

Advanced JavaScript

Speed up web development with the powerful features and benefits of JavaScript



Zachary Shute

Packt>
www.packt.com

Advanced JavaScript

Speed up web development with the powerful features and benefits of JavaScript

Zachary Shute



Advanced JavaScript

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Author: Zachary Shute

Reviewer: Housseem Yahiaoui

Managing Editor: Aritro Ghosh

Acquisitions Editor: Aditya Date

Production Editor: Samita Warang

Editorial Board: David Barnes, Ewan Buckingham, Shivangi Chatterji, Simon Cox, Manasa Kumar, Alex Mazonowicz, Douglas Paterson, Dominic Pereira, Shiny Poojary, Saman Siddiqui, Erol Staveley, Ankita Thakur, and Mohita Vyas

First Published: January 2019

Production Reference: 1310119

ISBN: 978-1-78980-010-4

Table of Contents

Preface	i
Introducing ECMAScript 6	1
Introduction	2
Beginning with ECMAScript	2
Understanding Scope	2
Function Scope	3
Function Scope Hoisting	4
Block Scope	4
Exercise 1: Implementing Block Scope	6
Declaring Variables	8
Exercise 2: Utilizing Variables	11
Introducing Arrow Functions	13
Exercise 3: Converting Arrow Functions	14
Arrow Function Syntax	15
Exercise 4: Upgrading Arrow Functions	18
Learning Template Literals	20
Exercise 5: Converting to Template Literals	21
Exercise 6: Template Literal Conversion	24
Enhanced Object Properties	26
Object Properties	26
Function Declarations	27
Computed Properties	28
Exercise 7: Implementing Enhanced Object Properties	30

Destructuring Assignment	31
Array Destructuring	31
Exercise 8: Array Destructuring	34
Rest and Spread Operators	36
Object Destructuring	39
Exercise 9: Object Destructuring	41
Exercise 10: Nested Destructuring	44
Exercise 11: Implementing Destructuring	45
Classes and Modules	47
Classes	48
Exercise 12: Creating Your Own Class	49
Classes – Subclasses	51
Modules	52
Export Keyword	53
Import Keyword	55
Exercise 13: Implementing Classes	58
Transpilation	60
Babel- Transpiling	62
Exercise 14: Transpiling ES6 Code	63
Iterators and Generators	65
Iterators	65
Generators	67
Exercise 15: Creating a Generator	68
Activity 1: Implementing Generators	70
Summary	72
Asynchronous JavaScript	75
Introduction	76

Asynchronous Programming	76
Sync Versus Async	76
Synchronous versus Asynchronous Timing	76
Introducing Event Loops	79
Stack	80
Heap and Event Queue	81
Event Loops	82
Things to Consider	84
Exercise 16: Handling the Stack with an Event Loop	86
Callbacks	88
Building Callbacks	89
Callback Pitfalls	91
Fixing Callback Hell	92
Exercise 17: Working with Callbacks	94
Promises	96
Promises States	96
Resolving or Rejecting a Promise	96
Using Promises	97
Exercise 18: Creating and Resolving Your First Promise	99
Handling Promises	100
Promise Chaining	102
Promises and Callbacks	108
Wrapping Promises in Callbacks	108
Exercise 19: Working with Promises	110
Async/Await	112
Async/Await Syntax	112
Asnyc/Await Promise Rejection	114

Using Async Await	116
Activity 2: Using Async/Await	119
Summary	121
DOM Manipulation and Event Handling	123
Introduction	124
DOM Chaining, Navigation, and Manipulation	124
Exercise 20: Building the HTML Document from a DOM Tree Structure .	126
DOM Navigation	127
Finding a DOM Node	127
Traversing the DOM	133
DOM Manipulation	133
Updating Nodes in the DOM	139
Updating Nodes in the DOM	142
Exercise 21: DOM Manipulation	144
DOM Events and Event Objects	147
DOM Event	147
Event Listeners	148
Event Objects and Handling Events	149
Event Propagation	150
Firing Events	150
Exercise 22: Handling Your First Event	152
Custom Events	153
Exercise 23: Handling and Delegating Events	155
jQuery	157
jQuery Basics	159
jQuery Selector	160
jQuery DOM Manipulation	161

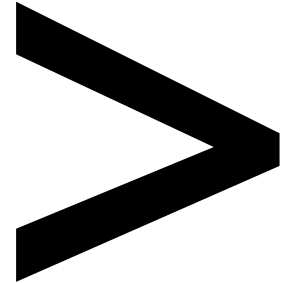
Selecting Elements	161
Traversing the DOM	162
Modifying the DOM	164
Chaining	165
jQuery Events	165
Firing Events	166
Custom Events	166
Activity 3: Implementing jQuery	166
Summary	168
Testing JavaScript	171

