Programación con JavaScript I

Funciones

Introducción

Una función es un bloque de código JavaScript que se define una vez, pero que puede ejecutarse o invocarse tantas veces como se desee. Las funciones de JavaScript están parametrizadas, es decir, una definición de función puede incluir una lista de parámetros que funcionan como variables locales para el cuerpo de la función.

```
34      if (FlagNewNum) {
35          FKeyPad.ReadOut.value  = Num;
36          FlagNewNum = false;
37          }
38      else {
39          if (FKeyPad.ReadOut.value == "0")
40          FKeyPad.ReadOut.value = Num;
41              else
42                  FKeyPad.ReadOut.value += Num;
43          }
44      }
45
46      function Operation (Op) {
47      var Readout = FKeyPad.ReadOut.value;
48      if (FlagNewNum && PendingOp != "=");
49      else
50          {
51              FlagNewNum = true;
52              if ( '+' == PendingOp )
53              FKeyPad.ReadOut.value = parseFloat(FKeyPad.ReadOut.value);
54              else if ( '-' == PendingOp )
55              Accumulate -= parseFloat(Readout);
56              else if ( '/' == PendingOp )
57              FKeyPad.ReadOut.value = parseFloat(FKeyPad.ReadOut.value);
58              else if ( '*' == PendingOp )
59              Accumulate *= parseFloat(Readout);
60              else
61                  Accumulate = parseFloat(Readout);
62                  FKeyPad.ReadOut.value = Accumulate;
63                  PendingOp = Op;
64          }
```

A menudo, las funciones usan sus valores de argumento para calcular un valor de retorno que se convierte en el valor de la expresión con que se manda llamar a la función. Además de los argumentos, cada invocación tiene otro valor: el contexto de invocación, que es el valor de la palabra clave this.
Si una función se asigna a una propiedad de un objeto, se conoce como método de ese objeto. Cuando se invoca una función en un objeto, ese objeto es el contexto de invocación. Las funciones diseñadas para inicializar un objeto recién creado se denominan constructores.

En JavaScript, las funciones son objetos y pueden ser manipulados por programas, pueden establecer propiedades en ellas e incluso invocar métodos en ellas. Además, JavaScript puede asignar funciones a variables y pasarlas a otras funciones.

Las definiciones de funciones de JavaScript se pueden anidar dentro de otras funciones y tienen acceso a cualquier variable que esté dentro del alcance donde están definidas, lo que significa que las funciones de JavaScript son cierres y habilitan técnicas de programación importantes y poderosas.

Explicación

Funciones

Como pudiste revisar en la introducción, una función es un bloque de código que se define una vez, pero que puede invocarse tantas veces como se desee. Esto permite a los programadores escribir código modular, es decir, que los programas con muchas líneas pueden ser divididos en funciones más pequeñas y manejables.

Antes de poder usar una función, es necesario definirla, ya que existen **diferentes tipos de funciones** o formas de crearlas. La forma más sencilla de **definir** una función es con la palabra clave ***function***, seguida del nombre de la nueva función, el cual debe ser único. Los nombres de las funciones obedecen las mismas reglas de las variables. Después, se coloca la lista de parámetros (si es que tuviera porque no es obligatorio) separados por comas y finalmente el bloque de instrucciones que se realizarán dentro de la función(cuerpo).

La sintaxis básica para la declaración de una función es la siguiente:

```
function nombreDeLaFunción(parámetros) {
  sentencias
}
```

Figura 1. Sintaxis de declaración de funciones.

De acuerdo con lo que explica De Andres (2019), cuando se trabaja con funciones, es importante diferenciar entre un **parámetro** y un **argumento.** Un parámetro es una variable que definimos cuando creamos la función, mientras que los argumentos son los datos que pasamos a la función que invocamos. En otras palabras, son los valores de los parámetros de la función.

```
function cuadrado(numero){
  return numero * numero
}
cuadrado(5);
```

Figura 2. Declaración de función en JavaScript.

