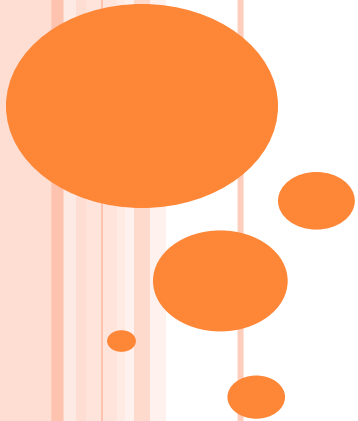


PROGRAMACIÓN CON JAVASCRIPT

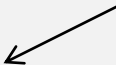


EVENTOS

➤ **Sucesos que pueden tener lugar sobre la interfaz de usuario (etiquetas HTML) y a los que se les puede asociar una función de respuesta.**

➤ **Se puede asociar desde la propia etiqueta:**

```
<input type="button" onclick="respuesta();" value="Pulsar">  
:  
<script>  
function respuesta(){  
    //codigo de respuesta al evento click  
}  
</script>
```



➤ **O directamente desde código:**

```
<input type="button" id="mybutton" value="Pulsar">  
:  
<script>  
let bt= document.querySelector("button");  
bt.addEventListener("click",()=>{  
    //código de respuesta al evento click  
});
```



EVENT BUBBLING

➤ Los eventos que se producen en objetos hijos son propagados hacia los objetos padre que los contienen

➤ Se puede llegar a detener mediante `stopPropagation()` del objeto `Event`:

```
<div>
  <p>Párrafo 1</p>
  <p>Párrafo 2</p>
</div>
:
let div = document.querySelector("div");
div.addEventListener("click", ()=>{
  console.log("click en div");
});
let ps = document.querySelectorAll("p");
ps.forEach((p)=>{
  p.addEventListener("click", (e)=>{
    console.log("click en p "+p.textContent);
  });
});
//click en p párrafo1
//click en div
```

```
<div>
  <p>Párrafo 1</p>
  <p>Párrafo 2</p>
</div>
:
let div = document.querySelector("div");
div.addEventListener("click", ()=>{
  console.log("click en div");
});
let ps = document.querySelectorAll("p");
ps.forEach((p)=>{
  p.addEventListener("click", (e)=>{
    console.log("click en p "+p.textContent);
    e.stopPropagation();
  });
});
//click en p párrafo1
```



EVENT DELEGATION

- **Consiste en delegar el manejo de eventos al elemento padre en lugar de cada hijo.**
- **Si se añade dinámicamente un hijo, también se capturará el evento sobre él de manera automática:**

```
<div>
  <p>Párrafo 1</p>
  <p>Párrafo 2</p>
</div>
:
let div = document.querySelector("div");
div.addEventListener("click", (e) => {
  if (e.target.tagName == "P") {
    console.log("click en p " + e.target.textContent);
  }
});

//el padre captura los eventos sobre los hijos
```



FUNCIONES AUTOINVOCADAS

➤ **Funciones que se ejecutan de forma inmediata**

➤ **No tienen nombre, se definen entre paréntesis y se invocan indicando la lista de argumentos a continuación:**

```
(function(){  
    //codigo función  
})();
```

➤ **Ejemplos:**

```
(function(){  
    console.log("autoinvocada!!!!");  
})();
```

```
(({})=>{  
    console.log("otra autoinvocada!!!!");  
})();
```

```
((n)=>{  
    console.log("con parámetro "+n);  
})("test");
```



FUNCIONES ANÓNIMAS

➤ Funciones que no tienen nombre

➤ Se les puede asignar a una variable e invocarlas a través de la variable:

```
var anonima=function(){  
    console.log("función anónima!!!!");  
};  
anonima();
```

➤ Se pueden enviar como parámetros a funciones que requieren otra función

```
setTimeout((n)=>{  
    console.log("temporizador");  
});
```



CLOSURES

- **Funciones que permiten mantener el estado de variables internas, una vez que se han ejecutado**
- **Devuelven una función interna, desde la que se puede acceder al estado:**

```
function contador(){  
  let count = 0;  
  return function (){  
    count++;  
    return count;  
  }  
};  
let c = contador();  
console.log(c());//1  
console.log(c());//2
```

