

# ÍNDICE

## 1. FUNCIONES

- a. Funciones predefinidas del lenguaje
- b. Funciones definidas por el usuario

## 2. ARRAYS

## 3. OBJETOS DEFINIDOS POR EL USUARIO

# Objetos definidos por el usuario

- JavaScript proporciona una serie objetos predefinidos, sin embargo es posible crear nuevos objetos definidos por el usuario.
- Cada uno de estos objetos puede tener sus propios métodos y propiedades.
- La creación de nuevos objetos resulta útil en el desarrollo de aplicaciones avanzadas.

# Objetos definidos por el usuario

- Algunas formas:

```
alumno = new Object();  
alumno.nombre="Juan";  
alumno.curso = "2ºDAW";  
alumno.materias = 5;
```

```
alumno = {nombre:"Juan",  
curso:"2ºDAW", materias:5}
```

# Objetos definidos por el usuario

- Declaración e inicialización de los objetos:
  - Un objeto es una entidad que posee unas *propiedades* que lo caracterizan y unos *métodos* que actúan sobre estas propiedades.
  - JavaScript dispone también de la palabra **this** para hacer referencia al objeto actual.
  - Su sintaxis es la siguiente:

```
function mi_objeto (valor_1, valor_2, valor_x){  
  this.propiedad_1 = valor_1;  
  this.propiedad_2 = valor_2;  
  this.propiedad_x = valor_x;  
}
```

# Objetos definidos por el usuario

- Ejemplo:

```
<script type="text/javascript">
  function Coche(marca_in, modelo_in, anyo_in){
    this.marca = marca_in;
    this.modelo = modelo_in;
    this.anyo = anyo_in;
  }

  //Instanciar un objeto
  coche1 = new Coche("Ferrari", "Scaglietti", "2010");
</script>
```

# Objetos definidos por el usuario

- Para acceder a una propiedad podemos utilizar:
  - Notación por `.` `coche.marca = "BMW" ;`
  - Notación por `[]` `coche['marca'] = "BMW" ;`
- Las propiedades de un objeto pueden ser valores de cualquier tipo:
  - Cadenas
  - Números
  - Boolean
  - Arrays
  - Funciones
  - Otros objetos

```
var obj = {  
  nombre: "Zanahoria",  
  detalles: {  
    color: "naranja",  
    medida: 12  
  }  
}
```

```
> obj.detalles.color  
naranja  
> obj["detalles"]["medida"]  
12
```

# Objetos definidos por el usuario

```
<script type="text/javascript">
  var coches = new Array(4);
  coches[0] = new Coche("Ferrari", "Scaglietti", "2010");
  coches[1] = new Coche("BMW", "Z4", "2010");
  coches[2] = new Coche("Seat", "Toledo", "1999");
  coches[3] = new Coche("Fiat", "500", "1995");
  for(i=0; i<coches.length; i++){
    document.write("Marca: " + coches[i].marca +
      " - Modelo: " + coches[i].modelo + " - Año  

      de fabricación: " + coches[i].anyo + "<br>");
  }
</script>
```

# Objetos definidos por el usuario

- Es posible añadir otras propiedades a cada instancia del objeto, por ejemplo:

```
function Coche (marca_in, modelo_in, anyo_in){  
  this.marca = marca_in;  
  this.modelo = modelo_in;  
  this.anyo = anyo_in;  
}  
var miCoche = new Coche("Pegeout", "206cc", 2003);  
miCoche.color = "azul";  
  
var tuCoche = new Coche("Fiat", "Punto", 2000);  
tuCoche.plazas = 5;  
  
var suCoche = new Coche("Audi", "A3", 2010);
```



# Objetos definidos por el usuario

```
function Factura(idFactura, idCliente) {  
    this.idFactura = idFactura;  
    this.idCliente = idCliente;  
  
    this.muestraCliente = function() {  
        alert(this.idCliente);  
    }  
  
    this.muestraId = function() {  
        alert(this.idFactura);  
    }  
}  
  
var laFactura = new Factura(3, 7);  
laFactura.muestraCliente();  
var otraFactura = new Factura(5, 4);  
otraFactura.muestraId();
```

# Objetos definidos por el usuario

```
function Factura(idFactura, idCliente) {  
    this.idFactura = idFactura;  
    this.idCliente = idCliente;  
}
```

```
Factura.prototype.muestraCliente = function() {  
    alert(this.idCliente);  
}
```

```
Factura.prototype.muestraId = function() {  
    alert(this.idFactura);  
}
```

# Objetos definidos por el usuario

- Si queremos trabajar con la propiedad **prototype** para poder realizar herencias entre objetos hay que crearlos mediante la función constructora.

```
function mi_objeto (valor_1, valor_2, valor_x){  
  this.propiedad_1 = valor_1;  
  this.propiedad_2 = valor_2;  
  this.propiedad_x = valor_x;  
}
```

- Crear objetos directamente mediante `{}` y `new Object()` **no permite heredar**.

