1

BLOCK BINDINGS

Traditionally, the way variable declarations work has been one tricky part of programming in JavaScript. In most C-based languages, variables (more formally known as *bindings*, as a name is bound to a value inside a scope) are created at the spot where the declaration occurs. In JavaScript, however, this is not the case. Where your variables are actually created depends on how you declare them, and ECMAScript 6 offers options to make controlling scope easier. This chapter demonstrates why classic `var` declarations can be confusing, introduces block-level bindings in ECMAScript 6, and then offers some best practices for using them.

www.itbook.store/books/9781593277574

var Declarations and Hoisting

Variable declarations using `var` are treated as if they're at the top of the function (or in the global scope, if declared outside of a function) regardless of where the actual declaration occurs; this is called *hoisting*. For a demonstration of what hoisting does, consider the following function definition:

```
function getValue(condition) {

    if (condition) {
        var value = "blue";

        // other code

        return value;
    } else {

        // value exists here with a value of undefined

        return null;
    }

    // value exists here with a value of undefined
}
```

If you are unfamiliar with JavaScript, you might expect the variable `value` to be created only if `condition` evaluates to `true`. In fact, the variable `value` is created regardless. Behind the scenes, the JavaScript engine changes the `getValue` function to look like this:

```
function getValue(condition) {

    var value;

    if (condition) {
        value = "blue";

        // other code

        return value;
    } else {

        return null;
    }
}
```

The declaration of `value` is hoisted to the top, and the initialization remains in the same spot. That means the variable `value` is still accessible from within the `else` clause. If accessed from the `else` clause, the variable would just have a value of `undefined` because it hasn't been initialized in the `else` block.

www.itbook.store/books/9781593277574

It often takes new JavaScript developers some time to get used to declaration hoisting, and misunderstanding this unique behavior can end up causing bugs. For this reason, ECMAScript 6 introduces block-level scoping options to give developers more control over a variable's life cycle.

Block-Level Declarations

Block-level declarations declare bindings that are inaccessible outside a given block scope. *Block scopes*, also called *lexical scopes*, are created in the following places:

- Inside a function
- Inside a block (indicated by the { and } characters)

Block scoping is how many C-based languages work, and the introduction of block-level declarations in ECMAScript 6 is intended to provide that same flexibility (and uniformity) to JavaScript.

let Declarations

The `let` declaration syntax is the same as the syntax for `var`. You can basically replace `var` with `let` to declare a variable but limit the variable's scope to only the current code block (there are a few other subtle differences, which are discussed in "The Temporal Dead Zone" on page 6). Because `let` declarations are not hoisted to the top of the enclosing block, it's best to place `let` declarations first in the block so they're available to the entire block. Here's an example:

```
function getValue(condition) {

    if (condition) {
        let value = "blue";

        // other code

        return value;
    } else {

        // value doesn't exist here

        return null;
    }

    // value doesn't exist here
}
```

This version of the `getValue` function behaves more similarly to how you'd expect it to in other C-based languages. Because the variable `value` is declared using `let` instead of `var`, the declaration isn't hoisted to the top

www.itbook.store/books/9781593277574

of the function definition, and the variable `value` is no longer accessible once execution flows out of the `if` block. If `condition` evaluates to `false`, then `value` is never declared or initialized.

No Redeclaration

If an identifier has already been defined in a scope, using the identifier in a `let` declaration inside that scope causes an error to be thrown. For example:

```
var count = 30;

// throws an error
let count = 40;
```

In this example, `count` is declared twice: once with `var` and once with `let`. Because `let` will not redefine an identifier that already exists in the same scope, the `let` declaration will throw an error. Conversely, no error is thrown if a `let` declaration creates a new variable with the same name as a variable in its containing scope, as demonstrated in the following code:

```
var count = 30;

if (condition) {

    // doesn't throw an error
    let count = 40;

    // more code
}
```

This `let` declaration doesn't throw an error because it creates a new variable called `count` within the `if` statement instead of creating `count` in the surrounding block. Inside the `if` block, this new variable shadows the global `count`, preventing access to it until execution exits the block.

const Declarations

You can also define bindings in ECMAScript 6 with the `const` declaration syntax. Bindings declared using `const` are considered *constants*, meaning their values cannot be changed once set. For this reason, every `const` binding must be initialized on declaration, as shown in this example:

```
// valid constant
const maxItems = 30;

// syntax error: missing initialization
const name;
```


www.itbook.store/books/9781593277574

The maxItems binding is initialized, so its const declaration will work without a problem. However, the name binding would cause a syntax error if you tried to run the program containing this code because name is not initialized.

