JavaScript FOR GURUS

Use JavaScript programming features, techniques and modules to solve everyday problems

OCKERT J. DU PREEZ

JavaScript FOR GURUS

Use JavaScript programming features, techniques and modules to solve everyday problems

OCKERT J. DU PREEZ

bpb

JavaScript for Gurus

Use JavaScript programming features, techniques and modules to solve everyday problems

by

Ockert J. Du Preez

FIRST EDITION 2020

Copyright © BPB Publications, India

ISBN: 978-93-89423-655

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

My Wife and Daughter

Elmarie du Preez
and
Michaela du Preez

About the Author

Ockert du Preez has always had a love for computers, but never thought he would end up working as a systems developer, never mind writing programming books!

His real interest in programming started in 1998 when he encountered Visual C++ 6 and Visual Basic 6.

After his wife, programming is his greatest love.

He has been writing online articles for platforms such as CodeGuru, DevX, Developer.com, and Database Journal for several years. He was a .NET Microsoft MVP from 2009 to 2017.

You can find his Visual Studio book here:

https://rb.gy/kgnsbg

Acknowledgements

First and foremost, I would like to thank God for giving me the courage to write this book. I would like to thank everyone at BPB Publications for giving me this opportunity to write and publish this book.

Thank you to my wife and my daughter for always believing in me.

Lastly, I would like to thank my critics. Without their criticism, I would never be able to write this book.

Preface

From start to finish, this book will cover all the intricacies of the JavaScript language! You will get an overview of all the Statements, Functions and Operators. Then you will learn the fun stuff which includes: Classes, Prototypes, Promises, and Generators. You will learn about arrays and how to make your scripts achieve logic in your scripts. Lastly, you will learn how to combine JavaScript with other languages.

Chapter 1 details the history of JavaScript and explains its purpose in the world of programming languages.

Chapter 2 teaches you about JavaScript objects. You learn what they are, how to use them properly, and how to dispose them.

Chapter 3 presents how to write decent JavaScript statements for real world scenarios. Have a deep dive into JavaScript keywords as well.

Chapter 4 explains JavaScript Operators in detail.

Chapter 5 explores the ins-and-outs of JavaScript functions.

Chapter 6 demonstrates how to create JavaScript Classes for better code!

Chapter 7 explains Prototypes in detail and why we need them.

Chapter 8 guides you on how to create Properties properly.

Chapter 9 teaches you about the complexities of writing Promises.

Chapter 10 introduces JavaScript Generators and Iterators.

Chapter 11 explains what modules are and how they affect your code clarity.

Chapter 12 teaches you how to use variables properly.

Chapter 13 shows you how to introduce logic into any JavaScript app.

Chapter 14 teaches you how to make use of the various JavaScript loops.

Chapter 15 provides a guideline on how to code.

Chapter 16 explains what arrays are.

Chapter 17 shows you how Regular Expressions make it easy to search or replace text.

Chapter 18 presents the differences between Currying and Partials.

Chapter 19 shows how JavaScript differs from other languages, how it works with other languages and how it has influenced the birth of other languages.

Table of Contents

SECTION I: Introductory Concepts

- Introduction

1. Overview of the Power of JavaScript and Its Purpose

- Introduction
- Structure
- Objectives
- What is JavaScript?
- History of JavaScript
  - *Browser wars*
  - *The start of JavaScript*
- Why JavaScript?
- JavaScript features
- Conclusion
- Questions

2. JavaScript Objects

- Introduction
- Structure
- Objectives
- JavaScript objects
  - *Primitive values and primitive data types*
    - *string*
    - *number*
    - *bigint*
    - *boolean*
    - *null*
    - *undefined*
    - *symbol*
  - *The object data type*
- Creating objects
  - *Object literal syntax*

        - *Using ‘new’ with in-built object constructor functions*
        - *Using ‘new’ with user defined constructor functions*
    - *Object.create()*
    - *Object.assign()*
    - *ES6 classes*
- The ‘this’ keyword
    - *Method*
    - *Alone*
    - *In a function*
    - *In a function in strict mode*
    - *In an event*
- Conclusion
- Questions

3. JavaScript Statements

- Introduction
- Structure
- Objectives
- Anatomy of a statement
- JavaScript code blocks
    - *Grouping statements*
        - *while loop*
        - *for loop*
        - *if statement*
- Whitespace and line breaks
    - *Whitespace*
        - *A while loop without whitespace*
        - *A for loop without whitespace*
        - *if without whitespace*
        - *A while loop with whitespace*
        - *A for loop with whitespace*
        - *if with whitespace*
    - *Line breaks*
        - *A while loop with line breaks and whitespace*
        - *A for loop with line breaks and whitespace*
        - *if with whitespace and line breaks*
- JavaScript keywords

Conclusion
Questions

4. JavaScript Operators
Introduction
Structure
Objectives
What are operators?
Major operators
Arithmetic operators
Comparison operators
Logical operators
Assignment operators
Operator sub-types
Unary operators
Binary operators
Ternary operators
Operator precedence
That’s not all
Conclusion
Questions

