Javascript

What You Can Do with JavaScript

There are lot more things you can do with JavaScript.

- You can modify the content of a web page by adding or removing elements.
- You can change the style and position of the elements on a web page.
- You can monitor events like mouse click, hover, etc. and react to it.
- You can perform and control transitions and animations.
- You can create alert pop-ups to display info or warning messages to the user.
- You can perform operations based on user inputs and display the results.
- You can validate user inputs before submitting it to the server.

Adding JavaScript to Your Web Pages

There are typically three ways to add JavaScript to a web page:

- Embedding the JavaScript code between a pair of `<script>` and `</script>` tag.
- Creating an external JavaScript file with the `.js` extension and then load it within the page through the `src` attribute of the `<script>` tag.
- Placing the JavaScript code directly inside an HTML tag using the special tag attributes such as `onclick`, `onmouseover`, `onkeypress`, `onload`, etc.

The following sections will describe each of these procedures in detail:

Embedding the JavaScript Code

You can embed the JavaScript code directly within your web pages by placing it between the `<script>` and `</script>` tags. The `<script>` tag indicates the browser that the contained statements are to be interpreted as executable script and not HTML. Here's an example:

ExampleTry this code »

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Embedding JavaScript</title>
</head>
<body>
<script>
    let greet = "Hello World!";
    document.write(greet); // Prints: Hello World!
    </script>
</body>
</html>
```

The JavaScript code in the above example will simply prints a text message on the web page. You will learn what each of these JavaScript statements means in upcoming chapters.

Note: The `type` attribute for `<script>` tag (i.e. `<script type="text/javascript">`) is no longer required since HTML5. JavaScript is the default scripting language for HTML5.

Calling an External JavaScript File

You can also place your JavaScript code into a separate file with a `.js` extension, and then call that file in your document through the `src` attribute of the `<script>` tag, like this:

<script src="js/hello.js"></script>

This is useful if you want the same scripts available to multiple documents. It saves you from repeating the same task over and over again, and makes your website much easier to maintain.

Well, let's create a JavaScript file named "hello.js" and place the following code in it:

ExampleTry this code »

```
// A function to display a message
function sayHello() {
    alert("Hello World!");
}

// Call function on click of the button
document.getElementById("myBtn").onclick = sayHello;
```

Now, you can call this external JavaScript file within a web page using the `<script>` tag, like this:

ExampleTry this code »

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Including External JavaScript File</title>
</head>
<body>
<button type="button" id="myBtn">Click Me</button>
<script src="js/hello.js"></script>
</body>
</html>
```

Note: Usually when an external JavaScript file is downloaded for first time, it is stored in the browser's cache (just like images and style sheets), so it won't need to be downloaded multiple times from the web server that makes the web pages load more quickly.

Placing the JavaScript Code Inline

You can also place JavaScript code inline by inserting it directly inside the HTML tag using the special tag attributes such as `onclick`, `onmouseover`, `onkeypress`, `onload`,

etc.

However, you should avoid placing large amount of JavaScript code inline as it clutters up your HTML with JavaScript and makes your JavaScript code difficult to maintain. Here's an example:

ExampleTry this code »

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Inlining JavaScript</title>
</head>
<body>
<button onclick="alert('Hello World!')">Click Me</button></bo
dy>
</html>
```

The above example will show you an alert message on click of the button element.

Tip: You should always keep the content and structure of your web page (i.e. HTML) separate out from presentation (CSS), and behavior (JavaScript).

Positioning of Script inside HTML Document

The `<script>` element can be placed in the `<head>`, or `<body>` section of an HTML document. But ideally, scripts should be placed at the end of the body section, just before the closing `</body>` tag, it will make your web pages load faster, since it prevents obstruction of initial page rendering.

Each `<script>` tag blocks the page rendering process until it has fully downloaded and executed the JavaScript code, so placing them in the head section (i.e. `<head>` element) of the document without any valid reason will significantly impact your website performance.

Tip: You can place any number of `<script>` element in a single document. However, they are processed in the order in which they appear in the document, from top to bottom.

Difference Between Client-side and Server-side Scripting

Client-side scripting languages such as JavaScript, VBScript, etc. are interpreted and executed by the web browser, while server-side scripting languages such as PHP, ASP, Java, Python, Ruby, etc. runs on the web server and the output sent back to the web browser in HTML format.

Client-side scripting has many advantages over traditional server-side scripting approach. For example, you can use JavaScript to check if the user has entered invalid data in form fields and show notifications for input errors accordingly in real-time before submitting the form to the web-server for final data validation and further processing in order to prevent unnecessary network bandwidth usages and the exploitation of server system resources.

Also, response from a server-side script is slower as compared to a client-side script, because server-side scripts are processed on the remote computer not on the user's local computer.

You can learn more about server-side scripting in PHP tutorial section.

JavaScript Syntax

In this tutorial you will learn how to write the JavaScript code.

Understanding the JavaScript Syntax

The syntax of JavaScript is the set of rules that define a correctly structured JavaScript program.

A JavaScript consists of JavaScript statements that are placed within the `<script>` `</script>` HTML tags in a web page, or within the external JavaScript file having `.js` extension.

The following example shows how JavaScript statements look like:

ExampleTry this code »

```
let x = 5;
let y = 10;
let sum = x + y;
document.write(sum); // Prints variable value
```

You will learn what each of these statements means in upcoming chapters.

Case Sensitivity in JavaScript

JavaScript is case-sensitive. This means that variables, language keywords, function names, and other identifiers must always be typed with a consistent capitalization of letters.

For example, the variable `myVar` must be typed `myVar` not `MyVar` or `myvar`. Similarly, the method name `getElementById()` must be typed with the exact case not as `getElementByID()`.

ExampleTry this code »

```
let myVar = "Hello World!";
console.log(myVar);
console.log(MyVar);
console.log(myvar);
```

If you checkout the browser console by pressing the `f12` key on the keyboard, you'll see a line something like this: "Uncaught ReferenceError: MyVar is not defined".

JavaScript Comments

A comment is simply a line of text that is completely ignored by the JavaScript interpreter. Comments are usually added with the purpose of providing extra information pertaining to source code. It will not only help you understand your

code when you look after a period of time but also others who are working with you on the same project.

JavaScript support single-line as well as multi-line comments. Single-line comments begin with a double forward slash (`//`), followed by the comment text. Here's an example:

ExampleTry this code »

```
// This is my first JavaScript program
document.write("Hello World!");
```

Whereas, a multi-line comment begins with a slash and an asterisk (`/*`) and ends with an asterisk and slash (`*/`). Here's an example of a multi-line comment.

ExampleTry this code »