Mismo ejemplo
con
autoinvocadas



```
let c=(function contador(){  
  let count = 0;  
  return function (){  
    count++;  
    return count;  
  }  
})();  
console.log(c());//1  
console.log(c());//2
```

CALLBACKS

- Son funciones que se pasan como parámetro a otras funciones.
- Un callback se puede ejecutar síncrona o asíncronamente
- Pueden acceder a la variables locales de la función a la que se pasa como parámetro

```
function process(data,funcallback){  
    console.log("se recibe "+data);  
    funcallback();  
};  
process("hello",()=>console.log("callback"));
```



PROMESAS

- Una promesa es un objeto que proporciona un resultado que puede ser consumido en otra parte del código.
- Realiza una llamada a la parte del código que va a consumir el resultado, tanto en una situación normal como en una situación de error.
- Utilizado en operaciones asíncronas, como peticiones a un API:

```
const myPromise=new Promise(resolve,reject)=>{  
  let x = Math.floor(Math.random()*10);  
  if(x>5){  
    resolve("superado");  
  }else{  
    reject("suspenso");  
  }  
});
```



CONSUMIR PROMESAS

➤ Para utilizar una promesa desde otra parte de la aplicación, se llaman a los métodos *when()* y *catch()* del objeto, con las funciones a ejecutar en cada situación:

```
myPromise
  .when((data)=>console.log(`Enhorabuena, ${data}`))
  .catch((err)=>console.log(`Lo siento, ${err}`));
```

Antonio Marín



PASO DE PARÁMETROS A PROMESAS

- Una promesa puede recibir parámetros para ser procesados de cara a enviar la posible respuesta a la aplicación consumidora.
- Para ello, la promesa debe ser encerrada en otra función que es la que recibe los parámetros:

```
function myPromise(data) {  
  return new Promise((resolve, reject) => {  
    if (data) {  
      resolve(`Hola, ${data} `);  
    } else {  
      reject("No se ha recibido el dato");  
    }  
  });  
}
```

```
myPromise("profe")  
  .then((response) => console.log(response))  
  .catch((error) => console.log(error));
```



UTILIZACIÓN DE ASYNC

- La utilización de *async* permite definir una promesa de forma más simple.
- Al definir una función como *async*, el resultado devuelto por está estará encapsulado en una promesa:

```
async function myPromise(data) {  
  if (data) {  
    return `Hola, ${data} `;  
  } else {  
    return "No se ha recibido el dato";  
  }  
}
```

```
myPromise("profe")  
  .then((response) => console.log(response))  
  .catch((error) => console.log(error));
```



ESPERAS MEDIANTE AWAIT

- La palabra reservada *await* se utiliza para esperar a la terminación de una promesa dentro de un programa.
- Al utilizar la expresión *await* promesa, el código queda bloqueado a la espera de que la promesa finalice:

```
function espera() {  
    return new Promise((resolve)=>setTimeout((resolve), 3000));  
}  
async function myPromise(data) {  
    await espera(); //se bloquea 3 segundos hasta que finalice la  
                  //otra promesa  
    if (data) {  
        return `Hola, ${data} `;  
    } else {  
        return "No se proporcionó un nombre";  
    }  
}
```



FUNCIÓN FETCH

- Alternativa a XMLHttpRequest para lanzar peticiones HTTP a recursos externos.
- Recibe como parámetro la URL y un objeto JavaScript con los datos de conexión y devuelve una promesa:

Petición GET simple

```
fetch("url")  
  .then((response)=>response.json())  
  .then((data)=>{  
    :  
  });
```

Petición POST

```
fetch("url", {  
  method: "POST",  
  headers: {  
    "Content-Type": "application/json"  
  },  
  body: JSON.stringify({ // Convertir objeto a JSON  
    name: "Prueba",  
    age: 45,  
    email: "prueb@gmail.com"  
  })  
})  
  .then(response => response.json())  
  .then(data => {...});
```

CLASES

- Para definir una clase utilizamos la palabra class.
- La clase incluye constructor y métodos:

Definición clase Calculadora

```
class Calculadora{  
  constructor(num1,num2){  
    this.num1=num1;  
    this.num2=num2;  
  }  
  sumar(){  
    return num1+num2;  
  }  
  multiplicar(){  
    return num1*num2;  
  }  
  static factorial(n){  
    let r=1;  
    for(var i=2;i<=n;i++){  
      r*=i;  
    }  
    return r;  
  }  
}
```

Utilización de la clase

```
let calc=new Calculadora(3,9);  
console.log(calc.sumar());  
console.log(calc.multiplicar());  
console.log(calc.factorial(5));
```



AÑADIR MÉTODOS DINÁMICAMENTE

➤ Una vez creada la clase, se pueden añadir nuevos métodos a posteriori dinámicamente:

```
class Calculadora{
  constructor(num1,num2){
    this.num1=num1;
    this.num2=num2;
  }
  sumar(){
    return num1+num2;
  }
  multiplicar(){
    return num1*num2;
  }
  static factorial(n){
    let r=1;
    for(var i=2;i<=n;i++){
      r*=i;
    }
    return r;
  }
}
```

```
let calc=new Calculadora(3,9);
calc.dividir= function(){
  return this.num1/this.num2;
};
console.log(calc.sumar()); //12
console.log(calc.multiplicar()); //27
console.log(calc.factorial(5)); //120
console.log(calc.dividir()); //0.3333
```



HERENCIA

- En JavaScript una clase puede heredar otra ya existente.
- La nueva clase adquiere los métodos de la existente y puede definir métodos propios:

Definición de clases

```
class Cuenta{
  constructor(codigo,saldo){
    this.codigo=codigo;
    this.saldo=saldo;
  }
  extraer(cantidad){
    this.saldo-=cantidad;
    return this.saldo;
  }
}
class CuentaMovimientos extends Cuenta{
  constructor(codigo,saldo){
    super(codigo,saldo);
    this.movs=[];
  }
  movimientos(){
    return this.movs;
  }
}
```

Utilización de la clase

```
let cuentaM=new CuentaMovimientos(1111,500);
console.log("Saldo: "+cuentaM.extraer(30));
cuentaM.movimientos().forEach(m=>console.log(m));
```



SOBRESCRITURA

➤ Una clase puede redefinir métodos heredados:

Definición de clases

```
class Cuenta{
  constructor(codigo,saldo){
    this.codigo=codigo;
    this.saldo=saldo;
  }
  extraer(cantidad){
    this.saldo-=cantidad;
    return this.saldo;
  }
}
class CuentaMovimientos extends Cuenta{
  constructor(codigo,saldo){
    super(codigo,saldo);
    this.movs=[];
  }
  movimientos(){
    return this.movs;
  }
  //sobrescritura de extraer()
  extraer(cantidad){
    this.movs.push({"tipo":"extraer","cant":cantidad});
    return super.extraer(cantidad);
  }
}
```

Utilización de la clase

```
let cuentaM=new CuentaMovimientos(1111,500);
//llama a la nueva versión del método
console.log("Saldo: "+cuentaM.extraer(30));
cuentaM.movimientos().forEach(m=>console.log(m));
```

Llamada al extraer de la
superclase



SOBRECARGA

➤ **En JavaScript no existe la sobrecarga de métodos y constructores**

➤ **Puede ser simulada utilizando número variable de argumentos en la llamada:**

Utilización de la clase

```
class Prueba{  
  metodo1(a,b,c){  
    if(a){  
      console.log("a vale "+a);  
    }  
    if(b){  
      console.log("b vale "+b);  
    }  
    if(c){  
      console.log("c vale "+c);  
    }  
  }  
}
```

```
let p = new Prueba();  
p.metodo1(1,2,3);  
p.metodo1(4,5);  
p.metodo1(6);  
/*  
a vale 1  
b vale 2  
c vale 3  
a vale 4  
b vale 5  
a vale 6  
*/
```



