Unidad 3. Estructuras definidas por el usuario en JavaScript.
Boletín de actividades

Funciones.

Una función es un conjunto de sentencias que se ejecutan en bloque. Se pueden entender como un subprograma que el desarrollador puede llevar a cabo cuando lo necesite. Las funciones constituyen una parte esencial del lenguaje JavaScript y de la programación funcional asociada al lenguaje.

Este es un ejemplo de *declaración de una función* es:

```
function decirHola() {
    //Aquí empieza el cuerpo de la función.
    console.log("Hello world!");
    console.log("¡Hola mundo!");
    //Aquí acaba el cuerpo de la función.
}

decirHola();
```

La declaración de la función `decirHola` no ejecuta el cuerpo de la función. Para *llamar* (*correr*, *ejecutar*, *invocar* o *despachar*) una función debe utilizarse el nombre seguido de paréntesis:

Valores de retorno.

La palabra reservada `return` en el cuerpo de una función hará que ésta finalice inmediatamente y se devuelva el valor especificado. En JavaScript la llamada a una función trabaja como una expresión y por tanto siempre se resolverá a un determinado valor. Si no se llama específicamente a `return`, se devolverá el valor `undefined`.

```
function obtenerSaludo() {
    return "Hello world!";
}

obtenerSaludo();    //Hello world!
```

Invocar versus referenciar.

Es importante entender la diferencia entre invocar a una función o referenciarla. En JavaScript las funciones son objetos y se pueden referenciar para pasarlas a otras funciones o asignarlas a variables.

Si se indican los paréntesis, la función se está nvocando. Si no se indican los paréntesis, la función se está referenciando.

```
obtenerSaludo();    //Hello world!
obtenerSaludo;      //function obtenerSaludo()
```

Argumentos.

El método principal para pasar información a una función son los argumentos, también llamados parámetros. La siguiente función recibe 2 parámetros:

```
function avg(a, b) {
    return (a + b) / 2;
}
```

Cuando una función se invoca, los argumentos formales reciben un valor. Hay que tener en cuenta que los argumentos sólo existen dentro del cuerpo de la función incluso si hay variables fuera con el mismo nombre:

```
const a = 5, b = 10;
avg(a, b);
```

Actividad 1. Argumentos por defecto.

En ES6 es posible definir valores por defecto para los argumentos de una función. Prueba el siguiente ejemplo y observa los valores de salida en la invocación de las funciones:

```
function f(a, b = "default", c = 3) {
    return `${a} - ${b} - ${c}`;
}

f(5, 6, 7);
f(5, 6);
f(5);
f();
```

Actividad 2. Número indeterminado de argumentos.

En ES6 es posible pasar a una función un número no determinado de argumentos gracias al operador de propagación (*spread operator*): "…". Prueba el siguiente ejemplo:

```
function anadirPrefijo(prefijo, ...palabras) {
    const palabrasConPrefijo = [];
    for (let i = 0; i < palabras.length; i++) {
        palabrasConPrefijo[i] = prefijo + palabras[i];
    }
    return palabrasConPrefijo;
}

anadirPrefijo("con", "verso", "vexo", "cavo");
```

Nota que tanto el operador de propagación como los argumentos por defecto siempre deben ser los últimos, en caso contrario JavaScript no será capaz de distinguir el resto.

Funciones anónimas.

Las funciones anónimas son aquellas que no tienen identificador. Las siguientes son funciones equivalentes:

```
//Función convencional
function obtenerSaludo() {
    return "Hello world!";
}
//Función anónima
function () {
    return "Hello world!";
}
```

Su uso dependerá del contexto, si la función se va a usar más tarde se le asignará un identificador para poder referenciarla. Si la función sólo se va a pasar a otra función o se va a asignar a una variable podemos crear una función anónima.

```
//Funciones equivalentes.
function decirHola() {
    consola.log("¡Hola mundo!");
}

boton.addEventListener("click", decirHola);

boton.addEventListener("click", function () {
    console.log("¡Hola Mundo!");
});
```

Notación flecha.

ES6 ha introducido una nueva sintáxis llamada *notación flecha* que permiten crear *funciones flecha*. El término flecha hace referencia al signo "=>" que se incorpora en la notación.

Las funciones flecha son funciones anónimas con una sintaxis simplificada para reducir el número de caracteres necesarios para declarar la función. Una función flecha sigue las siguientes reglas:

- Se puede omitir la palabra reservada `function`.
- Si la función tiene un único argumento, se pueden omitir los paréntesis.
- Si el cuerpo de la función tiene una única expresión, se pueden omitir las llaves y la palabra reservada `return`.

```
//Función anónima.
const decirHola2 = function () {
    return "¡Hola mundo!";
}
//Notación flecha equivalente.
const decirHola3 = () => "!Hola Mundo!";
```

Actividad 1. Utilizando la notación flecha.

Reescribe estas funciones anónimas utilizando la notación flecha:

a) `const f1 = function() { return “¡Hola!”; }`

b) `const f2 = function(nombre) { return `Hola ${nombre}`; }`

c) `const f3 = function(a, b) { return a + b; }`

Funciones: prácticas recomendadas.