# Objetos definidos por el usuario

- Si el número de parámetros que se va a pasar a una función es elevado o puede variar, se debe sustituir por un objeto. **(patrón configuración)**

```
//antipatrón  
function altaEmpleado(code,nombre, apellido, categoria,sueldo,dpto)  
altaEmpleado('E01','Manuel','Moral','aprendiz',200,'D1');
```

```
//correcto  
var empleado = {  
  code: 'E01',  
  nombre: 'Manuel',  
  apellido: 'Moral',  
  categoría: 'aprendiz',  
  sueldo: 200,  
  dpto.: 'D1'  
}  
altaEmpleado(empleado);
```

# Objetos definidos por el usuario

- **Modificar objetos definidos por el usuario**
  - Se pueden añadir/eliminar las propiedades y métodos cuando queramos
  - Para acceder a las propiedades *this.propiedad*
- **Modificar objetos predefinidos de Javascript(excepto Math)**
  - **Propiedades**
    - *String, Number y Boolean*
      - Objeto.prototype.nombre → SOLO lectura para datos primitivos
    - *Array*
      - Array.prototype.nombre → lectura/escritura en datos primitivos y objetos array
  - **Métodos** (String, Number, Boolean y Array)  
Lectura/escritura tanto datos primitivos como objetos  
Se hace referencia mediante *this*

# Objetos definidos por el usuario

- Para mostrar las propiedades de un objeto tenemos las siguientes opciones:

- For in

- Object.keys(obj)

} Devuelven todas las propiedades enumerables de un objeto

- Object.getOwnPropertyNames(miCoche)



Devuelve todas las propiedades enumerables o no de un objeto

Distintas formas para saltarnos las funciones:

- `typeof objeto[i] !== "function"`
- `!String(objeto[i]).includes('function')`
- `objeto[i].constructor.name.indexOf("Function") === -1`
- `objeto[i].constructor !== Function`

# Objetos definidos por el usuario

- Cada propiedad definida en un objeto tiene por defecto las siguientes características:
  - Es **enumerable**: será listada si se recorre el objeto con una estructura repetitiva *for..in* o utilizando el método *Object.keys()*.
  - Es **configurable**: Se podrá eliminar dicha propiedad
  - Es **modificable**: Se podrá cambiar su valor.

# Objetos definidos por el usuario

- Existe otra forma de crear un objeto, permitiendo además definir sus propiedades y cuál será su prototype.
- El *prototype* de un objeto es otro objeto, el cual guardará las propiedades y métodos que compartirá con el nuevo objeto.
- `Object.create(prototype, {propiedades})`

```
var dish = Object.create(Object.prototype, {  
  name: {  
    value: '',  
    writable: true,  
    configurable: false  
  },  
  ingredients: {  
    value: [],  
    writable: true,  
    configurable: false  
  }  
});
```

```
dish.name = 'Ceviche simple';  
dish.name;  
// "Ceviche simple"  
  
dish.ingredients.push('1 kilo de pescado');  
// 1
```



# Objetos definidos por el usuario

- Para crear o modificar propiedades de un objeto podemos utilizar `object.defineProperty(objeto, propiedad, {descriptor})`
- Permite un control más avanzado de cómo se podrá manipular la propiedad.
- Cuando se define una propiedad con este método, por defecto no es ni enumerable ni configurable ni modificable (excepto en Chrome, donde por defecto sí es modificable)

```
var siteTitle = {};  
  
Object.defineProperty(siteTitle, 'internalValue', {  
  writable: true,      // internalValue podrá cambiar de valor  
  configurable: false, // internalValue podrá cambiar de valor pero no  
                      // ser eliminado del objeto  
  enumerable: true     // internalValue aparecerá en Object.keys o  
                      // usando for..in  
});
```

# Objetos definidos por el usuario

- Si queremos definir más de una `object.defineProperty()`

```
var siteTitle = {};  
  
Object.defineProperty(siteTitle, {  
  internalValue: {  
    writable: true,      // internalValue podrá cambiar de valor  
    configurable: true,  // internalValue podrá cambiar de valor y ser  
                          // eliminado del objeto  
    enumerable: true     // internalValue aparecerá en Object.keys o  
                          // usando for..in  
  },  
  toTitle: {  
    writable: false,     // el método toTitle no podrá cambiar de valor  
    configurable: false, // el método toTitle no podrá ser eliminado ni  
                          // cambiado de valor,  
    enumerable: false,   // el método toTitle no aparecerá en  
                          // Object.keys o usando for..in  
    value: function() {  
      return this.internalValue.toUpperCase().split('').join(' ');  
    }  
  }  
});
```

# Objetos definidos por el usuario

## Copia de objetos:

- Un objeto se puede crear por copia de un objeto ya existente.

Ej: `var obj2 = obj1`

Se copia el contenido pero comparten la misma dirección de memoria.

Si se modifica `obj2` también lo hará `obj1`

```
var c1 = new Coche("Fiat", "Punto", 2000);  
var c2 = c1;  
c2.marca = "Mercedes";
```

# Objetos definidos por el usuario

## Copia de objetos:

- Tenemos dos formas para realizar copias de objetos y que sean independientes:

```
var c1 = new Coche("Fiat", "Punto", 2000);  
var c2 = new Coche(c1.marca, c1.modelo, c1.anyo);
```

```
function copia(obj){  
    obj.marca = this.marca;  
    obj.modelo = this.modelo;  
    obj.anyo = this.anyo;  
}  
  
var c3 = new Coche();  
c1.copia(c3);
```

# Objetos definidos por el usuario

## Quitar las referencias a los objetos:

- Asignando a *null* rompemos la referencia y destruimos el objeto para que más tarde el Garbage Collector de JavaScript lo limpie de la memoria.

```
var obj=new Object();  
obj=null;
```

