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

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

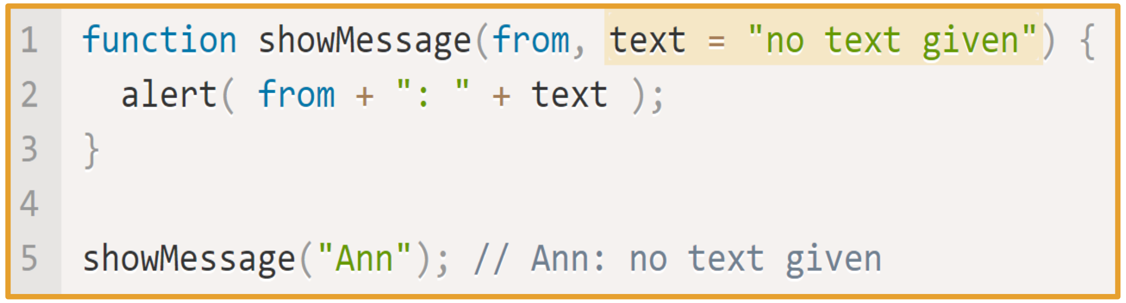
JS functions can declare variables inside the function scope.

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

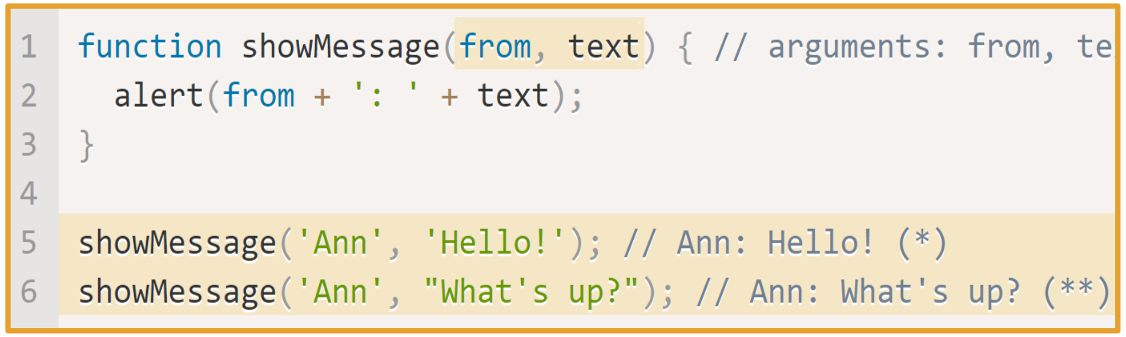
Primitives are passed by value in JS. Objects are passes by reference.

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.



```
function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
}

showMessage("Ann"); // Ann: no text given
```



