



JavaScript Functions

.NET

JavaScript (JS) programming language conforms to the ECMAScript specification. JavaScript is a high-level language that is just-in-time compiled, has curly-bracket syntax, dynamic typing, prototype-based object-orientation, and first-class functions.

[HTTPS://EN.WIKIPEDIA.ORG/WIKI/JAVASCRIPT](https://en.wikipedia.org/wiki/JavaScript)

Create Sample .html and .js docs

Create a **.html** document and create the HTML template inside (use 'doc' shortcut).

This can be used to experiment with the examples in the presentation.

The **.js** file and the **.html** file should be in the same folder.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>JS Example Document</title>
</head>
<body>
  <script src="functions.js"></script>
</body>
</html>
```

Function Declarations

<https://javascript.info/function-basics>

JS functions can declare variables inside the function scope.

If a function's local variable has the same name as a variable outside the function, the local variable **shadows** the outer variable.

Primitives are passed by value in JS.

Objects are passed by reference.

A function with multiple **parameters** can be called with fewer **arguments** than parameters. The unused parameters are shown as **undefined**.

A **parameter** can be given a default value.

```
1 function showMessage(from, text = "no text given") {  
2     alert( from + ": " + text );  
3 }  
4  
5 showMessage("Ann"); // Ann: no text given
```

```
1 function showMessage(from, text) { // arguments: from, te  
2     alert(from + ': ' + text);  
3 }  
4  
5 showMessage('Ann', 'Hello!'); // Ann: Hello! (*)  
6 showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

JavaScript – Functions

<https://javascript.info/function-basics>

A function can return a value at any point using **return**;

It can also **return**; without a value.

Never place return data on a separate line. JS assumes a ; after the keyword **return**.

```
1 function checkAge(age) {  
2   if (age >= 18) {  
3     return true;  
4   } else {  
5     return confirm('Do you have permission from your parents?');  
6   }  
7 }  
8  
9 let age = prompt('How old are you?', 18);  
10  
11 if ( checkAge(age) ) {  
12   alert( 'Access granted' );  
13 } else {  
14   alert( 'Access denied' );  
15 }
```

```
1 function showMovie(age) {  
2   if ( !checkAge(age) ) {  
3     return;  
4   }  
5  
6   alert( "Showing you the movie" ); // (*)  
7   // ...  
8 }
```

JavaScript – Function Expressions

<https://javascript.info/function-expressions>

In JavaScript, a function is considered a value. Figure 1 shows a *function expression*. It's called `sayHi()`. It is a value so it can also be passed.

A *Function Expression* is created when program execution reaches its declaration. It is usable only from that moment onward.

Figure 1

```
1 let sayHi = function() {  
2   alert( "Hello" );  
3 };
```

This *Function Declaration*:

- (1) creates the function called `sayHi`.
- (2) Stores `sayHi()` in a variable, `func`.
- (3) Now the function can be invoked as both `sayHi()` and `func()`.

```
1 function sayHi() { // (1) create  
2   alert( "Hello" );  
3 }  
4  
5 let func = sayHi; // (2) copy  
6  
7 func(); // Hello // (3) run the copy (it works)!  
8 sayHi(); // Hello // this still works too (why wouldn't it)
```

*If there were parentheses after `sayHi`, `func = sayHi()` would write the result of the call `sayHi()` into `func`.

Arrow Functions

<https://javascript.info/arrow-functions-basics>

Arrow Functions are a very simple and concise syntax for creating functions. Both the below expressions create a function that accepts arguments *arg1..argN*, then evaluates the expression and returns its result into *func*.

```
1 let func = function(arg1, arg2, ...argN) {  
2   return expression;  
3 };
```

Is the same as...

```
1 let func = (arg1, arg2, ...argN) => expression
```

This function accepts two arguments: a, b.
It returns the result of **a + b**.

```
1 let sum = (a, b) => a + b;  
2  
3 /* This arrow function is a shorter form of:  
4  
5 let sum = function(a, b) {  
6   return a + b;  
7 };  
8 */  
9  
10 alert( sum(1, 2) ); // 3
```

```
1 let sayHi = () => alert("Hello!");
```

```
1 let double = n => n * 2;
```

With one argument, **()** are not required. With zero arguments empty **()** are required.

```
1 let sum = (a, b) => { // the curly brace opens a multiline function  
2   let result = a + b;  
3   return result; // if we use curly braces, then we need an explicit "return"  
4 };  
5  
6 alert( sum(1, 2) ); // 3
```

JavaScript – Callback Functions

<https://javascript.info/function-expressions#callback-functions>
<https://gist.github.com/ericelliott/414be9be82128443f6df>

Pass functions as values. (Line 15) The arguments `showOk()` and `showCancel()` of the call to `ask()` are called *callback functions*.

A function can be passed to be “called back” later (if necessary). `showOk()` becomes the callback for a “yes” answer, and `showCancel()` for a “no” answer.

```
1 function ask(question, yes, no) {
2   if (confirm(question)) yes()
3   else no();
4 }
5
6 function showOk() {
7   alert( "You agreed." );
8 }
9
10 function showCancel() {
11   alert( "You canceled the execution." );
12 }
13
14 // usage: functions showOk, showCancel are passed as arguments to ask
15 ask("Do you agree?", showOk, showCancel);
```

We can use *Function Expressions* when calling `ask()`. It is the same function, but much shorter. These are called *Anonymous Functions*

```
1 function ask(question, yes, no) {
2   if (confirm(question)) yes()
3   else no();
4 }
5
6 ask(
7   "Do you agree?",
8   function() { alert("You agreed."); },
9   function() { alert("You canceled the execution."); }
10 );
```