En la figura 2 se define la función **cuadrado** y toma el parámetro **numero**. La función consta de una declaración que indica que devolverá el parámetro con el que hayamos invocado la función (numero) multiplicado por él mismo. La instrucción **return** especifica el valor que devolverá la función.

Explicación

Flanagan (2020) señala que el hoisting o elevación es una de las características más inusuales de este tipo de funciones. Este hoisting es una característica de JavaScript por la cual las definiciones se llevan a cabo al inicio de la ejecución del código, dicho de otra manera, es la característica de “mover declaraciones” al inicio del código. Por ejemplo:

```
cuadrado(5);

function cuadrado(numero){
  return numero * numero
}
```

Figura 3. Hoisting o elevación en JavaScript.

Con este tipo de definición de funciones no obtendremos un error al invocar la función cuadrado que aún no ha sido creada. Esto se debe a que JavaScript lo habrá ejecutado de la manera en que se muestra en la figura 2.

Las definiciones de funciones de JavaScript no especifican un tipo esperado para los **parámetros** de la función, mientras que las invocaciones de funciones no realizan ninguna verificación de tipo en los valores de argumento que pasa. De hecho, las invocaciones de funciones de JavaScript ni siquiera verifican el número de argumentos que se pasan. Los parámetros son usados dentro de la función como variables locales.

Cuando se invoca una función con menos argumentos que los parámetros declarados, los parámetros adicionales se establecen en su valor predeterminado, que normalmente no está definido. A menudo, es útil escribir funciones para que algunos argumentos sean opcionales. Ve el siguiente ejemplo:

```
// Agrega los nombres de las propiedades que se pueden numerar del objeto i al array r y
regresa r.
// Si este es omitido crea y devuelve una matriz.
function getPropertyNames(i, r) {
  if (r === undefined) r = []; // Si esta indefinido, usa nueva r
  array
  for(let property in i) r.push(property);
  return r;
}
// getPropertyNames() puede ser invocada con uno o dos parámetros:
let i={x:1},t={y:2,z:3}; //Dos objetos para probar
let r = getPropertyNames(i); // i == ["x"]; obtiene las propiedades de i en un nuevo arreglo
getPropertyNames(t, r); // i == ["x","y","z"]; agrega las propiedades de t al arreglo
```

Figura 4. Uso de parámetros opcionales.

Explicación

El código de una función se ejecuta en alguno de estos tres supuestos:

- Un usuario hace clic sobre un botón.
- Cuando se llama desde el código.
- Cuando es autoinvocada.

Haverbeke (2018) explica que por lo general las funciones producen un valor, sin embargo, hay algunas que no lo hacen y el único resultado es un efecto secundario. Normalmente, cuando la secuencia de devolución del código es alcanzada (si queremos que la función devuelva algo, debemos utilizar la palabra reservada **return**), la función dejará de ejecutarse y devuelve el valor retornado al código que invocó a la función. Las funciones que no producen un valor de retorno comoquiera retornan algo, aunque en esos casos regresarán un valor indefinido.

Otra forma de declarar una función es una forma muy similar a la anterior, la única diferencia es que la definición de la *función* no comienza con la instrucción function y el nombre de la función es opcional, lo que te permite almacenar el valor en una variable o constante.

```
const cuadrado = function(numero) { return numero*numero; };
let diezcuadrado = (function(x) {return x*x;}(10));
```

Figura 5. Declaración de funciones como expresión.

Si al crear una función no utilizamos un identificador como en el caso anterior, entonces se dice que estamos creando una **función anónima.**

Es importante resaltar que estas no son compatibles con el hoisting. Por ello, si invocas una de estas funciones antes de que haya sido declarada, se obtendrá un error.