Constants vs. let Declarations

Constants, like let declarations, are block-level declarations. That means constants are no longer accessible once execution flows out of the block in which they were declared, and declarations are not hoisted, as demonstrated in this example:

```
if (condition) {
    const maxItems = 5;

    // more code
}

// maxItems isn't accessible here
```

In this code, the constant maxItems is declared within an if statement. After the statement finishes executing, maxItems is not accessible outside that block.

In another similarity to let, a const declaration throws an error when made with an identifier for an already defined variable in the same scope. It doesn't matter whether that variable was declared using var (for global or function scope) or let (for block scope). For example, consider this code:

```
var message = "Hello!";
let age = 25;

// each of these throws an error
const message = "Goodbye!";
const age = 30;
```

The two const declarations would be valid alone, but given the previous var and let declarations in this case, they are syntax errors.

Despite those similarities, there is one significant difference between let and const. Attempting to assign a const to a previously defined constant will throw an error in both strict and non-strict modes:

```
const maxItems = 5;

// throws an error
maxItems = 6;
```

Much like constants in other languages, the maxItems variable can't be assigned a new value later on. However, unlike constants in other languages, the value a constant holds can be modified if it is an object.

www.itbook.store/books/9781593277574

Object Declarations with const

A const declaration prevents modification of the binding, not of the value. That means const declarations for objects don't prevent modification of those objects. For example:

```
const person = {
    name: "Nicholas"
};

// works
person.name = "Greg";

// throws an error
person = {
    name: "Greg"
};
```

Here, the binding person is created with an initial value of an object with one property. It's possible to change person.name without causing an error because this changes what person contains but doesn't change the value that person is bound to. When this code attempts to assign a value to person (thus attempting to change the binding), an error will be thrown. This subtlety in how const works with objects is easy to misunderstand. Just keep in mind that const prevents modification of the binding, not modification of the bound value.

The Temporal Dead Zone

A variable declared with either let or const cannot be accessed until after the declaration. Attempting to do so results in a reference error, even when using normally safe operations, such as the typeof operation in this if statement:

```
if (condition) {
    console.log(typeof value);  // throws an error
    let value = "blue";
}
```

Here, the variable value is defined and initialized using let, but that statement is never executed because the previous line throws an error. The issue is that value exists in what the JavaScript community has dubbed the *temporal dead zone (TDZ)*. The TDZ is never named explicitly in the ECMAScript specification, but the term is often used to describe why let and const bindings are not accessible before their declaration. This section covers some subtleties of declaration placement that the TDZ causes, and although the examples shown use let, note that the same information applies to const.

When a JavaScript engine looks through an upcoming block and finds a variable declaration, it either hoists the declaration to the top of the function or global scope (for var) or places the declaration in the TDZ (for let and const). Any attempt to access a variable in the TDZ results in a runtime

www.itbook.store/books/9781593277574

error. That variable is only removed from the TDZ, and therefore is safe to use, once execution flows to the variable declaration.

This is true anytime you attempt to use a variable declared with `let` or `const` before it's been defined. As the previous example demonstrated, this even applies to the normally safe `typeof` operator. However, you can use `typeof` on a variable outside the block where that variable is declared without throwing an error, although it may not produce the results you're after. Consider this code:

```
console.log(typeof value);     // "undefined"

if (condition) {
    let value = "blue";
}
```

The variable `value` isn't in the TDZ when the `typeof` operation executes because it occurs outside the block in which `value` is declared. That means there is no `value` binding, and `typeof` simply returns `"undefined"`.

The TDZ is just one unique aspect of block bindings. Another unique aspect has to do with their use inside loops.

