

Function Basics

Imagine being able to take one of your scripts and bundling it into a little package that you could use over and over again without having to rewrite the code. That's the power of functions.

Consider the following example:

```
function favoriteAnimal(animal) {  
    return animal + " is my favorite animal!"  
}  
  
console.log(favoriteAnimal('Goat'))
```

In JavaScript, parameters are the items listed between the parentheses () in the function declaration. Function arguments are the actual values we decide to pass to the function.

The function definition is: `function favoriteAnimal(animal)`. The parameter `animal` is found inside the parentheses. We could just as easily have replaced `animal` with `pet`, `x`, or `blah`. The name `animal` gives context on what the variable will/should contain.

Putting `animal` inside the parentheses tells JavaScript to send `some` value to our function. So `animal` is just a **placeholder** for some value.

The last line, `favoriteAnimal('Goat')`, is where we are calling the `favoriteAnimal` function and passing the value `'Goat'` inside that function. `Goat` is our argument. We are saying to use `'Goat'` wherever the `animal` placeholder is.

Calling `favoriteAnimal()` inside of `console.log()` prints the return value of the function to the console.

Built-in browser functions

Examples of functions are in the following:

```
const myText = "I am a string";  
const newString = myText.replace("string", "sausage");  
console.log(newString);  
// the replace() string function takes a source string,
```

```
// and a target string and replaces the source string,  
// with the target string, and returns the newly formed string
```

```
const myArray = ["I", "love", "chocolate", "frogs"];  
const madeAString = myArray.join(" ");  
console.log(madeAString);  
// the join() function takes an array, joins  
// all the array items together into a single  
// string, and returns this new string
```

```
const myNumber = Math.random();  
// the random() function generates a random number between  
// 0 and up to but not including 1, and returns that number
```

Some of the code you call when invoking a built-in browser function are written in other languages like C++. Some built-in browser functions are built on top of the default JavaScript language as part of browser APIs.

Functions that are part of objects are called **methods**.

A function to get a random whole number between 0 and a specified number (right-exclusive) is:

```
function random(number) {  
    return Math.floor(Math.random() * number);  
}
```

A function declaration is 'hoisted' meaning you can call the function above the function definition and it will work fine.

Sometimes parameters are optional. If you don't specify them, the function will generally adopt some kind of default behavior. For instance, the array `join()` function's parameter is optional:

```
const myArray = ["I", "love", "chocolate", "frogs"];  
const madeAString = myArray.join(" ");  
console.log(madeAString);  
// returns 'I love chocolate frogs'
```

```
const madeAnotherString = myArray.join();
console.log(madeAnotherString);
// returns 'I,love,chocolate,frogs'
```

If no parameter is included to specify a joining/delimiting character, a comma is used by default.

Default parameters

If you're writing a function and want to support optional parameters, you can specify default values by adding `=` after the name of the parameter, followed by the default value:

```
function hello(name = "Chris") {
  console.log(`Hello ${name}!`);
}

hello("Ari"); // Hello Ari!
hello(); // Hello Chris!
```

Anonymous functions and arrow functions

So far we have just created a function like so:

```
function myFunction() {
  alert("hello");
}
```

But you can also create a function that doesn't have a name:

```
(function () {
  alert("hello");
});
```

This is called an **anonymous function**, because it has no name. You'll often see anonymous functions when a function expects to receive another function as a parameter. In this case, the function parameter is often passed as an anonymous function.

This form of creating a function is also known as *function expression*. Unlike function declarations, function expressions are not hoisted.

Anonymous function example

Let's say you want to run some code when the user types into a text box. To do this you can call the `addEventListener()` function of the text box. It expects (at least) two parameters:

- the name of the event to listen for, which in this case is `keydown`
- a function to run when the event happens

When the user presses a key, the browser will call the function you provided, and will pass it a parameter containing information about this event, including the particular key that the user pressed:

```
function logKey(event) {  
  console.log(`You pressed "${event.key}".`);  
}  
  
textBox.addEventListener("keydown", logKey);
```

Instead of defining a separate `logKey()` function, you can pass an anonymous function into `addEventListener()`:

```
textBox.addEventListener("keydown", function (event) {  
  console.log(`You pressed "${event.key}".`);  
});
```