IIFE - Immediately Invoked Function Expression

<https://developer.mozilla.org/en-US/docs/Glossary/IIFE>

https://en.wikipedia.org/wiki/Immediately_invoked_function_expression

An *Immediately Invoked Function Expression* (IIFE, pronounced “iffy”) is a **JavaScript** function that runs as soon as it is defined. It’s also known as a Self-Executing Anonymous Function

IIFE’s contain two major parts:

- The first is the anonymous function with lexical scope enclosed within the Grouping Operator (). This prevents accessing variables within the **IIFE** idiom as well as polluting the global scope.
- The second part is another pair of (), which complete the statement. Now, the JavaScript engine will directly interpret the function.

```
1 | (function () {  
2 |     statements  
3 | })();
```

```
(function() {  
    alert('I am NOT an IIFE.');
```

```
})();  
alert('NOW I am an IIFE!');
```

```
(() => alert('I am also an IIFE.'))();
```

IIFE

Immediately Invoked Function Expression

<https://developer.mozilla.org/en-US/docs/Glossary/IIFE>

https://en.wikipedia.org/wiki/Immediately_invoked_function_expression

Any variable declared within an *IIFE* cannot be accessed from outside it.

Assigning an *IIFE* to a variable stores the function's return value, not the function definition itself.

```
1 | (function () {  
2 |     var aName = "Barry";  
3 | })();  
4 | // Variable aName is not accessible from the outside scope  
5 | aName // throws "Uncaught ReferenceError: aName is not defined"
```

```
1 | var result = (function () {  
2 |     var name = "Barry";  
3 |     return name;  
4 | })();  
5 | // Immediately creates the output:  
6 | result; // "Barry"
```

Scope with Nested Functions (and Closure)

<https://javascript.info/closure>

If a variable is declared inside a code block {...}, it's only visible inside that block.

A nested function can access variables declared inside it's code block and inside it's parent code block.

A nested function can be returned (as a property of a new object or as a result by itself). It can then be used anywhere else, and it will still have access to the same outer variables.

```
1 function sayHiBye(firstName, lastName) {  
2  
3     // helper nested function to use below  
4     function getFullName() {  
5         return firstName + " " + lastName;  
6     }  
7  
8     alert( "Hello, " + getFullName() );  
9     alert( "Bye, " + getFullName() );  
10  
11 }
```

```
1 function makeCounter() {  
2     let count = 0;  
3  
4     return function() {  
5         return count++;  
6     };  
7 }  
8  
9 let counter = makeCounter();  
10  
11 alert( counter() ); // 0  
12 alert( counter() ); // 1  
13 alert( counter() ); // 2
```

Scope and Closure

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>

A ***closure*** is a ***function*** enclosed with references to its surrounding state (the ***lexical environment***). A ***closure*** gives access to an outer ***function's*** scope from an inner ***function***.

```
1 function init() {  
2     var name = 'Mozilla'; // name is a local variable created by init  
3     function displayName() { // displayName() is the inner function, a closure  
4         alert(name); // use variable declared in the parent function  
5     }  
6     displayName();  
7 }  
8 init();
```

`init()` creates local variable (`name`) and a function, `displayName()`. `displayName()` is an inner function ***defined*** inside `init()`. `displayName()` is available only within the body of `init()`. `displayName()` has no local variables. Because inner functions have access to outer function variables, `displayName()` accesses the `name` variable declared in its parent function, `init()`. This is Lexical Scoping.

Scope and Closure Example

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>

`makeAdder()` takes a single argument, `x`, and returns a function. The returned function takes a single argument `y` and returns `(x + y)`.

`add5` and `add10` are both *closures*.

They are the function returned by `makeAdder()` but store different *lexical environments*.

In `add5`'s *lexical environment*, `x` is 5, while in `add10`'s, `x` is 10. When `add5` and `add10` are invoked, they still have access to the parameter `x` from `makeAdder()`;

```
1  function makeAdder(x) {  
2      return function(y) {  
3          return x + y;  
4      };  
5  }  
6  
7  var add5 = makeAdder(5);  
8  var add10 = makeAdder(10);  
9  
10 console.log(add5(2)); // 7  
11 console.log(add10(2)); // 12
```

Try/Catch/Finally

<https://javascript.info/try-catch#the-try-catch-syntax>

The JS *Try/Catch* block works similarly to the C# *Try/Catch* Block. There is only one 'error' object generated. The 'error' object has three parts

- Name – the Error Name, Like “Reference Error”.
- Message – a text message with error details
- Stack – a stack trace of the calls that led to the error.

JavaScript has many built-in, standard errors: **Error**, **SyntaxError**, **ReferenceError**, **TypeError**, and others.

The *Finally* Block always executes.

```
1 let error = new Error(message);
2 // or
3 let error = new SyntaxError(message);
4 let error = new ReferenceError(message);
5 // ...
```

```
1 try {
2
3   alert('Start of try runs'); // (1) <--
4
5   lalala; // error, variable is not defined!
6
7   alert('End of try (never reached)'); // (2)
8
9 } catch(err) {
10
11   alert(`Error has occurred!`); // (3) <--
12
13 }
```

```
1 let json = '{ "age": 30 }'; // incomplete data
2
3 try {
4
5   let user = JSON.parse(json); // <-- no errors
6
7   if (!user.name) {
8     throw new SyntaxError("Incomplete data: no name"); // (*)
9   }
10
11   alert( user.name );
12
13 } catch(e) {
14   alert( "JSON Error: " + e.message ); // JSON Error: Incomplete data
15 }
```


localStorage and sessionStorage

<https://blog.logrocket.com/localstorage-javascript-complete-guide/#sessionstoragevslocalstorage>

TODO.. This is also a TODO in the JS flow control PPT