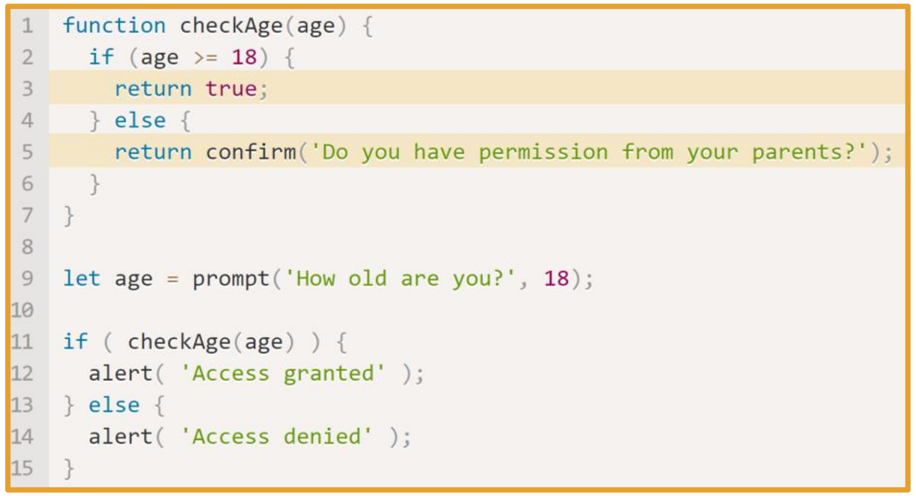
```
function showMessage(from, text) { // arguments: from, te
  alert(from + ': ' + text);
}

showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

JavaScript – Functions

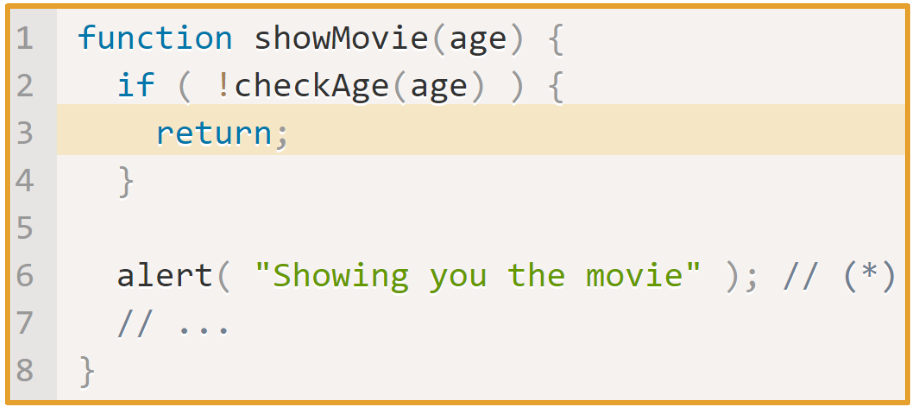
https://javascript.info/function-basics

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***.



```javascript
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
  alert( 'Access denied' );
}
```



```javascript
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Showing you the movie" ); // (*)
  // ...
}
```

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

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

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().

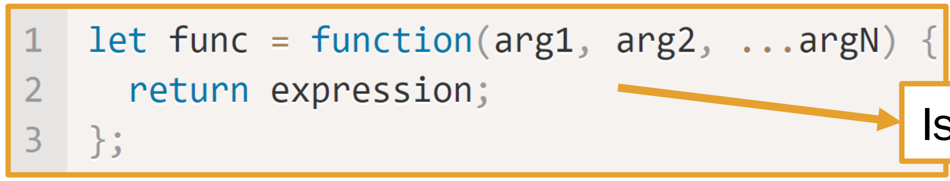
```
function sayHi() {   // (1) create
  alert( "Hello" );
}

let func = sayHi;    // (2) copy

func(); // Hello     // (3) run the copy (it works)!
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

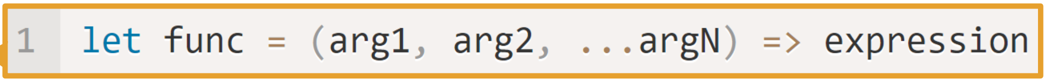
https://javascript.info/arrow-functions-basics

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***.

