



# JavaScript and jQuery Course

## Instructor Teresa Pelkie

### Session 5

## Topic: How to use Functions and Events

Chapter 3: How to work with objects, functions, events — pages 100 to end

### Functions - pages 100-105

A *function* is a block of code that executes only when you tell it to execute. Functions can be executed when an event occurs (for example, when a user clicks a button), or called from within a script. Functions let you reuse code, rather than write the same (or very similar code) more than once.

Up to this point, we have just been executing JavaScript code when the page loads. Functions allow us to “package up” the code so that we can use it when we want to. We do this by *calling* or *invoking* the function when an *event* occurs.

*Events* are actions that can be detected by JavaScript. Every element on a web page has certain events which can trigger a function invocation in JavaScript. There are also events connected to the page itself. Events are connected to a function by using an *event handler*.

Because objects and elements on the page can respond to events and be affected by events, we **need a way to identify the objects and elements on the page**. As you will see later in this course, jQuery is the preferred and more modern choice for accessing page elements.

## Events - pages 106-109

Events are actions that can be detected by JavaScript. Events occur regardless of whether or not they are *intercepted* or *handled* by JavaScript. **Most events occur as a result of a user action, when the user does something.**

Every element on a web page has certain events which can trigger a JavaScript event. There are also events connected to the page itself.

### Examples of events:

- User clicks something on the page (a click event)
- The page is loaded (the page-load event)
- User moves the mouse over something on the page (a mouse-over event)
- User types a key (a keystroke event)
- User enters an input field
- User submits a form

### Event Handlers

Events are *handled* or processed by using an *event handler*. Some common events and event handlers are:

Event	Event Handler	Description
click	onclick	Invoked when a button or another element on the page is clicked
load	onload	Invoked when the page has finished loading
mouseover	onmouseover	When the mouse cursor is moved over an element on the page

### Coding an Event Handler

There are three ways to code event handlers in JavaScript, **but the best way is by using jQuery (as you will see later in this course)**

- As an **inline attribute** - writing the JavaScript directly inside the HTML tag (this is an old technique that you will see in older code, it is **not to be used** for new development)
- As a **property of the element** - in the <head> section (page 269)
- Using the DOM - by using the addEventListener method (page 269)
  - Example: `window.addEventListener("load", init, false);` // page 272

# Functions

A *function* is a block of code that executes only when you tell it to execute. **It can be invoked when an event occurs**, such as when a user clicks a button, or from a call within a script.

A function can accept one or more **parameters**, used to pass values to the function. Parameters are sometimes referred to as **arguments**. You can use either term.

## Declaring a Named Function

Functions are created using the JavaScript keyword `function`, which is lowercase. You then code a name for the function, followed by the opening and closing parentheses characters ( and ). Later in this handout, you will see examples of how to code parameters inside the parentheses.

Following the closing right parentheses character ), you code an opening left curly brace character {. That is used to indicate where the code for the function begins. Most functions will contain one or more executable statements. You can also include comments inside functions. After you code all of the statements for the function, you indicate the end of the function with the closing right curly brace character }.

You can put the opening brace character { on the same line as the function declaration, or you can put it on the next line. If you put the opening brace character { on the next line, it is customary to align the { character under the letter "f" in the word `function`. Most JavaScript developers will use only one opening brace style (either on the same line, or on the next line). It is OK to mix the two styles in your code. However, if you use the "next line" style, you should make it a point to always align the opening brace the same way. For example, if you want to indent the opening brace character { by four spaces, use that same indentation for every occurrence.

- If you code the opening brace character { on the **same line** as the function declaration: the closing brace character } should be aligned under the opening "f" in the word `function`.
- If you code the opening brace character { on the **line following** the function declaration: align the closing brace character } so that it is under the opening brace character.

Later, when you start to work with jQuery, you will see that you frequently need to include other "closing" characters immediately before or after the closing brace. When there are additional closing characters, you can put all of the closing characters on the same line, aligned with the outermost block of code that is being closed.

The idea of aligning the opening and closing brace characters is so that you can see at a glance where the function begins and where it ends. In small example programs where there is not a lot of code, it is easy to see where a function begins and ends, so it may not seem to be that important to align the characters. As you'll see, you may have JavaScript code that includes dozens of functions, with each function containing `if` blocks and looping blocks that also have opening and closing brace characters. When you have code that has a lot of *nested blocks*, it is much more important to pay attention to the alignment of the opening and closing brace characters, so that you can easily see where a block begins and ends.