5. JavaScript Functions
Introduction
Structure
Objectives
Functions
Function declaration
Event-invoked function example
Directly invoked function example
Self-invoked function example
Parameters
Default values
Function expressions
Conclusion
Questions

SECTION II: The Power of JavaScript
Introduction

6. JavaScript Classes
Introduction
Structure
Objectives
What are classes?
Creating a class
Class declarations
Class expressions
Properties
Properties of properties
Methods
Static methods
Inheritance
Conclusion
Questions

7. JavaScript Prototypes
Introduction
Structure
Objectives
What are prototypes?
Adding properties to a prototype object
Prototypal inheritance
Mixins
Conclusion
Questions

8. JavaScript Properties
Introduction
Structure
Objectives
What are properties?
Properties of properties
Getters and setters

Adding tricks to getters and setters
Conclusion
Questions

9. JavaScript Promises
Introduction
Structure
Objectives
JavaScript Promises description
JavaScript promises
Synchronous versus asynchronous methods
Chaining Promises
Callbacks and Promisification
Callbacks
Microtasks and macrotasks
Promise methods
Promise.all
Promise.allSettled
Promise.any
Promise.race
Conclusion
Questions

10. JavaScript Generatorsand Iterators
Introduction
Structure
Objectives
What are JavaScript generators?
Advantages of generators
Lazy evaluation
Memory efficiency
Iterators
Iterables
Async iterators vs.async generators
Async iterators
Code segment 1: Regular iterator
Code segment 2: Async iterator

Async generators
Code segment 3: Regular generator function
Code sample 4: Async generator function
Conclusion
Questions

11. Modules
Introduction
Structure
Objectives
What are the modules?
Modules
Benefits of modules
Working with modules
Exporting modules
Importing modules
Cross-Origin Resource Sharing (CORS)
CORS preflight request
Moesif Origin & CORS Changer
Command Prompt
Out of blink CORS
Dynamic import
Nomodule
Bundlers
Quick notes on Bundlers
Conclusion
Questions

SECTION III: Using JavaScript Productively
Introduction

12. Variables
Introduction
Structure
Objectives
Declaring variables
Variable scope

Local Scope versus Global scope
Primitive types
String
Quotes
Backticks
Special characters
String properties
String methods
Number
Number properties
Number methods
BigInt
Null versus undefined
Boolean
Symbol
Conclusion
Questions

13. Control Flow Statements
Introduction
Structure
Objectives
Control-flow statements definition
if statement
else…if statement
else statement
switch statement
Nested statements
Conditional operator ‘?’
Conclusion
Questions

14. Loops
Introduction
Structure
Objectives
JavaScript loops

for
- *for…in*
- *for…of*

while
- *do…while*

Exiting loops
- *break*
- *break with label*

The continue statement
Conclusion
Questions

15. Code Quality
Introduction
Structure
Objectives
Debugging
JavaScript debugging language features
- *The console window*
  - *console.log*
  - *console.time and console.timeEnd keywords*
- *More console methods*
- *The debugger keyword*

try and catch blocks
Coding style
- *Variable and object names*
- *Operators and indentation*
- *Rules for objects*

Comments
- *Single line comments*
- *Multi-line comment*

Conclusion
Questions

SECTION IV: Advanced Concepts
Introduction

16. JavaScript Arrays
- Introduction
- Structure
- Objectives
- Arrays
  - *Creating arrays*
    - *Method 1*
    - *Method 2*
    - *Adding items to an array*
    - *Removing items from arrays*
  - *Multidimensional arrays*
- Conclusion
- Questions

17. Regular Expressions
- Introduction
- Structure
- Objectives
- Regular expression definition
  - *Using a regular expression literal*
  - *Calling the constructor function of the RegExp object*
- Regular expression modifiers/flags
- Regular expression patterns
- Conclusion
- Questions

18. Partials and Currying
- Introduction
- Structure
- Objectives
- Partials
- Currying
- Conclusion
- Questions

SECTION V: JavaScript and Friends

19. JavaScript and Other Languages
Introduction
Structure
Objectives
Other languages
jQuery and JavaScript
JSON and JavaScript
JavaScript objects
The Object data type
Creating JavaScript objects
Object literal syntax
The ‘new’ keyword
Creating JSON objects
Rules for valid JSON
AJAX and JavaScript
Languages based on or influenced by JavaScript
ActionScript
CoffeeScript
TypeScript
Conclusion
Questions

SECTION I

Introductory Concepts

Introduction

It is always best to start things off easy, especially when it is a topic that many people are scared of or find too complicated. Section I starts off easy with the general purpose and history of JavaScript. Then we start with objects, statements, operators, and functions, all relatively easy topics – so there is a lot to look forward to and nothing to be afraid of, as you will see.

