Function Contents

This lesson explains the structure of a function in detail.

WE'LL COVER THE FOLLOWING

Return Value

Here is a variation of our example program.

```
function sayHello() {
  return "Hello!";
}

console.log("Start of program");
const message = sayHello(); // Store the function return value in a variable
console.log(message);       // Show the return value
console.log("End of program");
```

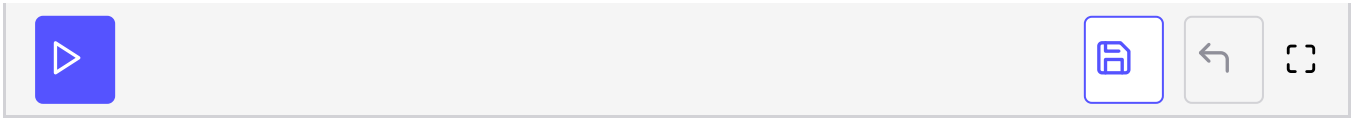
Run this code, and you'll see the same result as before. In this example, the body of the `sayHello()` function has changed: the statement `console.log("Hello!")` was replaced by `return "Hello!"`. The keyword `return` indicates that the function will return a value, which is specified immediately after the keyword. This *return value* can be retrieved by the caller.

```
// Declare myFunction
function myFunction() {
  let returnValue;
  // Calculate return value
  // returnValue = ...
  return returnValue;
```

```
}

// Get return value from myFunction

const result = myFunction();
// ...
```

This return value can be of any type (number, string, etc). However, a function can return only one value. Retrieving a function's return value is not mandatory, but in that case the return value is "lost". If you try to retrieve the return value of a function that does not actually have one, we get the JavaScript value `undefined`.

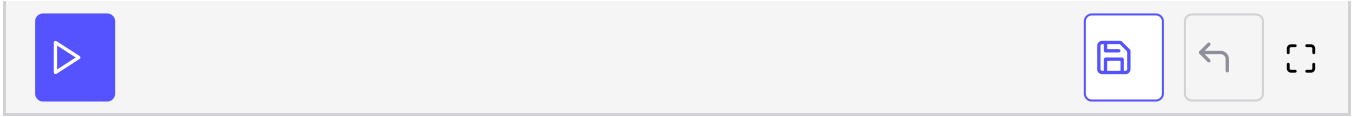
```
function myFunction() {
  // ...
  // No return value
}

const result = myFunction();
console.log(result); // undefined
```



> A function stops running immediately after the `return` statement is executed. Any further statements are never run.

Let's simplify our example a bit by getting rid of the variable that stores the function's return value.

```
function sayHello() {
  return "Hello!";
}

console.log(sayHello()); // "Hello!"
```

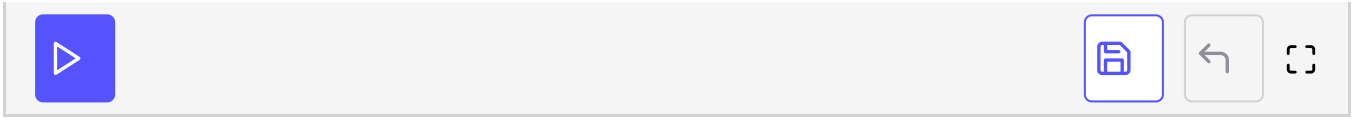


The return value of the `sayHello()` function is directly output through the `console.log()` command.

Local Variables

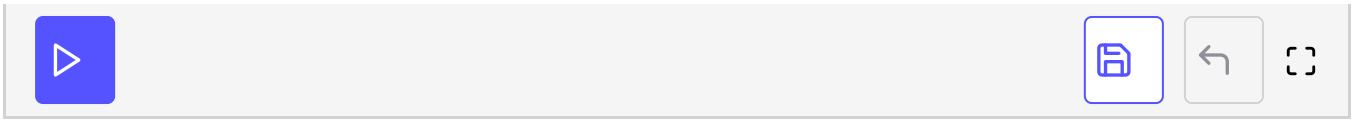
You can declare variables inside a function, as in the example below.

```
function sayHello() {
  const message = "Hello!";
  return message;
}

console.log(sayHello()); // "Hello!"
```



