Further Functions

FUNCTION PROPERTIES AND METHODS:

The fact that functions are first-class objects means they can have properties and methods themselves. For example, all functions have a length property that returns the number of parameters the function has.

square.length

CALL AND APPLY METHODS:

CALL() METHOD:

Used to set the value of this inside a function

It assigns it to an object provided as the first argument to the method call().

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

If the function that’s called requires any parameters, these need to be provided as arguments after the first argument

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

If a function doesn’t refer to an object as this in its body, it can still be called using the call() method, but you need provide null as its first argument.

Texto

Descripción generada automáticamente

APPLY() METHOD:

works in the same way, except the arguments of the function, are provided as an array, even if there is only one argument:

Texto

Descripción generada automáticamente

useful if the data you’re using as an argument is already in the form of an array, although it's not really needed in ES6, as the spread operator can be used to split an array of values into separate parameters.

These methods are useful because we can use generalized functions (that are not tied to any object), as methods of any object.

CUSTOM PROPERTIES:

We can create our own properties.

Texto

Descripción generada automáticamente

MEMOIZATION:

A useful feature of this is that it provides result caching, or **memoization**.

A result gets stored in the property.

If a function takes some time to compute a return value, we can save the result in a cache property. Then if the same argument is used again later, we can return the value from the cache, rather than having to compute the result again.

Texto

Descripción generada automáticamente

If we try calling the function a few times, we can see that the cache object stores the results:

Texto

Descripción generada automáticamente

IMMEDIATELY INVOKED FUNCTION EXPRESSIONS:

Texto

Descripción generada automáticamente

QUESTION:

When is this useful?

This means the global namespace is not polluted with lots of variable names.

There is no way to remove a variable from a scope once it’s been declared. If a variable is only required temporarily, it may cause confusion if it’s still available later in the code. Even worse, the name of the variable may clash with another piece of code (an external JavaScript library, for example) resulting in errors. Placing any code that uses the temporary variable inside an IIFE will ensure it’s only available while the IIFE is invoked, then it will disappear.

TEMPORARY VARIABLES:

The example that follows uses an IIFE to swap the value of two global variables, a and b. This process requires the use of a temporary variable, called temp, which only exists while the IIFE is invoked. Then it will disappear.

Captura de pantalla de computadora

Descripción generada automáticamente

Note that this technique is not needed to swap the values of two variables in ES6, as destructuring can be used, as shown below:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

INITIALIZATION CODE:

An IIFE can be used to set up any initialization code that there’ll be no need for again. Because the code is only run once, there’s no need to create any reusable, named functions, and all the variables will also be temporary. An IIFE will be invoked once, and can set up any variables, objects and event handlers when the page loads.

Texto

Descripción generada automáticamente

Note that much of this can be achieved in ES6 by simply placing the code inside a block. This is because variables have block scope when const or let are used, whereas in previous versions of JavaScript, only functions maintained the scope of variables. The example above would work just as well using the following code:

Texto

Descripción generada automáticamente

SAFE USE OF STRICT MODE:

One of the problems with simply placing 'use strict' at the beginning of a file is that it will enforce strict mode on all the JavaScript in the file, and if you’re using other people’s code, there’s no guarantee that they’ve coded in strict mode.

To avoid this, the recommended way to use strict mode is to place all your code inside an IIFE, like so:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

QUESTION:

Is this useful just for development purposes or is it good to use it in the final code? Is there another way to restrict the scope of the strict mode? Is this possible to achieve with simple blocks in ES6?

Again, this can be achieved in ES6 by simply placing the different parts of code into blocks. ES6 also supports a much more powerful module pattern that is covered in Chapter 15.

CREATING SELF-CONTAINED CODE BLOCKS:

An IIFE can be used to enclose a block of code inside its own private scope so it doesn’t interfere with any other part of the program.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

QUESTION:

When is this necessary?

Notice that both code blocks include a variable called name, but the modules don’t interfere with each other. This is a useful approach for separating parts of a program into discrete sections, especially for testing purposes.

Again, this can be achieved in ES6 by simply placing the different parts of code into blocks. ES6 also supports a much more powerful module pattern that is covered in Chapter 15.

FUNCTIONS THAT DEFINE AND REWRITE THEMSELVES:

This is called the Lazy Definition Pattern and is often used when some initialization code is required the first time it’s invoked. This means the initialization can be done the first time it’s called, then the function can be redefined to what you want it to be for every subsequent invocation.