This section will have the following chapters:

1. Overview of the Power of JavaScript and Its Purpose
2. JavaScript Objects
3. JavaScript Statements
4. JavaScript Operators
5. JavaScript Functions

CHAPTER 1

Overview of the Power of JavaScript and Its Purpose

Introduction

JavaScript came into being because of a need for more dynamic content and interactivity. Although it started off small and specific to web browsers, it has matured and grown into a necessity for any web page and any web and mobile developer.

Thefirst chapter of this book focuses on the origins of JavaScript. It provides a brief history of the language and a brief introduction to its features. Most importantly, it answers the question: Why JavaScript, and why do we need it?

Structure

- What is JavaScript?
- History of JavaScript
- Why JavaScript?
- JavaScript features

Objectives

- Learn what JavaScript is
- Understand the need for JavaScript
- Learn about the early browser wars
- Learn about the exciting features of JavaScript

What is JavaScript?

JavaScript is one of the core technologies of the **World Wide Web** (**WWW**).

JavaScript is a high-level, interpreted programming language, meaning that most of its instructions execute directly without having to be compiled. The most enticing feature of JavaScript is its ability to make web pages interactive. The JavaScript language supports dynamic typing, functions, prototype-based orientation, and a plethora of APIs allowing you to work with arrays, text, regular expressions, dates and the **Document Object Model** (**DOM**).

JavaScript was initially only implemented client-side in web browsers, but now, JavaScript engines are embedded in many other types of host software, which includes:

- Server-side in web servers and databases
- Non-web programs
- Runtime environments which make JavaScript available for writing mobile and desktop applications

By being embedded in other types of host, the software makes JavaScript more versatile, more powerful and more adaptable to change across platforms.

History of JavaScript

In order to shed more light on the history of JavaScript, let's quickly have a look at the term browser wars.

Browser wars

As the name implies, a browser war is a competition between web browsers, mostly about usage share. During the late 1990s, the first browser war occurred between Microsoft Internet Explorer and Netscape Navigator.

The first browser war lasted from 1995 and continued until 2001. Netscape Navigator was the most widely used web browser because Microsoft only released Internet Explorer 1.0 as part of the Microsoft Windows 95 Plus! pack in August 1995.

Microsoft released Internet Explorer 2.0 as a free download three months later. Internet Explorer 4 changed everything. Internet Explorer 4 was integrated into Microsoft Windows, and this started Netscape Navigator's downfall, as users didn’t feel the need to download another browser when they have a browser in Internet Explorer already.

The first browser war ended with Internet Explorer having no remaining serious competition for its market share thus, bringing an end to the rapid innovation in web browsers.

With the decline of Internet Explorer’s usage share and rise in popularity of browsers such as Firefox, Google Chrome, Safari, Opera browser wars intensified in the early 2010s.

The start of JavaScript

Marc Andreessen, the founder of *Netscape Communications*, envisioned a more dynamic web experience. Instead of displaying only static web pages that couldn’t cater much for user activity, he wanted more. The problem was that wanting a more dynamic web experience which included automation, animations and basic user interaction, needed a lightweight scripting language which was accessible to non-programmers and a language that could directly interact with the DOM.

Mocha was born. Mocha would become a web scripting language which was simple and dynamic. Enter *Brendan Eich*, the father of JavaScript. Netscape Communications contracted Eich to develop Scheme (lightweight Lisp dialect) for their browser. Scheme was powerful, dynamic, and functional. After a few months, a prototype of Mocha was integrated into Netscape Communicator during May 1995. Quickly thereafter it was renamed to LiveScript. In December 1995, Mocha/LiveScript was renamed to JavaScript, and it was presented as a scripting language for client-side browser tasks.

Why JavaScript?

It is hard to imagine a web browser with no scripting, except for HTML, in 2019. There would be no interaction, no dynamic content, no communication with the DOM. Worse than that, imagine a world without JavaScript entirely. There would be no JSON, no jQuery and no AJAX. If there had been mobile applications, it would have been more difficult to write.

JavaScript features

In this book, we will explore the ins-and-outs of JavaScript, so some of the features listed here in the next list might be covered sooner than the others. It just depends on the complexity of the feature as well as our level of understanding of JavaScript and its features.

The following list highlights a few awesome features of JavaScript:

- Validating the user’s input
- Client-side calculations
- Browser control
- Platform independent
- Handling dates and time
- Generating HTML content
- Detecting the user’s operating system information
- Create new functions within scripts

During the course of this book, we will learn more about these features and make use of them practically.

Conclusion

In this chapter, we explored the general purpose of the JavaScript language. It is important to understand why a language exists, and what gap it filled within the web development landscape. We then briefly looked into its history and touched on some of its most common uses and features.

As you saw, it is important to understand the origins of JavaScript in order to fully appreciate its power. You have learned the history of JavaScript, what influence the browser wars had on the creation of JavaScript, and lastly answered the question of why there is a need for JavaScript.

In the next chapter, wecontinue with the introductory section and explain JavaScript objects. We will learn how to create them and how to use them properly within our web pages. Without objects, there simply is no JavaScript.

Questions

1. Who created JavaScript?

2. Explain the term browser war.
3. Name four features of the JavaScript language.
4. Explain the term DOM.
5. What is JavaScript’s benefit over a language such as HTML?

