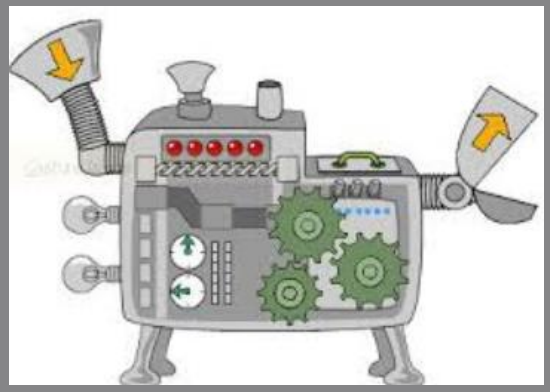


UD2: Objetos predefinidos y estructuras definidas por el usuario

Parte 2: Clases y objetos

María Rodríguez Fernández mariarfer@educastur.org

Al final de este documento...

- Sabrás cómo se **declaran** las clases en JS, con sus propiedades y métodos
- Habrás recordado el concepto de **herencia** entre clases y sabrás cómo se hace en JS
- Podrás modularizar tus aplicaciones

Objetos, clases y JavaScript

- JS no permitía crear clases hasta su versión ECMAScript 2015, donde se empieza a usar la palabra reservada *class*
  - *Antes se usaba function para crear Prototypes*

```
class NombreClase {
  constructor(parametro1 [,parametro 2...]) {
    this.propiedad1 = parametro1;
    [this.propiedad2 = parametro2;]
  }
}
```

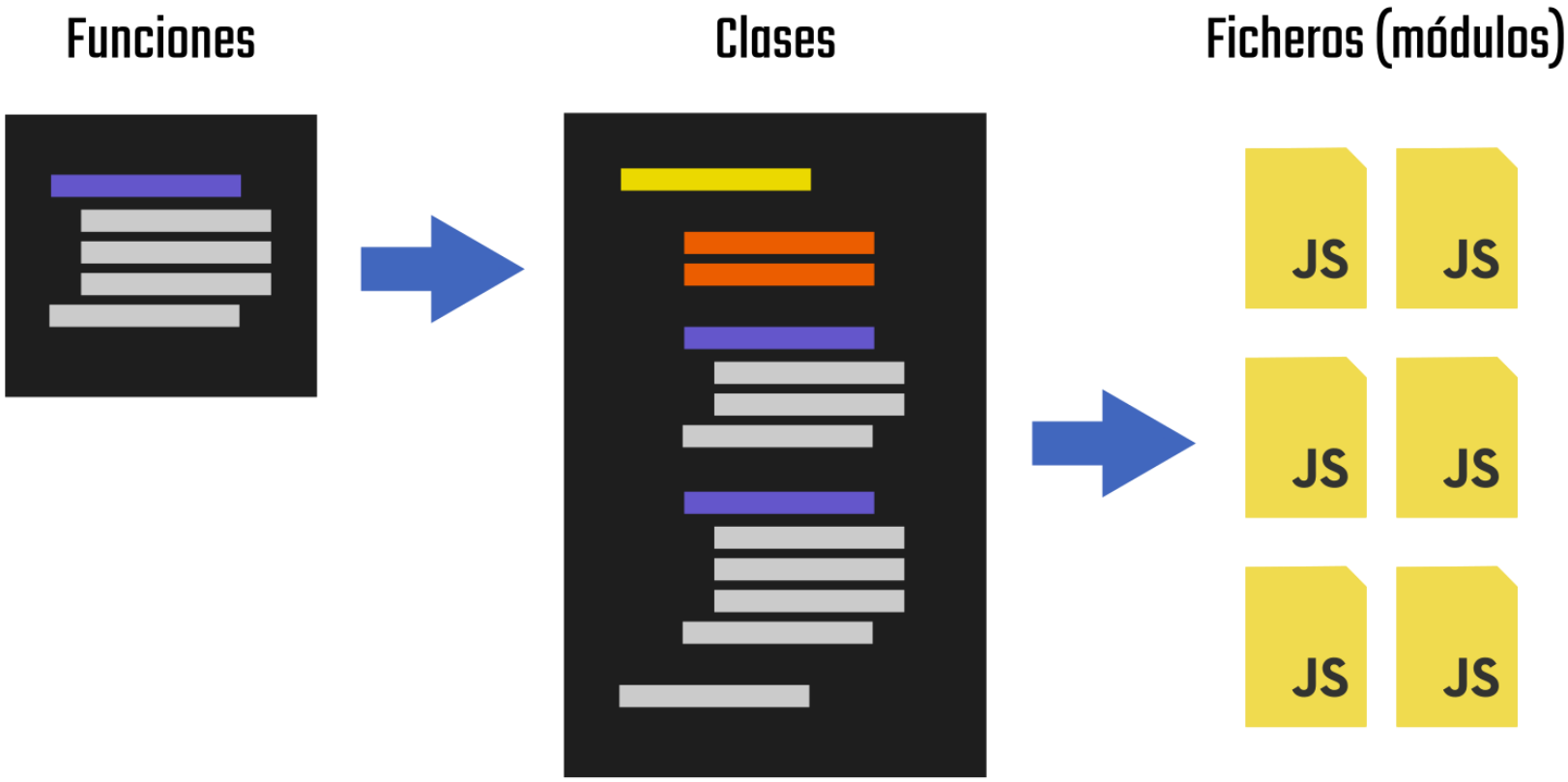
DECLARACIÓN

```
let nombreObjeto = new NombreClase (argumentos);
```