```
/* This is my first program
in JavaScript */
document.write("Hello World!");
```

JavaScript Variables

In this tutorial you will learn how to create variables to store data in JavaScript.

What is Variable?

Variables are fundamental to all programming languages. Variables are used to store data, like string of text, numbers, etc. The data or value stored in the variables can be set, updated, and retrieved whenever needed. In general, variables are symbolic names for values.

There are three ways to declare variables in JavaScript: `var`, `let` and `const`.

The `var` keyword is the older way of declaring variables, whereas the `let` and `const` keywords are introduced in JavaScript ES6. The main difference between them is the variables declared with the `let` and `const` keywords are **block scoped** (`{}`), that means they will only be

available inside the code blocks (functions, loops and conditions) where they are declared and it sub-blocks, whereas the variables declared with the `var` keyword are **function scoped** or **globally scoped**, depending on whether they are declared within a function or outside of any function.

We'll learn more about them in upcoming chapters. Now let's take a look at the following example where we've created some variables with the `let` keyword, and simply used the assignment operator (`=`) to assign values to them, like this: `let varName = value;`

ExampleTry this code »

```
let name = "Peter Parker";
let age = 21;
let isMarried = false;
```

Tip: Always give meaningful names to your variables. Additionally, for naming the variables that contain multiple words, camelCase is commonly used. In this convention all words after the first should have uppercase first letters, e.g. `myLongVariableName`.

In the above example we have created three variables, first one has assigned with a string value, the second one has assigned with a number, whereas the last one assigned with a boolean value. Variables can hold different types of data, we'll learn about them in later chapter.

In JavaScript, variables can also be declared without having any initial values assigned to them. This is useful for variables which are supposed to hold values like user inputs.

ExampleTry this code »

```
// Declaring Variable
let userName;

// Assigning value
userName = "Clark Kent";
```

Note: In JavaScript, if a variable has been declared, but has not been assigned a value explicitly, is automatically assigned the value `undefined`.

The `const` keyword works exactly the same as `let`, except that variables declared using `const` keyword cannot be reassigned later in the code. Here's an example:

ExampleTry this code »

```
// Declaring constant
const PI = 3.14;
console.log(PI); // 3.14

// Trying to reassign
PI = 10; // error
```

Note: The `let` and `const` keywords are not supported in older browsers like IE10. IE11 support them partially. See the JS ES6 features chapter to know how to start using ES6 today.

Declaring Multiple Variables at Once

In addition, you can also declare multiple variables and set their initial values in a single statement. Each variable are separated by commas, as demonstrated in the following example:

ExampleTry this code »

```
// Declaring multiple Variables
let name = "Peter Parker", age = 21, isMarried = false;

/* Longer declarations can be written to span
multiple lines to improve the readability */
let name = "Peter Parker",
age = 21,
isMarried = false;
```

Naming Conventions for JavaScript Variables

These are the following rules for naming a JavaScript variable:

- A variable name must start with a letter, underscore (`_`), or dollar sign (`$`).
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters (`A-z`, `0-9`) and underscores.
- A variable name cannot contain spaces.
- A variable name cannot be a JavaScript keyword or a JavaScript reserved word.

Note: Variable names in JavaScript are case sensitive, it means `$myvar` and `$myVar` are two different variables. So be careful while defining variable names.

JavaScript Generating Output

In this tutorial you will learn how to generate outputs in JavaScript.

Generating Output in JavaScript

There are certain situations in which you may need to generate output from your JavaScript code. For example, you might want to see the value of variable, or write a message to browser console to help you debug an issue in your running JavaScript code, and so on.

In JavaScript there are several different ways of generating output including writing output to the browser window or browser console, displaying output in dialog boxes, writing output into an HTML element, etc. We'll take a closer look at each of these in the following sections.

Writing Output to Browser Console

You can easily outputs a message or writes data to the browser console using the `console.log()` method. This is a simple, but very powerful method for generating detailed output. Here's an example:

ExampleTry this code »

```
// Printing a simple text message
console.log("Hello World!"); // Prints: Hello World!

// Printing a variable value
let x = 10;
let y = 20;
let sum = x + y;
console.log(sum); // Prints: 30
```

Tip: To access your web browser's console, first press F12 key on the keyboard to open the *developer tools* then click on the console tab. It looks something like the screenshot here.

Displaying Output in Alert Dialog Boxes

You can also use alert dialog boxes to display the message or output data to the user. An alert dialog box is created using the `alert()` method. Here's is an example:

ExampleTry this code »

```
// Displaying a simple text message
alert("Hello World!"); // Outputs: Hello World!

// Displaying a variable value
let x = 10;
let y = 20;
let sum = x + y;
alert(sum); // Outputs: 30
```

Writing Output to the Browser Window

You can use the `document.write()` method to write the content to the current document only while that document is being parsed. Here's an example:

ExampleTry this code »

```
// Printing a simple text message
document.write("Hello World!"); // Prints: Hello World!

// Printing a variable value
let x = 10;
let y = 20;
let sum = x + y;
document.write(sum); // Prints: 30
```

If you use the `document.write()` method method after the page has been loaded, it will overwrite all the existing content in that document. Check out the following example:

ExampleTry this code »

```
<h1>This is a heading</h1><p>This is a paragraph of text.</p>
<button type="button" onclick="document.write('Hello Worl
d!')">Click Me</button>
```

Inserting Output Inside an HTML Element

You can also write or insert output inside an HTML element using the element's `innerHTML` property. However, before writing the output first we need to select the element using a method such as `getElementById()`, as demonstrated in the following example:

ExampleTry this code »

```
<p id="greet"></p><p id="result"></p><script>
// Writing text string inside an element
```

```
document.getElementById("greet").innerHTML = "Hello World!";

// Writing a variable value inside an element
let x = 10;
let y = 20;
let sum = x + y;
document.getElementById("result").innerHTML = sum;
</script>
```

JavaScript Data Types

In this tutorial you will learn about the data types available in JavaScript.

Data Types in JavaScript

Data types basically specify what kind of data can be stored and manipulated within a program.

There are six basic data types in JavaScript which can be divided into three main categories: primitive (or *primary*), *composite* (or *reference*), and *special* data types. String, Number, and Boolean are primitive data types. Object, Array, and Function (which are all types of objects) are composite data types. Whereas Undefined and Null are special data types.

Primitive data types can hold only one value at a time, whereas composite data types can hold collections of values and more complex entities. Let's discuss each one of them in detail.

The String Data Type

The *string* data type is used to represent textual data (i.e. sequences of characters). Strings are created using single or double quotes surrounding one or more characters, as shown below:

ExampleTry this code »