PROTOTIPOS

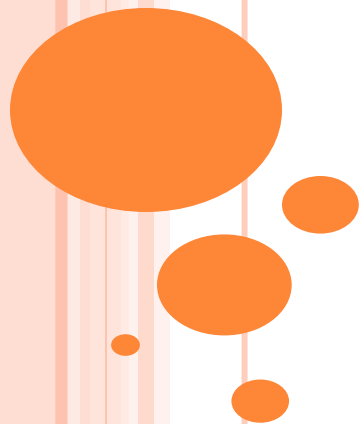
➤ **Es un mecanismo, anterior a la existencia de las clases, que permite asignar métodos y propiedades a un tipo de objeto.**

➤ **Todos los objetos de ese tipo adquieren los métodos y propiedades.**

```
function Empleado(nombre){
    this.nombre=nombre;
}
let e1=new Empleado("Ana");
//añade propiedades y métodos al prototipo
Empleado.prototype.salario=0;
Empleado.prototype.calcularSalario=function(){
    return this.salario+1000;
};
let e2=new Empleado("Juan");
e2.salario=1000;
//nuevos objetos y antiguos adquieren métodos y propiedades del
//prototipo
console.log(e2.calcularSalario()); //2000
console.log(e1.calcularSalario()); //1000
```



JQUERY



FUNDAMENTOS

- **Framework para JavaScript que proporciona funciones para la realización de tareas habituales, simplificando el código de las aplicaciones.**
- **Para utilizarlo en una página, se debe incluir la siguiente referencia al script:**

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
```



SELECTORES

➤ **Funciones especiales que permiten obtener una referencia a elementos HTML de la página.**

➤ **Entre los más importantes:**

- **\$(#id).** Obtiene una referencia al elemento por su id
- **\$(tipo).** Obtiene una colección de elementos de un tipo
- **\$(.class).** Referencia a los elementos que pertenezcan a una clase de estilo
- **\$(tipo:first).** Referencia al primer elemento de un tipo
- **\$("*").** Referencia a todos los elementos



CONTENIDO DE UN ELEMENTO

➤ Para recuperar y modificar el contenido de un elemento HTML se utilizan los siguientes métodos:

- **html().** Equivale a la propiedad innerHTML de JavaScript:

```
<div id="resultado"></div>
:
<script>
    $("#resultado").html("<b>No existe</b>");
</script>
```

- **val().** Accede al valor de un campo de formulario

```
<input type="text" id="name">
<input type="text" id="res">

:
<script>
    if($("#name").val()==""){
        $("#res").val("vacío");
    }
</script>
```



OCULTAR/MOSTRAR ELEMENTOS

➤ Para mostrar/ocultar etiquetas JQuery ofrece los siguientes métodos a aplicar sobre los objetos:

▪ **hide().** Oculta el elemento/elementos:

```
<div id="resultado">Resultado de la operación</div>
:
<script>
    $("#resultado").hide();
</script>
```

▪ **show().** Muestra el elemento/elementos:

```
<input type="text" id="name">
<input type="text" id="res">

:
<script>
    $("input").show(); //muestra todas las etiquetas
</script>
```



AÑADIR ELIMINAR ELEMENTOS

➤ Para añadir/eliminar elementos, JQuery ofrece los siguientes métodos a aplicar sobre los objetos:

▪ **append().** Añade un nuevo elemento como hijo:

```
<select id="datos"></select>
:
<script>
  const select = $('#datos');
  select.append('<option value="">Seleccione un curso</option>');

</script>
```

▪ **remove().** Elimina el elemento y sus descendientes:

```
:
<script>
  $("#datos").remove();
</script>
```



FUNCIONES AJAX

➤ **Permiten realizar peticiones HTTP a recursos remotos de forma sencilla:**

▪ **\$.ajax(). Se le proporciona un JSON con los datos de la petición y devuelve una promesa:**

```
altaAlumno(alumno) {  
  return $.ajax({  
    url: `${this.baseUrl}/alta`,  
    method: 'POST',  
    contentType: 'application/json',  
    data: JSON.stringify(alumno)  
  }).then(() => true)  
    .catch(() => false);  
}
```

▪ **\$.get(). Específicamente pensada para peticiones get:**

```
buscarAlumno(email) {  
  return $.get(`${this.baseUrl}/buscar/${email}`)  
    .then((data) => data || null)  
    .catch(() => null);  
}
```