# Novedades ES6 - objetos

- Eliminar repeticiones en la creación de objetos



The screenshot shows a code editor with a dark background. At the top, a function `buildUser` is defined, which takes `first` and `last` as arguments and returns an object containing `first`, `last`, and `fullName`. A green checkmark icon is next to the function definition. An orange arrow points from the text "Way cleaner!" with a thumbs-up icon to the object literal `{ first, last, fullName }` in the `return` statement. Below this, the text "Yields the same result:" is displayed. Underneath, a code snippet shows the function being called with `"Sam"` and `"Williams"`, and the resulting object being logged to the console. An orange arrow points from the console output to the object literal in the function call. The console output shows three lines: `> Sam`, `> Williams`, and `> Sam Williams`. A blue circular icon with a 'C' is in the bottom right corner of the console area.

```
function buildUser(first, last){  
  let fullName = first + " " + last;  
  
  return { first, last, fullName };  
}
```

Way cleaner! 👍

Yields the same result:

```
let user = buildUser("Sam", "Williams");  
console.log( user.first );  
console.log( user.last );  
console.log( user.fullName );
```

```
> Sam  
> Williams  
> Sam Williams
```

# Novedades ES6 - objetos

- Eliminar repeticiones en la creación de objetos



The screenshot shows a code editor with a dark background. At the top, a function `buildUser` is defined, which takes `first` and `last` as arguments, concatenates them into `fullName`, and returns an object containing `first`, `last`, and `fullName`. An orange arrow points from the returned object to a text annotation: "Returning objects with keys and variables with the same name looks repetitive". Below the function, the text "Calling the *buildUser* function:" is followed by code that calls `buildUser` with "Sam" and "Williams", and logs `user.first`, `user.last`, and `user.fullName` to the console. An orange arrow points from the console logs to a terminal window showing the output: `> Sam`, `> Williams`, and `> Sam Williams`. The terminal window has a blue 'C' icon in the top right corner.

```
function buildUser(first, last){  
  let fullName = first + " " + last;  
  
  return { first: first, last: last, fullName: fullName }  
}
```

Calling the *buildUser* function:

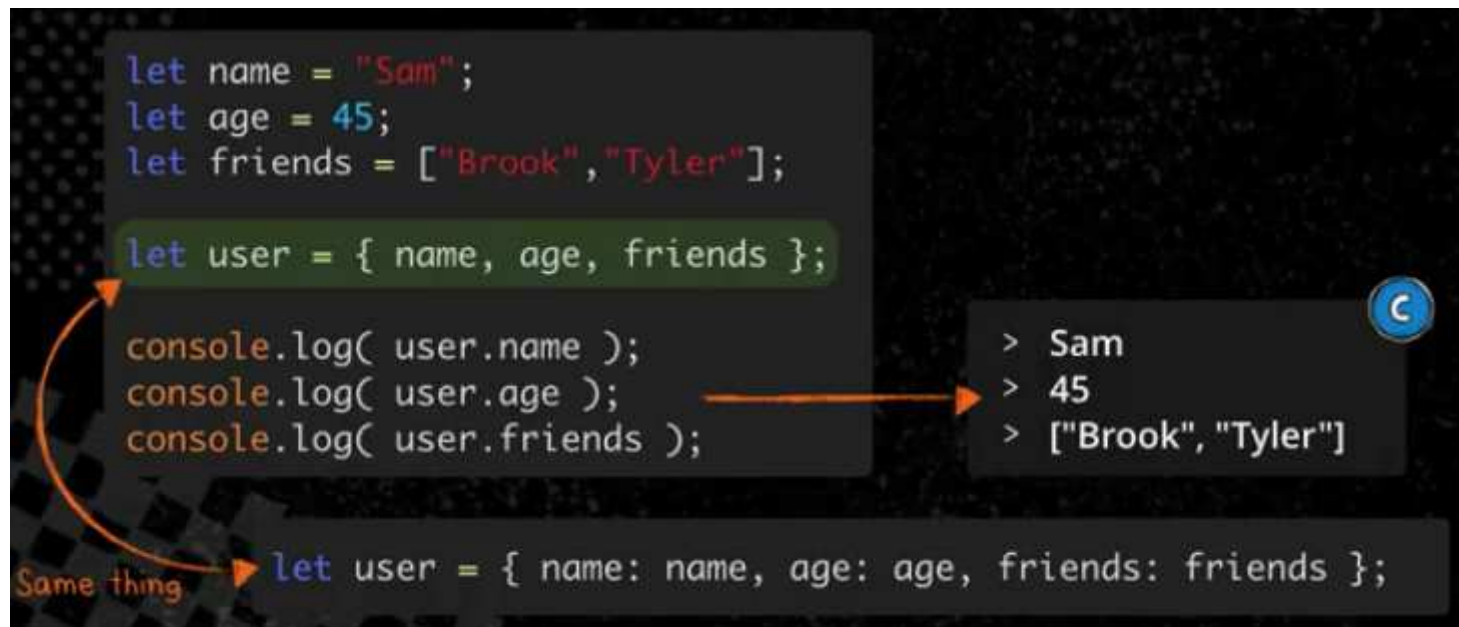
```
let user = buildUser("Sam", "Williams");  
  
console.log( user.first );  
console.log( user.last );  
console.log( user.fullName );
```

