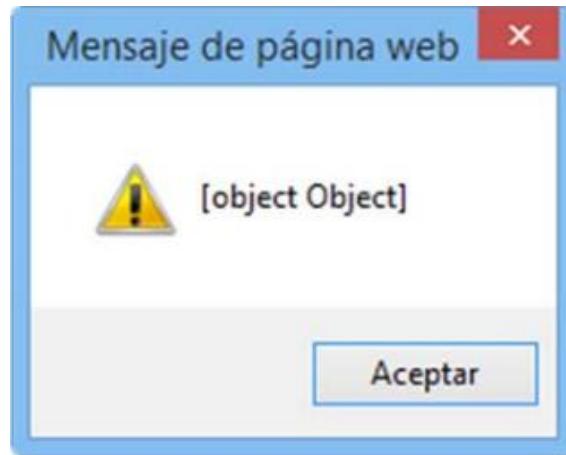
- De esta forma, cuando se crea un nuevo **Manager**, hereda las propiedades **nombreProp** y **dept** del objeto **Employee**.

# Aspectos peculiares de los prototipos

- Cuando creamos un objeto en JavaScript suceden algunas cosas curiosas "entre bambalinas".
  - Por ejemplo si declaramos un objeto vacío, a continuación podemos llamar a su función **valueOf**, u otra de las preexistentes en el objeto predefinido **Object**, ya que implícitamente, hereda su prototipo de **Object**.
  - Esto sucede porque todos los objetos disponen de prototipos que ayudan a modelar su construcción. En el caso anterior, como el objeto vacío no tiene prototipo definido, buscará una propiedad con ese nombre dentro de **Object**, y la ejecutará.
  - Si no la encontrase, seguiría buscando en la cadena de prototipos hasta producir un error, así que, podemos implementar la herencia en JavaScript, mediante prototipos (**prototypes**), que hacen las veces de clases bien definidas.
  - Si creamos un objeto vacío, este hereda un conjunto de elementos (funciones, por ejemplo) de su prototipo original (**Object.prototype**, al no indicarle otra cosa), y podemos usarlas directamente con ese objeto.

## Aspectos peculiares de los prototipos

- Por tanto, podríamos llamar a la función ***valueOf*** en la segunda instrucción, y obtendríamos una salida indicando que se trata del objeto ***Object ( [object Object] )***, como se aprecia en la figura:



- Como hemos visto antes, en la práctica podemos referirnos al prototipo de un objeto mediante la palabra reservada ***prototype***, y realizar operaciones con él e incluso (si no está explícitamente prohibido) cambiar el prototipo.



## Pongámoslo en práctica: Elementos DOM

- Añade el método remove al objeto Array, de tal manera que le puedas pasar cualquier número de argumentos
  
- **Tips:** usar la variable arguments para obtener la lista de argumentos

## Funciones, Prototipos y el operador new

- De forma que las funciones, y especialmente las que tienen estado (propiedades), pueden usarse de forma similar a lo que en otros lenguajes sería la definición de una clase.
- En parte debido al comportamiento dinámico de **this**, es posible utilizar una definición de función junto al operador **new** para crear nuevas "instancias" de una clase "definida" por una función (y téngase en cuenta el entrecomillado de los términos **instancias** y **definida**).
- Resumiendo lo dicho hasta aquí: si tenemos una función **Persona** definida previamente, podríamos crear nuevas instancias mediante la sintaxis:

# Funciones, Prototipos y el operador new

```
function Persona(nombre, apellidos) {  
    this.nombre= nombre;  
    this.apellidos= apellidos;  
}  
  
var Luis= new Persona('Luis', 'Marcos');  
var Irene= new Persona('Irene', 'García');  
Luis.apellidos === 'Marcos'; //true  
  
Persona.prototype.nombreCompleto= function() {  
    return this.nombre + " " + this.apellidos;  
}  
Irene.nombreCompleto() === "Irene García"; //true
```

