Anonymous Functions and Callbacks

Two ways to declare a function:

1 - Standard function declaration

We've already seen that functions can be declared using this syntax:

```
function functionName(parameters) {
// code to be executed
}
```

A function declared this way can be called like this:

```
functionName(parameters);
```

Notice that we do not add a semicolon at the end of a function declaration. Semicolons are used to separate executable JavaScript statements, and a function declaration is not an executable statement.

Here is an example:

```
function sum(a, b) {
  // this function returns a result
  return (a + b);
}

function displayInPage(message, value) {
  // this function does not return anything
  document.body.innerHTML += message + value + "<br>";
}

var result = sum(3, 4);
displayInPage("Result: ", result);

// we could have written this
displayInPage("Result: ", sum(10, 15));
```

In the above example, the `sum` function returns a value, and the `displayInPage` function does not return anything.

2 - Use a function expression

A JavaScript function can also be defined using an expression that can be stored in a variable. Then, the variable can be used as a function:

Here is a typical example:

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

```
var displayInPage = function(message, value) {
  // this function does not return anything
  document.body.innerHTML += message + value + "<br>";
};

var result = sum(3, 4);
displayInPage("Result: ", result);

// we could have written this
displayInPage("Result: ", sum(10, 15));
```

Notice how the sum and `displayInPage` functions have been declared. We used a variable to store the function expression, then we can call the functions using the variable name. And we added a semicolon at the end, since we executed a JavaScript instruction, giving a value to a variable.

The "function expression" is an **anonymous function**, a function without a name, that represents a value that can be assigned to a variable. Then, **the variable can be used to execute the function.**

We say that functions are **first-class objects** which can be manipulated like any other object/value in JavaScript.

This means that functions can also be used as parameters to other functions. In this case they are called **callbacks**.

Callbacks

Indeed, as functions are first-class objects, we can pass a function as an argument, as a parameter to another function and later execute that passed-in function or even return it to be executed later. When we do this, we talk about **callback** *functions* in JavaScript**: a function passed to another function,** and executed inside the function we called.

All the examples of **event listeners** that you've seen **used callback functions**. Here is another one that registers mouse click listeners on the window object (the window objects represent the whole HTML document):

```
// Add a click event listener on the whole document
// the processClick function is a callback:
// a function called by the browser when a click event
// is detected
window.addEventListener('click',processClick);

function processClick(event){
 document.body.innerHTML += "Button clicked<br>";
}

// We could have written this, with the body of the callback as an argument of
the addEventListener function

window.addEventListener('click', function(evt) {
  document.body.innerHTML += "Button clicked version 2<br>";
});
```

In this case, the `processClick` function is passed as a parameter to the `addEventListener` method/function.

Callback functions are derived from a programming paradigm known as **functional programming**. They are very, very common in JavaScript. We'll use them a lot in **handling events**.