> Sam  
> Williams  
> Sam Williams

Returning objects with keys and variables with the same name looks repetitive

# Novedades ES6 - objetos

- Eliminar repeticiones en la creación de objetos
- Otro ejemplo





# Novedades ES6 - objetos

- Construye un objeto user utilizando la nueva sintaxis. Este objeto tendrá como propiedades: name, totalReplies y avatar.

```
let name = "Brook";  
let totalReplies = 249;  
let avatar = "/users/avatars/brook-user-1.jpg";  
  
addUserToSidebar(user);
```

```
let name = "Brook";  
let totalReplies = 249;  
let avatar = "/users/avatars/brook-user-1.jpg";  
  
let user = {name, totalReplies, avatar} ;  
  
addUserToSidebar(user);
```

# Novedades ES6 - objetos

- Destructuring

The image shows a code editor with two versions of a JavaScript snippet. The top version (marked with a red 'X') uses the old syntax: `let user = buildUser("Sam", "Williams");` followed by `let first = user.first;`, `let last = user.last;`, and `let fullName = user.fullName;`. An annotation "Unnecessary repetition" points to these lines. The bottom version (marked with a green checkmark) uses ES6 destructuring: `let { first, last, fullName } = buildUser("Sam", "Williams");`. An annotation "Same names as properties from return object" points to the destructuring syntax. Another annotation "This function returns { first, last, fullName }" points to the function call. Below the code, a console log shows the output: `> Sam`, `> Williams`, and `> Sam Williams`. A blue 'C' icon is visible next to the console output.

```
let user = buildUser("Sam", "Williams");  
let first = user.first;  
let last = user.last;  
let fullName = user.fullName;
```

Unnecessary repetition

Same names as properties from return object

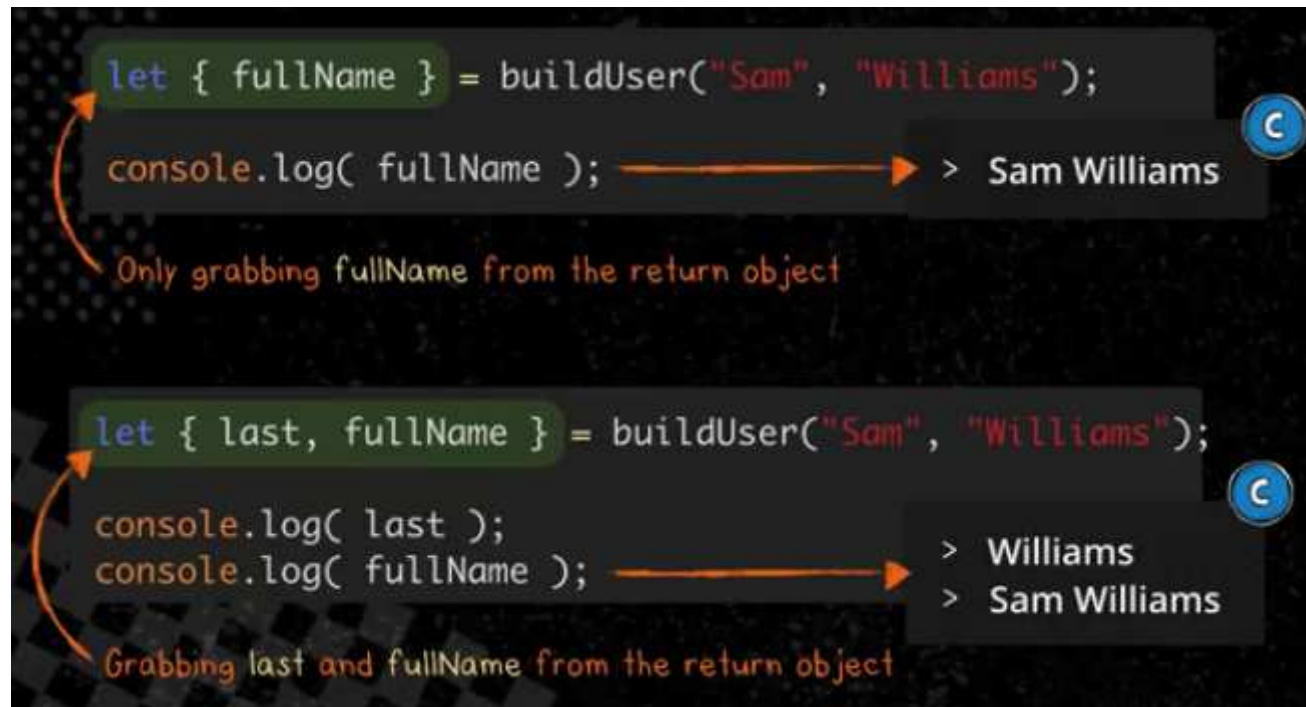
This function returns { first, last, fullName }

```
let { first, last, fullName } = buildUser("Sam", "Williams");  
console.log( first );  
console.log( last );  
console.log( fullName );
```

```
> Sam  
> Williams  
> Sam Williams
```

# Novedades ES6 - objetos

- **Destructuring**



# Novedades ES6 - objetos

- Inicializador de objetos vs Destructuring

**Object Initializer Shorthand Syntax**

```
let name = "Sam";  
let age = 45;  
  
let user = { name, age };  
  
console.log( user.name );  
console.log( user.age );
```

From variables to object properties

**Object Destructuring**

```
let { first, last, fullName } = buildUser("Sam", "Williams");  
  
console.log( first );  
console.log( last );  
console.log( fullName );
```