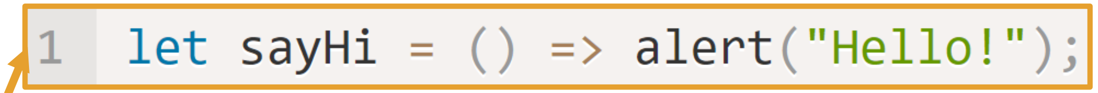


```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

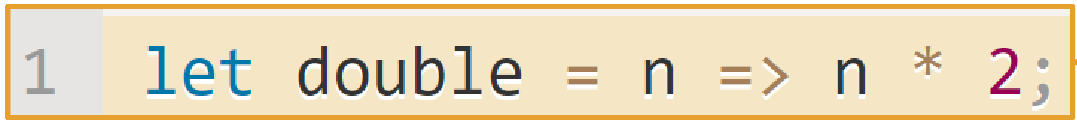
Is the same as...



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



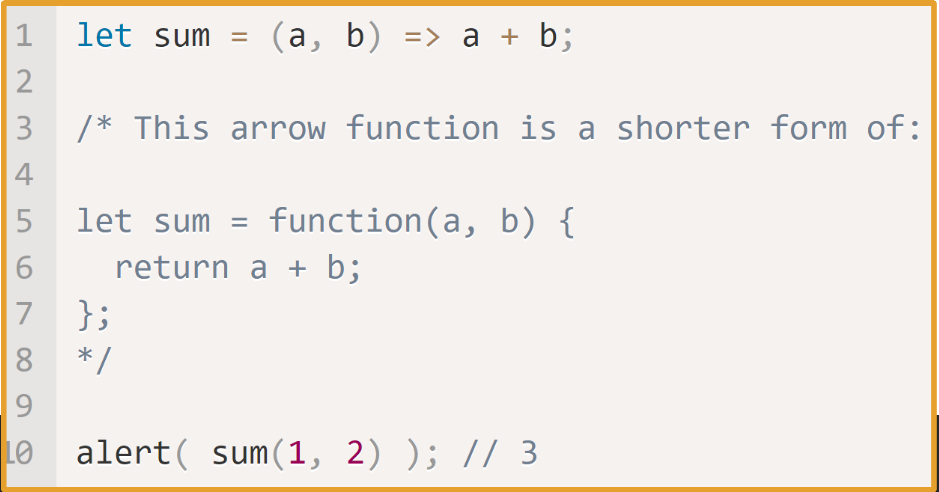
```
let sayHi = () => alert("Hello!");
```



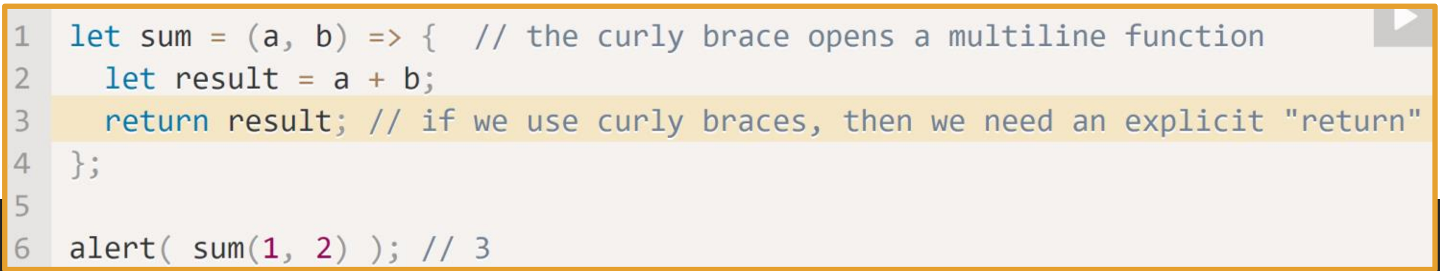
```
let double = n => n * 2;
```

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

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



```
let sum = (a, b) => a + b;

/* This arrow function is a shorter form of:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

