Funciones

Muy a menudo necesitamos realizar acciones similares en muchos lugares del script.

Por ejemplo, debemos mostrar un mensaje atractivo cuando un visitante inicia sesión, cierra sesión y tal vez en otros momentos.

Las funciones son los principales "bloques de construcción" del programa. Permiten que el código se llame muchas veces sin repetición.

Ya hemos visto ejemplos de funciones integradas, como `alert(message)`, `prompt(message, default)` y `confirm(question)`. Pero también podemos crear funciones propias.

Declaración de funciones

Para crear una función podemos usar una *declaración de función*.

Se ve como aquí:

```
function showMessage() {
  alert( '¡Hola a todos!' );
}
```

La palabra clave `function` va primero, luego va el *nombre de función*, luego una lista de *parámetros* entre paréntesis (separados por comas, vacía en el ejemplo anterior) y finalmente el código de la función entre llaves, también llamado "el cuerpo de la función".

```
function name(parameter1, parameter2, ... parameterN) {
  ...body...
}
```

Nuestra nueva función puede ser llamada por su nombre: `showMessage()`.

Por ejemplo:

```
function showMessage() {
  alert( '¡Hola a todos!' );
}


```

鰆 鯽

```
6
showMessage();
showMessage();
```

La llamada `showMessage()` ejecuta el código de la función. Aquí veremos el mensaje dos veces.

Este ejemplo demuestra claramente uno de los propósitos principales de las : evitar la duplicación de código... Si alguna vez necesitamos cambiar el mensaje o la forma en que se muestra, es suficiente modificar el código en un lugar: la función que lo genera.

Variables Locales

Una variable declarada dentro de una función solo es visible dentro de esa función.

Por ejemplo:

```
function showMessage() {
  let message = "Hola, ¡Soy JavaScript!"; // variable local

  alert( message );
}

showMessage(); // Hola, ¡Soy JavaScript!

alert( message ); // <-- ¡Error! La variable es local para esta función
```

Variables Externas

Una función también puede acceder a una variable externa, por ejemplo:

```
let userName  = 'Juan';

function showMessage() {
  let message = 'Hola, ' + userName ;
  alert(message);
}

showMessage(); // Hola, Juan
```

La función tiene acceso completo a la variable externa. Puede modificarlo también.

Por ejemplo:

```
let userName = 'Juan';

function showMessage() {
```

```
  userName = "Bob"; // (1) Cambió la variable externa

let message = 'Hola, ' + userName;
  alert(message);
}

alert( userName ); // Juan antes de llamar la función

showMessage();

alert( userName ); // Bob, el valor fué modificado por la función
```

La variable externa solo se usa si no hay una local.

Si una variable con el mismo nombre se declara dentro de la función, le *hace sombra* a la externa. Por ejemplo, en el siguiente código, la función usa la variable `userName` local. La exterior se ignora:

```
let userName = 'John';

function showMessage() {
  let userName = "Bob"; // declara variable local

let message = 'Hello, ' + userName; // Bob
  alert(message);
}

// la función crea y utiliza su propia variable local userName
showMessage();

alert( userName ); // John, se mantiene, la función no accedió a la variable ex
```

Variables globales

Variables declaradas fuera de cualquier función, como la variable externa `userName` en el código anterior, se llaman *globales*.

Las variables globales son visibles desde cualquier función (a menos que se les superpongan variables locales con el mismo nombre).

Es una buena práctica reducir el uso de variables globales. El código moderno tiene pocas o ninguna variable global. La mayoría de las variables residen en sus . Aunque a veces puede justificarse almacenar algunos datos a nivel de proyecto.

Parámetros

Podemos pasar datos arbitrarios a usando parámetros.

En el siguiente ejemplo, la función tiene dos parámetros: `from` y `text`.

```
function showMessage(from, text) { // parámetros: from, text
  alert(from + ': ' + text);
}

showMessage('Ann', '¡Hola!'); // Ann: ¡Hola! (*)
showMessage('Ann', "¿Cómo estás?"); // Ann: ¿Cómo estás? (**)
```

Cuando la función se llama `(*)` y `(**)`, los valores dados se copian en variables locales `from` y `text`. Y la función las utiliza.