Also, for each opening brace or parentheses character, **you must have a corresponding closing brace or parentheses character**. If you have a mismatch (too many closing braces/parentheses, or not enough), your code will end in error.

## "Golden Rule" of Opening Block Characters



Use this rule to always make sure you have the correct "match up" of opening and closing brace and parenthesis characters

As soon as you type an opening brace { or opening parenthesis ( character, **immediately** type the corresponding closing brace } or closing parenthesis ) character.

**Do not type any of the code that will be inside the block until you type the corresponding closing character.**

**After** you type the closing character, you can go back inside the block and start typing your statements.

## Function Example

This example shows the basic syntax for a function that does not have any parameters passed to it.

**Example 1:** Opening brace character on the same line.

```
<html>
<head>
  <script>

    // This function shows a message
    function showMessage() {
      // This is a comment inside the function
      alert("Hello world!");
    }

  </script>
</head>

<body>
</body>
</html>
```

**Example 2:** Opening brace character on the next line.

```
<html>
<head>
  <script>

    // This function shows a message
    function showMessage()
    {
      // This is a comment inside the function
      alert("Hello world!");
    }

  </script>
</head>

<body>
</body>
</html>
```

## Rules for the name of the function:

- Must start with a letter or an underscore
- Can contain letters, digits, and underscores in any combination
- Cannot contain a space (blank) character
- Cannot contain special characters

## Calling or Invoking a Function

Functions can be *called* or *invoked* in many ways. The most efficient way is by using jQuery, which we will be using in the second half of this course. **Functions are usually called in response to an event.** Some traditional methods of invoking a function are:

- Calling the function inside the `<script>` tags - **Inline Events**
- Using an event handler as an HTML attribute of an element in the `<body>` section (this is an old technique that you will see in older code, it is **not to be used** for new development)
- Using the event handler as a property of an object in the `<head>` section

## Traditional methods of calling a function

This is **not preferred** because the JavaScript code is mixed with HTML code. However, you may see many examples of this if you have to work with older code.

**Example 3:** Calling a function inside the body using the `<script>` tags.

```
<html>
<head>
  <script>

    function showMessage()
    {
      alert("Hello world!");
    }

  </script>
</head>

<body>
  <script>
    showMessage();
  </script>
</body>
</html>
```

**Example 4:** Using an event handler as an HTML attribute of an element in the `<body>` section. **This technique is not allowed in this course.**

```
<html>
<head>
  <script>

    function showMessage()
    {
      alert("Hello world!");
    }

  </script>
</head>

<body>
  <h2 onclick="showMessage();">Click Me!</h2>

  <form>
    <input type="button" value="Click Me Too!" onclick="showMessage();">
  </form>
</body>
</html>
```

## Preferred methods of calling a function

**Example 5:** Using the event handler as a property of an object in the <head> section.

```
<html>
<head>
  <script>
    function showMessage()
    {
      alert("Hello world!");
    }

    window.onload = showMessage;
  </script>
</head>

<body>
</body>
</html>
```

**Example 6:** In this example, all of the JavaScript code is contained in the <script> section. This is called **unobtrusive JavaScript**, meaning that all of the JavaScript code is separate from the content (the HTML). Ultimately, the code in the <script> section can be put into an external file.

```
<html>
<head>
  <script>
    function init()
    {
      function showMessage()
      {
        alert("Hello world!");
      }

      document.getElementById("button1").onclick = showMessage;
    }

    window.onload = init;
  </script>
</head>

<body>
  <form>
    <input type="button" id="button1" value="Click Me">
  </form>
</body>
</html>
```

**Note:** we use a function (init) to enclose the JavaScript and call it after the page is completely loaded. That way, the JavaScript code can access all of the document objects (in this example, the button1 object).

## Parameters and Arguments

The variables in a function definition are technically called *parameters*. The values passed to a function when it is invoked are called *arguments*. You can use either term.

### Examples of using parameters/arguments with functions

The parameter acts as a variable that will hold the value that is “passed” to it from some source. Nothing is passed to the function in this example.