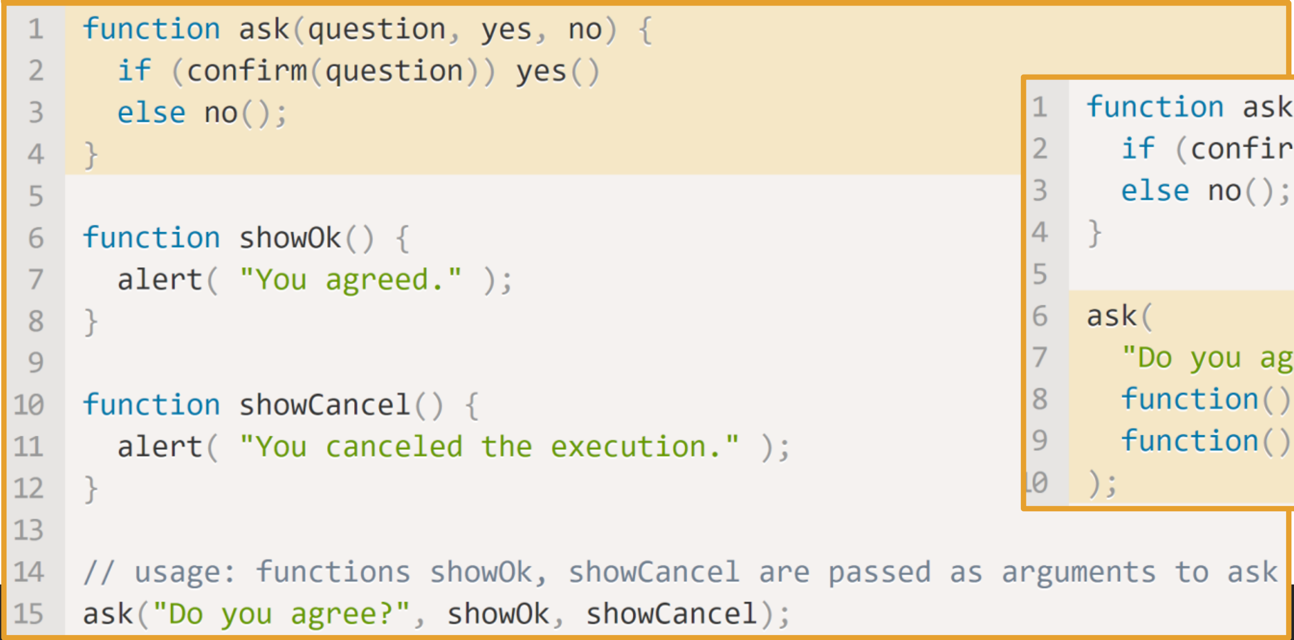


```
let sum = (a, b) => {  // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, then we need an explicit "return"
};

alert( sum(1, 2) ); // 3
```

JavaScript – Callback Functions

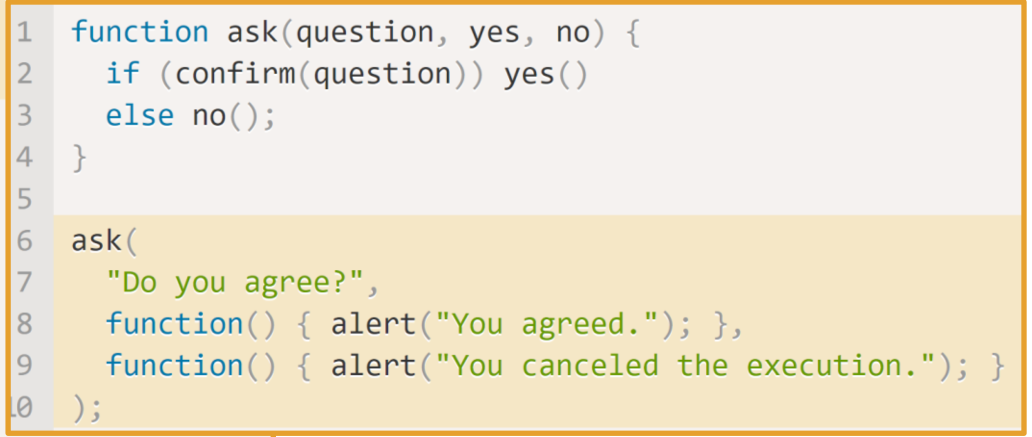
https://javascript.info/function-expressions#callback-functions
https://gist.github.com/ericelliott/414be9be82128443f6df

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.



```javascript
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***