Existen otro tipo de **funciones que se ejecutan inmediatamente** y no son accesibles después. Este tipo de funciones se deben crear dentro de un operador de agrupación (), seguido de otro operador de agrupación (), esto evitará que sean interpretadas por el motor de JavaScript. De esta manera se evita “contaminar” el ambiente con el uso o asignación de una variable. Por ejemplo:

```
(function () {
  var nombre = "Juan";
})();
console.log(nombre);
// Devuelve Uncaught ReferenceError: nombre is not defined
```

Figura 6. Función expresada invocadas inmediatamente.

Explicación

Otro tipo de funciones que podemos declarar son aquellas que pueden usarse como método de un objeto o clase. Para crear estas funciones se un asigna nombre, lista de parámetros y las sentencias deben estar entre llaves. Ejemplo:

```
const alumnos = {
  personas: [],
  agrega(...personas) {
   this.personas.push(...personas);
  },
  saludo(i) {
   return `Hola mi nombre es ${this.personas[i]}`;
  }
 };

 alumnos.agrega('Hugo', 'Paco', 'Luis');
 alumnos.saludo(0); // => 'Hola mi nombre es Hugo'
```

Figura 7. Definición de funciones con métodos de propiedad.

En este caso, agrega() y saludo() son métodos de la clase alumnos. Posteriormente, hemos invocado a los dos métodos, en primer lugar, alumnos.agrega para añadir tres personas. A continuación, hemos invocado al método alumnos.saludo, enviando como parámetro 0 que nos ha devuelto el saludo de la persona en la posición 1 de nuestro arreglo.

En ES6 se utiliza una nueva forma importante de definir funciones sin la palabra clave función: las **funciones de flecha**, las cuales tienen una sintaxis particularmente compacta y son muy útiles cuando se pasa una función como argumento a otra función. En estas funciones definimos la lista de parámetros entre paréntesis (si no lleva parámetros, se requieren los paréntesis; si tiene un parámetro, no requiere los paréntesis, y si lleva más de un parámetro, sí requiere paréntesis). Luego, se colocan los símbolos => y las llaves {} para indicar el cuerpo de la función. Adicional a lo anterior, De Andres (2019) enlista las siguientes características de estas funciones:

- No crean su propio contexto al ejecutarse. Las funciones expresadas y las funciones declaradas con la palabra función sí lo crean.
- Las funciones de flecha son anónimas.
- El objeto argumento no se encuentra en el contexto de la función.
- Si al definir la función no utilizamos el símbolo de las llaves, la función devolverá como resultado de la ejecución la instrucción que hayamos indicado.

Explicación

Ejemplo:

```
const saludo = (nombre) => {
  return `Hola ${nombre}`;
}

console.log(saludo('Pedro'))  // => Devuelve Hola Pedro

// Compactando un poco más el código.
const salu2 = (nombre) => `Hola ${nombre}`;

console.log(salu2('Pedro')); // => Devuelve Hola Pedro
```

Figura 8. Definición de funciones de flecha.

Ahora que conoces las formas más comunes de declarar una función, debes aprender que no puedes tener acceso a variables que fueron definidas en una función desde cualquier parte del programa. Esto es porque la variable fue definida solo en el ámbito de la función, es decir, la variable solamente tendrá valor y podrá ser procesada dentro de la función. Por otro lado, una función puede tener acceso a las variables y funciones definidas dentro del ámbito en el que estas están definidas, el cual es considerado como un ámbito global.

Una función que ha sido definida dentro de otra función puede tener acceso a todas las variables de su función principal, así como a cualquier otra variable a la que tenga acceso la función principal.

Explicación

Ejemplo:

```
//Variables definidas en ámbito global
var numero1 = 5,
numero2 = 4,
nombre = 'Jordan';

//Función definida en ámbito global
function multiplica() {
  return numero1 * numero2;
}

multiplica(); //Regresa 20
console.log(multiplica())

//Funciones anidadas
function getMarcador() {
  //Variables definidas en ámbito local
  let numero1 = 10,
  numero2 = 7;

  function agregar() {
    return nombre + ' anotó ' + (numero1 + numero2);
 }

return agregar();
}

getMarcador(); // Devuelve "Jordan anotó 17"
```

Figura 9. Ámbito de variables y funciones.