Consider the following function:

Captura de pantalla con letras

Descripción generada automáticamente

Every time the function is called after the first time, it will log the message 'Been there, got the T-Shirt':

Texto

Descripción generada automáticamente

If the function is also assigned to another variable, this variable will maintain the original function definition and not be rewritten.

Texto

Descripción generada automáticamente

If any properties have previously been set on the function, these will be lost when the function redefines itself.

Texto

Descripción generada automáticamente

INIT-TIME BRANCHING:

This can be a useful pattern to initialize functions the first time they’re called, optimizing them for the browser being used.

functions that rewrite themselves, known as init-time branching. This enables the functions to work more effectively in the browser, and avoid checking for features every time they’re invoked.

Captura de pantalla de computadora

Descripción generada automáticamente

Right at the end of the function, we call it again so that the rewritten function is now invoked, and the relevant value returned. One thing to be aware of is that the function is invoked twice the first time, although it becomes more efficient each subsequent time it’s invoked. Let’s take a look at how it works:

Texto

Descripción generada automáticamente

RECURSIVE FUNCTIONS:

A recursive function is one that invokes itself until a certain condition is met

Pantalla de computadora con letras

Descripción generada automáticamente

The function will continue to invoke itself until finally the argument is 0 and 1 is returned. This will result in a multiplication of 1, 2, 3 and all the numbers up to the original argument.

Another example from the world of mathematics is the [Collatz Conjecture](http://en.wikipedia.org/wiki/Collatz_conjecture" \t "_blank). This is a problem that is simple to state, but, so far, has not been solved. It involves taking any positive integer and following these rules:

* If the number is even, divide it by two
* If the number is odd, multiply it by three and add one

For example, if we start with the number 18, we would have the following sequence

18, 9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, ...

As you can see, the sequence becomes stuck in a loop at the end, cycling through “4,2,1”. The Collatz Conjecture states that every positive integer will create a sequence that finishes in this loop. This has been verified for all numbers up to 5 × 2⁶⁰, but there is no proof it will continue to be true for all the integers higher than this. To test the conjecture, we can write a function that uses recursion to keep invoking the function until it reaches a value of 1 (because we want our function to avoid being stuck in a recursive loop at the end!):

Texto

Descripción generada automáticamente

CALLBACKS:

EVENT-DRIVEN ASYNCHRONOUS PROGRAMMING:

By using callbacks, we ensure that waiting for these tasks to complete doesn't hold up the execution of other parts of the program.

Here’s an example of a function called wait() that accepts a callback. To simulate an operation that takes some time to happen, we can use the setTimeout() function to call the callback after a given number of seconds:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

The setTimeout() function is asynchronous, which means that the callback provided as an argument is placed on top of a stack that gets cleared once the rest of the program has run. This means that control is handed back to the program and the next line in the program is run, which displays the message ‘Hmmm, should I accept this mission or not ... ?’ Then, after five seconds, the callback is retrieved from the stack and invoked. This demonstrates that the setTimeout() function did not block the rest of the program from running.

Remember, though, that JavaScript is still single-threaded, so only one task can happen at once. If an event only takes a small amount of time to happen, it will still have to wait until other parts of the program have executed before the callback is invoked. For example, let’s see what happens if we set the waiting time to be zero seconds:

QUESTION:

SO, WE’RE ONLY CHANGING THE ORDER BUT NOT HAVING TO THREADS HAPPENING? IS IT STILL USEFUL?

Texto

Descripción generada automáticamente

We would have expected the callback to have been invoked immediately, but a callback always has to wait for the current execution stack to complete before it’s invoked. In this case, the current execution stack is the rest of the function and code already entered in the console. Once these have executed, the callback is invoked before handing control back to the main program.

QUESTION:

THEN, THE CALLBACK FUNCTIONS START EXECUTING AND WAITING 5 SECONDS AFTER THE OTHER FUNCTIONS HAVE BEEN COMPLETED? OR DOES IT STARTS AT THE TOP OF THE STACK BUT WAITS UNTIL THE OTHER PARTS ARE EXECUTED TO COMPLETE ITS TASK?

CALLBACK HELL:

Texto

Descripción generada automáticamente

Error-first Callbacks

The code example above uses the *error-first* callback style popularized by Node.js. In this coding pattern, callbacks have two arguments. The first is the error argument, which is an error object provided if something goes wrong when completing the operation. The second argument is any data returned by the operation that can be used in the body of the callback.