From object properties to variables

Returns { first, last, fullName }

# Novedades ES6 - objetos

- **Inicializador de objetos vs Destructuring**
  - ¿Qué se está realizando en los siguientes snippets?

```
// Snippet #1
let { tags, isLocked } = topicInfo(17);

// Snippet #2
let reply = { author, body, repliedAt };

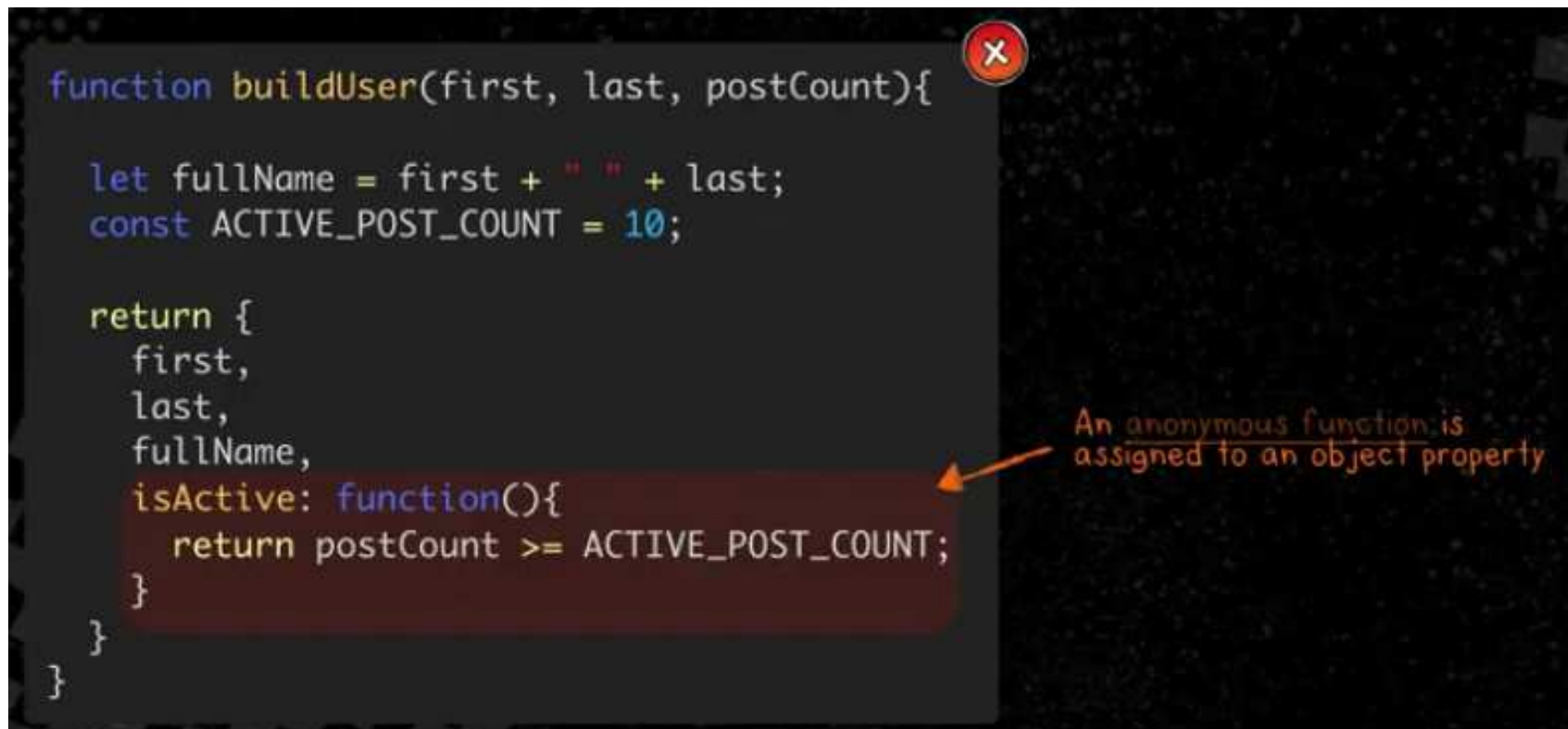
// Snippet #3
function buildMetadata(object){
  let id = parseInt(object.id);
  let lastUpdatedAt = object.updatedAt || object.createdAt;
  let hashCode = _buildHashCode(object);

  return { id, lastUpdateAt, hashCode };
}
```

- ☐ Destructuring, Initializer, Destructuring
- ☐ Destructuring, Initializer, Initializer
- ☐ Initializer, Initializer, Initializer

# Novedades ES6 - objetos

- Añadir una función a un objeto
  - ES5



```
function buildUser(first, last, postCount){  
  
  let fullName = first + " " + last;  
  const ACTIVE_POST_COUNT = 10;  
  
  return {  
    first,  
    last,  
    fullName,  
    isActive: function(){  
      return postCount >= ACTIVE_POST_COUNT;  
    }  
  }  
}
```

An anonymous function is assigned to an object property

# Novedades ES6 - objetos

- Añadir una función a un objeto
- ES6

```
function buildUser(first, last, postCount){  
  let fullName = first + " " + last;  
  const ACTIVE_POST_COUNT = 10;  
  
  return {  
    first,  
    last,  
    fullName,  
    isActive(){  
      return postCount >= ACTIVE_POST_COUNT;  
    }  
  }  
}
```

