JavaScript

When to Use a Function Expression vs. Function Declaration

Vanilla JavaScript

Paul Wilkins
March 22, 2022

Share

> There are two ways to create functions in JavaScript: function expressions and function declarations. In this article, we will discuss when to use function expressions vs. function declarations, and explain the differences between them.

Function declarations have been used for a long time, but function expressions have been gradually taking over. Many developers aren't sure when to use one or the other, so they end up using the wrong one.

There are a few key differences between function expressions and function declarations. Let's take a closer look at those differences, and when to use function expressions vs. function declarations in your code.

```
function funcDeclaration() {
    return 'A function declaration';
}

let funcExpression = function () {
    return 'A function expression';
}
```

Privacy

What Are Function Declarations?

Function declarations are when you create a function and give it a name. You declare the name of the function when you write the function keyword, followed by the function name. For example:

```
let myFunction = function() {
  // do something
};
```

As you can see, the function name (`myFunction`) is declared when the function is created. This means that you can call the function before it is defined.

Here is an example of a function declaration:

```
function add (a, b) {
  return a + b;
};
```

What Are Function Expressions?

Function expressions are when you create a function and assign it to a variable. The function is anonymous, which means it doesn't have a name. For example:

```
let myFunction = function() {
  // do something
};
```

As you can see, the function is assigned to the `myFunction` variable. This means that you must define the function before you can call it.

Here is an example of a function expression:

```
let add = function (a, b) {
  return a + b;
};
```

The Differences Between Function Expressions & Declarations

There are a few key differences between function expressions and function declarations:

- Function declarations are hoisted, while function expressions are not. This means that you can call a function declaration before it is defined, but you cannot do this with a function expression.
- With function expressions, you can use a function immediately after it is defined. With function declarations, you have to wait until the entire script has been parsed.
- Function expressions can be used as an argument to another function, but function declarations cannot.
- Function expressions can be anonymous, while function declarations cannot.

Understanding Scope in Your Function Expression: JavaScript Hoisting Differences

Similar to the `let` statement, function declarations **are hoisted to the top** of other code.

Function expressions aren't hoisted. This allows them to retain a copy of the local variables from the scope where they were defined.

Normally, you can use function declarations and function expressions interchangeably. But there are times when function expressions result in easier-to-understand code without the need for a temporary function name.

How to Choose Between Expressions and Declarations

So, when should you use function expressions vs. function declarations?

The answer depends on your needs. If you need a more flexible function or one that is not hoisted, then a function expression is the way to go. If you need a more readable and understandable function, then use a function declaration.

As you've seen, the two syntaxes are similar. The most obvious difference is that function expressions are anonymous, while function declarations have a name.

Today, you would typically use a function declaration when you need to do something that function expressions cannot do. If you don't need to do anything that can only be done with a function declaration, then it is generally best to use a function expression.

Use function declarations when you need to create a function that is recursive, or when you need to call the function before you define it. As a rule of thumb, use function expressions for cleaner code when you don't need to do either of those things.

Benefits of function declarations

There are a few key benefits to using function declarations.

- **It can make your code more readable.** If you have a long function, giving it a name can help you keep track of what it's doing.
- **Function declarations are hoisted**, which means that they are available before they are defined in your code. This helps if you need to use the function before it is defined.

Benefits of function expressions

Function expressions also have a few benefits.

- **They are more flexible than function declarations.** You can create function expressions and assign them to different variables, which can be helpful when you need to use the same function in different places.
- **Function expressions are not hoisted**, so you can't use them before they are defined in your code. This helps if you want to make sure that a function is only used after it is defined.

When to choose a function declaration vs. function expression

In most cases, it's easy to figure out which method of defining a function is best for your needs. These guidelines will help you make a quick decision in most situations.

Use a function declaration when:

- you need a more readable and understandable function (such as a long function, or one you'll need to use in different places)
- an anonymous function won't suit your needs
- you need to create a function that is recursive
- you need to call the function before it is defined

Use a function expression when:

- you need a more flexible function
- you need a function that isn't hoisted
- the function should only used when it is defined
- the function is anonymous, or doesn't need a name for later use
- you want to control when the function is executed, using techniques like immediately-invoked function expressions (IIFE)
- you want to pass the function as an argument to another function

That said, there are a number of cases where the flexibility of function expressions becomes a powerful asset.

Unlocking the Function Expression: JavaScript Hoisting Differences

There are several different ways that function expressions become more useful than function declarations.

- Closures
- Arguments to other functions
- Immediately Invoked Function Expressions (IIFE)

Creating Closures with Function Expressions

Closures are used when you want to give parameters to a function before that function is executed. A good example of how this can benefit you is when looping through a `NodeList`.