```javascript
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

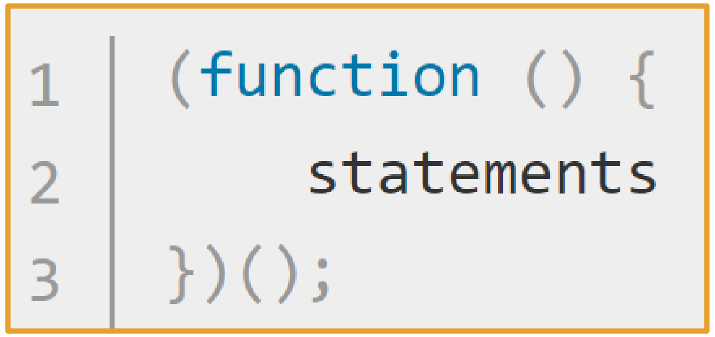
IIFE - Immediately Invoked Function Expression

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.



```
(function () {
    statements
})();
```

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

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

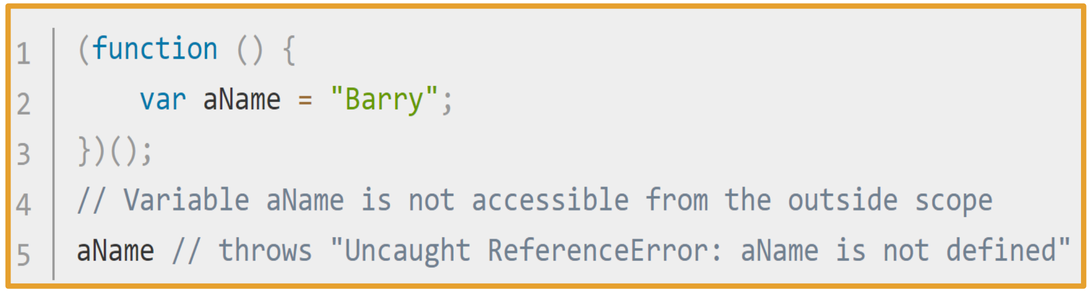
```
(() => alert('I am also an IIFE.')();
```

IIFE
Immediately Invoked Function Expression

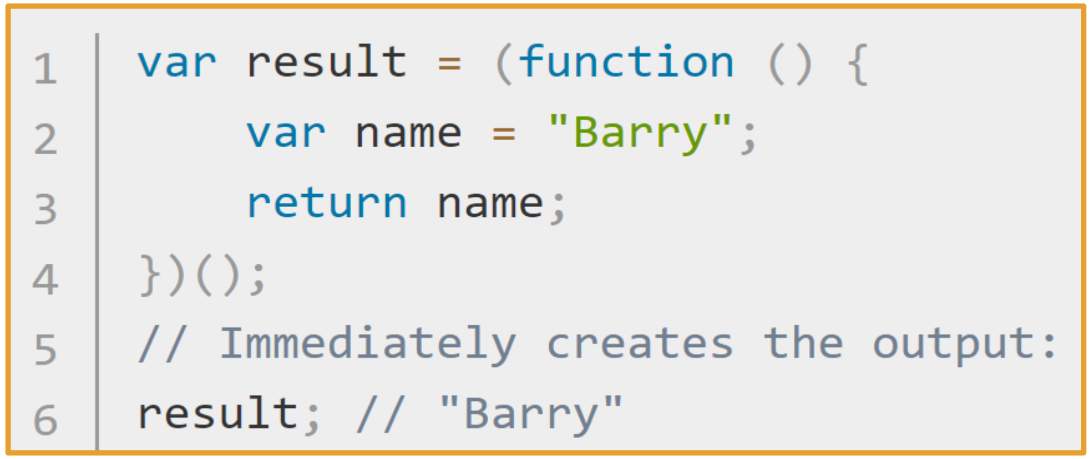
https://developer.mozilla.org/en-US/docs/Glossary/IIFE
https://en.wikipedia.org/wiki/Immediately_invoked_function_expression

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.



