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 puede entender como un subprograma que el desarrollador puede llevar a cabo cuando lo necesite.

Las funciones constituyen una parte esencial del lenguaje JavaScript y son una de las características más utilizadas. Empecemos aclarando la terminología. Este es un ejemplo de *declaración de una función*:

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

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:

```
decirHola();
```

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!”;
}
```

La llamada a la función, se resolverá al valor indicado.

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

Invocar versus referenciar.

Es importante entender la diferencia entre invocar a una función o simplemente referenciarla. En JavaScript las funciones son objetos por lo que pueden ser pasados o asignados como cualquier otro objeto del lenguaje.

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. Mira la siguiente función:

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

a y b se llaman *argumentos formales*. Cuando una función se invoca, los argumentos formales reciben un valor. Por ejemplo:

```
avg(5, 7);
```

Algo importante a tener en cuenta es que los argumentos sólo existen dentro del cuerpo de la función incluso si las variables tienen el mismo nombre fuera de la función:

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

Nuevas posibilidades en ES6.

Dos características interesantes relacionadas con los argumentos e introducidas en ES6 son la posibilidad de indicar argumentos para las funciones con un valor por defecto y poder pasar a una función un número argumentos variable.

Actividad 1. Argumentos por defecto.

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.

Para pasar a una función un número no determinado de argumentos se puede utilizar el 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");
```

Nota que el operador de propagación siempre debe ser el último argumento, en caso contrario JavaScript no será capaz de distinguir el resto.

Funciones anónimas.

JavaScript permite declarar funciones con y sin identificador. Podemos crear por tanto:

- **Funciones convencionales**, aquellas compuestas por un cuerpo que define lo que hace y un identificador que permite referenciarlas e invocarlas. Por ejemplo:

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

- **Funciones anónimas**, aquellas que no tienen identificador. Por ejemplo:

```
function () {
      return “Hello world!”;
}
```

Su uso dependerá del contexto, si la función se está creando para llamarla más tarde crearás una función convencional, si necesitas crear una función para asignar o pasar a otra función entonces usarás una función anónima. Ejemplos equivalentes:

```
boton.addEventListener(“click”, decirHola);

function decirHola() {
      consola.log(“¡Hola mundo!”);
}

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

Notación flecha.

ES6 ha introducido una nueva sintáxis llamada *notación flecha* que permiten crear las llamadas *funciones flecha*. El término flecha hace referencia al signo “->”.

Las funciones flecha simplifican la sintaxis reduciendo el número de caracteres necesarios para declarar una función. Esta inclusión ha sido muy bien recibida siendo de las características más usadas del lenguaje. Simplifican la sintaxis de la siguiente forma:

- 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`.

Las funciones flecha son siempre anónimas.

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. Por tanto siempre deberíamos crear este tipo de funciones. Para entenderlas veamos algunos conceptos previos:

Funciones como subrutinas.

La idea de una subrutina ha estado presente desde los inicios de la programación como una forma de gestionar la complejidad de los programas. Normalmente una subrutina 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:

```
const year = new Date().getFullYear();
if(year % 4 !== 0) console.log(`${year} is NOT a leap year.`)
else if(year % 100 != 0) console.log(`${year} IS a leap year.`)
else if(year % 400 != 0) console.log(`${year} is NOT a leap year.`)
else console.log(`{$year} IS a leap year`);

function printLeapYearStatus() {
      const year = new Date().getFullYear();
      if(year % 4 !== 0) console.log(`${year} is NOT a leap year.`)
      else if(year % 100 != 0) console.log(`${year} IS a leap year.`)
      else if(year % 400 != 0) console.log(`${year} is NOT a leap year.`)
      else console.log(`{$year} IS a leap year`);
}
```

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.

Funciones como subrutinas que devuelven un valor

El primer paso para crear funciones puras es verlas como subrutinas que devuelven un valor. Teniendo en cuenta la función anterior, 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 de tu código. Podríamos reescribir la función de la siguiente forma:

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

Fíjate también en el nuevo identificador, muy utilizado cuando la función devuelve un valor booleano.

Funciones puras.

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

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

- 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 la variable `colorIndex` que no es parte de la función, por lo que se produce un efecto colateral.

```
const colors = ['red', 'orange', 'yellow', 'green', 'blue', 'indigo',
'violet'];
let colorIndex = -1;
function getNextRainbowColor() {
   if(++colorIndex >= colors.length) colorIndex = 0;
   return colors[colorIndex];
}
```

Actividad 1. Crear una función pura.

Convierte la función para saber si un año es bisiesto para que sea una función pura.