CHAPTER 2
JavaScript Objects

Introduction

Objects, in any programming language, are the building blocks of any application. Being able to store information inside objects and retrieve their values is crucial to the speed of your page.

There are many ways to store information inside objects and make use of them. We will learn what JavaScript objects are, and then have a quick look at primitives. We will create objects in five different ways. Lastly, we will explore `this` keyword and how its meaning differs depending on where it is used.

Structure

- JavaScript objects
- Creating objects
- The `this` keyword

Objectives

- Understand JavaScript objects and how to create them
- Learn what primitives are
- Learn how to create objects
- Understand and use the `new` keyword
- Understand and use the `this` keyword

JavaScript objects

An object is something that has properties and methods. Properties describe an object. Methods are the actions which that object can take. Basically, everything can be an object in JavaScript. All JavaScript values, except primitives, are objects. Before we go into detail about built-in objects, let's explore primitives quickly.

Primitive values and primitive data types

Primitive values are values that do not have properties or methods. A primitive data type is simply data that has a primitive value. A primitive value or a primitive data type is not an object and contains no methods. JavaScript has 7 primitive data types. They are:

- string
- number
- bigint
- boolean
- null
- undefined
- symbol

Let's explore these primitive data types one by one.

string

A string represents textual data. If you will allow me to get a bit technical;the JavaScript string primitive data type is simply a set of items of 16-bit unsigned integer values. Each item in the String occupies a position in the string, starting at 0.

number

The numberJavaScript primitive data type is a double-precision 64-bit binary format IEEE 754 value. Integers do not have a specific type. The number In addition to being able to represent floating-point numbers, the numberprimitive type also has three special symbolic values, namely: +Infinity, -Infinity, and NaN (not-a-number). In order to check for the largest or smallest value in +Infinity or -Infinity, use the `Number.MAX_VALUE` or `Number.MIN_VALUE` constants. You can also check if a number is in the double-precision floating-point range by using `Number.isSafeInteger()` or

`Number.MAX_SAFE_INTEGER` and `Number.MIN_SAFE_INTEGER`.

bigint

The `bigint` JavaScript primitive data type stores and operates on very large integers. These large integers can be beyond the safe integer limit for numbers. You create a `bigint` by appending n to the end of an integer.

boolean

The boolean JavaScript primitive data type can have one of two values, true or false.

null

The null JavaScript primitive data means that there is no value.

undefined

When a variable has a value that is undefined, it means that it has not been assigned a value.

symbol

The symbol JavaScript primitive data type is unique and immutable and is mostly used as a key to an object property.

The object data type

Objects are variables too, but unlike with primitives, objects can store more than one value in it. Objects store keyed collections of data and complex entities. The information inside an object is stored as key-value pairs. The key is usually the property name. The value is the property’s value, for example:

```
var objStudent = {
Name:”Ockert”,
Course:”JavaScript”,
Year: 2019
};
```

In the above code segment, a new JavaScript object called `objStudent` is created. `objStudent` contains three properties, namely: `Name`, `Course`, and `Year`.

Creating objects

Now that we know a bit more about objects let's have a look at how to create them. Objects can be created using any of the following ways:

- Object literal syntax
- The `new` keyword
- `Object.create()`
- `Object.assign()`
- ES6 classes

Let me explain them one by one.

Object literal syntax

This method is quite easy to understand and quick to do. Here is an example:

```
var objCar = {
make: "BMW",
model: "i8 Roadster",
price: 2329300
};
```

This example may look almost familiar. If you look closely at the previous code segment in the Object data type section, you will notice that both code examples have the same similarities. The keyword `var` creates the object. The curly braces contain the properties and values separated with a colon, and each property is separated with a comma.

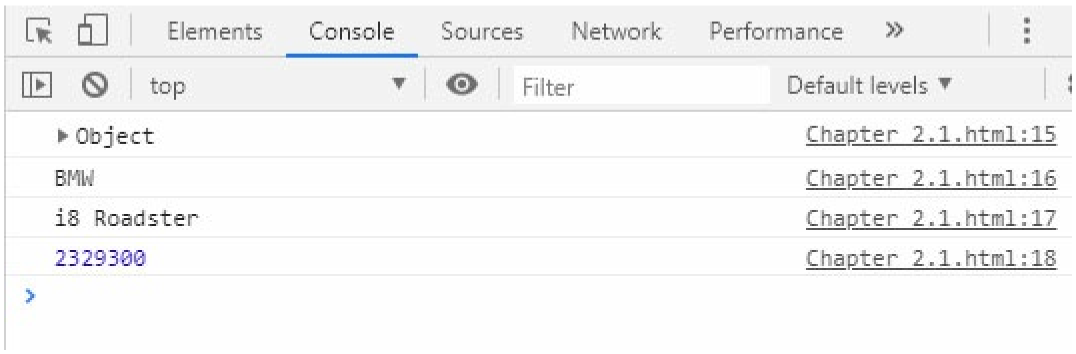
The object created now is named `objCar`. It contains the properties `make`, `model`, and `price`. To make use of the `objCar` object, you can have a statement like:

```
console.log(objCar); //Outputs {make: "BMW", model: "i8
Roadster", price: "2329300"}
console.log(objCar.make); //Outputs BMW
console.log(objCar.model); //Outputs i8 Roadster
console.log(objCar.price); //Outputs 2329300
```