Arrow functions

If you pass an anonymous function like this, there's an alternative form you can use, called an **arrow function**. Instead of `function(event)`, you write `(event) =>`:

```
textBox.addEventListener("keydown", (event) => {  
  console.log(`You pressed "${event.key}".`);  
});
```

If the function only takes one parameter, you can omit the parentheses around the parameter:

```
textBox.addEventListener("keydown", event => {
  console.log(`You pressed "${event.key}".`);
});
```

Finally, if your function contains only one line that's a `return` statement, you can also omit the braces and the `return` keyword and implicitly return the expression. In this example, we're using the `map()` method of `Array` to double every value in the original array:

```
const originals = [1, 2, 3];

const doubled = originals.map(item => item * 2);

console.log(doubled); // [2, 4, 6]
```

The `map()` method takes each item in the array in turn, passing it into the given function. It then takes the value returned by that function and adds it to a new array.

So in the example above, `item => item * 2` is the arrow function equivalent of:

```
function doubleItem(item) {
  return item * 2;
}
```

You can use the same concise syntax to rewrite the `addEventListener` example:

```
textBox.addEventListener("keydown", (event) =>
  console.log(`You pressed "${event.key}".`),
);
```

This is equivalent to:

```
function logKey(event) {
  return console.log(`You pressed "${event.key}".`);
}
```

This will call `console.log()` and implicitly return its return value which is `undefined` from the callback function.

Arrow functions are recommended.

There are some subtle differences between arrow functions and normal functions though.

Here's a complete working example of the "keydown" example:

HTML

```
<input id="textBox" type="text" />
<div id="output"></div>
```

JS

```
const textBox = document.querySelector("#textBox");
const output = document.querySelector("#output");

textBox.addEventListener("keydown", (event) => {
  output.textContent = `You pressed "${event.key}".`;
});
```

Function scope and conflicts

When you create a function, the variables and other things defined inside the function are inside their own separate **scope**, meaning they are locked away in their own separate compartments, unreachable from code outside the functions.

The top-level outside all your functions is called the global scope. Values defined in the global scope are accessible from everywhere in the code.

Here is an example of name conflicts:

HTML

```
<!-- Excerpt from my HTML -->
<script src="first.js"></script>
<script src="second.js"></script>
<script>
```

```
greeting();
</script>
```

JS

```
// first.js
const name = "Chris";
function greeting() {
  alert(`Hello ${name}: welcome to our company.`);
}
```

JS

```
// second.js
const name = "Zaptec";
function greeting() {
  alert(`Our company is called ${name}.`);
}
```

Both functions you want to call are called `greeting()` , but you can only ever access the `first.js` file's `greeting()` function (the second one is ignored). In addition, an error results when attempting (in the `second.js` file) to assign a new value to the `name` variable because it was already declared with `const` , and so can't be reassigned.

Keeping variables locked away in functions avoid things like re-definitions.

Function return values

Example:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>Function library example</title>
  <style>
```

```
    input {
      font-size: 2em;
      margin: 10px 1px 0;
    }
  </style>
</head>
<body>

  <input class="numberInput" type="text">
<p></p>

  <script>
    const input = document.querySelector('.numberInput');
    const para = document.querySelector('p');

    function squared(num) {
      return num * num;
    }

    function cubed(num) {
      return num * num * num;
    }

    function factorial(num) {
      if (num < 0) return undefined;
      if (num === 0) return 1;
      let x = num - 1;
      while (x > 1) {
        num *= x;
        x--;
      }
      return num;
    }

    input.addEventListener("change", () => {
      const num = parseFloat(input.value);
      if (isNaN(num)) {
        para.textContent = "You need to enter a number!";
      } else {
        para.textContent = `${num} squared is ${squared(num)}. `;
      }
    });
  </script>
</body>
</html>
```



```

        para.textContent += `${num} cubed is ${cubed(num)}. `;
        para.textContent += `${num} factorial is ${factorial(num)}. `;
    }
});