**Example 7:** Defining a function that has one parameter. The parameter named *what* is used inside of the function.

```
<html>
<head>
  <script>
    function showMessage(what)
    {
      alert(what);
    }
  </script>
</head>

<body>
</body>
</html>
```

**Example 8:** Passing an argument to a function that uses one parameter

```
<html>
<head>
  <script>
    function showMessage(what)
    {
      alert(what);
    }

    window.onload = showMessage("Hello world");
  </script>
</head>

<body>
</body>
</html>
```

**Example 9:** Passing an argument to a function that uses one parameter, use *prompt* function to get the value to be passed.

```
<html>
<head>
  <script>
    var yourName = prompt("what is your name?", "Your Name");

    function showMessage(what)
    {
      alert(what);
    }

    window.onload = showMessage("Your Name is " + yourName);
  </script>
</head>

<body>
</body>
</html>
```

**Example 10:** Defining a function that uses two parameters.

```
<html>
<head>
  <script>
    function showMessage(what, who)
    {
      alert("I say " + what + " to you " + who);
    }

    window.onload = showMessage("Hello " , "Teresa");
  </script>
</head>

<body>
</body>
</html>
```

**Example 11:** Passing arguments to a function that uses two parameters, getting one of the values from the prompt function.

```
<html>
<head>
  <script>
    var yourName = prompt("What is your name?", "Your Name");

    function showMessage(what, who)
    {
      alert("I say " + what + " to you " + who);
    }

    window.onload = showMessage("Hello world", yourName);
  </script>
</head>

<body>
</body>
</html>
```

**Example 12:** A function can contain a lot of code and perform many calculations.

```
<html>
<head>
  <script>
    function calculateTheCost()
    {
      var itemPrice    = 50.00;
      var itemQuantity = 10;
      var theTotal     = itemPrice * itemQuantity;
      var theOutput    = "The Total is: $" + theTotal;

      document.getElementById("theDisplay").firstChild.nodeValue = theOutput;
    }

    window.onload = calculateTheCost;
  </script>
</head>

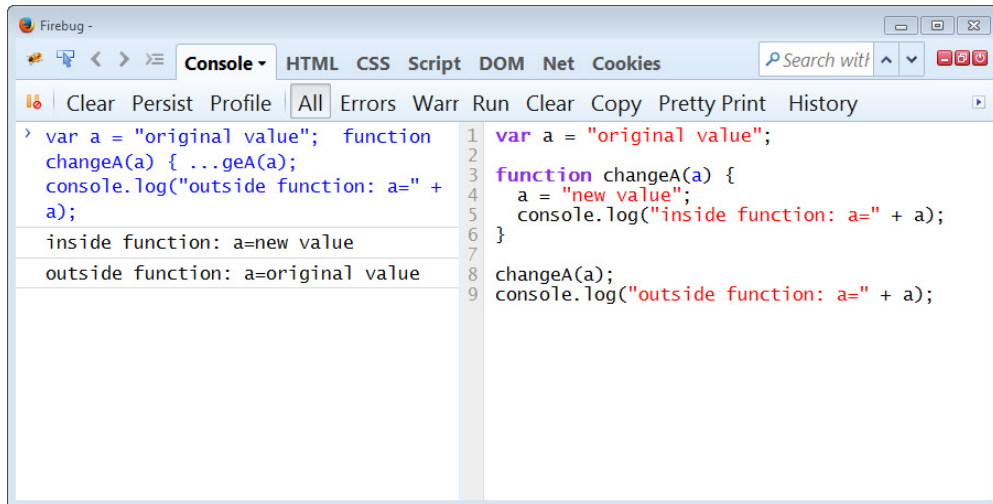
<body>
  <h2>Function Performs Calculation</h2>
  <span id="theDisplay"> </span>
</body>
</html>
```



## Passing "by value" and passing "by reference"

Most of the time, an argument is used to pass a value to the parameter, which is actually a variable that contains the value. When a parameter is passed *by value*, any changes that are made to the value of the parameter inside the function are not propagated back to the original argument.