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



```
var result = (function () {
    var name = "Barry";
    return name;
})();
// Immediately creates the output:
result; // "Barry"
```

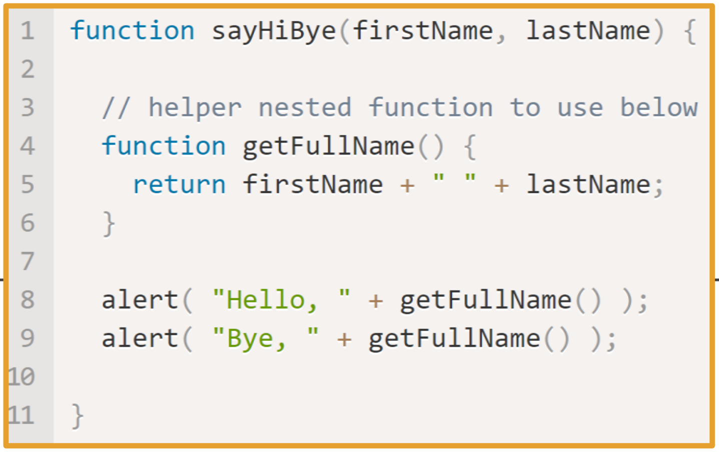
Scope with Nested Functions (and Closure)

https://javascript.info/closure

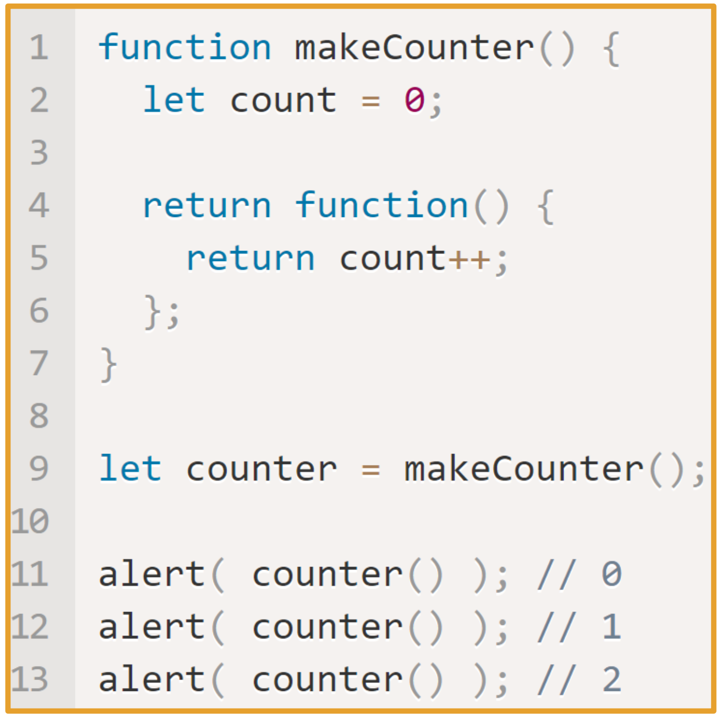
If a variable is declared inside a code block {...}, it’s only visible inside that block.

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.



```
function sayHiBye(firstName, lastName) {

  // helper nested function to use below
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );

}
```