```
let a = 'Hi there!';  // using single quotes
let b = "Hi there!";  // using double quotes
```

You can include quotes inside the string as long as they don't match the enclosing quotes.

ExampleTry this code »

```
let a = "Let's have a cup of coffee."; // single quote inside
double quotes
let b = 'He said "Hello" and left.';  // double quotes inside
single quotes
let c = 'We\'ll never give up.';     // escaping single quote
with backslash
```

You will learn more about the strings in JavaScript strings chapter.

The Number Data Type

The *number* data type is used to represent positive or negative numbers with or without decimal place, or numbers written using exponential notation e.g. 1.5e-4 (equivalent to 1.5×10-4).

ExampleTry this code »

```
let a = 25;         // integer
let b = 80.5;       // floating-point number
let c = 4.25e+6;    // exponential notation, same as 4.25e6 o
r 4250000
let d = 4.25e-6;    // exponential notation, same as 0.000004
25
```

The Number data type also includes some special values which are: `Infinity`, `-Infinity` and `NaN`. Infinity represents the mathematical Infinity `∞`, which is greater than any number. Infinity is the result of dividing a nonzero number by 0, as demonstrated below:

ExampleTry this code »

```
alert(16 / 0);  // Output: Infinity
alert(-16 / 0); // Output: -Infinity
alert(16 / -0); // Output: -Infinity
```

While `NaN` represents a special *Not-a-Number* value. It is a result of an invalid or an undefined mathematical operation, like taking the square root of -1 or dividing 0 by 0, etc.

ExampleTry this code »

```
alert("Some text" / 2);       // Output: NaN
alert("Some text" / 2 + 10);  // Output: NaN
alert(Math.sqrt(-1));         // Output: NaN
```

You will learn more about the numbers in JavaScript numbers chapter.

The Boolean Data Type

The Boolean data type can hold only two values: `true` or `false`. It is typically used to store values like yes (`true`) or no (`false`), on (`true`) or off (`false`), etc. as demonstrated below:

ExampleTry this code »

```
let isReading = true;   // yes, I'm reading
let isSleeping = false; // no, I'm not sleeping
```

Boolean values also come as a result of comparisons in a program. The following example compares two variables and shows the result in an alert dialog box:

ExampleTry this code »

```
let a = 2, b = 5, c = 10;
```

```
alert(b > a) // Output: true
alert(b > c) // Output: false
```

You will learn more about the comparisons in JavaScript if/else chapter.

The Undefined Data Type

The undefined data type can only have one value-the special value `undefined`. If a variable has been declared, but has not been assigned a value, has the value `undefined`.

ExampleTry this code »

```
let a;
let b = "Hello World!";

alert(a) // Output: undefined
alert(b) // Output: Hello World!
```

The Null Data Type

This is another special data type that can have only one value-the `null` value. A `null` value means that there is no value. It is not equivalent to an empty string (`""`) or 0, it is simply nothing.

A variable can be explicitly emptied of its current contents by assigning it the `null` value.

ExampleTry this code »

```
let a = null;
alert(a); // Output: null

let b = "Hello World!";
alert(b); // Output: Hello World!
```

```
b = null;
alert(b) // Output: null
```

The Object Data Type

The `object` is a complex data type that allows you to store collections of data.

An object contains properties, defined as a key-value pair. A property key (name) is always a string, but the value can be any data type, like strings, numbers, booleans, or complex data types like arrays, function and other objects. You'll learn more about objects in upcoming chapters.

The following example will show you the simplest way to create an object in JavaScript.

ExampleTry this code »

```
let emptyObject = {};
let person = {"name": "Clark", "surname": "Kent", "age": "3
6"};

// For better reading
let car = {
    "modal": "BMW X3",
    "color": "white",
    "doors": 5
}
```

You can omit the quotes around property name if the name is a valid JavaScript name. That means quotes are required around `"first-name"` but are optional around `firstname`. So the car object in the above example can also be written as:

ExampleTry this code »

```
let car = {
    modal: "BMW X3",
    color: "white",
```

```
    doors: 5
}
```

You will learn more about the objects in JavaScript objects chapter.

The Array Data Type

An array is a type of object used for storing multiple values in single variable. Each value (also called an element) in an array has a numeric position, known as its index, and it may contain data of any data type-numbers, strings, booleans, functions, objects, and even other arrays. The array index starts from 0, so that the first array element is `arr[0]` not `arr[1]`.

The simplest way to create an array is by specifying the array elements as a comma-separated list enclosed by square brackets, as shown in the example below:

ExampleTry this code »

```
let colors = ["Red", "Yellow", "Green", "Orange"];
let cities = ["London", "Paris", "New York"];

alert(colors[0]);   // Output: Red
alert(cities[2]);   // Output: New York
```

You will learn more about the arrays in JavaScript arrays chapter.

The Function Data Type

The function is callable object that executes a block of code. Since functions are objects, so it is possible to assign them to variables, as shown in the example below:

ExampleTry this code »

```
let greeting = function(){
    return "Hello World!";
```

```
}

// Check the type of greeting variable
alert(typeof greeting) // Output: function
alert(greeting());     // Output: Hello World!
```

In fact, functions can be used at any place any other value can be used. Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to other functions, and functions can be returned from functions. Consider the following function:

ExampleTry this code »

```
function createGreeting(name){
    return "Hello, " + name;
}
function displayGreeting(greetingFunction, userName){
    return greetingFunction(userName);
}

let result = displayGreeting(createGreeting, "Peter");
alert(result); // Output: Hello, Peter
```

You will learn more about the functions in JavaScript functions chapter.

The typeof Operator

The `typeof` operator can be used to find out what type of data a variable or operand contains. It can be used with or without parentheses (`typeof(x)` or `typeof x`).

The `typeof` operator is particularly useful in the situations when you need to process the values of different types differently, but you need to be very careful, because it may produce unexpected result in some cases, as demonstrated in the following example:

ExampleTry this code »

```
// Numbers
typeof 15;  // Returns: "number"
typeof 42.7;  // Returns: "number"
typeof 2.5e-4;  // Returns: "number"
typeof Infinity;  // Returns: "number"
typeof NaN;  // Returns: "number". Despite being "Not-A-Numbe
r"