Introduction	172
Testing	172
Reasons to Test Code	173
Test-driven Development	175
TDD Cycle	176
Conclusion	177
Exercise 24: Applying Test-Driven Development	178
Types of Testing	180
Black Box and White Box Testing	180
Unit Tests	181
Exercise 25: Building Unit Tests	183
Functional Testing	185
Integration Tests	187
Building Tests	189
Exercise 26: Writing Tests	190
Test Tools and Environments	194
Testing Frameworks	194

Mocha	195
Setting Up Mocha	195
Mocha Basics	197
Exercise 27: Setting Up a Mocha Testing Environment	197
Mocha Async	198
Mocha Hooks	199
Activity 4: Utilizing Test Environments	200
Summary	201
Functional Programming	201
Introduction	202
Introducing Functional Programming	202
Object-Oriented Programming	203
Functional Programming	203
Declarative Versus Imperative	203
Imperative Functions	204
Declarative Functions	204
Exercise 28: Building Imperative and Declarative Functions	205
Pure Functions	207
Same Output Given Same Input	207
No Side Effects	208
Referential Transparency	209
Exercise 29: Building Pure Controllers	209
Higher Order Functions	211
Exercise 30: Editing Object Arrays	212
Shared State	214
Exercise 31: Fixing Shared States	215

Immutability	216
Immutability in JavaScript	218
Side Effects	219
Avoiding Side Effects	220
Function Composition	221
Activity 5: Recursive Immutability	222
Summary	223
The JavaScript Ecosystem	227
Introduction	228
JavaScript Ecosystem	228
Frontend JavaScript	228
Command-Line Interface	229
Mobile Development	229
Backend Development	229
Node.js	230
Setting Up Node.js	230
Node Package Manager	231
Loading and Creating Modules	234
Exercise 32: Exporting and Importing NPM Modules	235
Basic Node.js Server	236
Exercise 33: Creating a Basic HTTP Server	238
Streams and Pipes	240
Types of Streams	240
Writable Stream Events:	241
Readable Stream Events:	242
Filesystem Operations	245
Express Server	246

Exercise 34: Creating a Basic Express Server	248
Routing	250
Advanced Routing	251
Middleware	255
Error Handling	256
Exercise 35: Building a Backend with Node.js	258
React	262
Installing React	262
React Basics	264
React Specifics	266
JSX	266
ReactDOM	267
React.Component	268
State	269
Conditional Rendering	271
List of Items	273
HTML Forms	273
Activity 6: Building a Frontend with React	275
Summary	276
Appendix	279
Index	301



Preface

About

This section briefly introduces the author, the coverage of this book, the technical skills you'll need to get started, and the hardware and software required to complete all of the included activities and exercises.

About the Book

JavaScript is a core programming language for web technology that can be used to modify both HTML and CSS. It is frequently abbreviated to just JS. JavaScript is used for processes that go on in the user interfaces of most web browsers, such as Internet Explorer, Google Chrome, and Mozilla Firefox. It is the most widely-used client-side scripting language today, due to its ability to make the browser do its work.

In this book, you will gain a deep understanding of JavaScript. You will learn how to write JavaScript in a professional environment using the new JavaScript syntax in ES6, how to leverage JavaScript's asynchronous nature using callbacks and promises, and how to set up test suites and test your code. You will be introduced to JavaScript's functional programming style and you will apply everything you learn to build a simple application in various JavaScript frameworks and libraries for backend and frontend development.

About the Author

Zachary Shute studied computer and systems engineering at RPI. He is now the lead full-stack engineer at a machine learning start-up in San Francisco, CA. For his company, Simple Emotion, he manages and deploys Node.js servers, a MongoDB database, and JavaScript and HTML websites.

Objectives

- Examine major features in ES6 and implement those features to build applications
- Create promise and callback handlers to work with asynchronous processes
- Develop asynchronous flows using Promise chaining and `async/await` syntax
- Manipulate the DOM with JavaScript
- Handle JavaScript browser events

- Explore Test Driven Development and build code tests with JavaScript code testing frameworks.
- List the benefits and drawbacks of functional programming compared to other styles
- Construct applications with the Node.js backend framework and the React frontend framework

Audience

This book is designed to target anyone who wants to write JavaScript in a professional environment. We expect the audience to have used JavaScript in some capacity and be familiar with the basic syntax. This book would be good for a tech enthusiast wondering when to use generators or how to use Promises and Callbacks effectively, or a novice developer who wants to deepen their knowledge on JavaScript and understand TDD.

Approach

This book thoroughly explains the technology in an easy-to-understand way, while perfectly balancing theory and exercises. Each chapter is designed to build on what was learned in the previous chapter. The book contains multiple activities that use real-life business scenarios for you to practice and apply your new skills in a highly relevant context.

Minimum Hardware Requirements

For the optimal student experience, we recommend the following hardware configuration:

- Processor: Intel Core i5 or equivalent
- Memory: 4 GB RAM
- Storage: 35 GB available space
- An internet connection

Software Requirements

You'll also need the following software installed in advance:

- Operating system: Windows 7 SP1 64-bit, Windows 8.1 64-bit, or Windows 10 64-bit
- Google Chrome (<https://www.google.com/chrome/>)
- Atom IDE (<https://atom.io/>)
- Babel (<https://www.npmjs.com/package/babel-install>)
- Node.js and Node Package Manager (npm) (<https://nodejs.org/en/>)

Access to installation instructions can be provided separately to book material for large training centers and organizations. All source code is publicly available on GitHub and fully referenced within the training material.