## Funciones, Prototipos y el operador new

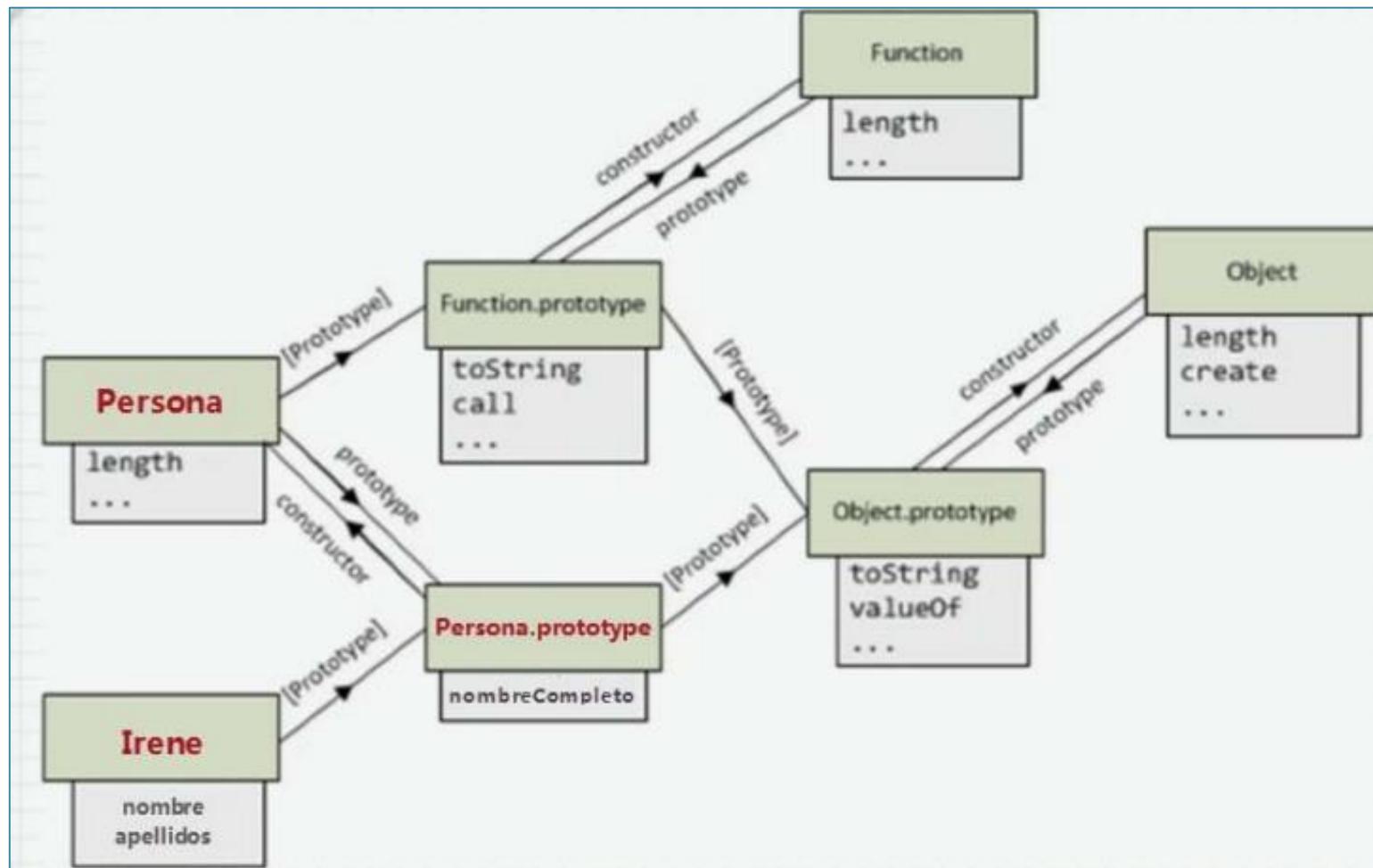
- Después de crear la función, podemos extenderla modificando su prototipo para añadir otra función adicional que nos devuelva el nombre completo.
- Al cambiar el prototipo, cualquier llamada a esa función por parte de cualquier objeto que herede del prototipo funcionará correctamente.
- En el código que nos ocupa, cuando se utiliza ***new*** en la construcción, primero se crea un objeto vacío a partir de ***Object***.
- A continuación, establece su prototipo al que corresponda con ***Persona.prototype***.
- Finalmente, ese objeto es pasado a la función ***Persona*** como un puntero ***this*** y la función queda inicializada.