</script>
</body>
</html>

```

By adding a listener to the `change` event, this function runs whenever the `change` event fires on the text input, that is, when a new value is entered into the text `input`, and submitted (for example, enter a value, then un-focus the input by pressing `TAB` or `RETURN`). When this anonymous function runs, the value in the `input` is stored in the `num` constant.

It's generally a good idea to check that any necessary parameters are validated (as the input's value is here), and that any optional parameters have some kind of default value provided.

You can change an outer variable inside a function without something like the `global` keyword in Python.

Example:



```

1  let userName = 'John';
2
3  function showMessage() {
4      userName = "Bob"; // (1) changed the outer variable
5
6      let message = 'Hello, ' + userName;
7      alert(message);
8  }
9
10 alert( userName ); // John before the function call
11
12 showMessage();
13
14 alert( userName ); // Bob, the value was modified by the function

```

The outer variable is only used if there's no local one.

If a same-named variable is declared inside the function then it *shadows* the outer one. For instance, in the code below the function uses the local `userName`. The outer one is ignored:

```

1  let userName = 'John';
2
3  function showMessage() {
4      let userName = "Bob"; // declare a local variable
5
6      let message = 'Hello, ' + userName; // Bob
7      alert(message);
8  }
9
10 // the function will create and use its own userName
11 showMessage();
12
13 alert( userName ); // John, unchanged, the function did not access the outer

```

It's a good practice to limit the use of global variables.

Parameters

In this example, we have a variable `from` and pass it to the function. Please note: the function changes `from`, but the change is not seen outside, because a function always gets a copy of the value:

```

1  function showMessage(from, text) {
2
3      from = '*' + from + '*'; // make "from" look nicer
4
5      alert( from + ': ' + text );
6  }
7
8  let from = "Ann";
9
10 showMessage(from, "Hello"); // *Ann*: Hello
11
12 // the value of "from" is the same, the function modified a local copy
13 alert( from ); // Ann

```

Default values

If a function is called, but an argument is not provided, then the corresponding value becomes `undefined`. For instance, the aforementioned function `showMessage(from, text)` can be called with a single argument:

```
1 showMessage("Ann");
```

That's NOT an error. Such a call would output `"*Ann*: undefined"`.

We can specify the so-called "default" (to use if omitted) value for a parameter in the function declaration, using `=`:

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

Now if the `text` parameter is not passed, it will get the value `"no text given"`.

The default value also jumps in if the parameter exists, but STRICTLY equals `undefined`, like this:

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

Here `"no" text given` is a string, but it can be a more complex expression, which is only evaluated and assigned if the parameter is missing. So, this is also possible:

```
1 function showMessage(from, text = anotherFunction()) {  
2   // anotherFunction() only executed if no text given  
3   // its result becomes the value of text  
4 }
```

In JavaScript, a default parameter is evaluated every time the function is called without the respective parameter.

In the example above, `anotherFunction()` isn't called at all, if the `text` parameter is provided.

On the other hand, it's INDEPENDENTLY called EVERY TIME when `text` is missing.

When JavaScript didn't support syntax for default parameters, people would try using other ways to specify them like:

```

1 function showMessage(from, text) {
2   if (text === undefined) {
3     text = 'no text given';
4   }
5
6   alert( from + ": " + text );
7 }

```

or using the `||` operator:

```

1 function showMessage(from, text) {
2   // If the value of text is falsy, assign the default value
3   // this assumes that text == "" is the same as no text at all
4   text = text || 'no text given';
5   ...
6 }

```

These tests can be done anywhere in the function.

Modern JavaScript engines support the nullish coalescing operator `??` which is better when most falsy values, such as `0`, should be considered "normal" (different than the behavior of `||`):

```

1 function showCount(count) {
2   // if count is undefined or null, show "unknown"
3   alert(count ?? "unknown");
4 }
5
6 showCount(0); // 0
7 showCount(null); // unknown
8 showCount(); // unknown

```

`confirm` is like `alert` or `prompt` but returns a boolean:

```

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 }

```

Returning a value

If a function does not return a value, it is the same as if it returns `undefined`:

```

1  function doNothing() { /* empty */ }
2
3  alert( doNothing() === undefined ); // true

```

An empty `return` is also the same as `return undefined`:

```

1  function doNothing() {
2      return;
3  }
4
5  alert( doNothing() === undefined ); // true

```

For a long expression in `return`, it might be tempting to put it on a separate line:

```

1  return
2  (some + long + expression + or + whatever * f(a) + f(b))

```

But this doesn't work, because JavaScript assumes a semicolon after `return`. This'll work the same as:

```
1 return;  
2 (some + long + expression + or + whatever * f(a) + f(b))
```

So, it effectively becomes an empty return.

If we want the returned expression to wrap across multiple lines, we should start it at the same line as `return`. Or at least put the opening parentheses there as follows:

```
1 return (  
2     some + long + expression  
3     + or +  
4     whatever * f(a) + f(b)  
5     )
```

This will work as we expect it to.

Naming a function

A function should only do the action it is named for, nothing more. A few examples of breaking this rule are:

- `getAge` would be bad if it shows an `alert` with the age (should only get).
- `createForm` would be bad if it modifies the document, adding a form to it (should only create it and return).
- `checkPermission` would be bad if it displays the `access granted/denied` message (should only perform the check and return the result).

Function expressions

Function declaration looks like:

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

Another syntax is called a *Function Expression*. It allows us to create a new function in the middle of any expression. For example:

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

Here we can see a variable `sayHi` getting a value, the new function, created as `function() { alert("Hello"); }`.

The function creation happens in the context of the assignment expression.

There's no name after the `function` keyword. Omitting a name is allowed for Function Expressions.

We are creating a function and putting it into the variable `sayHi`.

In more advanced situations, a function may be created and immediately called or scheduled for a later execution, not stored anywhere, thus remaining anonymous.

Function is a value

No matter how a function is created, a function is a value.

We can even print out that value using `alert`:

```
1 function sayHi() {  
2   alert( "Hello" );  
3 }  
4  
5 alert( sayHi ); // shows the function code
```

The last line does not run the function, because there are no parentheses after `sayHi`. The code above shows its string representation, which is the source code.

A function is a special value since for instance we can call it like `sayHi()` but it's still a value. So we can work with it like other kinds of values.

We can copy a function to another variable:

```
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)
```

We could also have used a Function Expression to declare `sayHi` , in the first line:

```
1 let sayHi = function() { // (1) create
2   alert( "Hello" );
3 };
4
5 let func = sayHi;
6 // ...
```

Callback functions

We'll write a function `ask(question, yes, no)` with three parameters:

`question`

Text of the question

`yes`

Function to run if the answer is "Yes"

`no`

Function to run if the answer is "No"

The function should ask the `question` and, depending on the user's answer, call `yes()` or `no()` :


```

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);

```

In practice, such functions are useful. The major difference between a real-life `ask` and the example above is that real-life functions use more complex ways to interact with the user than a simple `confirm`. In the browser, such functions usually draw a nice-looking question window.

The arguments `showOk` and `showCancel` of `ask` are called *callback functions*.

The idea is that we pass a function and expect it to be "called back" later if necessary. In our case, `showOk` becomes the callback for the "yes" answer, and `showCancel` for "no" answer.

We can use Function Expressions to write an equivalent, shorter function:

```

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 );

```

Here, functions are declared right inside the `ask(...)` call. They have no name, and so are called *anonymous*. Such functions are not accessible outside of `ask` (because they are not assigned to variables), but that's just what we want here.

Such code is in the spirit of JavaScript.

Function Expression vs. Function Declaration

A first key difference between Function Declarations and Expressions is the syntax.

Function Declaration

A function declared as a separate statement, in the main code flow:

```
1 // Function Declaration
2 function sum(a, b) {
3   return a + b;
4 }
```

Function Expression

A function created inside an expression or inside another syntax construct. Here the function is created on the right side of the assignment:

```
1 // Function Expression
2 let sum = function(a, b) {
3   return a + b;
4 };
```

The more subtle difference is *when* a function is created by the JavaScript Engine.

A Function Expression is created when the execution reaches it and is usable only from that moment.

Once the execution flow passes to the right side of the assignment `let sum = function...` - the function is created and can be used (assigned, called, etc.) from now on.

Function Declarations are different.

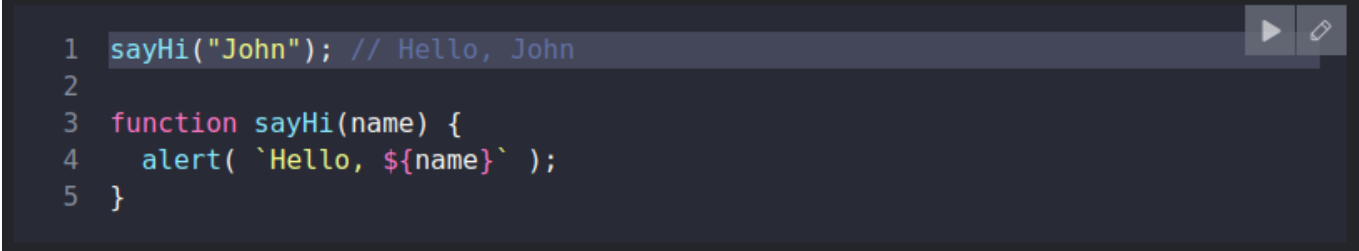
A Function Declaration can be called earlier than it is defined.

For example, a global Function Declaration is visible in the whole script, no matter where it is.

That's due to internal algorithms. When JavaScript prepares to run the script, it first looks for global Function Declarations in it and creates the functions. We can think of it as an "initialization stage".

And after all Function Declarations are processed, the code is executed. So it has access to these functions.

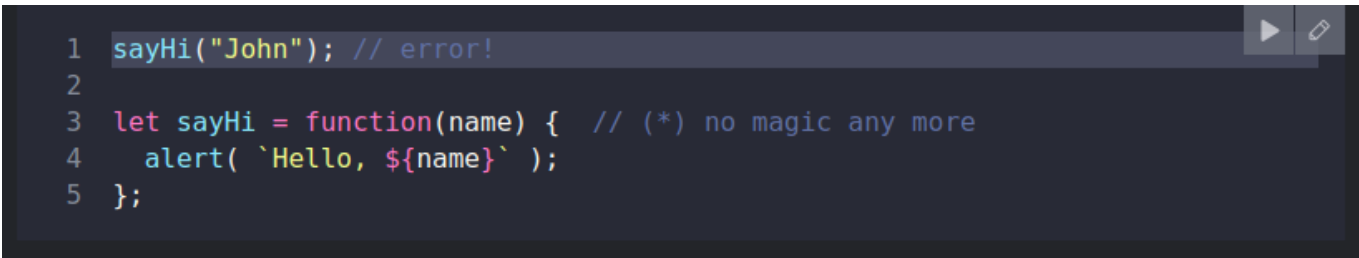
For example, this works:

A code editor with a dark background. Line 1: `sayHi("John"); // Hello, John`. Line 2: (empty). Line 3: `function sayHi(name) {`. Line 4: `alert(`Hello, ${name}`);`. Line 5: `}`. There are play and edit icons in the top right corner.

```
1 sayHi("John"); // Hello, John
2
3 function sayHi(name) {
4   alert( `Hello, ${name}` );
5 }
```

The Function Declaration `sayHi` is created when JavaScript is preparing to start the script and so is visible everywhere in it.

A Function Expression wouldn't work:

A code editor with a dark background. Line 1: `sayHi("John"); // error!`. Line 2: (empty). Line 3: `let sayHi = function(name) { // (*) no magic any more`. Line 4: `alert(`Hello, ${name}`);`. Line 5: `};`. There are play and edit icons in the top right corner.

```
1 sayHi("John"); // error!
2
3 let sayHi = function(name) { // (*) no magic any more
4   alert( `Hello, ${name}` );
5 };
```

Function Expressions are created when the execution reaches them. That would happen only in the line `(*)`. Too late.

Another special feature of Function Declarations is their block scope.

In strict mode, when a Function Declaration is within a code block, it's visible everywhere inside that block. But not outside of it.

For instance, let's imagine we need to declare a function `welcome()` depending on the `age` variable that we get during runtime. We will then use it later (outside the conditional if statement).

If we use Function Declaration, it won't work as intended:

```

1 let age = prompt("What is your age?", 18);
2
3 // conditionally declare a function
4 if (age < 18) {
5
6     function welcome() {
7         alert("Hello!");
8     }
9
10 } else {
11
12     function welcome() {
13         alert("Greetings!");
14     }
15
16 }
17
18 // ...use it later
19 welcome(); // Error: welcome is not defined

```

That's because a Function Declaration is only visible inside the code block in which it resides.

Another example:

```

1 let age = 16; // take 16 as an example
2
3 if (age < 18) {
4     welcome(); // \ (runs)
5                 // |
6     function welcome() { // |
7         alert("Hello!"); // | Function Declaration is available
8     }                 // | everywhere in the block where it's declared
9                 // |
10    welcome(); // / (runs)
11
12 } else {
13
14     function welcome() {
15         alert("Greetings!");
16     }
17 }
18
19 // Here we're out of curly braces,
20 // so we can not see Function Declarations made inside of them.
21
22 welcome(); // Error: welcome is not defined

```

The correct approach would be to use a Function Expression and assign `welcome` to the variable that is declared outside of `if` and has the proper visibility.

This code works as intended:

```
1 let age = prompt("What is your age?", 18);
2
3 let welcome;
4
5 if (age < 18) {
6
7     welcome = function() {
8         alert("Hello!");
9     };
10
11 } else {
12
13     welcome = function() {
14         alert("Greetings!");
15     };
16
17 }
18
19 welcome(); // ok now
```

We could simplify it further using the `?` ternary operator:

```
1 let age = prompt("What is your age?", 18);
2
3 let welcome = (age < 18) ?
4     function() { alert("Hello!"); } :
5     function() { alert("Greetings!"); };
6
7 welcome(); // ok now
```

When to choose which?

As a rule of thumb, when we need to declare a function, the first thing to consider is Function Declaration syntax. It gives more freedom in how to organize our code, because we can call such functions before they are declared.

That's also better for readability as it's easier to look up `function f(...) {...}` in the code than `let f = function(...) {...}`. Function Declarations are more "eye-catching".

But if a Function Declarations does not suit us for some reason, or we need a conditional declaration, then Function Expression should be used.

Arrow functions, the basics

There's another very simple and concise syntax for creating functions, that's often better than doing Function Expressions.

These are "arrow functions" that look like:

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

This creates a function `func` that accepts arguments `arg1..argN`, then evaluates the `expression` on the right side with their use and returns its result.

It's a shorter version of:

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

A concrete example:

```
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
```

`(a, b) => a + b` means a function that accepts two arguments named `a` and `b`. Upon execution, it evaluates the expression `a + b` and returns the result.

- If we have only one argument, then we can omit the parentheses around the parameter:

```
1 let double = n => n * 2;
2 // roughly the same as: let double = function(n) { return n * 2 }
3
4 alert( double(3) ); // 6
```

If there are no arguments, parentheses are empty but must be included still:

```
1 let sayHi = () => alert("Hello!");
2
3 sayHi();
```

Arrow functions can be used in the same way as Function Expressions. For example, to dynamically create a function:

```
1 let age = prompt("What is your age?", 18);
2
3 let welcome = (age < 18) ?
4   () => alert('Hello!') :
5   () => alert("Greetings!");
6
7 welcome();
```

Multiline arrow functions

Sometimes we need a more complex function, with multiple expressions and statements. In this case, we enclose them in curly braces. The major difference is that curly braces require a `return` within them to return a value (just like a regular function does).

Like this:

```
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
```

There are other features of arrow functions which will be discussed later.

JavaScript Call Stack

Stack Overflow

The call stack has a fixed size, depending on the implementation of the host environment, either the web browser or Node.js.

If there are too many execution contexts on the call stack, stack overflow will occur.

For instance, when a recursive function has no exit condition, the JavaScript engine will issue a stack overflow error.

Asynchronous JavaScript

JavaScript is a single-threaded programming language. This means that the JavaScript engine has only one call stack. Therefore, it can only do one thing at a time.

When executing a script, the JavaScript engine executes code from top to bottom, line by line. In other words, it is synchronous.

Asynchronous means the JavaScript engine can execute other tasks while waiting for another task to be completed. For example, the JavaScript engine can:

- Request for data from a remote server
- Display a spinner
- When the data is available, display it on the webpage

To do this, the JavaScript engine uses an event loop.