Installing the Code Bundle

Copy the code bundle for the class to the **C:/Code** folder.

Additional Resources

The code bundle for this book is also hosted on GitHub at <https://github.com/TrainingByPackt/Advanced-JavaScript>.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The three ways to declare variables in JavaScript: **var**, **let**, and **const**."

A block of code is set as follows:

```
var example; // Declare variable
example = 5; // Assign value
console.log( example ); // Expect output: 5
```


Any command-line input or output is written as follows:

```
npm install babel --save-dev
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "This means that variables created with block scope are subject to the **Temporal Dead Zone (TDZ)**."

Installing Atom IDE

1. To install Atom IDE, go to <https://atom.io/> in your browser.
2. Click on **Download Windows Installer** for Windows to download the setup file called **AtomSetup-x64.exe**.
3. Run the executable file.
4. Add the **atom** and **apm** commands to your path.
5. Create shortcuts on the desktop and Start menu.

Babel is installed locally to each code project. To install Babel in a NodeJs project, complete the following steps:

1. Open a command, line interface and navigate to a project folder.
2. Run the command **npm init command**.
3. Fill in all the required questions. If you are unsure about the meaning of any of the prompts, you can press the 'enter' key to skip the question and use the default value.
4. Run the **npm install --save-dev babel-cli** command.
5. Run the command **install --save-dev babel-preset-es2015**.
6. Verify that the **devDependencies** field in **package.json** has **babel-cli** and **babel-preset-es2015**.
7. Create a file called **.babelrc**.
8. Open this file in a text editor and add the code **{ "presets": ["es2015"] }**.

Installing Node.js and npm

1. To install Node.js, go to <https://nodejs.org/en/> in your browser.
2. Click on **Download for Windows (x64)**, to download the LTS setup file recommended for most users called **node-v10.14.1-x64.msi**.
3. Run the executable file.
4. Ensure that you select the npm package manager bundle during the setup.
5. Accept the license and default installation settings.
6. Restart your computer for the changes to take effect.

1

Introducing ECMAScript 6

Learning Objectives

By the end of this chapter, you will be able to:

- Define the different scopes in JavaScript and characterize variable declaration
- Simplify JavaScript object definitions
- Destructure objects and arrays, and build classes and modules
- Transpile JavaScript for compatibility
- Compose iterators and generators

In this chapter, you'll be learning how to use the new syntax and concepts of ECMAScript.

Introduction

JavaScript, often abbreviated as JS, is a programming language designed to allow the programmer to build interactive web applications. JavaScript is one of the backbones of web development, along with HTML and CSS. Nearly every major website, including Google, Facebook, and Netflix, make heavy use of JavaScript. JS was first created for the Netscape web browser in 1995. The first prototype of JavaScript was written by Brendan Eich in just a mere 10 days. Since its creation, JavaScript has become one of the most common programming languages in use today.

In this book, we will deepen your understanding of the core of JavaScript and its advanced functionality. We will cover the new features that have been introduced in the ECMAScript standard, JavaScript's asynchronous programming nature, DOM and HTML event interaction with JavaScript, JavaScript's functional programming paradigms, testing JavaScript code, and the JavaScript development environment. With the knowledge gained from this book, you will be ready to begin using JavaScript in a professional setting to build powerful web applications.

Beginning with ECMAScript

ECMAScript is a scripting language specification standardized by **ECMA International**. It was created to standardize JavaScript in an attempt to allow for independent and compatible implementations. **ECMAScript 6**, or **ES6**, was originally released in 2015 and has gone through several minor updates since then.

Note

You may refer to the following link for more information about ECMA specification: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources.

Understanding Scope

In computer science, **scope** is the region of a computer program where the binding or association of a name to an entity, such as a variable or function, is valid. JavaScript has the following two distinct types of scope:

- **Function scope**
- **Block scope**

Until ES6, function scope was the only form of scope in JavaScript; all variable and function declarations followed function scope rules. Block scope was introduced in ES6 and is used only by the variables declared with the new variable declaration keywords **let** and **const**. These keywords are discussed in detail in the *Declaring Variables* section.

Function Scope

Function scope in JavaScript is created inside functions. When a function is declared, a new scope block is created inside the body of that function. Variables that are declared inside the new function scope cannot be accessed from the parent scope; however, the function scope has access to variables in the parent scope.

To create a variable with function scope, we must declare the variable with the **var** keyword. For example:

```
var example = 5;
```

The following snippet provides an example of function scope:

```
var example = 5;
function test() {
  var testVariable = 10;
  console.log( example ); // Expect output: 5
  console.log( testVariable ); // Expect output: 10
}
test();
console.log( testVariable ); // Expect reference error
```

Snippet 1.1: Function Scope

Parent scope is simply the scope of the section of code that the function was defined in. This is usually the global scope; however, in some cases, it may be useful to define a function inside a function. In that case, the nested function's parent scope would be the function in which it is defined. In the preceding snippet, the function scope is the scope that was created inside the function test. The parent scope is the global scope, that is, where the function is defined.

Note

Parent scope is the block of code, which the function is defined in. It is not the block of code in which the function is called.

Function Scope Hoisting

When a variable is created with function scope, it's declaration automatically gets hoisted to the top of the scope. **Hoisting** means that the interpreter moves the instantiation of an entity to the top of the scope it was declared in, regardless of where in the scope block it is defined. Functions and variables declared using **var** are hoisted in JavaScript; that is, a function or a variable can be used before it has been declared. The following code demonstrates this, as follows:

```
example = 5; // Assign value
console.log( example ); // Expect output: 5
var example; // Declare variable
```

Snippet 1.2: Function Scope Hoisting

Note

Since a hoisted variable that's been declared with **var** can be used before it is declared, we have to be careful to not use that variable before it has been assigned a value. If a variable is accessed before it has been assigned a value, it will return the value as **undefined**, which can cause problems, especially if variables are used in the global scope.

Block Scope

A new block scope in JavaScript is created with curly braces (**{}**). A pair of **curly braces** can be placed anywhere in the code to define a new scope block. If statements, loops, functions, and any other curly brace pairs will have their own block scope. This includes floating curly brace pairs not associated with a keyword (if, for, etc). The code in the following snippet is an example of the block scope rules:

```
// Top level scope
function scopeExample() {
  // Scope block 1
  for ( let i = 0; i < 10; i++ ){ /* Scope block 2 */ }
  if ( true ) { /* Scope block 3 */ } else { /* Scope block 4 */ }
  // Braces without keywords create scope blocks
  { /* Scope block 5 */ }
  // Scope block 1
}
```



```
}  
// Top level scope
```

Snippet 1.3: Block Scope

Variables declared with the keywords **let** and **const** have **block scope**. When a variable is declared with block scope, it does NOT have the same variable hoisting as variables that are created in function scope. Block scoped variables are not hoisted to the top of the scope and therefore cannot be accessed until they are declared. This means that variables that are created with block scope are subject to the **Temporal Dead Zone (TDZ)**. The TDZ is the period between when a scope is entered and when a variable is declared. It ends when the variable is declared rather than assigned. The following example demonstrates the TDZ:

```
// console.log( example ); // Would throw ReferenceError  
let example;  
console.log( example ); // Expected output: undefined  
example = 5;  
console.log( example ); // Expected output: 5
```

Snippet 1.4: Temporal Dead Zone

Note

If a variable is accessed inside the Temporal Dead Zone, then a runtime error will be thrown. This is important because it allows our code to be built more robustly with fewer semantic errors arising from variable declaration.

To get a better understanding of scope blocks, refer to the following table:

	Function Scope	Block Scope
Scope Creation	Block of scope for each function	Block of scope for curly braces {}
Variable Keyword	Variables defined with var	Variables defined with let and const
Hoisting and Instantiation	Variable hoisting	Temporal Dead Zone

Figure 1.1: Function Scope versus Block Scope

In summary, scope provides us with a way to separate variables and restrict access between blocks of code. Variable identifier names can be reused between blocks of scope. All new scope blocks that are created can access the parent scope, or the scope in which they were created or defined. JavaScript has two types of scope. A new function scope is created for each function defined. Variables can be added to function scope with the **var** keyword, and these variables are hoisted to the top of the scope. Block scope is a new ES6 feature. A new block scope is created for each set of curly braces. Variables are added to block scope with the **let** and **const** keywords. The variables that are added are not hoisted and are subject to the TDZ.

Exercise 1: Implementing Block Scope

To implement block scope principles with variables, perform the following steps:

1. Create a function called **fn1** as shown (**function fn1()**).
2. Log the string as **scope 1**.
3. Create a variable called **scope** with the value of 5.
4. Log the value of the variable called **scope**.
5. Create a new block scope inside of the function with curly braces (**{}**).
6. Inside the new scope block, log the string called **scope 2**.
7. Create a new variable called **scope**, inside the scope block and assign the value **different scope**.
8. Log the value variable **scope** inside our block scope (scope 2).
9. Outside of the block scope defined in step 5 (scope 2), create a new block scope (use curly braces).
10. Log the string called **scope 3**.
11. Create a variable inside the scope block (scope 3) with the same name as the variables (call it **scope**) and assign it the value **a third scope**.
12. Log the new variable's value.
13. Call **fn1** and observe its output

Code

index.js:

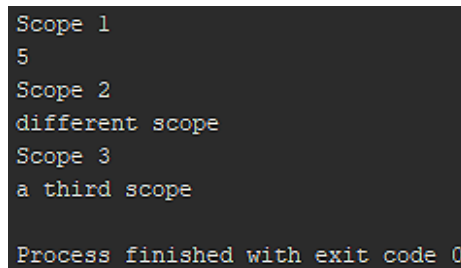
```
function fn1(){  
  console.log('Scope 1');
```