# Funciones, Prototipos y el operador new

- **Nota:** Por convención (aunque no es obligatorio), la notación de las funciones que ejercen de definiciones de objetos se escriben con la primera letra mayúscula.
- Recordemos que no debemos confundir el prototipo con el constructor (**constructor**, en la sintaxis del lenguaje).
- Si consideramos el objeto **Irene**, el esquema representativo de la herencia y sus relaciones de ancestros sería la siguiente:

# Funciones, Prototipos y el operador new

- Jerarquía de prototipos generada a partir del código anterior



# Funciones, Prototipos y el operador new

- Como vemos en el esquema, el objeto Irene tiene unas definiciones propias (**nombre**, **apellidos**), que compartiría cualquier objeto instanciado de esta forma.
- Pero también se ha definido una propiedad directamente en su prototipo (**Persona.prototype**), que cuenta con una propiedad extra (**nombreCompleto**).
- El prototipo, a su vez, dispone de un constructor **Persona**, que dispone de una propiedad **length**, por el hecho de heredar de **Function**. Y así podemos seguir la cadena ascendente hasta el final.
  - El sistema parece un tanto lioso, pero así es como funciona.
- Por otra parte, las nuevas funciones de JavaScript 5, permiten progresar en la inspección de las cadenas de herencia hasta llegar al comienzo, sin importar si se trata de un elemento propio del lenguaje o de un elemento del DOM, ya que estos también heredan –al final de la cadena- de **Object**.

# Métodos y sobre escritura (Overriding)

- No se trata exactamente de la característica que nosotros conocemos en otros lenguajes orientados a objetos.
- Pero, lo cierto es que esta forma declarativa del lenguaje permite sobrescribir métodos existentes, con la sencilla técnica de volverlos a definir en nuestra instancia del objeto.
- Por ejemplo, para sobrescribir el método **valueOf**, en nuestro código anterior, podemos indicar lo siguiente:

```
var obj2 = {  
    valueOf: function () { return 3; },  
    Propiedad1: 123  
}
```

- Nótese que debemos declarar la propiedad como una función, ya que **valueOf** es una función (si asignásemos un literal, se generaría un error de ejecución), por lo que, en el tipo definido, el prototipo **sí** debe de respetarse.

## *Object.create()*

- La versión 5.0 de JavaScript nos permite crear nuevos objetos a partir de cualquier otro definido por el contexto o por nosotros, mediante el método ***Object.create()***.
- Por ejemplo, podemos crear un nuevo objeto **obj3** basado en el anterior, y añadirle las propiedades que nos convenga:

```
Var obj3 = Object.create(obj2);  
obj3.Propiedad2 = "Valor";
```

- El nuevo objeto contiene, entonces, la “herencia” dos prototipos: el suyo directo (**obj2**) y el del antecesor de éste (**Object**).
- Y su método ***valueOf***, tendrá el valor que reasignamos en su objeto padre (3: lo que devuelve la función).
- Además, heredará por defecto los valores de su antecesor, de forma que su propiedad ***Propiedad1*** valdrá **123**, como en el original, ya que –en realidad- apunta a aquél.