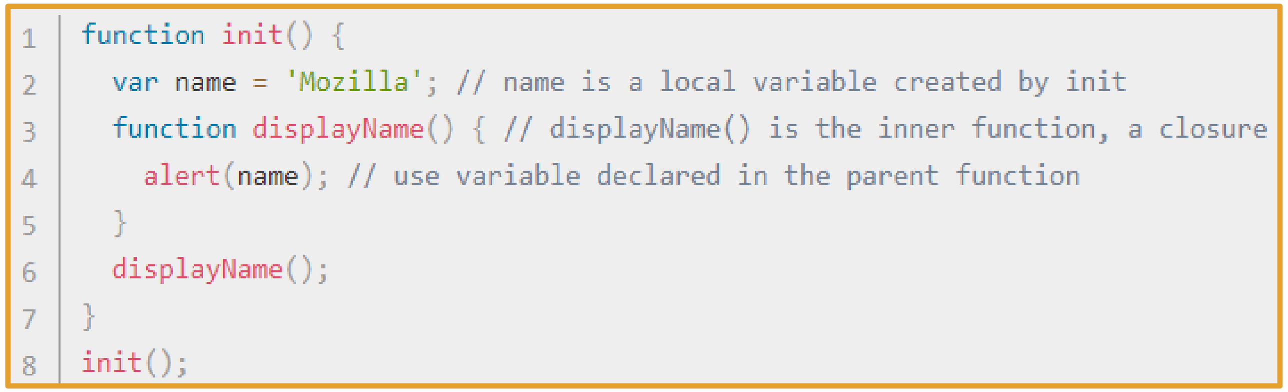


```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
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***.



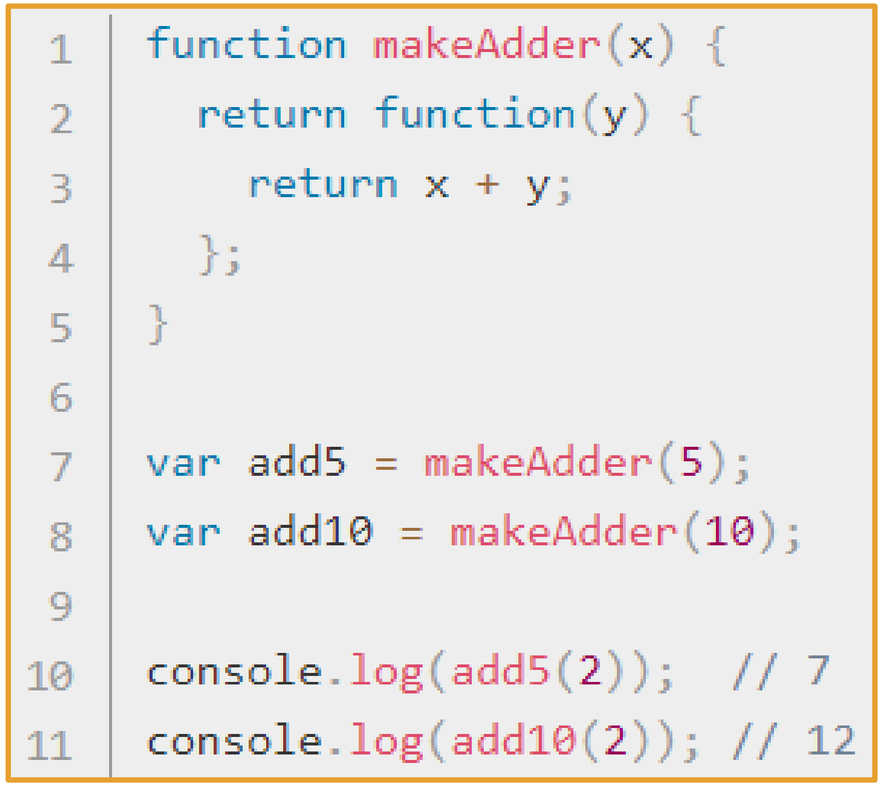
```javascript
function init() {
  var name = 'Mozilla'; // name is a local variable created by init
  function displayName() { // displayName() is the inner function, a closure
    alert(name); // use variable declared in the parent function
  }
  displayName();
}
init();
```

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();