Texto

Descripción generada automáticamente

QUESTION:

Why do we need to put one setTimeout function within another instead of putting one after another as in the image above?

PROMISES:

Pantalla negra con letras blancas

Descripción generada automáticamente

(The basic structure of a promise:

* To create it, we use the constructor
* We assign it to a variable
* Inside of the constructor, we pass an arrow function known as “executor”
* This “executor” has two arguments. Those arguments are both functions
* The first is the resolve() function, which executes when the process is successful.
* The second is the reject() function, which executes when the process is unsuccessful.
* We start the block with the initialization code (variables, etc).
* Then we open the if…else block that checks if the operation is successful or not according to the condition we pass as the argument.
* If it is successful, we invoke the resolve() function which returns a needed value or information
* Else, we invoke reject() function which returns a needed value or information.
* All of this is the PENDING STATE

)

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

(When a promise is settled: meaning, it has returned a response by resolve() or reject()

* We use the method .then() to handle the returned value from the promise.
* It can accept one or two arguments, both can be functions that deal, or do something with the result we got from the promise.
* The first function argument executes if the result has been gotten from resolve(), and the second function argument executes is if has been gotten from reject().
  + promise.then( result => console.log(`Yes! I rolled a ${result}`), result => console.log(`Drat! ... I rolled a ${result}`) );
* We can just use the first argument to deal with the successful result and a .catch() method to deal with an unsuccessful result.

)

The setTimeout() function is just used to delay the operation. But the promise object is asynchronous as itself.

CHAINING MULTIPLE PROMISES:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media Texto

Descripción generada automáticamente

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

ASYNC FUNCTIONS:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Always used with promises or delayed tasks. (Ajax, etc)

This is achieved by using the await operator before an asynchronous function. This will wrap the return value of the function in a promise that can then be assigned to a variable. The next line of code is not executed until the promise is resolved.

Pantalla de computadora con letras

Descripción generada automáticamente

In the example, the loadGame function is preceded by the async keyword, meaning the function will run in an asynchronous fashion. We then wrap each step of the process in a try block, so any errors are caught. Inside this block, we can write each step in the order it’s meant to be processed, so we start by assigning the variable user to the return value of the login() function. The await operator will ensure the next line of code is not executed until the login() function returns a user object. The getPlayerInfo() function is also preceded by the await operator. Once this function returns a result, it’s assigned to the variable info, and this can then be used to load the actual game. A catch block is used to deal with any errors that may occur.

GENERALIZED FUNCTIONS:

FUNCTIONS THAT RETURN FUNCTIONS:

The example below shows a function called returnHello() that returns a 'Hello World' function:

Texto

Descripción generada automáticamente

When the returnHello() function is invoked, it logs a message to the console then returns another function:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

To make use of the function that is returned, we need to assign it to a variable: Now we can invoke the 'Hello World' function by placing parentheses after the variable that it was assigned to:

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

This might seem a bit pointless, but let's now take it a step further and use this technique to create a generic 'greeter' function that takes a particular greeting as a parameter, then returns a more specific greeting function:

Texto

Descripción generada automáticamente

CLOSURES:

A closure is a reference to a variable that was created inside the scope of another function, but is then kept alive and used in another part of the program.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

We can now assign a variable to the return value of the outer() function:

Texto

Descripción generada automáticamente con confianza media

The variable closure now points to the inner() function that is returned by the outer() function.

What makes this a closure is that it now has access to the variables created inside both the outer() and inner() functions, as we can see when we invoke it:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Another example:

Pantalla negra con letras blancas

Descripción generada automáticamente

*const* toFahrenheit = closure();

Texto

Descripción generada automáticamente

Counter example:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

GENERATORS:

ES6 introduced support for generators. These are special functions used to produce iterators that maintain the state of a value. As we did with the counter example.

To define a generator function, an asterisk symbol (\*) is placed after the function declaration, like so:

Texto

Descripción generada automáticamente

Calling a generator function doesn’t actually run any of the code in the function; it returns a Generator object that can be used to create an iterator that implements a next() method that returns a value every time the next() method is called.

For example, we can create a generator to produce a Fibonacci-style number series (a sequence that starts with two numbers and the next number is obtained by adding the two previous numbers together), using the following code:

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente con confianza media

Pantalla de un celular con letras

Descripción generada automáticamente

It uses while infinite loops generally.

