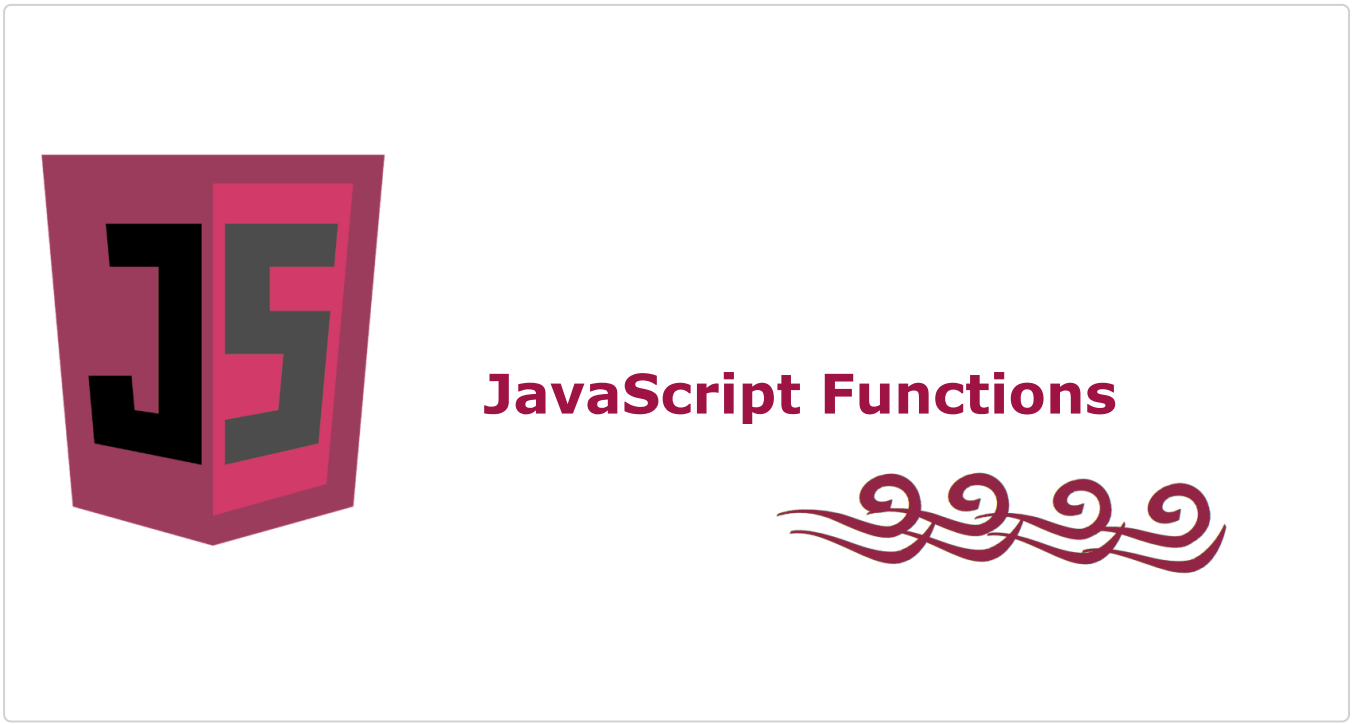
Functions

In this lesson, we'll get to know JavaScript functions.
Let's begin!

WE'LL COVER THE FOLLOWING



As you learned in the previous chapter, functions are very important concepts in JavaScript.

In this section, you'll dive deeper into the implementation of functions and get acquainted with many exciting features that make JavaScript functions powerful.

that make JavaScript functions powerful.

Function return values

Functions may retrieve results.

- You can use the `return` statement to pass back a result.
- If you specify `return` with no value, the function will immediately return without a value.
- If you utilize the result of a function that does not return a value, be prepared that you could get `undefined`.

Listing 8-3 demonstrates this behavior:

Listing 8-3: Using `return` in various ways in a function

```
<!DOCTYPE html>
<html>
<head>
  <title>Function return values</title>
  <script>
    function hasReturnValue() {
      return "Hey!";
    }

    function noReturnValue1() {
    }

    function noReturnValue2() {
      return;
    }

    console.log(hasReturnValue()); // "Hey!"
    console.log(noReturnValue1()); // undefined
    console.log(noReturnValue2()); // undefined
  </script>
</head>
<body>
  Listing 8-3: View the console output
</body>
</html>
```

Function arguments

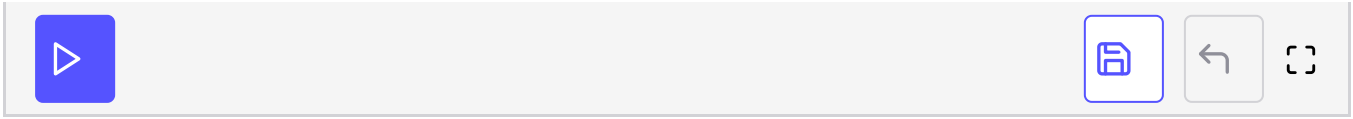
Functions may have arguments. In many languages, functions are identified by their signature that covers the name of the function and the type of their

arguments in the order of their occurrence. In JavaScript, a function is identified only by its name. It does not matter how many arguments you defined when declaring the function or how many parameters you pass when invoking the very same function.