// Strings
typeof '';  // Returns: "string"
typeof 'hello';  // Returns: "string"
typeof '12';  // Returns: "string". Number within quotes is t
ypeof string

// Booleans
typeof true;  // Returns: "boolean"
typeof false;  // Returns: "boolean"

// Undefined
typeof undefined;  // Returns: "undefined"
typeof undeclaredVariable; // Returns: "undefined"

// Null
typeof Null;  // Returns: "object"

// Objects
typeof {name: "John", age: 18};  // Returns: "object"

// Arrays
typeof [1, 2, 4];  // Returns: "object"

// Functions
typeof function(){};  // Returns: "function"
```

As you can clearly see in the above example when we test the `null` value using the `typeof` operator (*line no-22*), it returned "object" instead of "null".

This is a long-standing bug in JavaScript, but since lots of codes on the web written around this behavior, and thus fixing it would create a lot more problem, so idea of fixing this issue was rejected by the committee that design and maintains JavaScript.

JavaScript Events

In this tutorial you will learn how to handle events in JavaScript.

Understanding Events and Event Handlers

An event is something that happens when user interact with the web page, such as when he clicked a link or button, entered text into an input box or textarea, made selection in a select box, pressed key on the keyboard, moved the mouse pointer, submits a form, etc. In some cases, the Browser itself can trigger the events, such as the page load and unload events.

When an event occur, you can use a JavaScript event handler (or an event listener) to detect them and perform specific task or set of tasks. By convention, the names for event handlers always begin with the word "on", so an event handler for the click event is called `onclick`, similarly an event handler for the load event is called `onload`, event handler for the blur event is called `onblur`, and so on.

There are several ways to assign an event handler. The simplest way is to add them directly to the start tag of the HTML elements using the special event-handler attributes. For example, to assign a click handler for a button element, we can use `onclick` attribute, like this:

ExampleTry this code »

```
<button type="button" onclick="alert('Hello World!')">Click M
e</button>
```

However, to keep the JavaScript seperate from HTML, you can set up the event handler in an external JavaScript file or within the `<script>` and `</script>` tags, like

this:

ExampleTry this code »

```
<button type="button" id="myBtn">Click Me</button>
<script>
    function sayHello() {
        alert('Hello World!');
    }
    document.getElementById("myBtn").onclick = sayHello;
</script>
```

Note: Since HTML attributes are case-insensitive so `onclick` may also be written as `onClick`, `OnClick` or `ONCLICK`. But its *value* is case-sensitive.

In general, the events can be categorized into four main groups — mouse events, keyboard events, form events and document/window events. There are many other events, we will learn about them in later chapters. The following section will give you a brief overview of the most useful events one by one along with the real life practice examples.

Mouse Events

A mouse event is triggered when the user click some element, move the mouse pointer over an element, etc. Here're some most important mouse events and their event handler.

The Click Event (onclick)

The click event occurs when a user clicks on an element on a web page. Often, these are form elements and links. You can handle a click event with an `onclick` event handler.

The following example will show you an alert message when you click on the elements.

ExampleTry this code »

```
<button type="button" onclick="alert('You have clicked a butt
on!');">Click Me</button>
<a href="#" onclick="alert('You have clicked a link!');">Clic
k Me</a>
```

The Contextmenu Event (oncontextmenu)

The contextmenu event occurs when a user clicks the right mouse button on an element to open a context menu. You can handle a contextmenu event with an `oncontextmenu` event handler.

The following example will show an alert message when you right-click on the elements.

ExampleTry this code »

```
<button type="button" oncontextmenu="alert('You have right-cl
icked a button!');">Right Click on Me</button>
<a href="#" oncontextmenu="alert('You have right-clicked a li
nk!');">Right Click on Me</a>
```

The Mouseover Event (onmouseover)

The mouseover event occurs when a user moves the mouse pointer over an element.

You can handle the mouseover event with the `onmouseover` event handler. The following example will show you an alert message when you place mouse over the elements.

ExampleTry this code »

```
<button type="button" onmouseover="alert('You have placed mou
se pointer over a button!');">Place Mouse Over Me</button>
```

```
<a href="#" onmouseover="alert('You have placed mouse pointer
over a link!');">Place Mouse Over Me</a>
```

The Mouseout Event (onmouseout)

The mouseout event occurs when a user moves the mouse pointer outside of an element.

You can handle the mouseout event with the `onmouseout` event handler. The following example will show you an alert message when the mouseout event occurs.

ExampleTry this code »

```
<button type="button" onmouseout="alert('You have moved out o
f the button!');">Place Mouse Inside Me and Move Out</button>
<a href="#" onmouseout="alert('You have moved out of the lin
k!');">Place Mouse Inside Me and Move Out</a>
```

Keyboard Events

A keyboard event is fired when the user press or release a key on the keyboard. Here're some most important keyboard events and their event handler.

The Keydown Event (onkeydown)

The keydown event occurs when the user presses down a key on the keyboard.

You can handle the keydown event with the `onkeydown` event handler. The following example will show you an alert message when the keydown event occurs.

ExampleTry this code »

```
<input type="text" onkeydown="alert('You have pressed a key i
nside text input!')">
```

```
<textarea onkeydown="alert('You have pressed a key inside tex
tarea!')"></textarea>
```

The Keyup Event (onkeyup)

The keyup event occurs when the user releases a key on the keyboard.

You can handle the keyup event with the `onkeyup` event handler. The following example will show you an alert message when the keyup event occurs.

ExampleTry this code »

```
<input type="text" onkeyup="alert('You have released a key in
side text input!')">
<textarea onkeyup="alert('You have released a key inside text
area!')"></textarea>
```

The Keypress Event (onkeypress)

The keypress event occurs when a user presses down a key on the keyboard that has a character value associated with it. For example, keys like Ctrl, Shift, Alt, Esc, Arrow keys, etc. will not generate a keypress event, but will generate a keydown and keyup event.

You can handle the keypress event with the `onkeypress` event handler. The following example will show you an alert message when the keypress event occurs.

ExampleTry this code »

```
<input type="text" onkeypress="alert('You have pressed a key
inside text input!')">
<textarea onkeypress="alert('You have pressed a key inside te
xtarea!')"></textarea>
```

Form Events

A form event is fired when a form control receive or loses focus or when the user modify a form control value such as by typing text in a text input, select any option in a select box etc. Here're some most important form events and their event handler.

The Focus Event (onfocus)

The focus event occurs when the user gives focus to an element on a web page.

You can handle the focus event with the `onfocus` event handler. The following example will highlight the background of text input in yellow color when it receives the focus.

ExampleTry this code »