## El método *hasOwnProperty()*

- El método ***hasOwnProperty()*** devuelve un booleano indicando si el objeto tiene la propiedad especificada.
- Todo objeto descendiente de ***Object*** hereda ***hasOwnProperty***.
- Este método puede ser usado para determinar si un objeto tiene la propiedad especificada como una propiedad directa de ese objeto.
- A diferencia del operador ***in***, el método no elimina la cadena de ***prototipada*** del objeto.
- Ejemplo: determina si el objeto ***o*** contiene una propiedad ***prop***:

```
o = new Object();
o.prop = 'exists';

function changeO() {
    o.newprop = o.prop;
    delete o.prop;
}

o.hasOwnProperty('prop'); // true
changeO();
o.hasOwnProperty('prop'); // false
```

# El método *hasOwnProperty()*

- El siguiente ejemplo diferencia entre propiedades directas y propiedades heredadas a través de la cadena:

```
o = new Object();
o.prop = 'exists';
o.hasOwnProperty('prop');      // true
o.hasOwnProperty('toString');  // false
o.hasOwnProperty('hasOwnProperty'); // false
```

- Podemos realizar una iteración sobre las propiedades de un objeto sin ejecutar, solamente a través de sus propiedades heredadas.
- El bucle ***for..in*** puede iterar elementos enumerables, mostrando la capacidad de *hasOwnProperty* que no está solamente para iterar elementos "clásicos" (como con ***Object.getOwnPropertyNames()***).

## El método *hasOwnProperty()*

```
var objeto = {  
    propiedad: 'Valor de la propiedad'  
};  
  
for (var nombreProp in objeto) {  
    if (objeto.hasOwnProperty(nombreProp)) {  
        alert("Hay propiedad (" + nombreProp + "). Valor: " + objeto[nombreProp]);  
    }  
    else {  
        alert(nombreProp); // toString u otra cosa  
    }  
}
```



## Pongámoslo en práctica: Elementos DOM

- Usa el paso de contexto *this* para imprimir la descripción de un objeto león

2

# Closures

# Closures (*Namespaces* en JS)

- Una ***closure*** (o encerramiento), es el equivalente de un módulo o "***namespace***" en otros lenguajes.
- Permite separar funcional y operativamente un fragmento de código, de forma que –lo definido en él– no produzca "colisiones" con otro código existente o venidero.
- Lo que sigue es otro ejemplo de una "***closure***" más básica
- Hay que advertir dos cosas:
  - La función interna accede a lo declarado por la externa
  - Las llamadas posteriores no pierden el valor
  - Esto es debido a que se crea una zona privada de memoria asociada con el contexto.

# Functions: *Closures*

```
function leerContador() {  
    var i = 0;  
    return function () {  
        console.log(++i);  
    }  
}  
  
var valorContador = leerContador();  
valorContador(); //1  
valorContador(); //2
```

- Aunque parece evidente, conviene recordar que la razón de devolver una función es que –si no- estaríamos devolviendo el valor de retorno de la función log(), o sea, ***undefined***.

# Closures

- Quizá entenderemos mejor el funcionamiento si comparamos con este código:

```
function variante() {  
    var j = 0;  
    return function () {  
        console.log(++j);  
    }  
}  
var v = variante();  
v(); // undefined  
v(); // undefined
```

- En este caso, la función variante devuelve el valor de **j** incrementado en cada llamada
- Por tanto podemos decir que una "*closure*" es un **función con estado**.

# Closures

- Una de las características más importantes de las funciones es que **pueden crear un contexto**
- Ese **contexto** es lo que se asocia con la idea de **estado**
- Vamos a analizar el siguiente código y su resultado de salida:

```
var colores = ["Azul", "Verde", "Rojo", "Amarillo"];
for (var i = 0; i < colores.length; i++) {
    setTimeout(function () {
        console.log(colores);
        console.log(i);
        console.log(colores[i]);
    }, 200);
}
```