Aquí hay un ejemplo más: tenemos una variable `from` y la pasamos a la función. Tenga en cuenta: la función cambia `from`, pero el cambio no se ve afuera, porque una función siempre obtiene una copia del valor:

```
function showMessage(from, text) {

  from = '*' + from + '*'; // hace que "from" se vea mejor

alert( from + ': ' + text ); 6
} 7

let from = "Ann";

showMessage(from, "Hola"); // *Ann*: Hola

// el valor de "from" es el mismo, la función modificó una copia local
alert( from ); // Ann
```

Cuando un valor es pasado como un parámetro de función, también se denomina *argumento*.

Para poner los términos claros:

- Un parámetro es una variable listada dentro de los paréntesis en la declaración de función (es un término para el momento de la declaración)
- Un argumento es el valor que es pasado a la función cuando esta es llamada (es el término para el momento en que se llama).

Declaramos listando sus parámetros, luego las llamamos pasándoles argumentos.

En el ejemplo de arriba, se puede decir: "la función `showMessage` es declarada con dos parámetros, y luego llamada con dos argumentos: `from` y `"Hola"`".

Valores predeterminados

Si una función es llamada pero no se le proporciona un argumento, su valor correspondiente se convierte en `undefined`.

Por ejemplo, la función mencionada anteriormente `showMessage(from, text)` se puede llamar con un solo argumento:

```
showMessage("Ann");
```

Eso no es un error. La llamada mostraría `"Ann: undefined"`. Como no se pasa un valor de `text`, este se vuelve `undefined`.

Podemos especificar un valor llamado "predeterminado" o "default" (que se usa si el argumento fue omitido) en la declaración de función usando `=`:

```
function showMessage(from, text = "sin texto" ) {
  alert( from + ": " + text   );
}

showMessage("Ann"); // Ann: sin texto
```

Ahora, si no existe el parámetro `text`, obtendrá el valor `"sin texto"`

Aquí `"sin texto"` es un string, pero puede ser una expresión más compleja, la cual solo es evaluada y asignada si el parámetro falta. Entonces, esto también es posible:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() solo se ejecuta si text no fue asignado
// su resultado se convierte en el valor de texto
}
```

Evaluación de parámetros predeterminados

En JavaScript, se evalúa un parámetro predeterminado cada vez que se llama a la función sin el parámetro respectivo.

En el ejemplo anterior, `anotherFunction()` no será llamado en absoluto si se provee el parámetro `text`.

Por otro lado, se llamará independientemente cada vez que `text` se omita.

Parámetros predeterminados alternativos

A veces tiene sentido asignar valores predeterminados no en la declaración de función sino en un estadio posterior.

Podemos verificar si un parámetro es pasado durante la ejecución de la función comparándolo con `undefined`:

```
function showMessage(text) {
  // ...

  if (text === undefined) { // si falta el parámetro
    text = 'mensaje vacío';
  }

```

```
  alert(text);
}

showMessage(); // mensaje vacío
```

...O podemos usar el operador `||`:

```
function showMessage(text) {
  // si text es indefinida o falsa, la establece a 'vacío'
text = text || 'vacío';
  ...
}
```

Los intérpretes de JavaScript modernos soportan el operador nullish coalescing `??`, que es mejor cuando el valor de `0` debe ser considerado "normal" en lugar de falso:

```
function showCount(count) {
  // si count es undefined o null, muestra "desconocido"
  alert(count ?? "desconocido");
}

showCount(0); // 0
showCount(null); // desconocido
showCount(); // desconocido
```

Devolviendo un valor

Una función puede devolver un valor al código de llamada como resultado.

El ejemplo más simple sería una función que suma dos valores:

```
function sum(a, b) {
  return a + b;
}

let result = sum(1, 2);
alert( result ); // 3
```

La directiva `return` puede estar en cualquier lugar de la función. Cuando la ejecución lo alcanza, la función se detiene y el valor se devuelve al código de llamada (asignado al `result` anterior).

Puede haber muchos casos de `return` en una sola función. Por ejemplo:

```
function checkAge(age) {
  if (age > 18) {
```

```
    return true;
} else {
return confirm('¿Tienes permiso de tus padres?');
  }
}

let age = prompt('¿Qué edad tienes?', 18);

if ( checkAge(age) ) {
  alert( 'Acceso otorgado' );
} else {
  alert( 'Acceso denegado' );
}
```

Es posible utilizar `return` sin ningún valor. Eso hace que la función salga o termine inmediatamente.

Por ejemplo:

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

alert( "Mostrándote la película" ); // (*)
  // ...
}
```

En el código de arriba, si `checkAge(age)` devuelve `false`, entonces `showMovie` no mostrará la `alert`.