QUESTION:

Every time we invoke then(), where does the code starts running, just the part inside the while loop? Why?

Texto

Descripción generada automáticamente

It’s also possible to iterate over the generator to invoke it multiple times:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

Note that the sequence continued from the last value produced using the next() method. This is because a generator will maintain its state throughout the life of a program.

Generator functions employ the special yield keyword that is used to return a value. The difference between the yield and the return keywords is that by using yield, the state of the value returned is remembered the next time yield is called.

FUNCTIONAL PROGRAMMING:

It’s a programming style, just like OOP. IT’s based on the use of only functions that perform one single task to build a complex system composed just for functions. All the functions should follow some requisites. If they follow these requisites, it is called a Pure Function.

PURE FUNCTIONS:

A key aspect of functional programming is its use of pure functions. A pure function is a function that adheres to the following rules:

1) The return value of a pure function should only depend on the values provided as arguments. It doesn't rely on values from somewhere else in the program.

2) There are no side-effects. A pure function doesn't change any values or data elsewhere in the program. It only makes non-destructive data transformations and returns new values, rather than altering any of the underlying data.

3) Referential transparency. Given the same arguments, a pure function will always return the same result.

In order to follow these rules, any pure function must have:

* At least one argument; otherwise the return value must depend on something other than the arguments of the function, breaking the first rule
* A return value; otherwise there’s no point in the function (unless it has changed something else in the program – in which case, it’s broken the 'no side-effects' rule).

HIGHER-ORDER FUNCTIONS:

Higher-order functions are functions that accept another function as an argument, or return another function as a result, or both.

Closures are used extensively in higher-order functions as they allow us to create a generic function that can be used to then return more specific functions based on its arguments. This is done by creating a closure around a function's arguments that keeps them 'alive' in a return function. For example, consider the following multiplier() function:

Pantalla de un celular de un mensaje en letras blancas

Descripción generada automáticamente

The multiplier() function returns another function that traps the argument x in a closure. This is then available to be used by the returned function.

We can now use this generic multiplier() function to create more specific functions, as can be seen in the example below:

doubler = multiplier(2);

This creates a new function called doubler(), which multiplies a parameter by the argument that was provided to the multiplier() function (which was 2 in this case). The end result is a doubler() function that multiplies its argument by two:

doubler(10);

<< 20

The multiplier() function is an example of a higher-order function. This means we can use it to build other, more specific functions by using different arguments. For example, an argument of 3 can be used to create a tripler() function that multiplies its arguments by 3:

Texto

Descripción generada automáticamente

Here's another example

Texto

Descripción generada automáticamente Texto

Descripción generada automáticamente

When a higher-order function returns another function, we can use a neat trick to create an anonymous return function and immediately invoke it with a value instead by using double parentheses. The following example will calculate 3 to the power 5:

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

CURRYING:

Texto

Descripción generada automáticamente

Simplificado:

Pantalla de un video juego

Descripción generada automáticamente con confianza media

**Currying** is a process that involves the partial application of functions.

Partial because it actually doesn’t execute the operation until all the arguments have been provided. They return another function asking for the following argument until all of them are provided.

A function is said to be curried when not all arguments have been supplied to the function, so it returns another function that retains the arguments already provided, and expects the remaining arguments that were omitted when the original function was called. A final result is only returned once all the expected arguments have eventually been provided.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

This new function can then be used to calculate the tax, without requiring 0.22 as an argument:

Imagen que contiene Texto

Descripción generada automáticamente

A GENERAL CURRY FUNCTION:

In the last example, we hard-coded the multiplier() function so it could be curried. It’s possible to use a curry() function to take any function and allow it to be partially applied.(or curried) The curry function is the following:

Texto

Descripción generada automáticamente

This is the general curry function that makes any function to be curried.

This function accepts a function as its first argument, which is stored as func. The rest operator is used to collect all the other arguments together as ...oldArgs. These are the arguments of the function that is the first argument. It then returns a function that accepts some new arguments that are stored in the variable ...newArgs. These are then lumped together with ...oldArgs to make ...newArgs using the spread operator. The return value of this function is obtained by invoking the original function, which is accessed using a closure over func and passed the combined arguments ...allArgs.

Now let's create a generic divider() function that returns the result of dividing its two arguments:

*const* divider = (x,y) => x/y; // the generic function

*const* reciprocal = curry(divider,1); // making it partial applied, providing first just 1 argument

reciprocal(2);

<< 0.5