# Closures

- El problema es que ***setTimeout***, está programado para funcionar cuando transcurra una décima de segundo, y en ese momento, el bucle ha concluido, con lo que la variable ***i==4***.
- Además, ***colors[i]*** está fuera de rango, con lo que genera ***undefined***.
- Por tanto, la variable ***i*** esta fuera de contexto, cuando ***setTimeout*** va a utilizarla

```
var colores = ["Azul", "Verde", "Rojo", "Amarillo"];
for (var i = 0; i < colores.length; i++) {
    setTimeout(function () {
        console.log(colores);
        console.log(i);
        console.log(colores[i]);
    }, 200);
}
```

```
Consola × 0 ⚠ 0 ⓘ 1 ✎ X
HTML1300: Navegación realizada.
Archivo: 4-Funciones(Closures)-2.html
Azul,Verde,Rojo,Amarillo
4
undefined
Azul,Verde,Rojo,Amarillo
4
undefined
Azul,Verde,Rojo,Amarillo
4
undefined
Azul,Verde,Rojo,Amarillo
4
undefined
```

# Closures

- Al crear el contexto, para que la variable *i* sea reconocida, usamos una "**closure**"
- Encerramos la función dentro de un par de paréntesis de llamada y la llamamos con el argumento *i* (y modificamos la función para que reciba esa *i*)
- Con eso, resolvemos el problema del contexto, pero introducimos uno nuevo: **setTimeout** espera una función. Luego...
- Hacemos que nuestra "**closure**" devuelva una función que realiza las tareas indicadas

```
var masColores = ["Blanco", "Purpura", "Negro", "Gris"];
for (var i = 0; i < masColores.length; i++) {
    setTimeout( function (i) {
        return function () {
            console.log(masColores);
            console.log(i);
            console.log(masColores[i]);
        }
    })(i), 200);
}
```

```
▶ Array[4]
  0
  Blanco
  ▶ Array[4]
  1
  Purpura
  ▶ Array[4]
  2
  Negro
  ▶ Array[4]
  3
  Gris
```

# Una nota sobre **this**

- Siempre se refiere al "propietario" de la función que se está ejecutando, o al objeto al cual pertenece esa función.
- La duda surge al establecer exactamente, quién es el propietario de algo.
- La forma de indicar esa pertenencia es mediante el operador ***new***
- En JavaScript solo hay dos propietarios posibles: los objetos declarados o Global (el DOM u objeto ***window***). Por tanto:

```
var myFunction = function() {  
    console.log(this);  
};  
  
myFunction();  
var myvariable = new myFunction();
```



```
► DOMWindow  
▼ myFunction  
► __proto__: Object
```



# Modularidad

- Como consecuencia de todo lo anterior, es lícito declarar funciones que generen módulos usando closures:

```
// Modularidad y ámbito de variables
(function () {
    var variableLocal = "Esto es local por el contexto de la función";
    nolocal = "Esto pasa a pertenecer a Global"; //falta -var-
    noLocalAsignada = function () { // funcion "exportada" pero "ve" su contenedor
        console.log(variableLocal);
        return variableLocal;
    }
})();
typeof noLocalAsignada; // function
typeof noLocalAsignada(); // string
```



## Pongámoslo en práctica: Closure verificador

- Crea un closure que cuente las veces que se ha clicado en un botón y lo exprese en el html

2

# Promises

# Promises (promesas)

- El objeto Promise representa la eventual finalización (o fallo) de una operación asíncrona y su valor resultante.
- Una promesa es un proxy para un valor no necesariamente conocido cuando se crea la promesa.
- Permite asociar **handlers** para el caso de éxito o fallo de una acción asíncrona.
- Esto permite que los métodos asíncronos devuelvan valores como métodos sincronos: en lugar de devolver inmediatamente el valor final: **el método asíncrono devuelve una promesa de suministrar el valor en algún momento en el futuro.**
- Las promesas se crean usando el constructor **Promise**. Y la programación de su resolución se hace mediante el método **then**

# Uso básico

```
var p = new Promise(function(resolve, reject) {  
    // Do an async task async task and then...  
    if(/* good condition */) {  
        resolve('Success!');  
    } else {  
        reject('Failure!');  
    }  
});  
  
p.then(function() {  
    /* do something with the result */  
}).catch(function() {  
    /* error :( */  
})
```