A closure allows you to retain other information such as the index, in situations where that information isn't available once the function is executed.

```
function tabsHandler(index) {
    return function tabClickEvent(evt) {
        // Do stuff with tab.
        // The index variable can be accessed from within here.
    };
}

let tabs = document.querySelectorAll('.tab'),
    i;

for (i = 0; i &lt; tabs.length; i += 1) {
    tabs[i].onclick = tabsHandler(i);
}
```

Privacy

The attached event handlers are executed at a later time (after the loop is finished), so a closure is needed to retain the appropriate value of the `for` loop.

```
// Bad code, demonstrating why a closure is needed
let i;

for (i = 0; i &lt; list.length; i += 1) {
    document.querySelector('#item' + i).onclick = function doSomething(evt)
        // Do something with item i
        // But, by the time this function executes, the value of i is alway
    }
}
```

It's easier to understand why the problem occurs by extracting the `doSomething()` function out from within the `for` loop.

```
// Bad code, demonstrating why a closure is needed

let list = document.querySelectorAll('.item'),
    i,
    doSomething = function (evt) {
        // Do something with item i.
        // But, by the time this function executes, the value of i is not w
    };

for (i = 0; i &lt; list.length; i += 1) {
    item[i].onclick = doSomething;
}
```

The solution here is to pass the index as a function argument to an outer function so that it can pass that value to an inner function. You'll commonly see handler functions used to organize the information that an inner returning function needs.

```
// The following is good code, demonstrating the use of a closure
```

```
let list = ['item1', 'item2', 'item3'],
    i,
    doSomethingHandler = function (itemIndex) {
        return function doSomething(evt) {
            // now this doSomething function can retain knowledge of
            // the index variable via the itemIndex parameter,
            // along with other variables that may be available too.
            console.log('Doing something with ' + list[itemIndex]);
        };
    };

for (i = 0; i &lt; list.length; i += 1) {
    list[i].onclick = doSomethingHandler(i);
}
```

Learn more about **closures and their usage**.

Passing Function Expressions as Arguments

Function expressions can be passed directly to functions without having to be assigned to an intermediate temporary variable.

You'll most often see them in the form of an anonymous function. Here's a familiar jQuery function expression example:

```
$(document).ready(function () {
    console.log('An anonymous function');
});
```

A function expression is also used to handle the array items when using methods such as `forEach()`.

They don't have to be unnamed anonymous functions, either. It's a good idea to name the function expression to help express what the function is supposed to do and to aid in debugging:

Privacy

```
let productIds = ['12356', '13771', '15492'];

productIds.forEach(function showProduct(productId) {
    ...
});
```

Immediately Invoked Function Expressions (IIFE)

IIFEs help prevent your functions and variables from affecting the global scope.

All the properties within fall inside the anonymous function's scope. This is a common design pattern that's used to prevent your code from having unwanted or undesired side-effects elsewhere.

It's also used as a module pattern to contain blocks of code in easy-to-maintain sections. We take a deeper look at these in **Demystifying JavaScript closures, callbacks, and IIFEs**.

Here's a simple example of an IIFE:

```
(function () {
    // code in here
}());
```

… which when used as a module, can result in some easy-to-achieve maintainability for your code.

```
let myModule = (function () {
    let privateMethod = function () {
        console.log('A private method');
    },
    someMethod = function () {
        console.log('A public method');
    },
    anotherMethod = function () {
```

Privacy

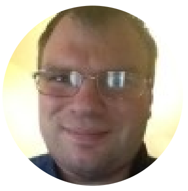
```
        console.log('Another public method');
    };

    return {
        someMethod: someMethod,
        anotherMethod: anotherMethod
    };
}());
```

Conclusion

As we've seen, function expressions aren't radically different from function declarations, but they can often result in cleaner and more readable code.

Their widespread use makes them an essential part of every developer's toolbox. Do you use function expressions in your code in any interesting ways that I haven't mentioned above? Comment and let me know!

Share This Article



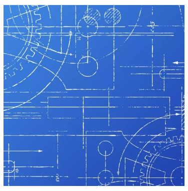
Paul Wilkins

I'm a web developer living in Christchurch (thanks for all the quakes) where JavaScript is my forte. When off the computer I volunteer down at the local community centre, or enjoy playing tabletop games such as Carcassonne or Stone Age with friends.

functional-js **functions** **Hoisting** **iife** **JavaScript functions** **javascript scope** **variable scope**

Privacy

Up Next



Why Maven Cannot Generate Your Module Declaration

Robert Scholte

Basic JavaScript Regular Expression Example

Sam Deering

Will JavaScript Function Without the 'function' Statement?

Craig Buckler

Expression Blend Behaviors

Andrew Gardner

Expression Blend Behaviors

Andrew Gardner

Designing with Microsoft Expression Blend

Shane Morris

Privacy