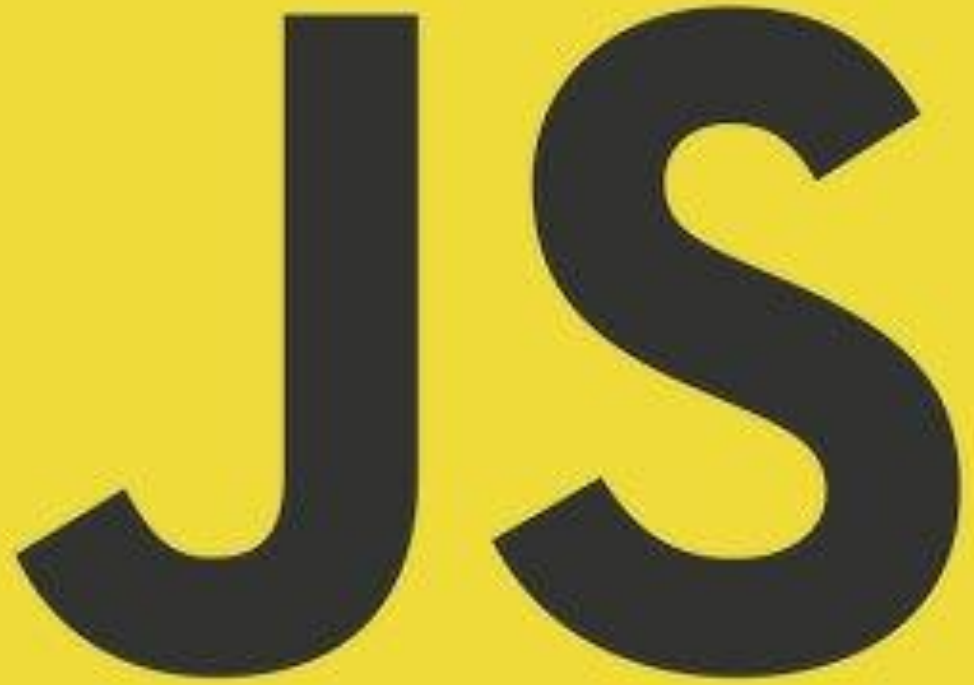
UT 4_2 - Programación con colecciones, funciones y objetos definidos por el usuario: Objetos



Marina Navarro Pina

Colecciones

Las colecciones en JavaScript permiten almacenar y manipular grupos de datos.

Arrays, Maps y *Sets* son las principales colecciones utilizadas en JavaScript.

Map

Colección de parejas de [clave,valor]. Permite que la clave sea **cualquier cosa** (array, objeto, ...) . Se puede acceder a un valor con **.clave** o **[clave]**.

```
const persona = {
  nombre: 'John',
  apellido: 'Doe',
  edad: 39
}
console.log(persona.nombre)      // John
console.log(persona['nombre'])   // John
```

Ejemplo 1

Set

Es como un *Map* pero que no almacena los valores sino sólo la **clave**.

Podemos verlo como una colección que **no permite duplicados**.

Ejemplo 1

Objetos

JavaScript siempre ha soportado objetos, aunque no de la manera tradicional de las **clases** como en otros lenguajes orientados a objetos.

Prácticamente todo lo que utilizamos en Javascript, son objetos. Son como una variable especial que puede contener más variables de la misma temática en su interior.

Un objeto en JavaScript es *similar a un array*, pero en lugar de estar indexado por números, está **indexado por nombres** (similar a un array asociativo en PHP o un diccionario en Python).

Permiten almacenar colecciones de datos y funcionalidades relacionadas.

Objetos - Creación

Se recomienda crearlos usando **notación literal** (forma abreviada para crear objetos):

```
const alumno = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
};
```

aunque también podemos crearlos utilizando la palabra clave **new** (NO recomendado):

```
const alumno = new Object()
alumno.nombre = 'Carlos'     // se crea la propiedad 'nombre' y se le asigna un valor
alumno['apellidos'] = 'Pérez Ortiz'    // se crea la propiedad 'apellidos'
alumno.edad = 19
```

Objetos - Propiedades

Podemos añadir propiedades al objeto después de haberlo creado, y no sólo en el momento de crear el objeto.