# Ejemplo

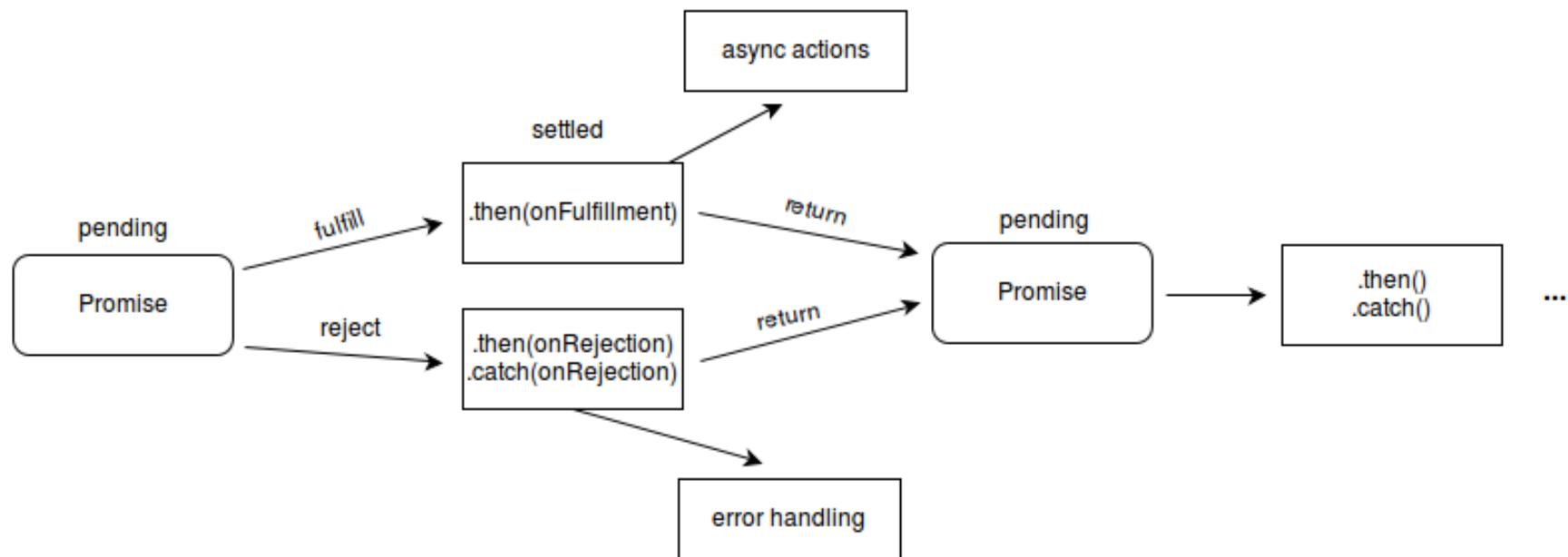
```
let myFirstPromise = new Promise((resolve, reject) => {
  // We call resolve(...) when what we were doing made async successful, and reject(...) when it failed.
  // In this example, we use setTimeout(...) to simulate async code.
  // In reality, you will probably be using something like XHR or an HTML5 API.
  setTimeout(function(){
    resolve("Success!"); // Yay! Everything went well!
  }, 250);
});
```

```
myFirstPromise.then((successMessage) => {
  // successMessage is whatever we passed in the resolve(...) function above.
  // It doesn't have to be a string, but if it is only a succeed message, it probably will be.
  console.log("Yay! " + successMessage);
});
```

# Promises - estados

- Una promesa está en uno de estos estados:
  - Pendiente (pending): estado inicial, no cumplido ni rechazado.
  - Cumplido (fulfilled): lo que significa que la operación se completó correctamente.
  - Rechazado (rejected): significa que la operación falló.
- Una promesa pendiente puede ser cumplida con un valor, o rechazada con una razón (error).
- Cuando ocurre cualquiera de estas opciones, los **handlers** asociados en cola mediante el método **then** se llaman.