Less characters and easier to read! 👍



# Novedades ES6 - objetos

- **Object.assign**

```
function countdownTimer(target, timeLeft, options = {}){  
  
  let container = options.container || ".timer-display";  
  let timeUnit = options.timeUnit || "seconds";  
  let clonedDataAttribute = options.clonedDataAttribute || "cloned";  
  let timeoutClass = options.timeoutClass || ".is-timeout";  
  let timeoutSoonClass = options.timeoutSoonClass || ".is-timeout-soon";  
  let timeoutSoonTime = options.timeoutSoonSeconds || 10;  
  
  //...  
}
```

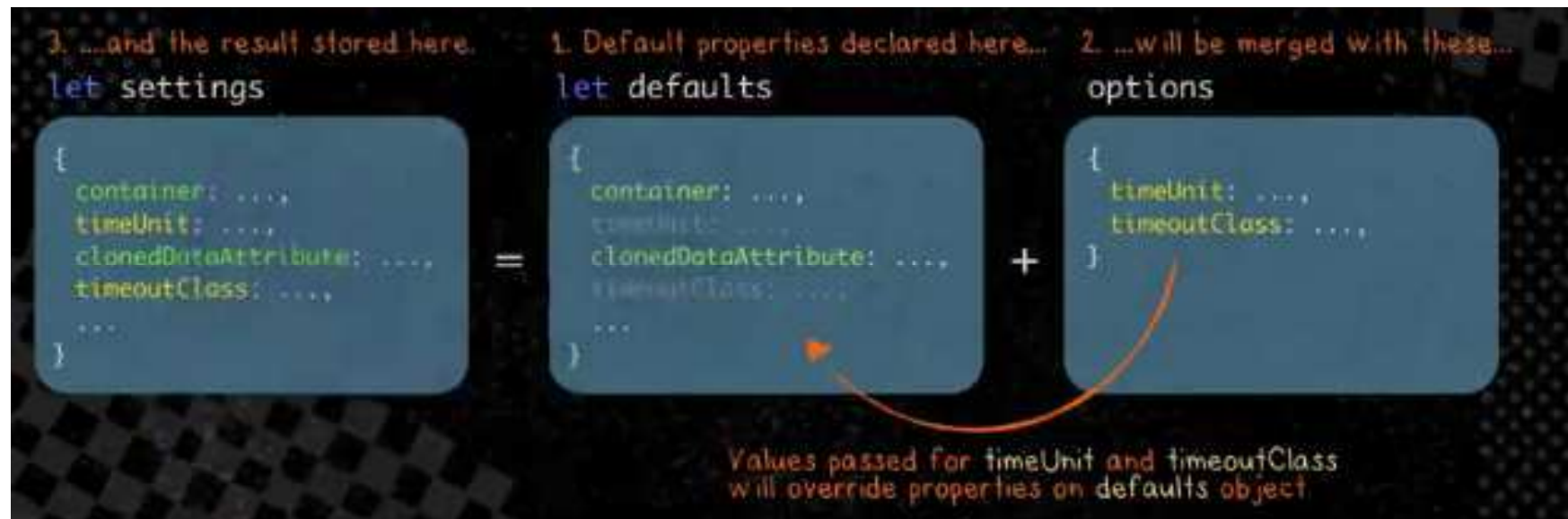
```
function countdownTimer(target, timeLeft, options = {}){  
  
  let defaults = {  
    container:      ".timer-display",  
    timeUnit:       "seconds",  
    clonedDataAttribute: "cloned",  
    timeoutClass:    ".is-timeout",  
    timeoutSoonClass: ".is-timeout-soon",  
    timeoutSoonTime: 10  
  };  
  
  //...  
}
```