Ahora que conoces las formas más comunes de declarar una función, debes aprender que no puedes tener acceso a variables que fueron definidas en una función desde cualquier parte del programa. Esto es porque la variable fue definida solo en el ámbito de la función, es decir, la variable solamente tendrá valor y podrá ser procesada dentro de la función. Por otro lado, una función puede tener acceso a las variables y funciones definidas dentro del ámbito en el que estas están definidas, el cual es considerado como un ámbito global.

Una función que ha sido definida dentro de otra función puede tener acceso a todas las variables de su función principal, así como a cualquier otra variable a la que tenga acceso la función principal.

```
function factorial(num) {
  if (num <= 1) return 1; return num * factorial(num-1);
}
```

Figura 10. Recursividad.

Cierre

En este tema aprendiste a escribir funciones en sus diferentes formas. Las funciones de flecha son las que más me gustan a mí, ¿cuál es la manera de definir funciones que más te gustó a ti o cuál crees que podría ser más útil?

Para comprender las funciones, es esencial entender el ámbito. Además, es importante saber que puedes utilizar el mismo nombre de variables para ámbitos locales y globales, pero no siempre es la mejor de las prácticas porque esto te podría ocasionar confusiones innecesarias.

Otro aspecto que debes analizar ahora que comienzas en el mundo de la programación es que el uso de funciones te permitirá ahorrar tiempo y esfuerzo, pues no tendrás que repetir tanto código, además de que te pueden ayudar a organizar un programa de mejor manera.

Referencias bibliográficas

- De Andres, V. (2019). *Funciones en Javascript. 7 formas de declarar una función.* Recuperado de https://dev.to/victordeandres/funciones-en-javascript-7-formas-de-declarar-una-funcion-523a
- Flanagan, D. (2020). *JavaScript: the Definitive Guide* (7a ed.). Estados Unidos: O'Reilly Media, Inc.
- Haverbeke, M. (2018). *ELOQUENT JAVASCRIPT* (3ª ed.). Estados Unidos: No Starch Press.

Para saber más

Lecturas

- LenguajeJS. (s.f.). *Funciones: Fragmentos de código en bloques reutilizables.* Recuperado de https://lenguajejs.com/javascript/fundamentos/funciones/
- Kantor, I. (s.f.). *El lenguaje JavaScript.* Recuperado de https://es.javascript.info/js
- MDN. (s.f.). *Funciones.* Recuperado de https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Functions

Videos

- Jonmircha. (2020, 20 de febrero). *Curso JavaScript: 10. Funciones* - #jonmircha [Archivo de video] Recuperado de https://www.youtube.com/watch?v=H6U1Pm7x60E
- Jonmircha. (2020, 6 de marzo). *Curso JavaScript: 21. Arrow Functions* - #jonmircha [Archivo de video] Recuperado de https://www.youtube.com/watch?v=WuCw9agV3Rc

Checkpoints

Asegúrate de:

- Utilizar todos los tipos de funciones y objetos.
- Desarrollar diferentes formas de funciones y objetos, escribiendo código en el ambiente que has instalado para practicar los conocimientos que vas adquiriendo.
- Identificar el funcionamiento y el uso adecuado en cada una de las situaciones a las que tengas que darles solución a través de un programa.

Requerimientos técnicos

- Computadora con acceso a Internet.
- Editor de texto.
- Permisos de administrador y/o Git instalado previamente.

Prework

- Leer detenidamente y comprender el material explicado en este tema.
- Practicar todos los ejemplos que se describen previamente en este tema.
- Revisar cada uno de los recursos adicionales que se proponen en este tema.