Una función con un `return` vacío, o sin return, devuelve `undefined`

Si una función no devuelve un valor, es lo mismo que si devolviera `undefined`:

```
function doNothing() { /* empty */ }

alert( doNothing() === undefined ); // true
```

Un `return` vacío también es lo mismo que `return undefined`:

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

Nunca agregue una nueva línea entre `return` y el valor

Para una expresión larga de `return`, puede ser tentador ponerlo en una línea separada, como esta:

```
return
 (una + expresion + o + cualquier + cosa * f(a) + f(b))
```

Eso no funciona, porque JavaScript asume un punto y coma después del `return`. Eso funcionará igual que:

```
return;
 (una + expresion + o + cualquier + cosa * f(a) + f(b))
```

Entonces, efectivamente se convierte en un return vacío. Deberíamos poner el valor en la misma línea.

Nomenclatura de

Las son acciones. Entonces su nombre suele ser un verbo. Debe ser breve, lo más preciso posible y describir lo que hace la función, para que alguien que lea el código obtenga una indicación de lo que hace la función.

Es una práctica generalizada comenzar una función con un prefijo verbal que describe vagamente la acción. Debe haber un acuerdo dentro del equipo sobre el significado de los prefijos.

Por ejemplo, que comienzan con `"show"` usualmente muestran algo.

 que comienza con…

- `"get…"` – devuelven un valor,
- `"calc…"` – calculan algo,
- `"create…"` – crean algo,
- `"check…"` – revisan algo y devuelven un boolean, etc.

Ejemplos de este tipo de nombres:

```
showMessage(..)     // muestra un mensaje
getAge(..)          // devuelve la edad (la obtiene de alguna manera)
calcSum(..)         // calcula una suma y devuelve el resultado
createForm(..)// crea un formulario (y usualmente lo devuelve)
checkPermission(..) // revisa permisos, y devuelve true/false
```

Con los prefijos en su lugar, un vistazo al nombre de una función permite comprender qué tipo de trabajo realiza y qué tipo de valor devuelve.

Una función – una acción

Una función debe hacer exactamente lo que sugiere su nombre, no más.

Dos acciones independientes por lo general merecen dos , incluso si generalmente se convocan juntas (en ese caso, podemos hacer una tercera función que llame a esas dos).

Algunos ejemplos de cómo se rompen estas reglas:

- `getAge` – está mal que muestre una `alert` con la edad (solo debe obtenerla).
- `createForm` – está mal que modifique el documento agregándole el form (solo debe crearlo y devolverlo).
- `checkPermission` – está mal que muestre el mensaje `acceso otorgado/denegado` (solo debe realizar la verificación y devolver el resultado).

En estos ejemplos asumimos los significados comunes de los prefijos. Tú y tu equipo pueden acordar significados diferentes, aunque usualmente no muy diferente. En cualquier caso, debe haber una compromiso firme de lo que significa un prefijo, de lo que una función con prefijo puede y no puede hacer. Todas las con el mismo prefijo deben obedecer las reglas. Y el equipo debe compartir ese conocimiento.

Nombres de ultracortos

Las que se utilizan *muy a menudo* algunas veces tienen nombres ultracortos.

Por ejemplo, el framework jQuery define una función con `$`. La librería LoDash tiene como nombre de función principal `_`.

Estas son excepciones. En general, los nombres de las deben ser concisos y descriptivos.

== Comentarios

Las deben ser cortas y hacer exactamente una cosa. Si esa cosa es grande, tal vez valga la pena dividir la función en algunas más pequeñas. A veces, seguir esta regla puede no ser tan fácil, pero definitivamente es algo bueno.