```
<script>
    function highlightInput(elm){
        elm.style.background = "yellow";
    }
</script>
<input type="text" onfocus="highlightInput(this)">
<button type="button">Button</button>
```

Note: The value of `this` keyword inside an event handler refers to the element which has the handler on it (i.e. where the event is currently being delivered).

The Blur Event (onblur)

The blur event occurs when the user takes the focus away from a form element or a window.

You can handle the blur event with the `onblur` event handler. The following example will show you an alert message when the text input element loses focus.

ExampleTry this code »

```
<input type="text" onblur="alert('Text input loses focus!')">
<button type="button">Submit</button>
```

To take the focus away from a form element first click inside of it then press the tab key on the keyboard, give focus on something else, or click outside of it.

The Change Event (onchange)

The change event occurs when a user changes the value of a form element.

You can handle the change event with the `onchange` event handler. The following example will show you an alert message when you change the option in the select box.

ExampleTry this code »

```
<select onchange="alert('You have changed the selection!');">
<option>Select</option>
<option>Male</option>
<option>Female</option>
</select>
```

The Submit Event (onsubmit)

The submit event only occurs when the user submits a form on a web page.

You can handle the submit event with the `onsubmit` event handler. The following example will show you an alert message while submitting the form to the server.

ExampleTry this code »

```
<form action="action.php" method="post" onsubmit="alert('Form
data will be submitted to the server!');">
<label>First Name:</label>
<input type="text" name="first-name" required><input type="su
```

```
bmit" value="Submit">
</form>
```

Document/Window Events

Events are also triggered in situations when the page has loaded or when user resize the browser window, etc. Here're some most important document/window events and their event handler.

The Load Event (onload)

The load event occurs when a web page has finished loading in the web browser.

You can handle the load event with the `onload` event handler. The following example will show you an alert message as soon as the page finishes loading.

ExampleTry this code »

```
<body onload="window.alert('Page is loaded successfully!');">
<h1>This is a heading</h1>
<p>This is paragraph of text.</p>
</body>
```

The Unload Event (onunload)

The unload event occurs when a user leaves the current web page.

You can handle the unload event with the `onunload` event handler. The following example will show you an alert message when you try to leave the page.

ExampleTry this code »

```
<body onunload="alert('Are you sure you want to leave this pa
ge?');">
<h1>This is a heading</h1>
```

```
<p>This is paragraph of text.</p>
</body>
```

The Resize Event (onresize)

The resize event occurs when a user resizes the browser window. The resize event also occurs in situations when the browser window is minimized or maximized.

You can handle the resize event with the `onresize` event handler. The following example will show you an alert message when you resize the browser window to a new width and height.

ExampleTry this code »