Para acceder a las propiedades tenemos dos formas diferentes: a través de la notación con ***puntos*** *(más utilizada) o a través de la notación con* ***corchetes****.*

```
// FORMA 1: A través de notación con puntos
const player = {};

player.name = "Manz";
player.life = 99;
player.power = 10;

// FORMA 2: A través de notación con corchetes
const player = {};

player["name"] = "Manz";
player["life"] = 99;
player["power"] = 10;
```

```
console.log(alumno.nombre)       // imprime 'Carlos'
console.log(alumno['nombre'])    // imprime 'Carlos'
let prop = 'nombre'
console.log(alumno[prop])        // imprime 'Carlos'
```

Objetos - Propiedades

Si intentamos acceder a una propiedad que no existe, se devuelve *undefined:*

```
console.log(alumno.ciclo)      // muestra undefined
```

Se genera un error si intentamos acceder a propiedades de algo que no es un objeto:

```
console.log(alumno.ciclo)          // muestra undefined
console.log(alumno.ciclo.descrip)     // se genera un ERROR
```

Con ES2020 (ES11) se incluyó el operador de encadenamiento (?.) que permite acceder a propiedades de un objeto de manera segura, evitando errores si el objeto es null o undefined:

```
console.log(alumno.ciclo?.descrip)
// si alumno.ciclo es un objeto muestra el valor de
// alumno.ciclo.descrip y si no muestra undefined
```

Objetos - Métodos

Si dentro de una variable del objeto metemos una función (o una variable que contiene una función), tendríamos lo que se denomina un método de un objeto:

```
const user = {
  name: "Manz",
  talk: function() { return "Hola"; }
};

user.name;       // Es una variable (propiedad), devuelve "Manz"
user.talk();     // Es una función (método), se ejecuta y devuelve "Hola"
```

Ejemplo 2

Objetos - Método .toString()

Simplemente por generar una variable de tipo Object , esa variable «hereda» una serie de métodos que existen en cualquier variable que sea de tipo Object.

El método .toString() es un método que intenta representar la información de ese objeto en un String.

Ejemplo 2

```
const objeto = {};
objeto.toString();    // Devuelve "[object Object]"
                      // (representación textual de un objeto genérico)
```

Desestructuración de Objetos

Utilizando la desestructuración de objetos podemos separar en variables las propiedades que teníamos en el objeto.

Ejemplo 2

```
const user = {
  name: "Manz",
  role: "streamer",
  life: 99
}

const { name, role, life } = user;

console.log(name);
console.log(role, life);
```

```
const personaCarlos = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
};

function muestraNombre({nombre, apellidos}) {
    console.log('El nombre es ' + nombre + ' ' + apellidos)
}

muestraNombre(personaCarlos)
```

Desestructuración - Propiedades

Se pueden renombrar propiedades y establecer valores por defecto:

```
const { name, role: type, life } = user;

console.log({ name, type, life });
```

```
const { name, role = "normal user", life = 100 } = user;

console.log({ name, role, life });
```

Desestructuración - Reestructuración

Para reutilizar objetos y recrear nuevos objetos a partir de otros.

Ejemplo 2

```
const personaCarlos = {
    nombre: 'Carlos',
    apellidos: 'Pérez Ortiz',
    edad: 19,
};
const alumnoCarlos = {
    ...personaCarlos,
    ciclo: 'DAW',
    curso: 2,
};
```

Desestructuración - Copia de objetos

Cuando se realiza la copia de objetos, los valores **primitivos** (números, strings, booleanos...), son **pasados por valor**; pero los valores **no primitivos** (objetos, arrays, etc...) se pasan **por referencia**.

Ejemplo 2 -- structuredClone()

```
const user = {
  name: "Manz",
  role: "streamer",
  life: 99,
  features: ["learn", "code", "paint"]
}

const fullUser = {
  ...user,
  power: 25,
  life: 50
}
```

```
console.log(user.features);       // ["learn", "code", "paint"]
console.log(fullUser.features);   // ["learn", "code", "paint"]

fullUser.features[0] = "program";

console.log(fullUser.features);   // ["program", "code", "paint"]
console.log(user.features);       // ["program", "code", "paint"]
```

Objetos - Más información

https://www.w3schools.com/js/js_objects.asp

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Object

Interesante leer:
https://lenguajejs.com/javascript/objetos/iteradores/
https://lenguajejs.com/javascript/objetos/groupby/

Buenas prácticas

‘use strict’: no permite

- Usar una variable sin declarar
- Definir más de 1 vez una propiedad de un objeto
- Duplicar un parámetro en una función
- Usar números en octal
- Modificar una propiedad de sólo lectura

Buenas prácticas

Variables

- Elegir un buen nombre es fundamental. Evitar abreviaturas o nombres sin significado (a, b, c, …)
- Evitar en lo posible variables globales
- Usar let para declararlas
- Usar const siempre que una variable no deba cambiar su valor
- Declarar todas las variables al principio
- Inicializar las variables al declararlas
- Evitar conversiones de tipo automáticas
- Usar para nombrarlas la notación camelCase

Buenas prácticas

Otras

- Debemos ser coherentes a la hora de escribir código: Existen muchas guías de estilo: Airbnb, Google, Idiomatic, etc. Para obligarnos a seguir las reglas podemos usar alguna herramienta *linter*.
- También es conveniente para mejorar la legibilidad de nuestro código separar las líneas de más de 80 caracteres.
- Usar === en las comparaciones.
- Si un parámetro puede faltar al llamar a una función darle un valor por defecto.
- Y para acabar comentar el código cuando sea necesario, pero mejor que sea lo suficientemente claro como para no necesitar comentarios.