Looks much better



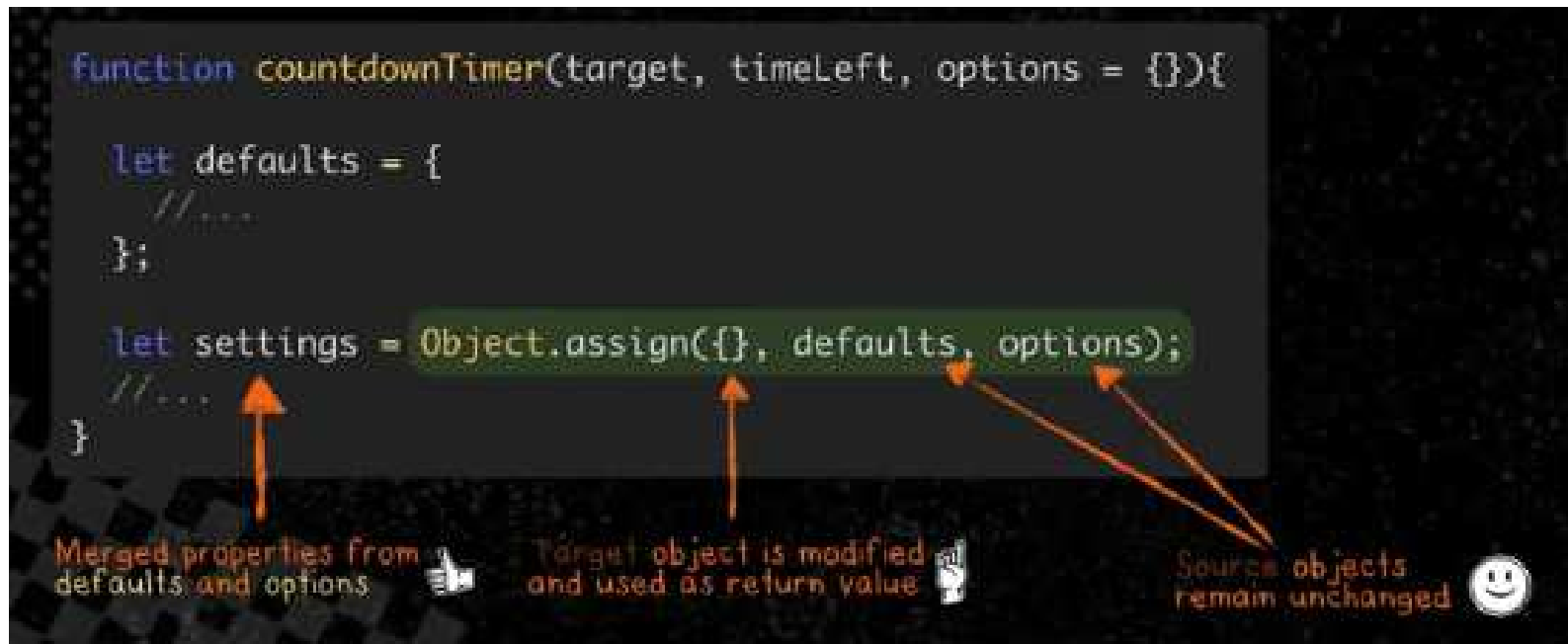
# Novedades ES6 - objetos

- **Object.assign**



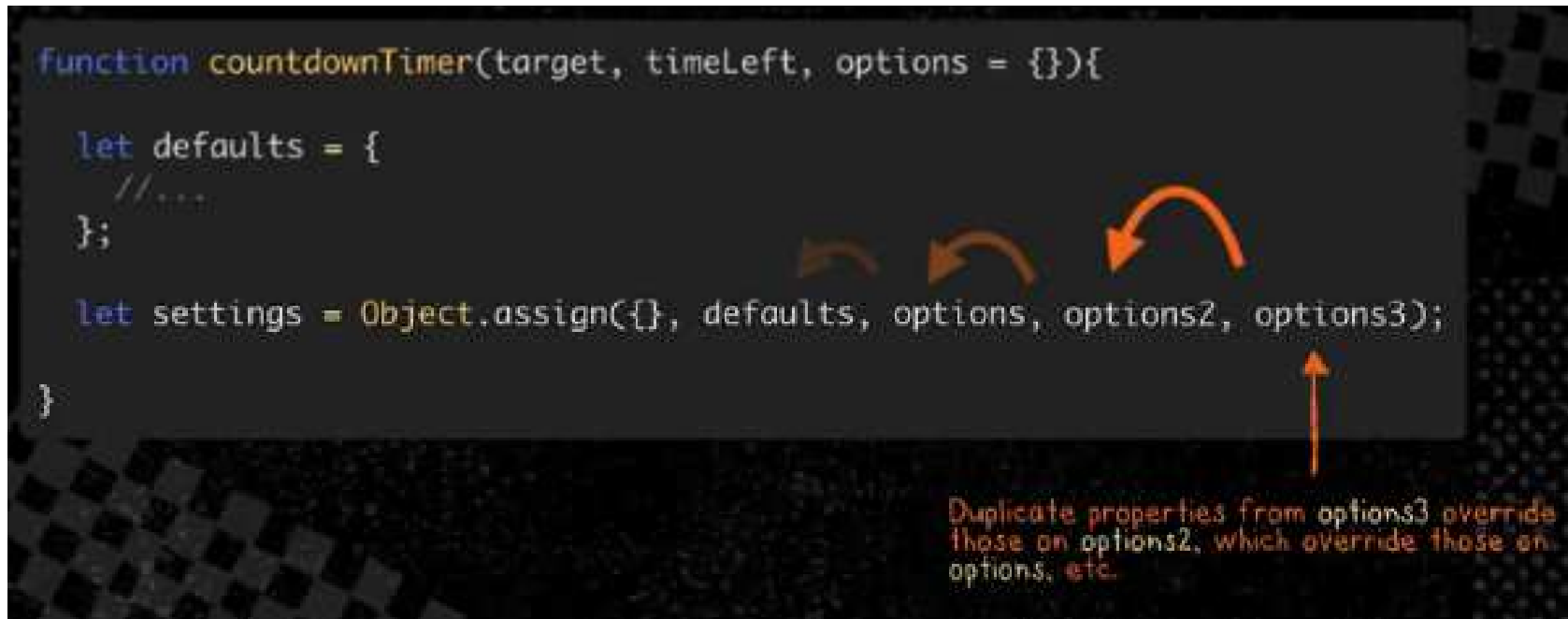
# Novedades ES6 - objetos

- **Object.assign**



# Novedades ES6 - objetos

- **Object.assign**



# Novedades ES6 - objetos

- **Object.assign**

The diagram illustrates three different ways to use `Object.assign` with annotations:

- Top Example (Red X):** `Object.assign(defaults, options);`  
Annotation: "defaults is mutated, so we can't go back and access original default values after the merge"
- Middle Example (Green Checkmark):** `let settings = Object.assign({}, defaults, options);`  
Annotation: "Can access original default values and looks functional" (with a thumbs up icon)
- Bottom Example (Red X):** `let settings = {};  
Object.assign(settings, defaults, options);  
console.log( settings.user );`  
Annotations: "Default values are not changed, but settings is passed as a reference" and "Not reading return value" (with an arrow pointing to the `console.log` statement)