Actividad 1. Función de pares e impares.

Crea una función sencilla que devuelva la palabra par o impar dependiendo del número que reciba como argumento.

Muestra a continuación 500 números aleatorios del 1 al 10.000 indicando al lado de ellos si el número es par o impar.

Actividad 2. Función que dibuja una tabla.

Crea una función que permita dibujar una tabla HTML. La función recibirá como parámetros el número de filas y columnas de la tabla. Por defecto la función creará una tabla de 10 filas y 4 columnas.

Funciones puras.

Las funciones puras son más fáciles de testear, más fáciles de entender y más portables. Además son la base de la programación funcional. Siempre deberíamos crear este tipo de funciones.

La idea de una función ha estado presente desde los inicios de la programación como una forma de gestionar la complejidad de los programas. Normalmente una función guarda un algoritmo, una receta para hacer una tarea.

Por ejemplo, si tengo el siguiente código para determinar si un año es bisiesto o no y necesito usarlo 10 o 100 veces, entonces es buena idea crear una función que pueda invocar tantas veces como necesite. Además si necesito hacer cambios sólo habrá que tocar el código de la función.

```
const oAnio = new Date().getFullYear();
if (oAnio % 4 !== 0) {
    console.log(`${oAnio} NO es un año bisiesto.`);
} else if (oAnio % 100 != 0) {
    console.log(`${oAnio} ES un año bisiesto.`);
} else if (oAnio % 400 != 0) {
    console.log(`${oAnio} NO es un año bisiesto.`);
} else {
    console.log(`${oAnio} ES un año bisiesto.`);
}

//Función que hace el código reutilizable.
function imprimirEstadoAnioBisiesto() {
    const oAnio = new Date().getFullYear();
    if (oAnio % 4 !== 0) {
        console.log(`${oAnio} NO es un año bisiesto.`);
    } else if (oAnio % 100 != 0) {
        console.log(`${oAnio} ES un año bisiesto.`);
    } else if (oAnio % 400 != 0) {
        console.log(`${oAnio} NO es un año bisiesto.`);
    } else {
        console.log(`${oAnio} ES un año bisiesto.`);
    }
}
```

Definiendo una función hemos creado un código reusable, si necesito cambiar algo no tengo que buscar todas las instancias repartidas por el código, ahora las líneas están bien localizadas y cualquier cambio sólo afectará a las líneas de la función.

Imagina que ahora no sólo necesitas imprimir por consola si el año es bisiesto, también quieres utilizarla para mostrar mensajes en el HTML, escribir la información en un fichero o hacer cálculos en otras partes del código. Podríamos reescribir la función de la siguiente forma:

```
function esAnioBisiesto() {
    const oAnio = new Date().getFullYear();
    if (oAnio % 4 !== 0) return false;
    else if (oAnio % 100 != 0) return true;
    else if (oAnio % 400 != 0) return false;
    else return true;
}
```

Requisitos

Para que una función se considere pura debe cumplir dos condiciones:

1. La primera es que siempre **devuelva la misma salida para el mismo conjunto de entradas**. Por ejemplo la función `esAnioBisiesto` no es pura porque devolverá un valor diferente dependiendo de cuando se llame, unas veces devolverá `true` y otras `false`.

2. La segunda es que **no debe tener efectos colaterales**. Esto quiere decir que la invocación de la función no debe alterar el estado del programa. Por ejemplo la siguiente función cambia el valor de una variable que no es parte de la función, por lo que se produce un efecto colateral.

```
const aColores = ['red', 'orange', 'yellow', 'green', 'blue'];
let iIndiceColor = -1;

function obtenerSiguienteColor() {
    if (++iIndiceColor >= aColores.length) iIndiceColor = 0;
    return aColores[iIndiceColor];
}
```

Actividad 1. Crea una función pura.

Convierte la función `esAnioBiesto` en una función pura.

Objetos

Los objetos son contenedores de propiedades. Una propiedad es un par clave:valor. Un valor puede ser un valor primitivo, otro objeto o una función.

Ya hemos utilizado objetos por ejemplo String, Date, Document o Node, de hecho casi todo en JavaScript es un objeto. El lenguaje también permite crear tus propios objetos y permite hacerlo de 2 formas diferentes:

1. Definiendo y creando un único objeto.

2. Definiendo un tipo de objeto y luego creando distintos objetos de dicho tipo.

1. Creando un único objeto.
La creación de un único objeto se puede hacer usando la notación literal o creando un objeto Object y luego añadiendo las propiedades deseadas. Aquí se muestra un ejemplo de ambas técnicas:

```
const oPersona = {
    nombre: "Pedro",
    apellidos: "Sánchez",
    edad: 50,
    colorOjos: "azul",
    nombreCompleto: function () {
        return `${this.nombre} ${this.apellidos}`;
    }
};
```

```
const oPersona = new Object();
oPersona.nombre = "Pedro";
oPersona.apellidos = "Sánchez";
oPersona.edad = 50;
oPersona.colorOjos = "azul";
oPersona.nombreCompleto = function () {
    return `${this.nombre} ${this.apellidos}`;
};
```

Estas dos técnicas hacen exactamente lo mismo. Por simplicidad, legibilidad y velocidad de ejecución **utiliza siempre la notación literal**.

