# Internet and Web Principal

Week 10

JavaScript 2

# Content

1. Native and custom function
2. Browser objects
3. Evet handler
4. Document Object Model

# Native and custom function

**Function**

A function is a piece of code that performs a task but does not execute until it is referred or called. Our browser includes a method called alert(). It's a piece of code that only executes when we tell it to. In certain ways, we may conceive of a function as a variable containing logic, in that referring that variable will execute all of the code included within it. Functions allow code to be reused whenever it is referenced, eliminating the need to rewrite it.

# Native and custom function

The function name is always followed by a set of parentheses (no space), followed by a pair of curly brackets with the accompanying code. Parentheses may include extra information utilized by the function arguments. Arguments are pieces of information that can affect how the function performs. For example, the well-known alert() method receives a string of text as a parameter and utilizes that data to create the resultant window.

# Native and custom function

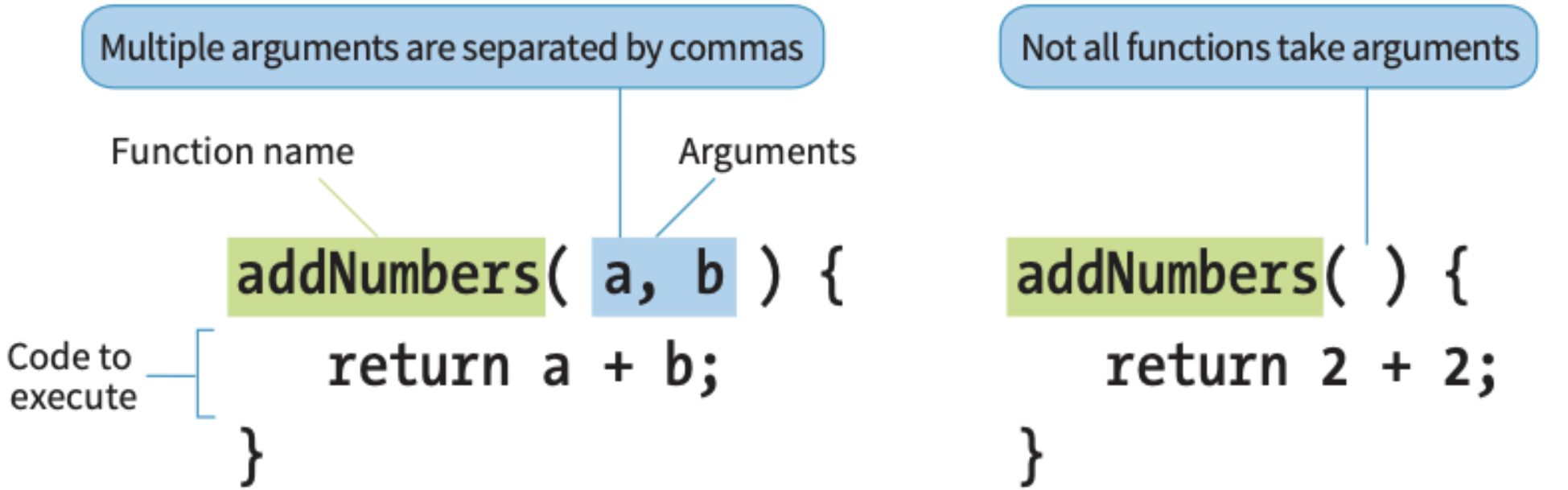


FIGURE 21-6. The structure of a function.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 608

# Native and custom function

## Native functions

Hundreds of predefined functions are built into JavaScript, including these:

**alert(), confirm(), *and* prompt()**
These functions trigger browser-level dialog boxes.

**Date()**

Returns the current date and time.

**parseInt("123")** This function will, among other things, take a string data type containing numbers and turn it into a number data type. The string is passed to the function as an argument.

**setTimeout(*functionName*, 5000)**

Executes a function after a delay. The function is specified in the first argument, and the delay is specified in milliseconds in the second argument (in the example, 5,000 milliseconds, which equals 5 seconds).