Let’s put it all together in a small exercise:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:
```
<!DOCTYPE html>
<html>
<body>
<h1>Object literal syntax</h1>
<p>The following code creates a JavaScript object and
displays its results in the console window of the
browser</p>
<script>
var objCar = {
make : “BMW”,
model : “i8 Roadster”,
price : 2329300 };
console.log(objCar); //Outputs {make: “BMW”, model: “i8
Roadster”, price: “2329300”}
console.log(objCar.make); //Outputs BMW
console.log(objCar.model); //Outputs i8 Roadster
console.log(objCar.price); //Outputs 2329300
</script>
</body>
</html>
```
2. Save the file with an HTML extension, for example,`Chapter 2.1.html`.
3. Double-click the file to open it in the default browser. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

Your Google Chrome Console window should resemble *Figure 2.1* below:



```
▶Object                                    Chapter 2.1.html:15
BMW                                        Chapter 2.1.html:16
i8 Roadster                                Chapter 2.1.html:17
2329300                                    Chapter 2.1.html:18
>
```

Figure 2.1: *objCar property values*

The ‘new’ keyword

Using the `new` keyword to create objects resembles the way objects are created in class-based programming languages, like C# or Java. There are 2 ways to use the `new` keyword to create objects, they are:

- Using `new` with in-built object constructor functions
- Using `new` with user defined constructor functions

Let’s explore the differences.

Using ‘new’ with in-built object constructor functions

A constructor, in class-based **object-oriented programming** (**OOP**), is a special type of subroutine called to *create an object*. A constructor prepares the new object for use by initializing the object’s members. To create a new object, you must use the `new` keyword with the `Object()` constructor, like this:

```
var house = new Object();
```

This creates and initializes a new object named `house`. We can add properties to this object by using code like the following:

```
house.bedrooms = 4;
house.bathrooms = 2;
house.garages = 2;
house.hasSwimmingPool = false;
house.hasOutsideWalls = true;
```

As you can see, creating properties for an object, which was created using the `Object()` constructor, can become quite a tedious task, and involves a lot of typing. The `house` object we created above has a bedrooms property, `bathrooms`, `garages`, as well as Boolean properties indicating the presence of a swimming pool, or outside walls.

What if we had to create hundreds of `house` objects? Let’s see how we can make this code a bit shorter, by using a user defined constructor function.

Using ‘new’ with user defined constructor functions

When creating a user-defined constructor function, you basically create a template for the object’s properties and methods. You create the function once, and just feed the necessary properties to it, instead of manually entering each property and each setting as shown earlier in the Using `new` with in-built object constructor functions section.

Let’s create the user defined constructor using the `house` object as an example again:

```
function House(_bed, _bath, _garage, _pool, _walls) {
this.bedRooms = _bed;
this.bathRooms = _bath;
this.garages = _garage;
this.hasSwimmingPool = _pool;
this.hasOutsideWalls = _walls;
}
```

The name of the constructor function is `House`. Inside the brackets, there are five parameters: (`_bed`, `_bath`, `_garage`, `_pool`, and `_walls`. These will be used to pass information to the `House` function. Inside the constructor function, each property gets set to its own parameter. Remember, this is simply a template, there are no values yet.

Now that we have the constructor function, it is easy to create objects based on it. Here is how to:

```
var houseOne = new House(4, 2, 2, false, true);
var houseTwo = new House(3, 1, 1, true, false);
```

The `houseOne`object above is a new `House` object. It has 4 bedrooms, 2 bathrooms, 2 garages, no swimming pool and it has outside walls. `houseTwo`, in contrast, has 3 bedrooms, 1 bathroom, 1 garage, a swimming pool and it doesn’t have outside walls.

Let’s put it all together in a small exercise:

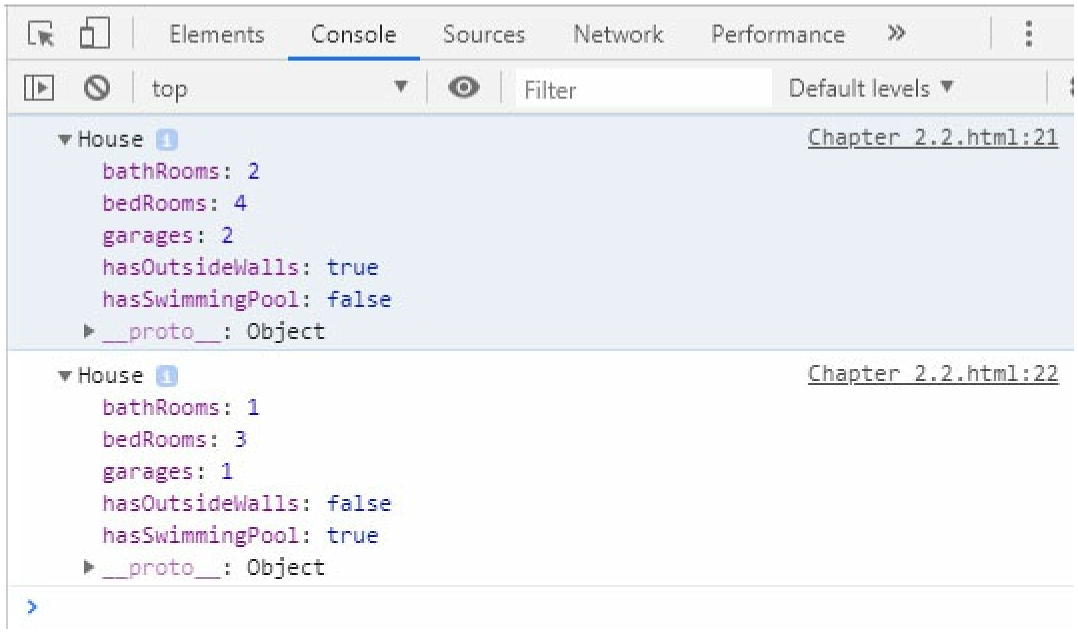
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!DOCTYPE html>
   <html>
   <body>
   <h1>new Keyword</h1>
   <p>The following code creates a JavaScript object and
   displays its results in the console window of the
   browser</p>
   <script>
   function House(_bed, _bath, _garage, _pool, _walls) {
   this.bedRooms = _bed;
   this.bathRooms = _bath;
   this.garages = _garage;
   this.hasSwimmingPool = _pool;
   ```

```
this.hasOutsideWalls = _walls;
}
var houseOne = new House(4, 2, 2, false, true);
var houseTwo = new House(3, 1, 1, true, false);
console.log(houseOne);
console.log(houseTwo);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example,`Chapter 2.2.html`.
3. Double-click the file to open it in the default browser. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

Your Chrome console window should resemble *Figure 2.2* below:



```
▼House                                    Chapter 2.2.html:21
    bathRooms: 2
    bedRooms: 4
    garages: 2
    hasOutsideWalls: true
    hasSwimmingPool: false
  ▶__proto__: Object
▼House                                    Chapter 2.2.html:22
    bathRooms: 1
    bedRooms: 3
    garages: 1
    hasOutsideWalls: false
    hasSwimmingPool: true
  ▶__proto__: Object
>
```

Figure 2.2: *House object property values*

Let's move onto the `Object.Create()` method.

Object.create()

By using the `Object.create()` method, you create a new object, using an existing object as the template or prototype. Let's look at a quick example:

Suppose you have a course object represented by `Course`:

```
let Course = {name: 'Web Programming'}
```

And you want to create subjects for the course. The subjects depend on the course. So, the subject object that we will see an example of next is based on the course object because the `Course` object acts as a template to it:

```
let Subject = Object.create(Course, { name:{ value: 'JavaScript'
} });
console.log(Subject.name); //Output: JavaScript
console.log(Subject.Course); //Output: Web Programming
```

Inside the `Object.create()` method you pass the object you'd like to use as a prototype, then, you supply a new property and value for the inherited object. We will cover Inheritance of objects in detail in *Chapter 7: JavaScript Prototypes*.

Object.assign()

The `Object.assign()` method takes `Object.create()` a level or two further. Whereas `Object.create()` makes use of one prototype object, `Object.assign()` can inherit or obtain properties from any number of prototype objects.

Suppose you have two objects as below:

```
let Course = {Coursename: 'Web Programming'}
let Subject = {Subjectname: 'JavaScript'}
```

And you want a `Student` object doing a `Web Programming` course and are currently busy with the subject named `JavaScript`. You can do that with the help of `Object.assign` as below:

```
let Student = Object.assign({}, Course, Subject);
```

Now, you have a Student object that has `Coursename` and `Subjectname` as its properties:

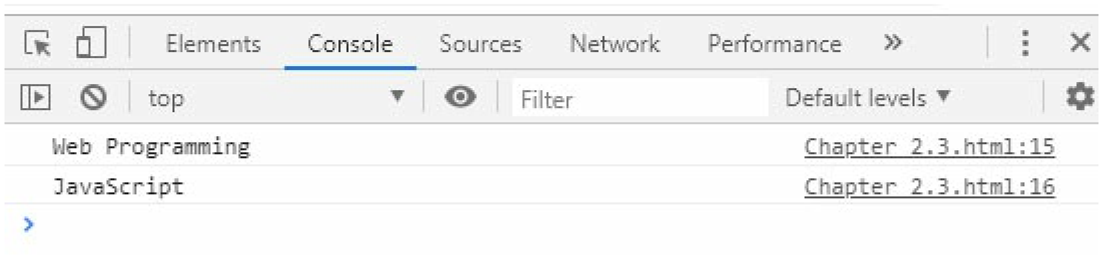
```
console.log(Student.Coursename) //Output: Web Programming
console.log(Student.Subjectname) //Output: JavaScript
```