# Promises - estados



# Uso básico de Promises

- A veces no es necesario completar tareas asíncronas dentro de la promesa. Pero si el código cliente espera una promesa, es posible devolverla como respuesta de una función dada .
- En ese caso, simplemente puede llamar a **Promise.resolve()** o **Promise.reject()** sin usar la palabra clave “**new**”.
- Puesto que siempre se devuelve una promesa, siempre puede utilizar los métodos `then` y `catch` en su valor de retorno.

# Ejemplo

```
var userCache = {};  
  
function getUserDetail(username) {  
    // In both cases, cached or not, a promise will be returned  
  
    if (userCache[username]) {  
        // Return a promise without the "new" keyword  
        return Promise.resolve(userCache[username]);  
    }  
  
    // Use the fetch API to get the information  
    // fetch returns a promise  
    return fetch('users/' + username + '.json')  
        .then(function(result) {  
            userCache[username] = result;  
            return result;  
        })  
        .catch(function() {  
            throw new Error('Could not find user: ' + username);  
        });  
}
```

# Then

- Todas las promises tienen un método que permite reaccionar a una promesa: **Then**.
- El primer método **then** de callback, recibe el resultado dado por la llamada **resolve()**:

```
new Promise(function(resolve, reject) {  
  // Una acción simulada con setTimeout  
  setTimeout(function() { resolve(10); }, 3000); })  
  .then(function(result) {  
    console.log(result); })  
  // Desde la consola: 10
```

# Then

- La llamada de retorno se activa cuando se resuelve la promise.
- También puede encadenar las devoluciones de llamada del método then:

```
new Promise(function(resolve, reject) {  
    // Una acción simulada con setTimeout  
    setTimeout(function() { resolve(10); }, 3000); }  
    .then(function(num) { console.log('first then: ', num); return num * 2;  
    })  
    .then(function(num) { console.log('second then: ', num); return num *  
    2; })  
    .then(function(num) { console.log('last then: ', num); })  
};  
// primer then: 10 segundo then: 20 último then: 40
```

# Catch

- **Catch:** El callback catch se ejecuta cuando se rechaza la promise:

```
new Promise(function(resolve, reject) {  
    // A mock async action using setTimeout  
    setTimeout(function() { reject('Done!'); }, 3000);  
}  
.then(function(e) { console.log('done', e); })  
.catch(function(e) { console.log('catch:', e); });  
  
// From the console:  
// 'catch: Done!'
```

- En este caso, un patrón muy usado es enviar un **Error** al **catch**:

```
reject(Error('Data could not be found'));
```

# Promise.all

- El método Promise.all toma una serie de promises y activa un callback de llamada una vez todas están resueltas:

```
Promise.all([promise1, promise2]).then(function(results) {  
    // Ambas promises resueltas  
})  
.catch(function(error) {  
    // Una o más promises fueron rechazadas  
});
```

- Una manera perfecta de pensar acerca de Promise.all es disparar múltiples solicitudes AJAX (via fetch) al mismo tiempo:

```
var request1 = fetch('/users.json');  
var request2 = fetch('/articles.json');  
  
Promise.all([request1, request2]).then(function(results) {  
    // Both promises done!  
});
```

# Promise.all

- Una manera perfecta de pensar acerca de Promise.all es disparar múltiples solicitudes AJAX (via fetch) al mismo tiempo:

```
var request1 = fetch('/users.json');
var request2 = fetch('/articles.json');
```

```
Promise.all([request1, request2]).then(function(results) {
    // Both promises done!
});
```

- Se pueden combinar APIs como fetch y Battery API ya que ambas devuelven promises:

```
Promise.all([fetch('/users.json'), navigator.getBattery()]).then(function(results) {
    // Ambas promises hechas!
});
```