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 609

# Native and custom function

**Custom functions**

To create a custom function, we type the **function** keyword followed by a name for the function, followed by opening and closing parentheses, followed by opening and closing curly brackets:

```
function name() {
// Our function code goes here.
}
```

Just as with variables and arrays, the function's name can be anything you want, but all the same naming syntax rules apply.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 605

# Native and custom function

If we were to create a function that just alerts some text (which is a little redundant, I know), it would look like this:

```
function foo() {
alert("Our function just ran!");
// This code won't run until we call the function 'foo()'
}
```

We can then call that function and execute the code inside it anywhere in our script by writing the following:

```
foo(); // Alerts "Our function just ran!"
```

We can call this function any number of times throughout our code. It saves a lot of time and redundant coding.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 605

# Native and custom function

Arguments

A function that performs the same code throughout your script is unlikely to be very helpful. To apply a function's logic to distinct sets of data at different times, we may "pass arguments" (give data) to native and custom functions. Create a variable name (or a sequence of comma-separated names) in the parenthesis following the function name when the function is defined to keep a location for the arguments. For example, suppose we wanted to write a simple function that alerts us to the amount of elements in an array. We already know how to use.length to retrieve the number of elements in an array, so all we need now is a mechanism to feed the array to be measured into our function. We do this by passing the array to be measured as an input.

# Native and custom function

In the code, I've added a new function called alertArraySize() and a variable called arr to contain the parameter. That variable will then be accessible within the function and will hold any argument we supply when calling the function.

```
function alertArraySize(arr) {

}

alert(arr.length);
```

When we call that function, whatever we put between the parentheses following the function name (in this example, test) will be sent to the arr placeholder as the function executes. The variable test is defined here as a five-item array. After passing that variable to the method, the array is plugged in and the length is returned.

```
var test = [1,2,3,4,5]; alertArraySize(test); // Alerts "5"
```

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 610

# Native and custom function

## Returning a value

It’s pretty common to use a function to calculate something and then give you back a value that you can use elsewhere in your script. We could accomplish this using what we know now, through clever application of variables, but there’s a much easier way.

The **return** keyword inside a function effectively turns that function into a variable with a dynamic value! This one is a little easier to show than it is to tell, so bear with me while we consider this example:

```
function addNumbers(a,b) {
return a + b; }
```

Learning Web Design, Jennifer Niederst Robbins, O’Reilly Media, Inc., 2018, Page 610

# Native and custom function

We now have a function that accepts two arguments and adds them together. That wouldn't be much use if the result always lived inside that function, because we wouldn't be able to use the result anywhere else in our script. Here we use the **return** keyword to pass the result out of the function. Now any reference you make to that function gives you the result of the function— just like a variable would:

```
alert( addNumbers(2,5) ); // Alerts "7"
```

In a way, the **addNumbers()** function is now a variable that contains a dynamic value: the value of our calculation. If we didn't return a value inside our func- tion, the preceding script would alert **undefined**, just like a variable that we haven't given a value.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 610

# Native and custom function

The **return** keyword has one catch. As soon as JavaScript sees that it’s time to return a value, the function ends. Consider the following:

function bar() {

return 3;

alert("We'll never see this alert."); }

When you call this function by using **bar()**, the alert on the second line never runs. The function ends as soon as it sees it’s time to return a value.

Learning Web Design, Jennifer Niederst Robbins, O’Reilly Media, Inc., 2018, Page 610

# Browser Object

In addition to being able to control elements on a web page, JavaScript also gives you access to and the ability to manipulate the parts of the browser window itself. For example, you might want to get or replace the URL that is in the browser's address bar, or open or close a browser window.

In JavaScript, the browser is known as the **window** object. The **window** object has a number of properties and methods that we can use to interact with it. TABLE 21-1 lists just a few of the properties and methods that can be used with **window** to give you an idea of what's possible. For a complete list, see the Window API reference at MDN Web Docs (*https://developer.mozilla.org/en-US/docs/Web/API/Window*).