INSTANCIA

Primero es necesario declarar la clase y luego acceder a ella para evitar los errores de referencia.

Organización



Clases: Setters y getters

- JS no permitía definir propiedades o métodos privados hasta la versión ES2020
  - El carácter # indica que la propiedad/método es privada/o
    - #propiedad
  - Hasta entonces, como convención, el nombre de los elementos privados de la clase comenzaba por "_"
    - _propiedad
- También por convención, el nombre de los *setters* y *getters* es el mismo que la propiedad, pero **en mayúscula la primera letra**
  - Si pusiésemos el mismo nombre al método que a la propiedad se entraría en un bucle

Clases: Ejemplo

```
class Personaje {
   name;
   #type = "Player";
   lifes = 5;
   energy = 10;

set Type(type) {
   this.#type=type;
   }

get Status() {
   return '★'.repeat(this.energy);
 }

}
const mario = new Personaje("Mario");
mario.#type    // Error de sintaxis (privada)
mario.Type = "Admin";  //Modifica el valor de #type
mario.energy    // 10
mario.Status    // '★★★★★★★★★★'
```

Equivalente

```
constructor() {
   this.name;
   this.#type = "Player";
   this.lifes = 5;
   this.energy = 10;
 }
```

Utilidad

- Para generar los getters y setters de manera más rápida puedes instalar la extensión

JavaScript (ES6) code snippets v1.8.0

charalampos karypidis | 6.820.638 | ★★★★★ (31)

Code snippets for JavaScript in ES6 syntax

Instalar

Trigger	Content
con→	adds default constructor in the class `constructor() {}`
met→	creates a method inside a class `add() {}`
pge→	creates a getter property `get propertyName() {return value;}`
pse→	creates a setter property `set propertyName(value) {}`

EJERCICIO PROPUESTO I: Clase Miembro

- Desarrolla un script con la clase **Miembro** con los siguientes elementos:
  - Atributos con sus setters y getters
    - Nombre
    - Apellidos
  - Constructor
    - Con nombre y apellidos – Si no se le pasan por defecto "Sin nombre" y "Sin apellidos".
  - Las siguientes funciones
    - **comer**, que muestre el mensaje "Estoy comiendo"
    - **cenar**, que muestre el mensaje "Estoy cenando"
- Crea un objeto de tipo Miembro, muestra su nombre y apellidos y llama a su método comer

Métodos estáticos

- Este tipo de métodos se utilizan sobre todo para crear funciones de utilidad en una aplicación
  - Ejemplo: `Math.random()`
- Al igual que ocurre en otros lenguajes de programación, **un método estático se llama directamente sin instanciar la clase**
- Se define anteponiendo la palabra `static`:

```
static nombreMetodo (parametros) { //código }
```

Herencia

- Podemos hacer que una clase (hija) herede la estructura y el comportamiento de otra clase (padre).
  - `class ClaseHijo extends ClasePadre`
- Para hacer referencia a atributos del padre:
  - super(atributo)
- Llamadas desde la clase hija a métodos de la clase padre:
  - `super.metodo()`