```
<p id="result"></p><script>
    function displayWindowSize() {
        let w = window.outerWidth;
        let h = window.outerHeight;
        let txt = "Window size: width=" + w + ", height=" +
h;
        document.getElementById("result").innerHTML = txt;
    }
    window.onresize = displayWindowSize;
</script>
```

JavaScript Array Reference

This chapter contains a brief overview of the properties and method of the global array object.

The JavaScript Array Object

The JavaScript Array object is a global object that is used in the construction of arrays. An array is a special type of variable that allows you to store multiple

values in a single variable.

To learn more about Arrays, please check out the JavaScript array chapter.

Array Properties

The following table lists the standard properties of the Array object.

Property	Description
`length`	Sets or returns the number of elements in an array.
`prototype`	Allows you to add new properties and methods to an Array object.

Note: Every object in JavaScript has a `constructor` property that refers to the constructor function that was used to create the instance of that object.

Array Methods

The following table lists the standard methods of the Array object.

Method	Description
`concat()`	Merge two or more arrays, and returns a new array.
`copyWithin()`	Copies part of an array to another location in the same array and returns it.
`entries()`	Returns a key/value pair Array Iteration Object.
`every()`	Checks if every element in an array pass a test in a testing function.
`fill()`	Fill the elements in an array with a static value.
`filter()`	Creates a new array with all elements that pass the test in a testing function.
`find()`	Returns the value of the first element in an array that pass the test in a testing function.
`findIndex()`	Returns the index of the first element in an array that pass the test in a testing function.
`forEach()`	Calls a function once for each array element.
`from()`	Creates an array from an object.

`includes()`	Determines whether an array includes a certain element.
`indexOf()`	Search the array for an element and returns its first index.
`isArray()`	Determines whether the passed value is an array.
`join()`	Joins all elements of an array into a string.
`keys()`	Returns a Array Iteration Object, containing the keys of the original array.
`lastIndexOf()`	Search the array for an element, starting at the end, and returns its last index.
`map()`	Creates a new array with the results of calling a function for each array element.
`pop()`	Removes the last element from an array, and returns that element.
`push()`	Adds one or more elements to the end of an array, and returns the array's new length.
`reduce()`	Reduce the values of an array to a single value (from left-to-right).
`reduceRight()`	Reduce the values of an array to a single value (from right-to-left).
`reverse()`	Reverses the order of the elements in an array.
`shift()`	Removes the first element from an array, and returns that element.
`slice()`	Selects a part of an array, and returns the new array.
`some()`	Checks if any of the elements in an array passes the test in a testing function.
`sort()`	Sorts the elements of an array.
`splice()`	Adds/Removes elements from an array.
`toString()`	Converts an array to a string, and returns the result.
`unshift()`	Adds new elements to the beginning of an array, and returns the array's new length.
`values()`	Returns a Array Iteration Object, containing the values of the original array.

JavaScript Boolean Reference

This chapter contains a brief overview of the properties and method of the global Boolean object.

The JavaScript Boolean Object

The Boolean object is an object wrapper around the boolean value `true` or `false`. This Boolean object type exists primarily to provide a `toString()` method to convert boolean values to strings.

To learn more about the Boolean, please check out the JavaScript data types chapter.

Boolean Properties

The following table lists the standard properties of the Boolean object.

Property	Description
`prototype`	Allows you to add new properties and methods to a Boolean object.

Note: Every object in JavaScript has a `constructor` property that refers to the constructor function that was used to create the instance of that object.

Boolean Methods

The following table lists the standard methods of the Boolean object.

Method	Description
`toString()`	Converts a Boolean value to a string, and returns the result.
`valueOf()`	Returns the primitive value of a Boolean object.

JavaScript Date Reference

This chapter contains a brief overview of the properties and method of the global Date object.

The JavaScript Date Object

The JavaScript Date object is a global object that is used to work with dates and times. The Date objects are based on a time value that is the number of

milliseconds since January 1, 1970 UTC.

To learn more about the Date, please check out the JavaScript date and time chapter.

Date Properties

The following table lists the standard properties of the Date object.

Property	Description
`prototype`	Allows you to add new properties and methods to a Date object.

Note: Every object in JavaScript has a `constructor` property that refers to the constructor function that was used to create the instance of that object.

Date Methods

The following table lists the standard methods of the Date object.

Method	Description
`getDate()`	Returns the day of the month (from 1-31).
`getDay()`	Returns the day of the week (from 0-6).
`getFullYear()`	Returns the year (four digits).
`getHours()`	Returns the hour (from 0-23).
`getMilliseconds()`	Returns the milliseconds (from 0-999).
`getMinutes()`	Returns the minutes (from 0-59).
`getMonth()`	Returns the month (from 0-11).
`getSeconds()`	Returns the seconds (from 0-59).
`getTime()`	Returns the number of milliseconds since midnight Jan 1, 1970.
`getTimezoneOffset()`	Returns the time difference between UTC time and local time, in minutes.
`getUTCDate()`	Returns the day of the month, according to universal time (from 1-31).

`getUTCDay()`	Returns the day of the week, according to universal time (from 0-6).
`getUTCFullYear()`	Returns the year, according to universal time.
`getUTCHours()`	Returns the hour, according to universal time (from 0-23).
`getUTCMilliseconds()`	Returns the milliseconds, according to universal time (from 0-999)
`getUTCMinutes()`	Returns the minutes, according to universal time (from 0-59).
`getUTCMonth()`	Returns the month, according to universal time (from 0-11).
`getUTCSeconds()`	Returns the seconds, according to universal time (from 0-59).
getYear()	Deprecated. Use the `getFullYear()` method instead.
`now()`	Returns the number of milliseconds since midnight Jan 1, 1970.
`parse()`	Parses a date string and returns the number of milliseconds since Jan 1, 1970.
`setDate()`	Sets the day of the month of a date object.
`setFullYear()`	Sets the full year of a date object.
`setHours()`	Sets the hours of a date object.
`setMilliseconds()`	Sets the milliseconds of a date object.
`setMinutes()`	Set the minutes of a date object.
`setMonth()`	Sets the month of a date object.
`setSeconds()`	Sets the seconds of a date object.
`setTime()`	Sets a date to a specified number of milliseconds after/before Jan 1, 1970.
`setUTCDate()`	Sets the day of the month of a date object, according to universal time.
`setUTCFullYear()`	Sets the year of a date object, according to universal time.
`setUTCHours()`	Sets the hours of a date object, according to universal time.
`setUTCMilliseconds()`	Sets the milliseconds of a date object, according to universal time.
`setUTCMinutes()`	Set the minutes of a date object, according to universal time.
`setUTCMonth()`	Sets the month of a date object, according to universal time.
`setUTCSeconds()`	Set the seconds of a date object, according to universal time.
`setYear()`	Deprecated. Use the `setFullYear()` method instead.

`toDateString()`	Converts the date portion of a Date object into a human readable form.
`toGMTString()`	Deprecated. Use the toUTCString() method instead
`toISOString()`	Returns the date as a string, formatted according to ISO standard.
`toJSON()`	Returns the date as a string, formatted as a JSON date.
`toLocaleDateString()`	Returns the date portion of a Date object as a locally formatted string.
`toLocaleTimeString()`	Returns the time portion of a Date object as a locally formatted string.
`toLocaleString()`	Converts a Date object to a locally formatted string.
`toString()`	Converts a Date object to a string.
`toTimeString()`	Converts the time portion of a Date object to a string.
`toUTCString()`	Converts a Date object to a string, according to universal time.
`UTC()`	Returns the number of milliseconds in a Date object since January 1, 1970, 00:00:00 (midnight), universal time.
`valueOf()`	Returns the primitive value of a Date object.

JavaScript Math Reference

This chapter contains a brief overview of the properties and method of the global Math object.

The JavaScript Math Object

The JavaScript Math object is used to perform mathematical tasks. The Math object is a static built-in object, so you won't need to instantiate it, all its properties and methods can be accessed directly.

To learn more about Math, please check out the JavaScript math operations chapter.

Math Properties

The following table lists the standard properties of the Math object.

Property	Description