For example, you can run the following code in Firebug and see that the value of the variable `a` remains the same, even though the value that is passed to the function is changed inside the function:



```
1 var a = "original value";
2 function changeA(a) {
3   a = "new value";
4   console.log("inside function: a=" + a);
5 }
6 changeA(a);
7 console.log("outside function: a=" + a);
```

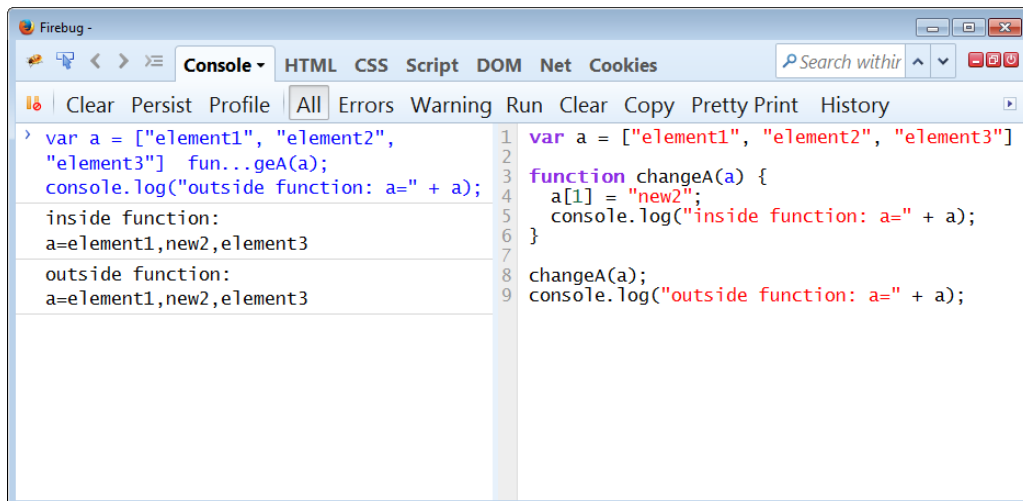
Console output:

```
inside function: a=new value
outside function: a=original value
```

jsjqh05001

Figure 1: This shows that a parameter passed "by value" is not changed upon returning from the function.

If you pass an array or an object reference to a function, a *reference* to the original argument is passed. If any changes are made to the parameter inside the function, the changes are propagated back to the original argument, as seen in the following Firebug example.



```
1 var a = ["element1", "element2", "element3"];
2 function changeA(a) {
3   a[1] = "new2";
4   console.log("inside function: a=" + a);
5 }
6 changeA(a);
7 console.log("outside function: a=" + a);
```

Console output:

```
inside function:
a=element1,new2,element3
outside function:
a=element1,new2,element3
```

jsjqh05002

Figure 2: An array is passed "by reference". Changes made to the array inside the function are also seen outside the function.

## Defining an anonymous function

Sometimes you will not need a name for a function, so you can define it as an *anonymous function* (a function with no name) as shown below.

**Example 13:** Defining an anonymous function. The anonymous function is invoked when the `window.onload` event occurs. The value of `yourName` (from the prompt function) is displayed.

```
<html>
<head>
  <script>
    var yourName = prompt("what is your name?", "Your Name");

    window.onload = function()
    {
      alert("Hello " + yourName);
    }
  </script>
</head>

<body>
</body>
</html>
```

## Returning a value from a function

A function can return a value. That is done by using the `return` statement. Upon encountering a `return` statement, the function stops executing and returns the specified value. Note, you do not need to return a value when you use the `return` statement, you can just code the `return` statement by itself to return from the function without returning a value.

You can have more than one `return` statement in a function, if you need to return from different parts of the function. It is generally considered to be a "best practice" to only have one `return` statement in a function (it is easier to understand the function when there is only one way for it to return).

**Example 14:** Return a value from a function that uses parameters. In this example, the `showSum` function returns the result of the calculation (`a` added to `b`).

```
<html>
<head>
  <script>
    function init()
    {
      function showSum(a, b)
      {
        return a + b;
      }

      document.getElementById("response").innerHTML = showSum(5, 8);
    }

    window.onload = init;
  </script>
</head>

<body>
  <h3 id="response"> </h3>
</body>
</html>
```

## Variable Scope - pages 104-105

### Local Scope

Variables defined inside a function using the keyword `var` exist only inside the function and can have the same name as a variable outside the function without interfering with that outside variable. This is referred to as *local scope*, since the variable is only known "locally" in the function in which it is defined.