Ejemplo de Herencia

```
class Padre {
   soloPadre() { console.log("Tarea en el padre..."); }
   padreHijo() { console.log("Tarea en el padre..."); }
   sobreHijo() { console.log("Tarea en el padre..."); }
 }

 class Hijo extends Padre {
   padreHijo() {
     super.padreHijo();
     console.log("Tarea en el hijo...");
   }

   soloHijo() { console.log("Tarea en el hijo..."); }
   sobreHijo() { console.log("Tarea en el hijo..."); }
 }
```

Ejecución del ejemplo

Método	Clase Padre	Clase Hija	¿Se ejecuta el método en una instancia de la clase hija?
soloPadre()	☑	✗	Se ejecuta porque **se hereda** el método del padre hacia el hijo.
soloHijo()	✗	☑	Se ejecuta porque simplemente existe en el hijo.
padreHijo()	☑	☑	Se ejecutan ambos porque super llama al padre primero.
sobreHijo()	☑	☑	Se ejecuta sólo el hijo, porque sobreescribe el heredado del padre.

Modularización

- Una **buena práctica** es la **modularización**
  - Dividir programas en pequeños **módulos independientes** que pueden ser importados
- En un principio, y de forma nativa, la forma más extendida era incluir varias etiquetas <script> desde nuestra página HTML
  - Varios ficheros JS separados, cada uno para una finalidad concreta.
    - Poco modular
    - Lento, sobrecarga al cliente con múltiples peticiones extra

EJERCICIO PROPUESTO I (Parte 2): Clases Padre e Hijo

- Crea una clase **Padre** y una clase **Hijo** que hereden de **Miembro** – usa archivos separados
  - **Hijo** tendrá además de nombre y apellido (por ser un Miembro de la familia), una **moto**. Si no se le pasa la moto en el constructor se le asignará "Sin moto".
  - **Padre** tendrá además de nombre y apellido (por ser un Miembro de la familia), un **coche**.
  - **Padre** sobrecargará los métodos **comer** y **cenar**, pero como buen padre comerá huevos, su mensaje será "estoy comiendo huevos" y "estoy cenando huevos"
- Crea un padre y un hijo y prueba a mostrar su coche/moto y sus métodos comer y cenar

RETO EXTRA:
Para saber más

- Crea una clase **Familia** que tenga tres atributos:
  - **domicilio**
  - **renta**. Por defecto será 0
  - **miembros**. Array que contendrá los miembros de la familia
- Crea un objeto de la clase **Familia**
  - Añade los miembros creados en el punto anterior a la misma.
  - Recorre todos los miembros y llama a comer mediante un bucle.

Módulos

- Tener en cuenta:
  - Los módulos utilizan automáticamente modo `strict mode`
  - Los módulos se ejecutan una única vez, aunque se haga referencia a ellos en varias etiquetas <script>.
  - El fichero se carga en diferido (como si tuviera la palabra `defer`)
  - Las características de un módulo no están disponibles a nivel global: solamente se puede acceder a las funciones importadas en el script en el que se importan

Para utilizar módulos en JS es necesario ejecutar los ficheros desde un servidor; si lo haces localmente obtendrás un error de CORS (Cross-Origin Request Blocked) debido a requisitos de seguridad de JS

Los módulos no son sólo para clases

- Puedo exportar funciones, constantes…

```
export function add(x,y){
    return x+y
}
export const puntos=[10, 20, 30]
export default puntos
```

En la etiqueta script del html indicamos que vamos a usar módulos con el atributo **type="module"**

- Puedo importar uno a uno – desestructurando con {}, o en general (necesitamos un **export default**)

```
import {add, puntos} from './add.js'
import porDefecto from "./add.js"

console.log(add(10,20)); //30
console.log(puntos); //[10, 20, 30]
console.log (porDefecto); //[10, 20, 30]
```

EJERCICIO PROPUESTO I (Parte 3): Uso de módulos

- Modifica la organización del ejercicio:
  - En el **html** enlaza sólo al script con el código del programa principal indicando que es de tipo módulo
  - En cada **js** importa las clases que necesites con la sentencia `import`
  - Exporta las clases para que puedan ser usadas con la sentencia `export default`

¿Cómo ha ido?