Block Bindings in Loops

Perhaps one area where developers most want block-level scoping of variables is within `for` loops, where the throwaway counter variable is meant to be used only inside the loop. For instance, it's not uncommon to see code like this in JavaScript:

```
for (var i = 0; i < 10; i++) {
    process(items[i]);
}

// i is still accessible here
console.log(i);                     // 10
```

In other languages where block-level scoping is the default, this example should work as intended—only the `for` loop should have access to the `i` variable. However, in JavaScript, the variable `i` is still accessible after the loop is completed because the `var` declaration is hoisted. Using `let` instead, as in the following code, should produce the intended behavior:

```
for (let i = 0; i < 10; i++) {
    process(items[i]);
}

// i is not accessible here - throws an error
console.log(i);
```

In this example, the variable `i` exists only within the `for` loop. When the loop is complete, the variable is no longer accessible elsewhere.

www.itbook.store/books/9781593277574

Functions in Loops

The characteristics of var have long made creating functions inside loops problematic, because the loop variables are accessible from outside the scope of the loop. Consider the following code:

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}

funcs.forEach(function(func) {
    func();     // outputs the number "10" ten times
});
```

You might ordinarily expect this code to print the numbers 0 to 9, but it outputs the number 10 ten times in a row. The reason is that `i` is shared across each iteration of the loop, meaning the functions created inside the loop all hold a reference to the same variable. The variable `i` has a value of 10 when the loop completes, so when `console.log(i)` is called, that value prints each time.

To fix this problem, developers use *immediately invoked function expressions (IIFEs)* inside loops to force a new copy of the variable they want to iterate over to be created, as in this example:

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    funcs.push((function(value) {
        return function() {
            console.log(value);
        }
    }(i)));
}

funcs.forEach(function(func) {
    func();     // outputs 0, then 1, then 2, up to 9
});
```

This version uses an IIFE inside the loop. The `i` variable is passed to the IIFE, which creates its own copy and stores it as `value`. This is the value used by the function for that iteration, so calling each function returns the expected value as the loop counts up from 0 to 9. Fortunately, block-level binding with `let` and `const` in ECMAScript 6 can simplify this loop for you.

www.itbook.store/books/9781593277574

let Declarations in Loops

A let declaration simplifies loops by effectively mimicking what the IIFE does in the previous example. On each iteration, the loop creates a new variable and initializes it to the value of the variable with the same name from the previous iteration. That means you can omit the IIFE altogether and get the results you expect, like this:

```
var funcs = [];

for (let i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}

funcs.forEach(function(func) {
    func();     // outputs 0, then 1, then 2, up to 9
})
```

This loop works exactly like the loop that used var and an IIFE but is arguably cleaner. The let declaration creates a new variable i each time through the loop, so each function created inside the loop gets its own copy of i. Each copy of i has the value it was assigned at the beginning of the loop iteration in which it was created. The same is true for for-in and for-of loops, as shown here:

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };

for (let key in object) {
    funcs.push(function() {
        console.log(key);
    });
}

funcs.forEach(function(func) {
    func();     // outputs "a", then "b", then "c"
});
```

In this example, the for-in loop shows the same behavior as the for loop. Each time through the loop, a new key binding is created, so each function has its own copy of the key variable. The result is that each function outputs a different value. If var were used to declare key, all functions would output "c".

www.itbook.store/books/9781593277574

NOTE *It's important to understand that the behavior of* `let` *declarations in loops is a specially defined behavior in the specification and is not necessarily related to the non-hoisting characteristics of* `let`*. In fact, early implementations of* `let` *did not exhibit this behavior, because it was added later in the process.*

const Declarations in Loops

The ECMAScript 6 specification doesn't explicitly disallow `const` declarations in loops; however, `const` behaves differently based on the type of loop you're using. For a normal `for` loop, you can use `const` in the initializer, but the loop will throw a warning if you attempt to change the value. For example:

```
var funcs = [];