You can have local variables with the same name in different functions, because local variables are only recognized by the function in which they are declared. Local variables are deleted ("go out of scope") upon returning from the function (either an explicit return with a return statement, or an implicit return by executing all of the statements in the function).

### Global Scope

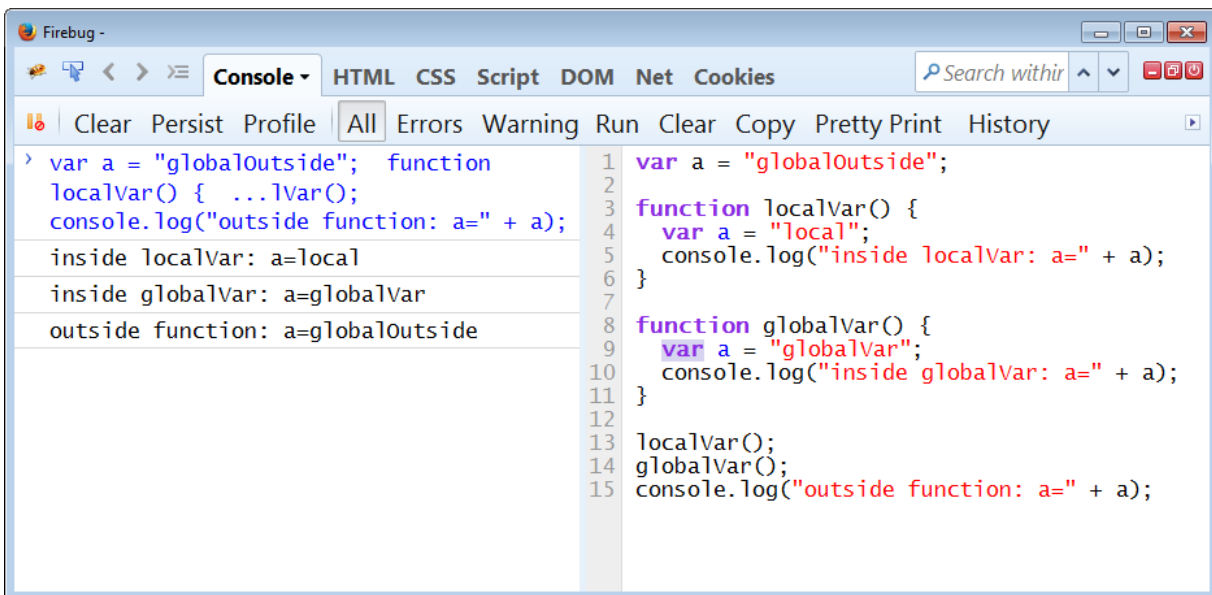
Variables defined outside functions have *global scope*. All scripts and functions on the web page can access variables that have global scope. Global variables are deleted when you close the page.

### Assigning values to undeclared JavaScript variables

If you assign a value to variable that has not yet been declared, the variable is **automatically declared as a global variable**. In other words, if a variable is defined inside a function without using the `var` keyword, it will have **global scope**.

☞ This is generally **not a good thing to do**, since it can be very difficult to track down where changes are made to the variable.