An example complete script and a screenshot of its results are as follows:

```
<!DOCTYPE html>
<html>
<body>
<h1>Object.assign</h1>
<p>The following code creates a JavaScript object and displays
```

```
its results in the console window of the browser</p>
<script>
let Course = {Coursename: ‘Web Programming’}
let Subject = {Subjectname: ‘JavaScript’}
let Student = Object.assign({}, Course, Subject);
console.log(Student.Coursename) //Output: Web Programming
console.log(Student.Subjectname) //Output: JavaScript
</script>
</body>
</html>
```

Your Chrome console window should resemble *Figure 2.3* below:



```
Web Programming                                   Chapter 2.3.html:15
JavaScript                                        Chapter 2.3.html:16
```

Figure 2.3: *Object.assign() output*

Let’s move onto the ES6 classes.

ES6 classes

Using ES6 (ECMAScript 6 or JavaScript 6) classes is like using the `new` keyword with user defined constructor functions. The constructor functions are simply replaced by classes. Here is a short example:

```
class Car {
 constructor(_make, _model) {
   this.carMake = _make;
   this.carModel = _model;
 }
}
```

Create a new `Car` object as shown below:

```
myCar = new Car(“BMW”, “i8 Roadster”);
```

The ‘this’ keyword

The `this` keyword generally refers to the object it belongs to. The `this` keyword has different meanings depending on where it is used.

The `this`keyword can be used in the following locations:

- In a method
- Alone
- In a function
- In a function in strict mode
- In an event

Method

Inside a method, `this` refers to the owner of the method. So, if a `Student` object has a method named `Write()` and `this` was used inside the `Write()` method, this would refer to the `Student` object:

```
var Student = {
 firstName: “Ockert”,
 lastName : “du Preez”,
 subject : “JavaScript”,
 score : 85
 Write : function() {
  return this.firstName + “ wrote “ + this.subject + “ and got “
  this.score + “%”;
 }
};
```

The above code will give the following output:

```
Ockert wrote JavaScript and got 85%
```

Alone

When `this` is used alone, the owner of it is the global object. In a browser the global object is `[object Window]`:

```
var objWindow = this; //returns [object window]
```

In a function

When `this` is used in a function, the owner of it is also the global object:

```
function mainWindow() {
 return this; //returns [object window]
}
```

In a function in strict mode

When `this` is used in a function, in strict mode, `this` is undefined:

```
“use strict”;
function mainWindow() {
 return this; //returns undefined
}
```

In an event

When `this` is used in an HTML event handler, `this` refers to the HTML element that received the event:

```
<button onclick=”this.style.display=’none’”>
 Click Me!
</button>
```

Conclusion

In this chapter, we explored objects inside JavaScript. We learned about the various object creation techniques including: the `new` keyword, user-defined constructors, the object constructor, ES6 classes, and then we learned the benefits of the `Object.create()` and `Object.assign()` methods. Lastly, we learned the different ways to implement the `this` keyword.

In the next chapter, we will learn about JavaScript statements (what they are, how to properly construct a statement) and JavaScript keywords.

Questions

1. What is the difference between the `Object.create()` and `Object.assign()` methods?
2. Explain the term primitives.
3. What are user-defined constructor functions?
4. Where can the `this` keyword be used?
5. Explain the term parameter.

CHAPTER 3

JavaScript Statements

Introduction

Without statements there simply would not be any program. A statement is an instruction that gets sent to an interpreter or a compiler. How you compose your statements is vital to the overall success of your script. Yes, it sounds a bit drastic, but if you use sloppy code without whitespace or line breaks, it can get messy quite quickly.

In this lesson, you will learn what JavaScript statements are and how important it is to compose them properly for readability and understandability.

Structure

The topics covered in this chapter are:

- Anatomy of a statement
- JavaScript code blocks
- Whitespace and line breaks
- JavaScript keywords

Objectives

- Learn what a JavaScript statement is
- Learn how to create JavaScript statements
- Learn what code blocks are
- Learn about whitespace and line breaks
- Get introduced to the JavaScript keywords

Anatomy of a statement

A statement in a programming language is an instruction line that gets executed by a compiler or an interpreter. Lots of statements become a program. In JavaScript, statements are usually composed of keywords and operators (forming an expression) and values. Optionally, comments may also be included.

A few examples follow:

```
var objWidth, objHeight;
objWidth = 50;
objHeight = 100;
var size = objWidth * objHeight; //Multiply height and width for
screen size
document.getElementById("displaySizeLabel").innerHTML = size;
/*Display screen size*/
```

A JavaScript statement should be ended with a semi-colon (`;`). You can also include multiple statements on one line, as long as they are separated by a semi-colon, as in the next example:

```
objWidth = 50; objHeight = 100; var size = objWidth * objHeight;
```

Three statements are combined into one line, by separating each with a semi-colon.