If you pass more arguments than defined, the extra ones will be passed to the function, but the function will ignore them unless it is prepared to handle them, as you'll see soon. If you pass fewer arguments, the missing ones will have a value of undefined within the function body. So, this code is totally legal in JavaScript:

```
function twoArgs(arg1, arg2) {
  return arg1 + arg2;
}

console.log(twoArgs());           // NaN
console.log(twoArgs(12));         // NaN
console.log(twoArgs(12, 23));     // 35
console.log(twoArgs(12, 23, 34)); // 35
```



Only the **third call** uses the function as intended, but all the other calls are valid. The **first two calls** result in NaN, because either `arg1` or both arguments are undefined. The last invocation simply ignores the **third argument**.

This behavior implies that there is no function overload in JavaScript.

So, you cannot define two different functions with the same name but different number of parameters and use both of them.

The following code snippet demonstrates this:

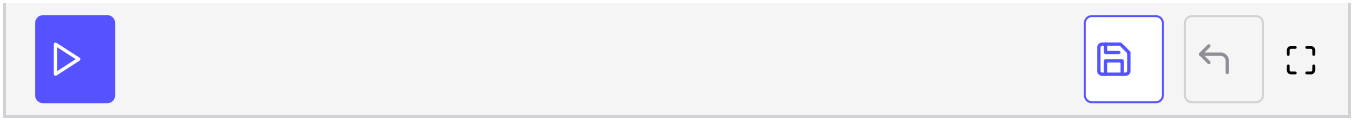
```
function add(arg1, arg2) {
  return arg1 + arg2;
}

console.log(add(12, 23)); // NaN

function add(arg1, arg2, arg3) {
  return arg1 + arg2 + arg3;
```

```
}

console.log(add(12, 23, 34)); // 69;
```



In JavaScript, it is totally legal to define a function twice, or even more times.

It looks as if there were two `add` functions, and the first `console.log()` uses the first one, the second `console.log()` uses the second one.

It does not work this way. Because of **function hoisting**, both log operations invoke the second function definition as this is the only one in this context, and it *overrides* the first one.

> 📜 **NOTE:** Just for a short recap: when a script is processed, the JavaScript engine recognizes named function declarations, and it automatically adds them to the context. This mechanism is called function hoisting. It means that a statement can refer to a function that is defined in the context only later.

Function arguments are internally represented by an **array**. This array is a real object, it is named `arguments`, and it can be accessed inside a function. The JavaScript engine takes care to assemble this array when a function is invoked.

The first element of this array (`arguments[0]`) represents the first argument, the second element (`arguments[1]`) holds the second argument, and so on.

Named arguments are interchangeable with their representation in the argument array.

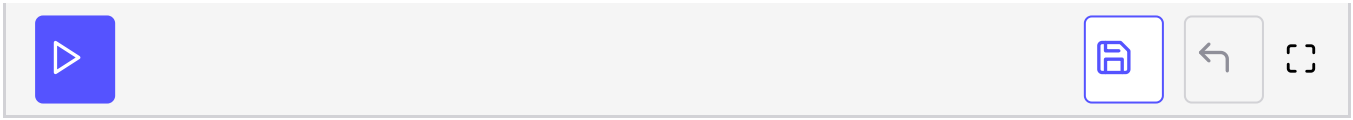
So, the following four functions are **equivalent**:

```
// Explicit arguments
function add(arg1, arg2) {
  return arg1 + arg2;
}
```

```
// Mixing explicit and implicit args
function add(arg1, arg2) {
  return arg1 + arguments[1];
}

function add(arg1, arg2) {
  return arguments[0] + arguments[1];
}

// Implicit arguments only
function add() {
  return arguments[0] + arguments[1];
}
```



You can also leverage the fact that `arguments` is an array instance.

The example in Listing 8-4 shows that you can easily write a function that sums up all passed arguments.

Listing 8-4: A function to sum up all passed arguments.

```
<!DOCTYPE html>
<html>
<head>
  <title>Function arguments</title>
  <script>
    function sum() {
      var result = 0;
      for (var i = 0; i < arguments.length; i++) {
        result += arguments[i];
      }
      return result;
    }

    console.log(sum(12));          // 12
    console.log(sum(12, 23));      // 35
    console.log(sum(12, 23, 34)); // 69
  </script>
</head>
<body>
  Listing 8-4: View the console output
</body>
</html>
```

Quiz time! :)

It’s time to test how much we’ve learned in this lesson with a short quiz!

Check your Understanding

1 What will the result if the following function is called?

```
sum(90, -90);
```

COMPLETED 0%

1 of 2

In the *next lesson*, we'll dive deeper into function expressions and how to exactly specify them.