# Browser Object

TABLE 21-1. Browser properties and methods.

| Property/method | Description |
|---|---|
| `event` | Represents the state of an event |
| `history` | Contains the URLs the user has visited within a browser window |
| `location` | Gives read/write access to the URI in the address bar |
| `status` | Sets or returns the text in the status bar of the window |
| `alert()` | Displays an alert box with a specified message and an OK button |
| `close()` | Closes the current window |
| `confirm()` | Displays a dialog box with a specified message and an OK and a Cancel button |
| `focus()` | Sets focus on the current window |

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 613

# Event Handler

JavaScript can access objects in the page and the browser window, but did you know it's also "listening" for certain events to happen? An event is an action that can be detected with JavaScript, such as when the document loads or when the user clicks an element or just moves her mouse over it. HTML 4.0 made it possible for a script to be tied to events on the page, whether initi- ated by the user, the browser itself, or other scripts. This is known as event binding. TABLE 21-2 lists some of the most common event han- dlers. Keep in mind that these are also case-sensitive.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 613

# Event Handler

**TABLE 21-2.** Common events.

| Event handler | Event description |
|---|---|
| onblur | An element loses focus. |
| onchange | The content of a form field changes. |
| onclick | The mouse clicks an object. |
| onerror | An error occurs when the document or an image loads. |
| onfocus | An element gets focus. |
| onkeydown | A key on the keyboard is pressed. |
| onkeypress | A key on the keyboard is pressed or held down. |
| onkeyup | A key on the keyboard is released. |
| onload | A page or an image is finished loading. |
| onmousedown | A mouse button is pressed. |
| onmousemove | The mouse is moved. |
| onmouseout | The mouse is moved off an element. |
| onmouseover | The mouse is moved over an element. |
| onmouseup | A mouse button is released. |
| onsubmit | The submit button is clicked in a form. |

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 613

# Event Handler

There are three common methods for applying event handlers to items within our pages:

As an HTML attribute

As a method attached to the element

Using**addEventListener()**

**As an HTML Attribute**

You can specify the function to be run in an attribute in the markup, as shown in the following example:

```
<body onclick="myFunction();"> /* myFunction will now run when the user clicks anything within 'body' */
```

Although still functional, this is an antiquated way of attaching events to elements within the page. It should be avoided for the same reason we avoid using **style** attributes in our markup to apply styles to individual elements.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 615

# Event Handler

**As a Method**

This is another somewhat dated approach to attaching events, though it does keep things strictly within our scripts. We can attach functions by using help- ers already built into JavaScript:

```
window.onclick = myFunction; /* myFunction will run when the user clicks anything within the browser window */
```

We can also use an anonymous function rather than a predefined one:

```
window.onclick = function() {

/* Any code placed here will run when the user clicks anything within the browser window */
};
```

This approach has the benefit of both simplicity and ease of maintenance, but does have a fairly major drawback: we can bind only one event at a time with this method.

```
window.onclick = myFunction; window.onclick = myOtherFunction;
```

In the example just shown, the second binding overwrites the first, so when the user clicks inside the browser window, only **myOtherFunction** will run. The reference to **myFunction** is thrown away.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 615

# Event Handler

**addEventListener**

Although a little more complex at first glance, this approach allows us to keep our logic within our scripts and allows us to perform multiple bindings on a single object. The syntax is a bit more verbose. We start by calling the **addEventListener()** method of the target object, and then specify the event in question and the function to be executed as two arguments:

window.addEventListener("click", myFunction);

Notice that we omit the preceding “on” from the event handler with this syntax.

Like the previous method, **addEventListener()** can be used with an anony- mous function as well:

window.addEventListener("click", function(e) { });

Learning Web Design, Jennifer Niederst Robbins, O’Reilly Media, Inc., 2018, Page 615

# Document Object Model

The DOM is a programming interface (an API) for HTML and XML pages. It provides a structured map of the document, as well as a set of methods to interface with the elements contained therein. Effectively, it translates our markup into a format that JavaScript (and other languages) can understand. The basic gist is that the DOM serves as a map to all the elements on a page and lets us *do* things with them. We can use it to find elements by their names or attributes, and then add, modify, or delete elements and their content.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 621

# Document Object Model

**Accessing DOM Nodes**

The **document** object in the DOM identifies the page itself, and more often than not will serve as the starting point for our DOM crawling. The **document** object comes with a number of standard properties and methods for accessing collections of elements.

To give you a general idea of what I mean, the statement in this example says to look on the page (**document**), find the element that has the **id** value “beginner”, find the HTML content within that element (**innerHTML**), and save those contents to a variable (**foo**):

```
var foo = document.getElementById("beginner").innerHTML;
```

Because the chains tend to get long, it is also common to see each property or method broken onto its own line to make it easier to read at a glance.