The function `sayHello()` declares a variable named message and returns its value. The variables declared in the body of a function are called *local variables*. Their *scope* is limited to the function body (hence their name). If you try to use these local variables outside the function, you won't be able to!

```
function sayHello() {
  const message = "Hello!";
  return message;
}

console.log(sayHello()); // "Hello!"
console.log(message); // Error: the message variable is not visible here
```



Each function call will redeclare the function's local variables, making the calls perfectly independent from one another. Not being able to use local variables outside the functions in which they are declared may seem like a limitation. Actually, it's a good thing! This means functions can be designed as autonomous and reusable. Moreover, this prevents *naming conflicts*: allowing variables declared in different functions to have the same name.

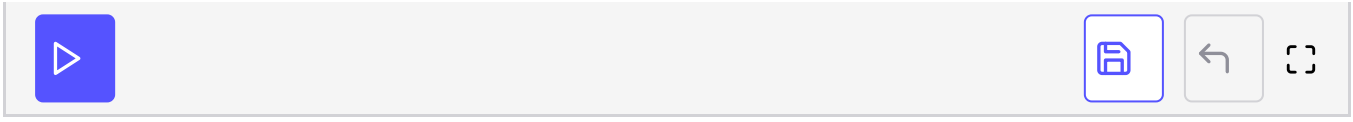
Parameter Passing

A *parameter* is information that the function needs in order to work. The function parameters are defined in parentheses immediately following the name of the function. You can then use the parameter value in the body of the function.

You supply the parameter value when calling the function. This value is called

You supply the parameter value when calling the function. This value is called an *argument*. Let's edit the above example to add a personalized greeting:

```
function sayHello(name) {
  const message = `Hello, ${name}!`;
  return message;
}

console.log(sayHello("Baptiste")); // "Hello, Baptiste!"
console.log(sayHello("Thomas"));   // "Hello, Thomas!"
```

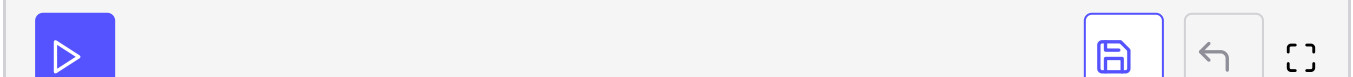


The declaration of the `sayHello()` function now contains a parameter called `name`. In this example, the first call to `sayHello()` is done with the argument `"Baptiste"` and the second one with the argument `"Thomas"`. In the first call, the value of the `name` parameter is `"Baptiste"`, and `"Thomas"` in the second. Here's the general syntax of a function declaration with parameters. The number of parameters is not limited, but more than 3 or 4 is rarely useful.

```
// Declare a function myFunction with parameters
function myFunction(param1, param2, ...) {
  // Statements using param1, param2, ...
}

// Function call
// param1 value is set to arg1, param2 to arg2, ...
myFunction(arg1, arg2, ...);
```

Just like with local variables, parameter scope is limited to the function body. Thus, an external variable used as an argument in a function call may have the same name as a function parameter. The following example is perfectly valid.

```
function sayHello(name) {
  // Here, "name" is the function parameter
  const message = `Hello, ${name}!`;
  return message;
}

// Here, "name" is a variable used as an argument
let name = "Baptiste";
console.log(sayHello(name)); // "Hello, Baptiste!"
name = "Thomas";
console.log(sayHello(name)); // "Hello, Thomas!"
```



When calling a function, respecting the number and order of parameters is paramount! Check out the following example.

```
function presentation(name, age) {
  console.log(`Your name is ${name} and you're ${age} years old`);
}

presentation("Garance", 9); // "Your name is Garance and you're 9 years old"
presentation(5, "Prosper"); // "Your name is 5 and you're Prosper years old"
```

The second call arguments are given in reverse order, so `name` gets the value `5` and `age` gets `"Prosper"` for that call.