Overview

This question is intended as an introduction to JavaScript functions. This editorial will cover their syntax and topics like closures and higher-order functions.

Function Syntax

In JavaScript, there are two main ways to declare a function. One of which is to use the function keyword.

Basic Syntax

The syntax is:

function f(a, b) {

const sum = a + b;

return sum;

}

console.log(f(3, 4)); // 7

In this example, f is the name of the function. (a, b) are the arguments. You can write any logic in the body and finally return a result. You are allowed to return nothing, and it will instead implicitly return undefined.

Anonymous Function

You can optionally exclude the name of the function after the function keyword.

var f = function(a, b) {

const sum = a + b;

return sum;

}

console.log(f(3, 4)); // 7

Immediately Invoked Function Expression (IIFE)

You can create a function and immediately execute it in Javascript.

const result = (function(a, b) {

const sum = a + b;

return sum;

})(3, 4);

console.log(result); // 7

Why would you write code like this? It gives you the opportunity to encapsulate a variable within a new scope. For example, another developer can immediately see that sum can't be used anywhere outside the function body.

Functions Within Functions

A powerful feature of JavaScript is you can actually create functions within other functions and even return them!

function createFunction() {

function f(a, b) {

const sum = a + b;

return sum;

}

return f;

}

const f = createFunction();

console.log(f(3, 4)); // 7

In this example, createFunction() returns a new function. Then that function can be used as normal.

Function Hoisting

JavaScript has a feature called hoisting where a function can sometimes be used before it is initialized. You can only do this if you declare functions with the function syntax.

function createFunction() {

return f;

function f(a, b) {

const sum = a + b;

return sum;

}

}

const f = createFunction();

console.log(f(3, 4)); // 7

In this example, the function is returned before it is initialized. Although it is valid syntax, it is sometimes considered bad practice as it can reduce readability.

Closures

An important topic in JavaScript is the concept of closures. When a function is created, it has access to a reference to all the variables declared around it, also known as it's lexical environment. The combination of the function and its enviroment is called a closure. This is a powerful and often used feature of the language.

function createAdder(a) {

function f(b) {

const sum = a + b;

return sum;

}

return f;

}

const f = createAdder(3);

console.log(f(4)); // 7

In this example, createAdder passes the first parameter a and the inner function has access to it. This way, createAdder serves as a factory of new functions, with each returned function having different behavior.

Arrow Syntax

The other common way to declare functions is with arrow syntax. In fact, on many projects, it is the preferred syntax.

Basic Syntax

const f = (a, b) => {

const sum = a + b;

return sum;

};

console.log(f(3, 4)); // 7

In this example, f is the name of the function. (a, b) are the arguments. You can write any logic in the body and finally return a result. You are allowed to return nothing, and it will instead implicitly return undefined.

Omit Return

If you can write the code in a single line, you can omit the return keyword. This can result in very short code.

const f = (a, b) => a + b;

console.log(f(3, 4)); // 7

Differences

There are 3 major differences between arrow syntax and function syntax.

More minimalistic syntax. This is especially true for anonymous functions and single-line functions. For this reason, this way is generally preferred when passing short anonymous functions to other functions.

No automatic hoisting. You are only allowed to use the function after it was declared. This is generally considered a good thing for readability.

Can't be bound to this, super, and arguments or be used as a constructor. These are all complex topics in themselves but the basic takeaway should be that arrow functions are simpler in their feature set. You can read more about these differences here.

The choice of arrow syntax versus function syntax is primarily down to preference and your project's stylistic standards.

Rest Arguments

You can use rest syntax to access all the passed arguments as an array. This isn't necessary for this problem, but it will be a critical concept for many problems. You can read more about ... syntax here.

Basic Syntax

The syntax is:

function f(...args) {

const sum = args[0] + args[1];

return sum;

}

console.log(f(3, 4)); // 7

In this example the variable args is [3, 4].

Why

It may not be immediately obvious why you would use this syntax because you can always just pass an array and get the same result.

The primary use-case is for creating generic factory functions that accept any function as input and return a new version of the function with some specific modification.

By the way, a function that accepts a function and/or returns a function is called a higher-order function, and they are very common in JavaScript.

For example, you can create a logged function factory:

function log(inputFunction) {

return function(...args) {

console.log("Input", args);

const result = inputFunction(...args);

console.log("Output", result);

return result;

}

}

const f = log((a, b) => a + b);

f(1, 2); // Logs: Input [1, 2] Output 3

Solutions to Problem

Now let's apply these different ways of writing JavaScript functions to solve this problem.

Function Syntax

var createHelloWorld = function() {

return function() {

return "Hello World";

}

};

Arrow Syntax

var createHelloWorld = function() {

return () => "Hello World";

};

Arrow Syntax + Rest Arguments

var createHelloWorld = function() {

return (...args) => "Hello World";

};