```
let scope = 5;
console.log(scope);
{
  console.log('Scope 2');
  let scope = 'different scope';
  console.log(scope);
}
{
  console.log('Scope 3');
  let scope = 'a third scope';
  console.log(scope);
}
}
fn1();
```

<https://bit.ly/2Ro0otW>

Snippet 1.5: Block implementation output

Outcome



```
Scope 1
5
Scope 2
different scope
Scope 3
a third scope

Process finished with exit code 0
```

Figure 1.2: Scope outputs

You have successfully implemented block scope in JavaScript.

In this section, we covered the two types of JavaScript scope, function and block scope, and the differences between them. We demonstrated how a new instance of function scope was created inside each function and how block scope was created inside each set of curly braces. We discussed the variable declaration keywords for each type of scope, **var** for function scope and **let/const** for block scope. Finally, we covered the basics of hoisting with both function and block scope.

Declaring Variables

Basic JavaScript uses the keyword **var** for **variable declaration**. ECMAScript 6 introduced two new keywords to declare variables; they are **let** and **const**. In the world of Professional JavaScript variable declaration, **var** is now the weakest link. In this topic, we will go over the new keywords, **let** and **const**, and explain why they are better than **var**.

The three ways to declare variables in JavaScript are by using **var**, **let**, and **const**. All function in slightly different ways. The key differences between the three variable declaration keywords are the way they handle variable reassignment, variable scope, and variable hoisting. These three features can be explained briefly as follows:

Variable reassignment: The ability to change or reassign the variable's value at any time.

Variable scope: The extent or area of the code from which the variable may be accessed.

Variable hoisting: The variable instantiation and assignment time in relation to the variable's declaration. Some variables can be used before they are declared.

The **var** keyword is the older variable declaration keyword that's used to declare variables in JavaScript. All variables created with **var** can be reassigned, have function scope, and have variable hoisting. This means that variables created with **var** are hoisted to the top of the scope block, where they are defined and can be accessed before declaration. The following snippet demonstrates this, as follows:

```
// Referenced before declaration
console.log( example ); // Expect output: undefined
var example = 'example';
```

Snippet 1.6: Variables created using var are hoisted

Since variables that are created with the keyword **var** are not constants, they can be created, assigned, and reassigned a value at will. The following code demonstrates this aspect of the functionality of **var**:

```
// Declared and assigned
var example = { prop1: 'test' };
console.log( 'example:', example );
// Expect output: example: {prop1: "test"}
// Value reassigned
example = 5;
```

```
console.log( example ); // Expect output: 5
```

Snippet 1.7: Variables created using var are not constant

Variables created with **var** can be reassigned at any time and once the variable is created, it can be accessed from anywhere in the function, even before the original declaration point.

The **let** keyword functions similar to the keyword **var**. As expected, the keyword **let** allows us to declare a variable that can be reassigned at any time. This is shown in the following code:

```
// Declared and initialized
let example = { prop1: 'test' };
console.log( 'example:', example );
// Expect output: example: {prop1: 'test'}
// Value reassigned
example = 5;
console.log( example ); // Expect output: 5
```

Snippet 1.8: Variables created with let are not constant

There are two significant differences between **let** and **var**. Where **let** and **var** differ is their scoping and variable hoisting properties. Variables declared with **let** are scoped at the block level; that is, they are only defined in the block of code contained within a matching pair of curly braces (**{}**).

Variables declared with **let** are not subject to variable hoisting. This means that accessing a variable declared with **let** before the assignment will throw a runtime error. As discussed earlier, this is the Temporal Dead Zone. An example of this is shown in the following code:

```
// Referenced before declaration
console.log( example );
// Expect ReferenceError because example is not defined
let example = 'example';
```

Snippet 1.9: Variables created with let are not hoisted

The last variable declaration keyword is **const**. The **const** keyword has the same scoping and variable hoisting rules as the **let** keyword; variables declared with **const** have block scoping and do not get hoisted to the top of the scope. This is shown in the following code:

```
// Referenced before declaration
console.log( example );
// Expect ReferenceError because example is not defined
const example = 'example';
```

Snippet 1.10: Variables created with **const** are not hoisted

The key difference between **const** and **let** is that **const** signifies that the identifier will not be reassigned. The **const** identifier signifies a read-only reference to a value. In other words, the value written in a **const** variable cannot be changed. If the value of a variable initialized with **const** is changed, a **TypeError** will be thrown.

Even though variables created with **const** cannot be reassigned, this does not mean that they are immutable. If an array or object is stored in a variable declared with **const**, the value of the variable cannot be overwritten. However, the array content or object properties can be changed. The contents of an array can be modified with functions such as **push()**, **pop()**, or **map()** and object properties can be added, removed, or updated. This is shown in the following code:

```
// Declared and initialized
const example = { prop1: 'test' };