// throws an error after one iteration
for (const i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i);
    });
}
```

In this code, the `i` variable is declared as a constant. The first iteration of the loop, where `i` is 0, executes successfully. An error is thrown when `i++` executes because it's attempting to modify a constant. As such, you can only use `const` to declare a variable in the loop initializer if you're not modifying that variable.

On the other hand, when used in a `for-in` or `for-of` loop, a `const` variable behaves similarly to a `let` variable. Therefore, the following should not cause an error:

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };

// doesn't cause an error
for (const key in object) {
    funcs.push(function() {
        console.log(key);
    });
}

funcs.forEach(function(func) {
    func();     // outputs "a", then "b", then "c"
});
```


www.itbook.store/books/9781593277574

This code functions almost the same as the second example in "let Declarations in Loops" on page 9. The only difference is that the value of key cannot be changed inside the loop. The for-in and for-of loops work with const because the loop initializer creates a new binding on each iteration through the loop rather than attempting to modify the value of an existing binding (as was the case in the for loop example).

Global Block Bindings

Another way in which let and const are different from var is in their global scope behavior. When var is used in the global scope, it creates a new global variable, which is a property on the global object (window in browsers). That means you can accidentally overwrite an existing global using var, as this code does:

```
// in a browser
var RegExp = "Hello!";
console.log(window.RegExp);     // "Hello!"

var ncz = "Hi!";
console.log(window.ncz);        // "Hi!"
```

Even though the RegExp global is defined on the window object, it is not safe from being overwritten by a var declaration. This example declares a new global variable RegExp that overwrites the original. Similarly, ncz is defined as a global variable and then defined as a property on window immediately afterward, which is the way JavaScript has always worked.

If you instead use let or const in the global scope, a new binding is created in the global scope but no property is added to the global object. That also means you cannot overwrite a global variable using let or const declarations; you can only shadow it. Here's an example:

```
// in a browser
let RegExp = "Hello!";
console.log(RegExp);                    // "Hello!"
console.log(window.RegExp === RegExp);  // false

const ncz = "Hi!";
console.log(ncz);                       // "Hi!"
console.log("ncz" in window);           // false
```

A new let declaration for RegExp creates a binding that shadows the global RegExp. Because window.RegExp and RegExp are not the same, there is no disruption to the global scope. Also, the const declaration for ncz creates a binding but does not create a property on the global object. This lack of global object modification makes let and const much safer to use in the global scope when you don't want to create properties on the global object.

www.itbook.store/books/9781593277574

NOTE *You might still want to use* `var` *in the global scope if you have code that should be available from the global object. This is most common in a browser when you want to access code across frames or windows.*

Emerging Best Practices for Block Bindings

While ECMAScript 6 was in development, there was widespread belief you should use `let` by default instead of `var` for variable declarations. For many JavaScript developers, `let` behaves exactly the way they thought `var` should have behaved, so the direct replacement made logical sense. In this case, you would use `const` for variables that needed modification protection.

However, as more developers migrated to ECMAScript 6, an alternate approach gained popularity: use `const` by default, and only use `let` when you know a variable's value needs to change. The rationale is that most variables should not change their value after initialization because unexpected value changes are a source of bugs. This idea has a significant amount of traction and is worth exploring in your code as you adopt ECMAScript 6.

Summary

The `let` and `const` block bindings introduce lexical scoping to JavaScript. These declarations are not hoisted and only exist within the block in which they're declared. Block bindings offer behavior that is more like other languages and less likely to cause unintentional errors, because variables can now be declared exactly where they're needed. As a side effect, you cannot access variables before they're declared, even with safe operators, such as `typeof`. Attempting to access a block binding before its declaration results in an error due to the binding's presence in the TDZ.

In many cases, `let` and `const` behave in a manner similar to `var`; however, this is not true in loops. Inside `for-in` and `for-of` loops, both `let` and `const` create a new binding with each iteration through the loop. As a result, functions created inside the loop body can access the loop bindings' current values rather than their values after the loop's final iteration (the behavior with `var`). The same is true for `let` declarations in `for` loops, whereas attempting to use a `const` declaration in a `for` loop may result in an error.

The current best practice for block bindings is to use `const` by default and only use `let` when you know a variable's value needs to change. Doing so ensures a basic level of immutability in code that can help prevent certain types of errors.

www.itbook.store/books/9781593277574