# Novedades ES6 - objetos

- **Object.assign**

```
function countdownTimer(target, timeLeft, options = {}){  
  let defaults = {  
    // ...  
  };  
  
  let settings = Object.assign({}, defaults, options);  
  
  if(settings.timeUnit !== defaults.timeUnit){  
    _conversionFunction(timeLeft, settings.timeUnit)  
  }  
}
```

Runs when value passed as argument for  
timeUnit is different than the original value

# Novedades ES6 - objetos

- **Object.assign**

```
countdownTimer($('btn-undo'), 60, { container: '.new-post-options' });  
  
...and using the default value for everything else.  
  
function countdownTimer(target, timeLeft, options = {}){  
  let defaults = {  
    container: ".timer-display",  
    timeUnit: "seconds",  
    //...  
  };  
  
  let settings = Object.assign({}, defaults, options);  
  
  console.log( settings.container );  
  console.log( settings.timeUnit );  
}
```

> .new-post-options  
> seconds

# Novedades ES6 - objetos

- **Object.assign**

- ¿Cuál será el resultado de settings ?

```
let defaults = {  
  container: ".main",  
  isActiveClass: ".is-active"  
};  
  
let options1 = {  
  container: ".main-container",  
  isActiveClass: ".is-active-element"  
};  
  
let options2 = {  
  isActiveClass: ".is-active-content"  
};
```

```
let settings = Object.assign({}, defaults, options1, options2);
```

- ☐ { container: '.main-container', isActiveClass: '.is-active-content' }
- ☐ { container: ".main", isActiveClass: ".is-active-content" }
- ☐ { container: ".main-container", isActiveClass: ".is-active-element" }

# Novedades ES6 - objetos

- **Object.assign**
  - Rellena los huecos que faltan de forma que ningún objeto sea modificado

```
function spinner(target, options = {}){  
  
  let defaults = {  
    message: "Please wait",  
    spinningSpeed: 5,  
    cssClass: ".is-spinning"  
  };  
  
  let settings = Object.assign(  
    ,  
    ,  
      
  );  
  
  if(settings.spinningSpeed !== defaults.spinningSpeed){  
    settings.cssClass = _addSpeedClass(target);  
  }  
  
  _renderSpinner(target, settings);  
}
```

```
function spinner(target, options = {}){  
  
  let defaults = {  
    message: "Please wait",  
    spinningSpeed: 5,  
    cssClass: ".is-spinning"  
  };  
  
  let settings = Object.assign(  
    {},  
    defaults,  
    options  
  );  
  
  if(settings.spinningSpeed !== defaults.spinningSpeed){  
    settings.cssClass = _addSpeedClass(target);  
  }  
  
  _renderSpinner(target, settings);  
}
```



# Novedades ES6 - objetos

- **Object.assign**
  - ¿Qué resultado se mostrará en la consola?

```
function spinner(target, options = {}) {  
  let defaults = {  
    message: "Please wait",  
    spinningSpeed: 5,  
    cssClass: ".is-spinning"  
  };  
  
  let settings = Object.assign({}, defaults, options);  
  
  console.log(`Message: ${settings.message}`);  
  console.log(`spinningSpeed: ${settings.spinningSpeed}`);  
  console.log(`cssClass: ${settings.cssClass}`);  
}
```

```
spinner(targetElement, {  
  cssClass: ".is-fast-spinning",  
  spinningSpeed: 8  
});
```

☐ Message: Please wait  
spinningSpeed: 5  
cssClass: .is-fast-spinning

☐ Message: Please wait  
spinningSpeed: 8  
cssClass: .is-fast-spinning

☐ Message: Please wait  
spinningSpeed: 8  
cssClass: .is-spinning

# Novedades ES8 - objetos

- **Object.entries** y **Object.values**
- Se añaden estos métodos para iterar sobre las propiedades enumerables propias del objeto. El resultado sigue la misma ordenación que `for in`.

```
const data = { a: 'hola', b: 1 };  
Object.values(data); // ['hola', 1]  
Object.entries(data); // [['a', 'hola'], ['b', 1]]
```

# Novedades ES8 - objetos

- **Object.entries** y **Object.values**
  - Recuerda que los arrays tratan sus elementos como un objeto con propiedades numéricas empezando por 0.

```
const obj = ['a', 'z', '5']; // equivale a { 0: 'a', 1: 'z', 2: '5' };  
Object.values(obj); // ['a', 'z', '5']  
Object.entries(obj); // [['0', 'a'], ['1', 'z'], ['2', '5']]
```

- For in ordena las claves numéricas en orden ascendente

```
const obj = { 2: 'a', 1: 'b', 3: 'c' };  
Object.values(obj); // ['b', 'a', 'c']  
Object.entries(obj); // [['1', 'b'], ['2', 'a'], ['3', 'c']]
```

# Novedades ES8 - objetos

- **Object.getOwnPropertyDescriptors**
  - Devuelve todos los descriptores de propiedades del objeto (no heredadas de su prototype).
  - Nace con la voluntad de facilitar el clonado entre 2 objetos, que hasta ahora se hacía a mano.

```
const shallowClone = (object) => Object.create(  
  Object.getPrototypeOf(object),  
  Object.getOwnPropertyDescriptors(object)  
);  
  
const shallowMerge = (target, source) => Object.defineProperties(  
  target,  
  Object.getOwnPropertyDescriptors(source)  
);
```