Extracted from:

Rediscovering JavaScript

Master ES6, ES7, and ES8

This PDF file contains pages extracted from *Rediscovering JavaScript*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Rediscovering JavaScript

Master ES6, ES7, and ES8

Venkat Subramaniam

edited by Jacquelyn Carter

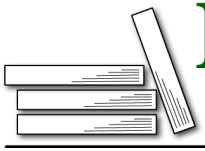
Rediscovering JavaScript

Master ES6, ES7, and ES8

Venkat Subramaniam

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Copy Editor: Liz Welch
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2018 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-546-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2018

CHAPTER 2

Variables and Constants

Traditionally JavaScript has used var to define variables. Moving forward, we should not use that keyword. Instead, we should choose between using const and let.

In this chapter, you'll start by learning why var is a bad idea, why it's still there, and why we should avoid it. Then you'll learn about the strengths, capabilities, and some limitations of using let. Finally, we'll explore const and discuss when to use it instead of let.

Out with var

Prior to ES6, JavaScript required var to define variables. If we forget to define a variable explicitly before assigning to it, we'll accidentally define a global variable. The 'use strict'; directive saves us from that error. In short, all variables should be defined before their first use. However, var is not the right choice, as we'll see here.

var does two things poorly. First, it does not prevent a variable from being redefined in a scope. Second, it does not have block scope. Let's explore these two issues with examples.

Redefining

It's poor programming practice to redefine a variable in the same scope as that often leads to errors in code. Here's an example where a variable max is redefined.

variables/redefine.js

```
Line 1 'use strict';
     2 var max = 100;
     3 console.log(max);
     4
     5 var max = 200;
     6 console.log(max);
```

On line 5 the variable max, which already exists, is redefined. If the programmer intended to assign a new value to an existing variable, then there should be no var declaration on that line. It appears, though, that the programmer intended to define a new variable, which happens to have the same name as an existing variable, thus accidentally erasing the previously stored value in that variable.

If a function were several lines long, it's possible that by accident we may redefine a variable for a different purpose or intent. Unfortunately, JavaScript doesn't give us any hint of the variable being redefined when var is used—tough luck.

No Block Scope

Variables defined using var within functions have function scope. Sometimes we may want to limit the scope of a variable to a smaller scope than the entire function. This is especially true for variables that are defined within a branch or a loop. Let's look at an example with a loop to illustrate the point.

variables/no-block.js

```
'use strict';

console.log(message);

console.log('Entering loop');
for(var i = 0; i < 3; i++) {
  console.log(message); //visible here, but undefined
  var message = 'spill ' + i;
}
console.log('Exiting loop');

console.log(message);
```

The variable message was defined within the loop—what happens in a loop should stay in the loop, but vars are not good at keeping secrets (poor encapsulation). The variable spills over the loop and is visible outside the loop—var hoists the variable to the top of the function. As a result, both message and the loop index variable i are visible throughout the function.

Not only is the variable, defined using var, visible following the block, it's also visible before the block. In other words, regardless of where in the function a variable is defined, it has the scope of the entire function.

Here's the output of running the previous code:

```
undefined
Entering loop
undefined
spill 0
spill 1
Exiting loop
spill 2
```

In short, var is a mess; don't use it.

var is terrible, but programmers have used it extensively for a few decades in JavaScript. Changing its behavior to fix these issues or removing var entirely will create compatibility issues between old and new JavaScript engines. This will turn into a nightmare for developers who deploy code on different browsers. That's the reason why var is still lingering around in the language. Even though the language can't get rid of it, we can and should. Quit using var and choose from the new let or const.

In with let

let is the sensible replacement for var. Anywhere we used var correctly before we can interchange it with let. let removes the issues that plague var and is less error prone.

No Redefinition

let does not permit a variable in a scope to be redefined. Unlike var, let behaves a lot like variable definitions in other languages that strictly enforce variable declarations and scope. If a variable is already defined, then using let to redefine that variable will result in an error, as in the next example.

variables/no-redefine.js

```
'use strict';
//BROKEN_CODE
let max = 100;
console.log(max);

let max = 200;
console.log(max);
```

This example is identical to the one we saw earlier, except that var was replaced with let. The compiler gives an error that max can't be redefined, as we see in the output:

```
let max = 200;
    ^

SyntaxError: Identifier 'max' has already been declared
```

let brings variable declaration semantics in JavaScript on par with what's expected in general programming.

What if we define a variable using var and then try to redefine it using let or vice versa? First, we should avoid such immoral thoughts—no reason to use var anymore. Second, JavaScript will not permit redefining a variable when let is used in the original definition or in the redefinition.

The fact that let does not allow redefinition is mostly good. There is, however, one place where that may not be to our advantage—in the REPL. As we saw in *Run Using the REPL*, on page ?, we can use node also as a quick experimentation tool. Likewise, as we saw in *Run in the Browser Console*, on page ?, we may also use the browser console to experiment and try out different code. When experimenting, we'd want to write and quickly change code to try out different ideas. In a few languages that have REPL and also prohibit variable redefinition, the rules of redefinition are favorably relaxed in REPLs for developer convenience. Sadly, node and the console in some of the popular browsers enforce the rule of prohibiting redefinition, thus making it a bit hard to retype chunks of code with variable definitions even within the console or REPL.

Block Scope

Variables declared using let have block scope. Their use and visibility is limited to the block of code enclosed by the {...} in which they're defined. Furthermore, unlike var, variables defined using let are available only after their point of definition. That is, the variables are not hoisted to the top of the function or the block in which they're defined.

Let's convert var to let in the code we saw earlier where we used a variable defined within a loop from outside the loop.

```
'use strict';

//console.log(message);  //ERROR if this line is uncommented

console.log('Entering loop');
for(let i = 0; i < 3; i++) {
  //console.log(message); //ERROR if this line is uncommented
  let message = 'spill ' + i;
}
console.log('Exiting loop');

//console.log(message);  //ERROR if this line is uncommented
```

This code illustrates the semantic difference between var and let. First, the variable defined within the block is not visible outside the block. Furthermore, even within the block, the variable is not visible before the point of definition. That's semantically sensible behavior—just the way it should be.