```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

Try/Catch/Finally

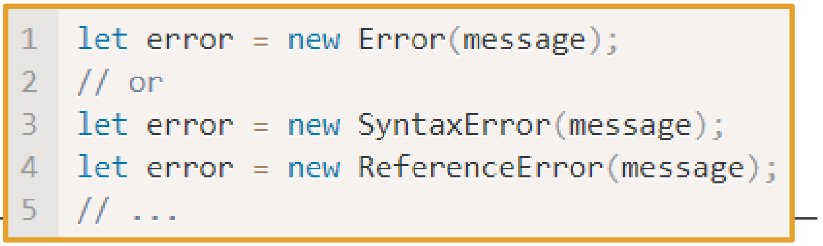
https://javascript.info/try-catch#the-try-catch-syntax

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

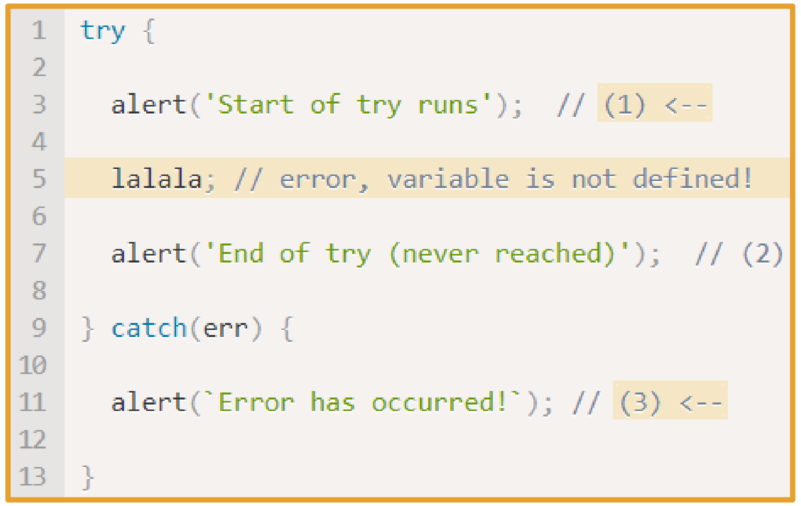
- Name – the Error Name, Like "Reference Error".
- Message – a text message with error details
- Stack – a stack trace of the calls that led to the error.

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

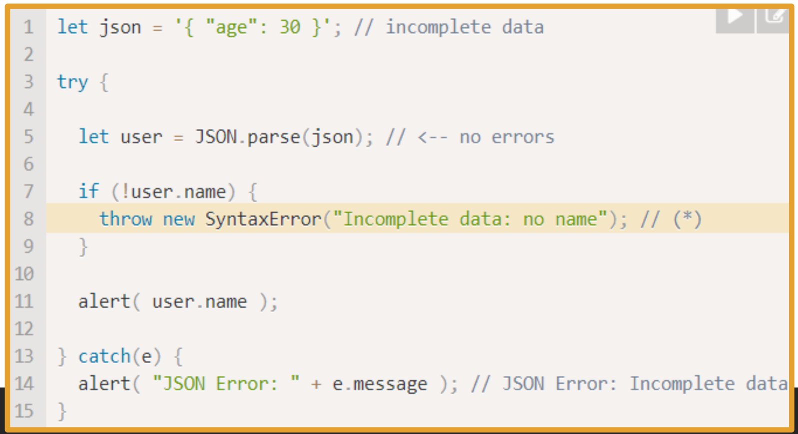
The *Finally* Block always executes.



```javascript
let error = new Error(message);
// or
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```



```javascript
try {

  alert('Start of try runs');  // (1) <--

  lalala; // error, variable is not defined!

  alert('End of try (never reached)');  // (2)

} catch(err) {

  alert(`Error has occurred!`); // (3) <--

}
```



```javascript
let json = '{ "age": 30 }'; // incomplete data

try {

  let user = JSON.parse(json); // <-- no errors

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // (*)
  }

  alert( user.name );

} catch(e) {
  alert( "JSON Error: " + e.message ); // JSON Error: Incomplete data
}
```