Una función separada no solo es más fácil de probar y depurar, – ¡su existencia es un gran comentario!

Por ejemplo, comparemos las dos `showPrimes(n)` siguientes. Cada una devuelve números primos hasta `n`.

La primera variante usa una etiqueta:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

for (let j = 2; j < i; j++) {
if (i % j == 0) continue nextPrime;
    }

alert( i ); // un número primo
```

```
  }
}
```

La segunda variante usa una función adicional `isPrime(n)` para probar la primalidad:

```
function showPrimes(n) {

for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;

alert(i);  // a prime
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

La segunda variante es más fácil de entender, ¿no? En lugar del código, vemos un nombre de la acción. (`isPrime`). A veces las personas se refieren a dicho código como *autodescriptivo*.

Por lo tanto, las se pueden crear incluso si no tenemos la intención de reutilizarlas. Estructuran el código y lo hacen legible.

Resumen

Una declaración de función se ve así:

```
function name(parámetros, delimitados, por, coma) {
/* code */
}
```

- Los valores pasados a una función como parámetros se copian a sus variables locales.
- Una función puede acceder a variables externas. Pero funciona solo de adentro hacia afuera. El código fuera de la función no ve sus variables locales.
- Una función puede devolver un valor. Si no lo hace, entonces su resultado es `undefined` .

Para que el código sea limpio y fácil de entender, se recomienda utilizar principalmente variables y parámetros locales en la función, no variables externas.

Siempre es más fácil entender una función que obtiene parámetros, trabaja con ellos y devuelve un resultado que una función que no obtiene parámetros, pero modifica las variables externas como un efecto secundario.

Nomenclatura de :

- Un nombre debe describir claramente lo que hace la función. Cuando vemos una llamada a la función en el código, un buen nombre nos da al instante una comprensión de lo que hace y devuelve.
- Una función es una acción, por lo que los nombres de las suelen ser verbales.
- Existen muchos prefijos de bien conocidos como `create…`, `show…`, `get…`, `check…` y así. Úsalos para insinuar lo que hace una función.

Las son los principales bloques de construcción de los scripts. Ahora hemos cubierto los conceptos básicos, por lo que en realidad podemos comenzar a crearlos y usarlos. Pero ese es solo el comienzo del camino. Volveremos a ellos muchas veces, profundizando en sus avanzadas.

Tareas

¿Es "else" requerido?

importancia: 4

La siguiente función devuelve `true` si el parámetro `age` es mayor a `18`.

De lo contrario, solicita una confirmación y devuelve su resultado:

```
function checkAge(age) {
if (age > 18) {
return true;
} else {
    // ...
    return confirm('¿Tus padres te permitieron?');
  }
}
```

¿Funcionará la función de manera diferente si se borra `else`?

```
function checkAge(age) {
if (age > 18) {
return true;
  }
  // ...
  return confirm('¿Tus padres te permitieron?');
}
```

¿Hay alguna diferencia en el comportamiento de estas dos variantes?

solución

Reescribe la función utilizando '?' o '||'

importancia: 4

La siguiente función devuelve `true` si el parámetro `age` es mayor que `18`.

De lo contrario, solicita una confirmación y devuelve su resultado.

```
function checkAge(age) {
if (age > 18) {
return true;
} else {
return confirm('¿Tienes permiso de tus padres?');
   }
 }
```

Reescríbela para realizar lo mismo, pero sin `if`, en una sola linea.

Haz dos variantes de `checkAge`:

1. Usando un operador de signo de interrogación `?`
2. Usando OR `||`

solución

Función min(a, b)

importancia: 1

Escriba una función `min(a,b)` la cual devuelva el menor de dos números `a` y `b`.

Por ejemplo:

```
min(2, 5) == 2
min(3, -1) == -1
min(1, 1) == 1
```

solución

Función pow(x,n)

importancia: 4

Escriba la función `pow(x,n)` que devuelva `x` como potencia de `n`. O, en otras palabras, multiplique `x` por si mismo `n` veces y devuelva el resultado.

```
pow(3, 2) = 3 * 3 = 9
pow(3, 3) = 3 * 3 * 3 = 27
pow(1, 100) = 1 * 1 * ...* 1 = 1
```