`E`	Returns Euler's number, the base of natural logarithms, `e`, approximately 2.718
`LN2`	Returns the natural logarithm of 2, approximately 0.693
`LN10`	Returns the natural logarithm of 10, approximately 2.302
`LOG2E`	Returns the base 2 logarithm of `e`, approximately 1.442
`LOG10E`	Returns the base 10 logarithm of `e`, approximately 0.434
`PI`	Returns the ratio of the circumference of a circle to its diameter (i.e. `π`). The approximate value of PI is 3.14159
`SQRT1_2`	Returns the square root of 1/2, approximately 0.707
`SQRT2`	Returns the square root of 2, approximately 1.414

Note: The Math object is just a collection of static functions and constants. The Math object is different from other built-in objects (e.g. Date, Array, String, etc.), because it has no constructor, so there is no way to create an instance of Math.

Math Methods

The following table lists the standard methods of the Math object.

Method	Description
`abs()`	Returns the absolute value of a number.
`acos()`	Returns the arccosine of a number, in radians.
`acosh()`	Returns the hyperbolic arccosine of a number.
`asin()`	Returns the arcsine of a number, in radians
`asinh()`	Returns the hyperbolic arcsine of a number.
`atan()`	Returns the arctangent of a number, in radians.
`atan2(y, x)`	Returns the arctangent of the quotient of its arguments.
`atanh()`	Returns the hyperbolic arctangent of a number.
`cbrt()`	Returns the cube root of a number.
`ceil()`	Returns the next integer greater than or equal to a given number (rounding up).

`cos()`	Returns the cosine of the specified angle. The angle must be specified in radians.
`cosh()`	Returns the hyperbolic cosine of a number.
`exp(x)`	Returns `e x`, where `x` is the argument, and `e` is Euler's number (also known as Napier's constant), the base of the natural logarithms.
`floor()`	Returns the next integer less than or equal to a given number (rounding down).
`log()`	Returns the natural logarithm (base `e`) of a number.
`max(x, y, ...)`	Returns the highest-valued number in a list of numbers.
`min(x, y, ...)`	Returns the lowest-valued number in a list of numbers.
`pow(x, y)`	Returns the base to the exponent power, that is, `x y`.
`random()`	Returns a random number between 0 and 1 (including 0, but not 1).
`round()`	Returns the value of a number rounded to the nearest integer.
`sin()`	Returns the sign of a number (given in radians).
`sinh()`	Returns the hyperbolic sine of a number.
`sqrt()`	Returns the square root of a number.
`tan()`	Returns the tangent of a number.
`tanh()`	Returns the hyperbolic tangent of a number.
`trunc(x)`	Returns the integer part of a number by removing any fractional digits.

JavaScript Number Reference

This chapter contains a brief overview of the properties and method of the global Number object.

The JavaScript Number Object

The JavaScript Number object acts as a wrapper for primitive numeric values. JavaScript has only one kind of number data type and it doesn't distinguish between integers and floating-point values.

To learn more about Number, please check out the JavaScript numbers chapter.

Number Properties

The following table lists the standard properties of the Number object.

Property	Description
`MIN_SAFE_INTEGER`	Represents the maximum safe integer in JavaScript (253 - 1).
`MAX_VALUE`	Returns the largest numeric value representable in JavaScript, approximately 1.79E+308. Values larger than `MAX_VALUE` are represented as `Infinity`.
`MIN_SAFE_INTEGER`	Represents the minimum safe integer in JavaScript (-(253 - 1)).
`MIN_VALUE`	Returns the smallest positive numeric value representable in JavaScript, approximately 5e-324. It is closest to 0, not the most negative number. Values smaller than `MIN_VALUE` are converted to 0.
`NEGATIVE_INFINITY`	Represents the negative infinity value.
`NaN`	Represents "Not-A-Number" value.
`POSITIVE_INFINITY`	Represents the infinity value.
`prototype`	Allows you to add new properties and methods to a Number object.

Note: Every object in JavaScript has a `constructor` property that refers to the constructor function that was used to create the instance of that object.

Number Methods

The following table lists the standard methods of the Number object.

Method	Description
`isFinite()`	Checks whether the passed value is a finite number.
`isInteger()`	Checks whether the passed value is an integer.
`isNaN()`	Checks whether the passed value is `NaN` and its type is Number.
`isSafeInteger()`	Checks whether a value is a safe integer.
`toExponential()`	Converts a number to exponential notation.
`toFixed()`	Formats a number using fixed-point notation.
`toPrecision()`	Returns a string representing the number to the specified precision.

`toString()`	Converts a number to a string.
`valueOf()`	Returns the primitive value of a Number object.

JavaScript String Reference

This chapter contains a brief overview of the properties and method of the global String object.

The JavaScript String Object

The JavaScript String object is a global object that is used to store strings. A string is a sequence of letters, numbers, special characters and arithmetic values or combination of all.

To learn more about String, please check out the JavaScript strings chapter.

String Properties

The following table lists the standard properties of the String object.

Property	Description
`length`	Returns the length of a string.
`prototype`	Allows you to add new properties and methods to an String object.

Note: Every object in JavaScript has a `constructor` property that refers to the constructor function that was used to create the instance of that object.

String Methods

The following table lists the standard methods of the String object.

Method	Description
`charAt()`	Returns the character at the specified index.
`charCodeAt()`	Returns the Unicode of the character at the specified index.
`concat()`	Joins two or more strings, and returns a new string.

`endsWith()`	Checks whether a string ends with a specified substring.
`fromCharCode()`	Converts Unicode values to characters.
`includes()`	Checks whether a string contains the specified substring.
`indexOf()`	Returns the index of the first occurrence of the specified value in a string.
`lastIndexOf()`	Returns the index of the last occurrence of the specified value in a string.
`localeCompare()`	Compares two strings in the current locale.
`match()`	Matches a string against a regular expression, and returns an array of all matches.
`repeat()`	Returns a new string which contains the specified number of copies of the original string.
`replace()`	Replaces the occurrences of a string or pattern inside a string with another string, and return a new string without modifying the original string.
`search()`	Searches a string against a regular expression, and returns the index of the first match.
`slice()`	Extracts a portion of a string and returns it as a new string.
`split()`	Splits a string into an array of substrings.
`startsWith()`	Checks whether a string begins with a specified substring.
`substr()`	Extracts the part of a string between the start index and a number of characters after it.
`substring()`	Extracts the part of a string between the start and end indexes.
`toLocaleLowerCase()`	Converts a string to lowercase letters, according to host machine's current locale.
`toLocaleUpperCase()`	Converts a string to uppercase letters, according to host machine's current locale.
`toLowerCase()`	Converts a string to lowercase letters.
`toString()`	Returns a string representing the specified object.
`toUpperCase()`	Converts a string to uppercase letters.
`trim()`	Removes whitespace from both ends of a string.
`valueOf()`	Returns the primitive value of a String object.

JavaScript Functions

In this tutorial you will learn how to define and call a function in JavaScript.

What is Function?

A function is a group of statements that perform specific tasks and can be kept and maintained separately form main program. Functions provide a way to create reusable code packages which are more portable and easier to debug. Here are some advantages of using functions:

- **Functions reduces the repetition of code within a program** — Function allows you to extract commonly used block of code into a single component. Now you can perform the same task by calling this function wherever you want within your script without having to copy and paste the same block of code again and again.
- **Functions makes the code much easier to maintain** — Since a function created once can be used many times, so any changes made inside a function automatically implemented at all the places without touching the several files.
- **Functions makes it easier to eliminate the errors** — When the program is subdivided into functions, if any error occur you know exactly what function causing the error and where to find it. Therefore, fixing errors becomes much easier.

The following section will show you how to define and call functions in your scripts.

Defining and Calling a Function

The declaration of a function start with the `function` keyword, followed by the name of the function you want to create, followed by parentheses i.e. `()` and finally place your function's code between curly brackets `{}`. Here's the basic syntax for declaring a function:

function functionName() { // Code to be executed}

Here is a simple example of a function, that will show a hello message:

ExampleTry this code »