2. Creando un tipo de objeto.

Notación ES5:

```
function Persona(nombreP, apellidosP, edadP, colorOjosP) {
    this.nombre = nombreP;
    this.apellidos = apellidosP;
    this.edad = edadP;
    this.colorOjos = colorOjosP;
    this.nombreCompleto = function () {
        return `${this.nombre} ${this.apellidos}`
    };
}

const oMiPadre = new Persona("Juan", "Sánchez", 70, "marrón");
const oMiMadre = new Persona("María", "Campos", 72, "verde");
```

ES6 introduce una nueva sintáxis para crear un tipo de objeto utilizando la palabra reservada class:

```
class Persona {
    constructor(nombreP, apellidosP, edadP, colorOjosP) {
        this.nombre = nombreP;
        this.apellidos = apellidosP;
        this.edad = edadP;
        this.colorOjos = colorOjosP;
    }

    nombreCompleto() {
        return `${this.nombre} ${this.apellidos}`;
    }
}

const oMiPadre = new Persona("Juan", "Sánchez", 70, "marrón");
const oMiMadre = new Persona("María", "Campos", 72, "verde");
```

Por mantenibilidad **utiliza siempre la sintáxis que introduce ES6**.

Programación Orientada a Objetos.

La programación orientada a objetos es un paradigma de programación que se basa en la idea de que los programas se pueden diseñar como una serie de objetos que se relacionan para realizar una tarea.

Los objetos deben diseñarse de forma que encapsulen datos y funcionalidades relacionadas. Por ejemplo un Coche podría ser un objeto de un fabricante, modelo, número de puertas o número de bastidor (datos) y con unas funciones como acelerar, cambiar de marcha, abrir puertas o encender las luces (funcionalidad).

En este paradigma hay una serie de términos clave que debes entender:

- *Clase:* plantilla que permite definir un objeto de forma abstracta (el coche en el ejemplo anterior).
- *Objeto o instancia*: datos y funcionalidades concreta con unos determinados valores (por ejemplo un coche concreto por ejemplo un Fiat Punto con matrícula 2345JHJ).
- *Método*: una funcionalidad del objeto (por ejemplo acelerar).
- *Constructor*: método que inicializa el objeto la primera vez que se crea.

Actividad 1. Creación de objetos de tipo Coche.

a) Creación de clase e instancias.
Realiza las siguientes tareas utilizando la nueva notación de ES6:

1. Crea una clase `Coche` con los datos `fabricante`, `modelo`, `marchas` y `marchaActual`. Y con la función de `cambiarMarcha`. Que permita establecer una nueva `marchaActual`. Si la marcha a la que se pretende cambiar no existe, se debe lanzar una excepción.
2. Crea dos instancias de `Coche`: un `TeslaModelS` y un `Mazda3i`. Cambia la marcha del *Tesla* a primera y la del *Mazda* a marcha atrás.
3. Muestra por consola el `fabricante`, el `modelo` y la `marcha` en la que se encuentran ambos coches.

b) Getters and setters
Muchos lenguajes orientados a objetos (Java entre ellos) proporcionan mecanismos de protección para propiedades y métodos que permiten especificar un nivel de acceso. JavaScript no dispone de esta función. Para conseguir un comportamiento similar se pueden usar *propiedades dinámicas* que ayudan a reducir esta debilidad sin eliminarla por completo.

Añade métodos *getter* y *setter* a la clase `Coche` para cada una de sus propiedades.

c) Métodos estáticos.
Añade dos métodos estáticos a la clase `Coche`, uno para generar el número de bastidor de cada instancia y otro para determinar si dos coches son iguales. Esto sucederá cuando tengan el mismo fabricante y modelo.

d) Herencia.
Realiza las siguientes tareas:

1. Crea una clase genérica `Vehículo` con una única propiedad `pasajeros`. Incorpora un método para añadir un nuevo pasajero.

2. Haz que la clase `Coche` herede de la clase `Vehículo`.

3. Crea un nuevo `Coche`, añade varios pasajeros y muéstralos.

e) Polimorfismo.
JavaScript proporciona el operador `instanceOf` para determinar si un objeto es de una determinada clase.

Ejecuta el siguiente código y comprueba la salida de la sentencias que utilizan `instanceOf`:

```
class Moto extends Vehiculo {}
const c = new Coche();
const m = new Moto();
c instanceof Coche;
c instanceof Vehiculo;
m instanceof Coche;
m instanceof Moto;
m instanceof Vehiculo;
```

f) Representación textual de objetos.
Redefine el método `toString` de la clase `Coche` con el siguiente formato:

`fabricante modelo: número de bastidor`. Por ejemplo, *Tesla Model S: 12234*