// Variable reassigned
example = 5;
// Expect TypeError error because variable was declared with const

// Object property updated
example.prop1 = 5;
// Expect no error because subproperty was modified
```

Snippet 1.11: Variables created with **const** are constant but not immutable

To understand the different keywords in more detail, refer to the following table:

	Var	Let	Const
Scope Creation	Function Scope	Block Scope	Block Scope
Reassignment	Can be reassigned	Can be reassigned	Cannot be reassigned
Hoisting	Hoisted	Not hoisted	Not hoisted

Figure 1.3: Differences between var, let, and const

Now that we understand the nuances among **var**, **let**, and **const**, we can decide on which one to use. In the professional world, we should always use **let** and **const**, because they provide all the functionality of **var** and allow the programmer to be specific and restrictive with the variable scope and usage.

In summary, **var**, **let**, and **const** all function similarly. The key differences are in the nature of **const**, the scope, and the hoisting. **Var** is function scoped, not constant, and hoisted to the top of the scope block. **let** and **const** are both block-scoped and not hoisted. **let** is not constant, while, **const** is constant but immutable.

Exercise 2: Utilizing Variables

To utilize the **var**, **const**, and **let** variable declaration keywords for variable hoisting and reassignment properties, perform the following steps:

1. Log the string **Hoisted before assignment:** and the value of the **hoisted** variable.
2. Define a variable called **hoisted** with the keyword **var** and assign it the value **this got hoisted**.
3. Log the string **hoisted after assignment:** and the value of the **hoisted** variable.
4. Create a try-catch block.
5. Inside the **try** block, log the value of the variable called **notHoisted1**.
6. Inside the **catch** block, give the catch block the **err** parameter, then log the string **Not hoisted1 with error:** and the value of **err.message**.
7. After the try-catch block, create the **notHoisted1** variable with the **let** keyword and assign the value 5.
8. Log the string **notHoisted1 after assignment** and the value of **notHoisted1**.
9. Create another try-catch block.
10. Inside the **try** block, log the value of the **notHoisted2** variable.

11. Inside the catch block, give the catch block the **err** parameter, then log the string **Not hoisted2 with error:** and the value of **err.message**.
12. After the second try-catch block, create the **notHoisted2** variable with the keyword **const** and assign the value **[1,2,3]**.
13. Log the string **notHoisted2 after assignment** and the value of **notHoisted2**.
14. Define a final try catch block.
15. Inside the **try** block, reassign **notHoisted2** to the **new value** string.
16. Inside the catch block, give the catch block the **err** parameter, then log the string **Not hoisted 2 was not able to be changed**.
17. After the try-catch block, push the value **5** onto the array in **notHoisted2**.
18. Log the string **notHoisted2 updated. Now is:** and the value of **notHoisted2**.

Code

index.js:

```
var hoisted = 'this got hoisted';
try{
  console.log(notHoisted1);
} catch(err){}
let notHoisted1 = 5;
try{
  console.log(notHoisted2);
} catch(err){}
const notHoisted2 = [1,2,3];
try{
  notHoisted2 = 'new value';
} catch(err){}
notHoisted2.push(5);
```

Snippet 1.12: Updating the contents of the object

<https://bit.ly/2RDEynv>

Outcome

```
Hoisted before assignment: undefined
Hoisted after assignment: this got hoisted
Not hoisted1 with error: notHoisted1 is not defined
notHoisted1 after assignment 5
Not hoisted2 with error: notHoisted2 is not defined
notHoisted1 after assignment [ 1, 2, 3 ]
Not hoisted 2 was not able to be changed
notHoisted2 updated. Now is: [ 1, 2, 3, 5 ]

Process finished with exit code 0
```

Figure 1.4: Hoisting the variables

You have successfully utilized keywords to declare variables.

In this section, we discussed variable declaration in ES6 and the benefits of using the **let** and **const** variable declaration keywords over the **var** variable declaration keyword. We discussed each keywords variable reassignment properties, variable scoping, and variable hoisting properties. The keywords **let** and **const** are both **create** variables in the block scope where **var** creates a variable in the function scope. Variables created with **var** and **let** can be reassigned at will. However, variables created with **const** cannot be reassigned. Finally, variables created with the keyword **var** are hoisted to the top of the scope block in which they were defined. Variables created with **let** and **const** are not hoisted.

Introducing Arrow Functions

Arrow functions, or **Fat arrow functions**, are a new way to create functions in ECMAScript 6. Arrow functions simplify function syntax. They are called **fat arrow functions** because they are denoted with the characters `=>`, which, when put together look like a fat arrow. Arrow functions in JavaScript are frequently used in callback chains, promise chains, array methods, in any situation where unregistered functions would be useful.

The key difference between arrow functions and normal functions in JavaScript is that arrow functions are **anonymous**. Arrow functions are not named and not bound to an identifier. This means that an arrow function is created dynamically and is not given a name like normal functions. Arrow functions can however be assigned to a variable to allow for reuse.

When creating an arrow function, all we need to do is remove the function keyword and place an arrow between the function arguments and function body. Arrow functions are denoted with the following syntax:

```
( arg1, arg2, ..., argn ) => { /* Do function stuff here */ }
```

Snippet 1.13: Arrow function syntax

As you can see from the preceding syntax, arrow functions are a more concise way of writing functions in JavaScript. They can make our code more concise and easier to read.

Arrow function syntax can also vary, depending on several factors. Syntax can vary slightly depending on the number of arguments passed in to the function, and the number of lines of code in the function body. The special syntax conditions are outlined briefly in the following list:

- Single input argument
- No input arguments
- Single line function body
- Single expression broken over multiple lines
- Object literal return value

Exercise 3: Converting Arrow Functions

To demonstrate the simplified syntax by converting a standard function into an arrow function, perform the following steps:

1. Create a function that takes in parameters and returns the sum of the two parameters. Save the function into a variable called **fn1**.
2. Convert the function you just created to an arrow function and save into another variable called **fn2**.

To convert the function, remove the **function** keyword. Next, place an arrow between the function arguments and the function body.

3. Call both functions and compare the output.

Code

index.js:

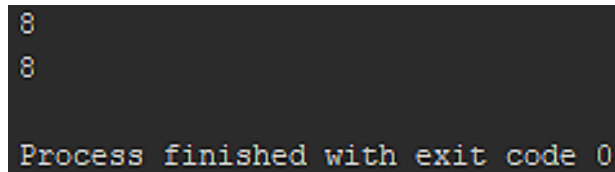
```
const fn1 = function( a, b ) { return a + b; };  
const fn2 = ( a, b ) => { return a + b; };
```

```
console.log( fn1( 3 ,5 ), fn2( 3, 5 ) );
```

Snippet 1.14: Calling the functions

<https://bit.ly/2M6uKwN>

Outcome



```
8
8
Process finished with exit code 0
```

Figure 1.5: Comparing the function's output

You have successfully converted normal functions into arrow functions.

Arrow Function Syntax

If there are multiple arguments being passed in to the function, then we create the function with the parentheses around the arguments as normal. If we only have a single argument to pass to the function, we do not need to include the parentheses around the argument.

There is one exception to this rule, and that is if the parameter is anything other than a simple identifier. If we include a default value or perform operations in the function arguments, then we must include the parentheses. For example, if we include a default parameter, then we will need the parentheses around the arguments. These two rules are shown in the following code:

```
// Single argument arrow function
arg1 => { /* Do function stuff here */ }

// Non simple identifier function argument
( arg1 = 10 ) => { /* Do function stuff here */ }
```

Snippet 1.15: Single argument arrow function

If we create an arrow function with no arguments, then we need to include the parentheses, but they will be empty. This is shown in the following code:

```
// No arguments passed into the function
( ) => { /* Do function stuff here */ }
```

Snippet 1.16: No argument

Arrow functions can also have varied syntax, depending on the body of the function. As expected, if the body of the function is multiline, then we must surround it with curly braces. However, if the body of the function is a single line, then we do not need to include the curly braces around the body of the function. This is shown in the following code:

```
// Multiple line body arrow function
( arg1, arg2 ) => {
  console.log( `This is arg1: ${arg1}` );
  console.log( `This is arg2: ${arg2}` );
  /* Many more lines of code can go here */
}

// Single line body arrow function
( arg1, arg2 ) => console.log( `This is arg1: ${arg1}` )
```

Snippet 1.17: Single line body

When using arrow functions, we may also exclude the return keyword if the function is a single line. The arrow function automatically returns the resolved value of the expression on that line. This syntax is shown in the following code:

```
// With return keyword - not necessary
( num1, num2 ) => { return ( num1 + num2 ) }
// If called with arguments num1 = 5 and num2 = 5, expected output is 10

// Without return keyword or braces
( num1, num2 ) => num1 + num2
// If called with arguments num1 = 5 and num2 = 5, expected output is 10
```

Snippet 1.18: Single line body when value is returned

Since arrow functions with single expression bodies can be defined without the curly braces, we need special syntax to allow us to split the single expression over multiple lines. To do this, we can wrap the multi-line expression in parentheses. The JavaScript interpreter sees that the line are wrapped in parentheses and treats it as if it were a single line of code. This is shown in the following code:

```
// Arrow function with a single line body
// Assume numArray is an array of numbers
( numArray ) => numArray.filter( n => n > 5 ).map( n => n - 1 ).every( n => n
< 10 )

// Arrow function with a single line body broken into multiple lines
// Assume numArray is an array of numbers
( numArray ) => (
  numArray.filter( n => n > 5 )
    .map( n => n - 1 )
    .every( n => n < 10 )
)
```

Snippet 1.19: Single line expression broken into multiple lines

If we have a single line arrow function returning an object literal, we will need special syntax. In ES6, scope blocks, function bodies, and object literals are all defined with curly braces. Since single line arrow functions do not need curly braces, we must use the special syntax to prevent the object literal's curly braces from being interpreted as either function body curly braces or scope block curly braces. To do this, we surround the returned object literal with parentheses. This instructs the JavaScript engine to interpret curly braces inside the parentheses as an expression instead of a function body or scope block declaration. This is shown in the following code:

```
// Arrow function with an object literal in the body
( num1, num2 ) => ( { prop1: num1, prop2: num2 } ) // Returns an object
```

Snippet 1.20: Object literal return value

When using arrow functions, we must be careful of the scope that these functions are called in. Arrow functions follow normal scoping rules in JavaScript, with the exception of the **this** scope. Recall that in basic JavaScript, each function is assigned a scope, that is, the **this** scope. Arrow functions are not assigned a **this** scope. They inherit their parent's **this** scope and cannot have a new **this** scope bound to them. This means that, as expected, arrow functions have access to the scope of the parent function, and subsequently, the variables in that scope, but the scope of **this** cannot be changed in an arrow function. Using the **.apply()**, **.call()**, or **.bind()** function modifiers will NOT change the scope of an arrow function's **this** property. If you are in a situation where you must bind **this** to another scope, then you must use a normal JavaScript function.

In summary, arrow functions provide us with a way to simplify the syntax of anonymous functions. To write an arrow function, simply omit the function keyword and add an arrow between the arguments and function body.

Special syntax can then be applied to the function arguments and body to simplify the arrow function even more. If the function has a single input argument, then we can omit the parentheses around it. If the function body has a single line, we can omit the **return** keyword and the curly braces around it. However, single-line functions that return an object literal must be surrounded with parentheses.

We can also use parentheses around the function body to break a single line body into multiple lines for readability.

Exercise 4: Upgrading Arrow Functions

To utilize the ES6 arrow function syntax to write functions, perform the following steps:

1. Refer to the **exercises/exercise4/exercise.js** file and perform the updates in this file.
2. Convert **fn1** with basic ES6 syntax.

Remove the function keyword before the function arguments. Add an arrow between the function arguments and function body.

3. Convert **fn2** with single statement function body syntax.

Remove the function keyword before the function arguments. Add an arrow between the function arguments and function body.

Remove the curly braces (**{}**) around the function body. Remove the return keyword.

4. Convert **fn3** with Single input argument syntax.

Remove the function keyword before the function arguments. Add an arrow between the function arguments and function body.

Remove the parentheses around the function input argument.

5. Convert **fn4** with no input argument syntax.

Remove the function keyword before the function arguments. Add an arrow between the function arguments and function body.

6. Convert **fn5** with object literal syntax.

Remove the function keyword before the function arguments. Add an arrow between the function arguments and function body.

Remove the curly braces (**{}**) around the function body. Remove the return keyword.

Surround the returned object with parentheses.

Code

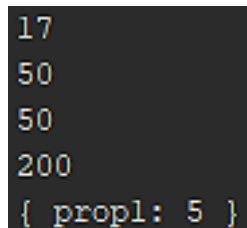
index.js:

```
let fn1 = ( a, b ) => { ... };  
let fn2 = ( a, b ) => a * b;  
let fn3 = a => { ... };  
let fn4 = () => { ... };  
let fn5 = ( a ) => ( ... );
```

Snippet 1.21: Arrow function conversion

<https://bit.ly/2M6qSfg>

Outcome



```
17  
50  
50  
200  
{ prop1: 5 }
```

Figure 1.6: Converting the function's output

You have successfully utilized the ES6 arrow function syntax to write functions.

In this section, we introduced arrow functions and demonstrated how they can be used to greatly simplify function declaration in JavaScript. First, we covered the basic syntax for arrow functions: `(arg1, arg2, argn) => { /* function body */ }`. We proceeded to cover the five special syntax cases for advanced arrow functions, as outlined in the following list:

- Single input argument: `arg1 => { /* function body */ }`
- No input arguments: `() => { /* function body */ }`
- Single line function body: `(arg1, arg2, argn) => /* single line */`
- Single expression broken over multiple lines: `(arg1, arg2, argn) => (/* multi line single expression */)`
- Object literal return value: `(arg1, arg2, argn) => ({ /* object literal */ })`

Learning Template Literals

Template literals are a new form of string that was introduced in ECMAScript 6. They are enclosed by the **backtick** symbol (```) instead of the usual single or double quotes. Template literals allow you to embed expressions in the string that are evaluated at runtime. Thus, we can easily create dynamic strings from variables and variable expressions. These expressions are denoted with the dollar sign and curly braces (`${ expression }`). The template literal syntax is shown in the following code:

```
const example = "pretty";
console.log( `Template literals are ${ example } useful!!!` );
// Expected output: Template literals are pretty useful!!!
```

Snippet 1.22: Template literal basic syntax

Template literals are escaped like other strings in JavaScript. To escape a template literal, simply use a backslash (`\`) character. For example, the following equalities evaluate to true: ``\`` === "`"`, ``\t` === "\t"`, and ``\n\r` === "\n\r"`.