# Promise.all

- Si alguna promise es rechazada el catch se dispara al primer rechazo:

```
var req1 = new Promise(function(resolve, reject) {
    // A mock async action using setTimeout
    setTimeout(function() { resolve('First!'); }, 4000);
});
var req2 = new Promise(function(resolve, reject) {
    // A mock async action using setTimeout
    setTimeout(function() { reject('Second!'); }, 3000);
});
Promise.all([req1, req2]).then(function(results) {
    console.log('Then: ', results);
}).catch(function(err) {
    console.log('Catch: ', err);
});

// From the console:
// Catch: Second!
```

- Promise.all será muy útil a medida que más APIs se muevan hacia promises.

## Promise.race

- Promise.race permite que, en lugar de esperar a que todas las promises sean resueltas o rechazadas, se active tan pronto como se resuelve o rechaza cualquier promise en la matriz:
- Un caso de uso podría estar provocando una solicitud a una fuente primaria y una fuente secundaria (en caso de que la primaria o la secundaria no estén disponibles).

# Ejemplo

```
var req1 = new Promise(function(resolve, reject) {  
    // A mock async action using setTimeout  
    setTimeout(function() { resolve('First!'); }, 8000);  
});  
var req2 = new Promise(function(resolve, reject) {  
    // A mock async action using setTimeout  
    setTimeout(function() { resolve('Second!'); }, 3000);  
});  
Promise.race([req1, req2]).then(function(one) {  
    console.log('Then: ', one);  
}).catch(function(one, two) {  
    console.log('Catch: ', one);  
});  
  
// From the console:  
// Then: Second!
```

# Promises y HTTP

- La API XMLHttpRequest es asíncrona pero no utiliza la API Promises. Sin embargo, hay algunas API nativas que ahora usan promises:
  - Battery API (<https://davidwalsh.name/javascript-battery-api>)
  - fetch API (<https://davidwalsh.name/fetch>)
  - ServiceWorker API
- Las promises van haciéndose cada vez más frecuentes, por lo que es importante que todos los desarrolladores front-end se acostumbren a ellos.
- También vale la pena señalar que Node.js es una plataforma para Promises (al ser Promise es una característica del lenguaje javascript).
- Corresponde al desarrollador llamar manualmente a **resolve** o **reject** dentro del cuerpo de la devolución de llamada en función del resultado de su tarea.

# Ejemplo

```
function get(url) {  
  // Return a new promise.  
  return new Promise(function(resolve, reject) {  
    // Do the usual XHR stuff  
    var req = new XMLHttpRequest();  
    req.open('GET', url);  
  
    req.onload = function() {  
      // This is called even on 404 etc  
      // so check the status  
      if (req.status == 200) {  
        // Resolve the promise with the response text  
        resolve(req.response);  
      }  
      else {  
        // Otherwise reject with the status text  
        // which will hopefully be a meaningful error  
        reject(Error(req.statusText));  
      }  
    };  
  
    // Handle network errors  
    req.onerror = function() {  
      reject(Error("Network Error"));  
    };  
  
    // Make the request  
    req.send();  
  });  
}
```

```
// Use it!  
get('story.json').then(function(response) {  
  console.log("Success!", response);  
}, function(error) {  
  console.error("Failed!", error);  
});
```



[...]**netmind**

WeKnowIT

Barcelona

C. Almogàvers, 123  
08018 Barcelona  
Tel. 93 304.17.20  
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7  
28020 Madrid  
Tel. 91 442.77.03  
Fax. 91 442.77.07

[www.netmind.es](http://www.netmind.es)



GOBIERNO  
DE ESPAÑA

MINISTERIO  
DE ENERGÍA, TURISMO  
Y AGENDA DIGITAL

**red.es**



ESTRATEGIA DE  
EMPRENDIMIENTO Y  
EMPLEO JUVENIL  
*garantía juvenil*



Agenda Digital para España



**UNIÓN EUROPEA**

Fondo Social Europeo  
*“El FSE invierte en tu futuro”*