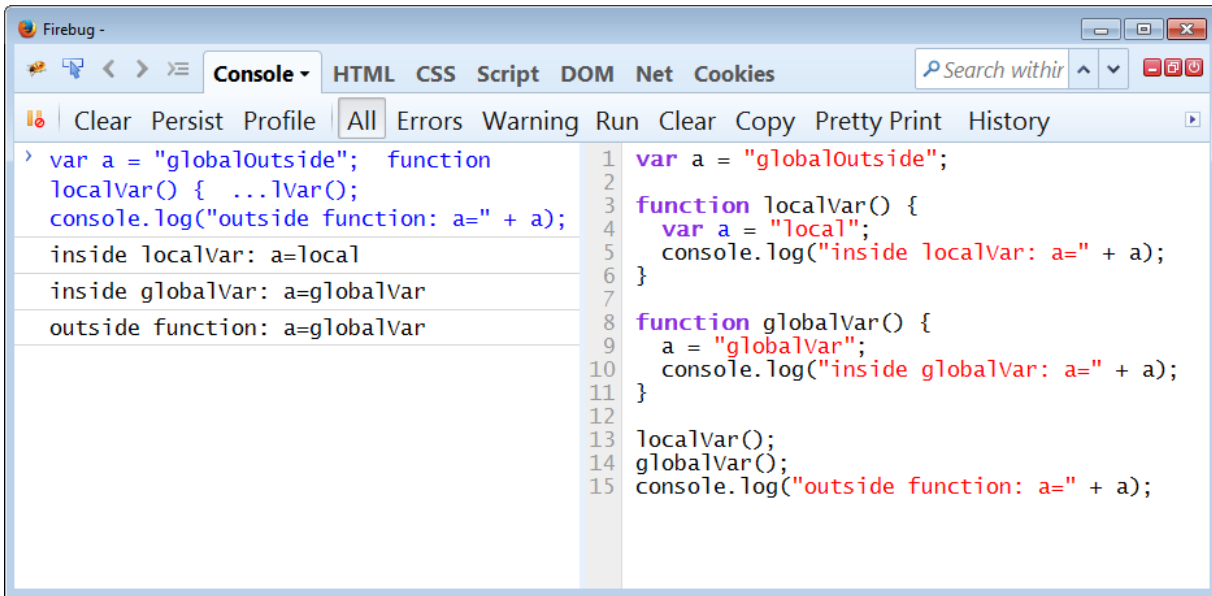
The following two examples in Firebug show how global scope and local scope act. In Figure 3, the variable named `a` is defined three times, as a global variable and inside each function as a local variable. When defined like this, setting the value inside the functions does not change the value of the global variable.



jsjqh05003

Figure 3: In this example, the variables inside the functions have local scope.

Figure 4 shows what happens when you forget to include the `var` keyword in the `globalVar` function. In this example, when the value is set in the `globalVar` function, it changes the value of the global variable `a` that is defined at the top of the code. You can see that in the output shown on the left side.



```
1 var a = "globalOutside"; function
2
3 function localVar() { ...lVar();
4   var a = "local";
5   console.log("inside localVar: a=" + a);
6 }
7
8 function globalVar() {
9   a = "globalVar";
10  console.log("inside globalVar: a=" + a);
11 }
12
13 localVar();
14 globalVar();
15 console.log("outside function: a=" + a);
```

Console output:

- outside function: a=globalOutside
- inside localVar: a=local
- inside globalVar: a=globalVar
- outside function: a=globalVar

jsjqh05004

Figure 4: In this example, the variable inside the function `globalVar` has global scope.

## Sample Files for this session

File Name	Description
05_assignment_sample1.html	Page that shows how to code a named function that validates values entered on the page and performs calculations using the values.
05_assignment_sample2.html	Page that shows how to code an anonymous function that validates values entered on the page and performs calculations using the values.
05_assignment_sample3.html	Page that shows how to code a named function that validates values entered on the page and performs calculations using the values.  The function invokes two other functions to perform the validations.
05_event_handler1.html	Page that shows how to code an anonymous function and attach it as the event handler for a button.
05_event_handler2.html	Page that shows how to code a named function and attach it as the event handler for a button.
05_event_handler_many.html	Page that shows how to code multiple event handler functions and assign them as event handlers to multiple functions.
05_function_anonymous_parameters.html	Page that shows how to code an anonymous function that takes two parameters and returns a value.
05_function_anonymous_reference_DOM_element.html	Page that shows how to code an anonymous function and how to get an object reference using the DOM and the \$ function.
05_function_anonymous1.html	Page that shows how to code an anonymous function that is assigned as the event handler for the window.onload event.
05_function_anonymous2.html	Page that shows how to code an anonymous function that is assigned as the event handler for the window.onload event.  The code inside the anonymous function is used to assign an event handler to a button onclick event.
05_function_named_parameters.html	Page that shows how to code a named function that takes two parameters, performs a calculation, and returns a value.
05_function_named1.html	Page that shows how to define and invoke a named function.

File Name	Description
05_function_named2.html	Page that shows how to define a named function and assign it as the event handler for a button onclick event.
05_function_return_value.html	Page that shows how to define a function that returns a value.
05_function_return_value_parameters.html	Page that shows how to define a function, passing parameters to the function.  The example shows multiple returns from the function.
05_textbook_email_application.html	This page is one of the example files from the textbook. It shows validation and error messages for three text fields on a form.
jsjqas03_starter_file.html	This file contains the "starter" HTML and CSS code for the MPG assignment.