```
// Defining function
function sayHello() {
    alert("Hello, welcome to this website!");
}

// Calling function
sayHello(); // Outputs: Hello, welcome to this website!
```

Once a function is defined it can be called (invoked) from anywhere in the document, by typing its name followed by a set of parentheses, like `sayHello()` in the example above.

Note: A function name must start with a letter or underscore character not with a number, optionally followed by the more letters, numbers, or underscore characters. Function names are case sensitive, just like variable names.

Adding Parameters to Functions

You can specify parameters when you define your function to accept input values at run time. The parameters work like placeholder variables within a function; they're replaced at run time by the values (known as argument) provided to the function at the time of invocation.

Parameters are set on the first line of the function inside the set of parentheses, like this:

function functionName(*parameter1*, *parameter2*, *parameter3*) { // Code to be executed}

The `displaySum()` function in the following example takes two numbers as arguments, simply add them together and then display the result in the browser.

ExampleTry this code »

```
// Defining function
function displaySum(num1, num2) {
```

```
    let total = num1 + num2;
    alert(total);
}

// Calling function
displaySum(6, 20); // Outputs: 26
displaySum(-5, 17); // Outputs: 12
```

You can define as many parameters as you like. However for each parameter you specify, a corresponding argument needs to be passed to the function when it is called, otherwise its value becomes `undefined`. Let's consider the following example:

ExampleTry this code »

```
// Defining function
function showFullname(firstName, lastName) {
    alert(firstName + " " + lastName);
}

// Calling function
showFullname("Clark", "Kent"); // Outputs: Clark Kent
showFullname("John"); // Outputs: John undefined
```

Default Values for Function Parameters ES6

With ES6, now you can specify default values to the function parameters. This means that if no arguments are provided to function when it is called these default parameters values will be used. This is one of the most awaited features in JavaScript. Here's an example:

ExampleTry this code »

```
function sayHello(name = 'Guest') {
    alert('Hello, ' + name);
}

sayHello(); // Outputs: Hello, Guest
sayHello('John'); // Outputs: Hello, John
```

While prior to ES6, to achieve the same we had to write something like this:

ExampleTry this code »

```
function sayHello(name) {
    let name = name || 'Guest';
    alert('Hello, ' + name);
}

sayHello(); // Outputs: Hello, Guest
sayHello('John'); // Outputs: Hello, John
```

To learn about other ES6 features, please check out the JavaScript ES6 features chapter.

Returning Values from a Function

A function can return a value back to the script that called the function as a result using the `return` statement. The value may be of any type, including arrays and objects.

The `return` statement usually placed as the last line of the function before the closing curly bracket and ends it with a semicolon, as shown in the following example.

ExampleTry this code »

```
// Defining function
function getSum(num1, num2) {
```

```
    let total = num1 + num2;
    return total;
}

// Displaying returned value
alert(getSum(6, 20)); // 0utputs: 26
alert(getSum(-5, 17)); // 0utputs: 12
```

A function *can not* return multiple values. However, you can obtain similar results by returning an array of values, as demonstrated in the following example.

ExampleTry this code »

```
// Defining function
function divideNumbers(dividend, divisor){
    let quotient = dividend / divisor;
    let arr = [dividend, divisor, quotient];
    return arr;
}

// Store returned value in a variable
let all = divideNumbers(10, 2);

// Displaying individual values
alert(all[0]); // 0utputs: 10
alert(all[1]); // 0utputs: 2
alert(all[2]); // 0utputs: 5
```

Working with Function Expressions

The syntax that we've used before to create functions is called function declaration. There is another syntax for creating a function that is called a function expression.

ExampleTry this code »

```
// Function Declaration
function getSum(num1, num2) {
    let total = num1 + num2;
    return total;
}

// Function Expression
let getSum = function(num1, num2) {
    let total = num1 + num2;
    return total;
};
```

Once function expression has been stored in a variable, the variable can be used as a function:

ExampleTry this code »

```
let getSum = function(num1, num2) {
    let total = num1 + num2;
    return total;
};

alert(getSum(5, 10)); // 0utputs: 15

let sum = getSum(7, 25);
alert(sum); // 0utputs: 32
```

Note: There is no need to put a semicolon after the closing curly bracket in a function declaration. But function expressions, on the other hand, should always end with a semicolon.

Tip: In JavaScript functions can be stored in variables, passed into other functions as arguments, passed out of functions as return values, and constructed at run-time.

The syntax of the *function declaration* and *function expression* looks very similar, but they differ in the way they are evaluated, check out the following example:

ExampleTry this code »

```
declaration(); // Outputs: Hi, I'm a function declaration!
function declaration() {
    alert("Hi, I'm a function declaration!");
}

expression(); // Uncaught TypeError: undefined is not a funct
ion
let expression = function() {
    alert("Hi, I'm a function expression!");
};
```

As you can see in the above example, the function expression threw an exception when it was invoked before it is defined, but the function declaration executed successfully.

JavaScript parse declaration function before the program executes. Therefore, it doesn't matter if the program invokes the function before it is defined because JavaScript has hoisted the function to the top of the current scope behind the scenes. The function expression is not evaluated until it is assigned to a variable; therefore, it is still undefined when invoked.

ES6 has introduced even shorter syntax for writing function expression which is called arrow function, please check out the JavaScript ES6 features chapter to learn more about it.

Understanding the Variable Scope

However, you can declare the variables anywhere in JavaScript. But, the location of the declaration determines the extent of a variable's availability within the JavaScript program i.e. where the variable can be used or accessed. This accessibility is known as *variable scope*.

By default, variables declared within a function have *local scope* that means they cannot be viewed or manipulated from outside of that function, as shown in the example below:

ExampleTry this code »

```
// Defining function
function greetWorld() {
    let greet = "Hello World!";
    alert(greet);
}

greetWorld(); // Outputs: Hello World!

alert(greet); // Uncaught ReferenceError: greet is not define
d
```

However, any variables declared in a program outside of a function has *global scope* i.e. it will be available to all script, whether that script is inside a function or outside. Here's an example:

ExampleTry this code »

```
let greet = "Hello World!";

// Defining function
function greetWorld() {
    alert(greet);
}

greetWorld();  // Outputs: Hello World!

alert(greet); // Outputs: Hello World!
```

example for arrow function

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
```

```
    <title>JavaScript ES6 Arrow Functions</title>
</head>
<body>
    <script>
    // Function Expression
    var sum = function(a, b) {
        return a + b;
    }
    document.write(sum(2, 3)); // 5
    document.write("<br>");

    // Arrow function
    var sum = (a, b) => a + b;
    document.write(sum(2, 3)); // 5
    </script>
</body>
</html>
```

example of anonymous function

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>JavaScript Construct Closure Using Function Expression Sy
</head>
<body>
    <script>
        // Anonymous function expression
        let myCounter = (function() {
            let counter = 0;

            // Nested anonymous function
            return function() {
                counter += 1;
                return counter;
```

```
            }
        })();

        document.write(myCounter() + "<br>"); // Prints: 1
        document.write(myCounter()); // Prints: 2
    </script>
</body>
</html>
```