Functions

Friday, March 29, 2024

12:34 PM

**Defining a function**

A function is created with an expression that starts with the keyword **function** . Functions have a set of parameters and a body, which contains the statements that are to be executed when the function is called. The function body of a function created this way must always be wrapped in braces, even when it consists of only a single statement.

*const square = function(x) {*

*return x \* x;*

*};*

*console.log(square(12));*

*// → 144*

Some functions produce a value, and some don’t, whose only result is a side effect. A **return** statement determines the value the function returns. When control comes across such a statement, it immediately jumps out of the current function and gives the returned value to the code that called the function.

**Binding and scope**

Each binding has a scope, which is the part of the program in which the binding is visible. For bindings defined outside of any function or block, the scope is the whole program—you can refer to such bindings wherever you want. These are called ***global***.

But bindings created for function parameters or declared inside a function can be referenced only in that function, so they are known as ***local bindings***.

Bindings declared with **let** and **const** are in fact local to the block that they are declared in, so if you create one of those inside of a loop, the code before and after the loop cannot “see” it.

**Nested scope**

Each local scope can also see all the local scopes that contain it, and all scopes can see the global scope. This approach to binding visibility is called ***lexical scoping***.

**Functions as values**

Declaration notation

When the function keyword is used at the start of a statement, it works differently.

*function square(x) {*

*return x \* x;*

*}*

This is a function declaration. The statement defines the binding square and points it at the given function. It is slightly easier to write and doesn’t require a semicolon after the function.

The function call works, even when the function is defined below the code that uses it. Function declarations are not part of the regular top-to-bottom flow of control.

Arrow functions

There’s a third notation for functions, which looks very different from the others. Instead of the function keyword, it uses an **arrow (=>)** made up of an equal sign and a greater-than character (not to be confused with the greaterthan-or-equal operator, which is written >=).

*const power = (base, exponent) => {*

*let result = 1;*

*for (let count = 0; count < exponent; count++) {*

*result \*= base;*

*}*

*return result;*

*};*

The arrow comes after the list of parameters and is followed by the function’s body. It expresses something like “this input (the parameters) produces this result (the body)”.

When there is only one parameter name, you can omit the parentheses around the parameter list.

When an arrow function has no parameters at all, its parameter list is just an empty set of parentheses.

*const horn = () => {*

*console.log("Toot");*

*};*

**The call stack**

The place where the computer stores this context is the ***call stack***. Every time a function is called, the current context is stored on top of this stack. When a function returns, it removes the top context from the stack and uses that context to continue execution.

**Optional arguments**

The following code is allowed and executes without any problem:

*function square(x) { return x \* x; }*

*console.log(square(4, true, "hedgehog"));*

*// → 16*

JavaScript is extremely broad-minded about the number of arguments you pass to a function. If you pass too many, the extra ones are ignored. If you pass too few, the missing parameters get assigned the value undefined.

If you write an = operator after a parameter, followed by an expression, the value of that expression will replace the argument when it is not given.

**Closure**

The feature—being able to reference a specific instance of a local binding in an enclosing scope—is called ***closure***.

*function multiplier(factor) {*

*return number => number \* factor;*

*}*

*let twice = multiplier(2);*

*console.log(twice(5));*

*// → 10*

A good mental model is to think of function values as containing both the code in their body and the environment in which they are created. When called, the function body sees the environment in which it was created, not the environment in which it is called.

**Recursion**

It is perfectly okay for a function to call itself, as long as it doesn’t do it so often that it overflows the stack. A function that calls itself is called ***recursive***.

**Functions and side effects**

Functions can be roughly divided into those that are called for their side effects and those that are called for their return value.