Learning Web Design, Jennifer Niederst Robbins, O’Reilly Media, Inc., 2018, Page 623

# Document Object Model

**By id attribute value**

**getElementById()** This method returns a single element based on that element's ID (the value of its **id** attribute), which we provide to the method as an argument. For example, to access this particular image

<img src="photo.jpg" alt="" id="lead-photo">

we include the **id** value as an argument for the **getElementById()** method: var photo = document.getElementById("lead-photo");

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 625

# Document Object Model

**By class attribute value**

**getElementsByClassName()**

Just as it says on the tin, this allows you to access nodes in the document based on the value of a **class** attribute. This statement assigns any element with a **class** value of “column-a” to the variable **firstColumn** so it can be accessed easily from within a script:

```
var firstColumn = document.getElementsByClassName("column-a");
```

Like **getElementsByTagName()**, this returns a nodeList that we can reference

by index or loop through one at a time.

Learning Web Design, Jennifer Niederst Robbins, O’Reilly Media, Inc., 2018, Page 625

# Document Object Model

**By selector**

**querySelectorAll()**

**querySelectorAll()** allows you to access nodes of the DOM based on a CSS-style selector. The syntax of the arguments in the following examples should look familiar to you. It can be as simple as accessing the child elements of a specific element:

var sidebarPara = document.querySelectorAll(".sidebar p"); or as complex as selecting an element based on an attribute:

var textInput = document.querySelectorAll("input[type='text']");
**querySelectorAll()** returns a nodeList, like **getElementsByTagName()** and

**getElementsByClassName()**, even if the selector matches only a single element.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 625

# Document Object Model

## Accessing an attribute value

### getAttribute()

As I mentioned earlier, elements aren't the only thing you can access with the DOM. To get the value of an attribute attached to an element node, we call **getAttribute()** with a single argument: the attribute name. Let's assume we have an image, *stratocaster.jpg*, marked up like this:

```
<img src="stratocaster.jpg" alt="electric guitar" id="lead-image">
```

In the following example, we access that specific image (**getElementbyId()**) and save a reference to it in a variable ("bigImage"). At that point, we could access any of the element's attributes (**alt**, **src**, or **id**) by specifying it as an argument in the **getAttribute()** method. In the example, we get the value of the **src** attribute and use it as the content in an alert message.

```
var bigImage = document.getElementById("lead-image");
alert( bigImage.getAttribute("src") ); // Alerts "stratocaster.jpg".
```

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 625

# Document Object Model

**setAttribute()** To continue with the previous example, we saw how we *get* the attribute value, but what if we wanted to *set* the value of that **src** attribute to a new pathname altogether? Use **setAttribute()**! This method requires two argu- ments: the attribute to be changed and the new value for that attribute.

In this example, we use a bit of JavaScript to swap out the image by changing the value of the **src** attribute:

```
var bigImage = document.getElementById("lead-image");
bigImage.setAttribute("src", "lespaul.jpg");
```

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 625

# Document Object Model

Just think of all the things you could do with a document by changing the values of attributes. Here we swapped out an image, but we could use this same method to make a number of changes throughout our document:

Update the **checked** attributes of checkboxes and radio buttons based on user interaction elsewhere on the page.

Findthe**link**elementforour.*css*fileandpointthe**href**valuetoadiffer- ent style sheet, changing all the page's styles.

Update a **title** attribute with information on an element's state ("this element is currently selected," for example).

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 625

# Document Object Model

## innerHTML

**innerHTML** gives us a simple method for accessing and changing the text and markup inside an element. It behaves differently from the methods we've covered so far. Let's say we need a quick way of adding a paragraph of text to the first element on our page with a class of **intro**:

```
var introDiv = document.getElementsByClassName("intro");

introDiv[0].innerHTML = "<p>This is our intro text</p>";
```

The second statement here adds the content of the string to **introDiv** (an element with the **class** value "intro") as a *real live element* because **innerHTML** tells JavaScript to parse the strings " " and " " as markup.

Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, Page 625

# Thank You

Alfred Tenggono, S.Kom., M.Kom.

alfred.tenggono@jiu.ac

# Reference


- Learning Web Design, Jennifer Niederst Robbins, O'Reilly Media, Inc., 2018, ISBN: 978-1-491-96020-2