JavaScript code blocks

Code blocks group statements together. Because the statements are grouped inside a code block, they get executed together. A common place to group statements together in code blocks is to use a function. JavaScript functions have the following format:

```
function functionName() {
//statement_1;
//statement_2;
 //statement_3;
}
```

A function starts with the keyword function. The open and closing brackets allow you to put in parameters for the function. The code block exists between the opening and closing curly braces of the function. In *Chapter 5: JavaScript Functions*, we will delve deeper into functions as a whole.

Separate code blocks can also appear inside a code block such as a function. For example a function can contain grouping statements such as a `while` loop, a `for` loop, or an `if` statement. Here is a very quick explanation and example on each.

Grouping statements

Grouping statements in JavaScript include loops and conditional tests. A loop in JavaScript is a block of code that gets executed repeatedly until a certain condition becomes true. A condition is something that gets tested upon, for example: if a certain value is less than 100, something should happen. A conditional statement is a statement that determines if a certain condition is true or false, for example: has a name been entered, or not.

The following few code blocks will demonstrate various loops and conditional tests.

while loop

A `while` loop loops through a block of code as long as a specified condition exists. Let’s look at an example:

```
var counter = 0;
while (counter <= 100)
{
document.write(counter + “<br>);
counter++;
}
```

In the above example, a variable named `counter` is created. The `while` loop starts. Notice that the statements inside the `while` loop are also grouped with enclosing curly braces. The `while` loop checks if the `counter` variable is less than or equal to `100` every time it loops. If it is, it increments the `counter` variable. Once it becomes more than `100`, the loop exits. I will not go into too much detail now, as loops will be explained in full in *Chapter 14: Loops*.

for loop

A `for` loop executes a block of code a specified number of times. So, if you want to repeat certain text 10 times, you would use a `for` loop, or if you want to loop through all the images on a page, you will use a `for` loop. Here is a small example:

```
for (let i = 0; i <= 100; i++)
{
 document.write(i + “<br>”);
}
```

The `for` loop differs slightly from the `while` loop. Whereas the `while` loop is what is called a *conditional* loop (it loops while a certain condition exists, or until a certain condition becomes true), a `for` loop is a counter loop. With a `for` loop, you specify exactly from where the loop should start, and where it should stop. To demonstrate the difference, I have kept the parameters the same as the `while` loop above. The `for` loop will loop from `0` to `100`.

It will then write the value of the loop counter on the document. As you can see, there is no need to increment the looping variable as it gets incremented automatically. I will not go into too much detail now, as loops will be explained in full in *Chapter 14: Loops*.

if statement

The JavaScript `if` statement executes the grouped statements if a certain condition is true. Here is a small example:

```
if (counter > 50)
{
alert(“stop it!”);
break;
}
```

The `if` statement above also groups a bunch of statements together by the using curly braces. In the above example, it determines if the `counter` variable is greater than `50`. If it is, it will display a popup box stating: `stop it!` if `counter` is smaller than or equal to `50`, nothing will happen. I will not go into too much detail now, as `if` statements will be explained in full in *Chapter 13: Control Flow Statements*.

Whitespace and line breaks

Code can become messy very quick, and you can get lost in code even quicker. Imagine a script containing 1000 lines. Imagine further that you have to change a few lines somewhere in there. Now, without whitespace you will struggle to find where your functions and grouping statements end. You will struggle immensely trying to find the place you need to edit.

This is where whitespace and line breaks come in. Whitespace allows you to format the flow of your code nicely in a way that is easy to follow.

Let’s make our code pretty!

Whitespace

White space is simply the addition of spaces, tabs or line-return characters so that your code is more readable. The JavaScript interpreter ignores multiple spaces, and it does not really add to the size of your script in any way.

Let’s take the same example code we used for the `while` loop, `for` loop, and `if` statement, and write each of them without whitespace.

A while loop without whitespace

```
var counter = 0;
while (counter <= 100)
{
document.write(counter + “<br>”);
counter++;
}
```

The above is a short example of what a small `while` loop looks like without whitespace. Let’s have a look at a `for` loop without any whitespace

A for loop without whitespace

```
for (let i = 0; i <= 100; i++)
{
document.write(i + “<br>”);
}
```

Next, here is an example of an `if` statement without whitespace.

if without whitespace

```
if (counter > 50)
{
alert(“stop it!”);
break;
}
```

As you can see, it is not easy to read the code now anymore, even though the code is not long. Now, imagine having a very long script with tens of `if` statements and loops combined. It will be extremely difficult to follow the code, not only for you but for the other developers on your team. You may now know everything going on in the script, but it may not be the case a month from now.

Let’s add whitespace to the above three examples.

A while loop with whitespace

```
var counter = 0;
while (counter <= 100)
{
document.write(counter + “<br>”);
counter++;
}
```

Above is a short example of what a small `while` loop can look like with a bit of whitespace. Let’s add some whitespace to a `for` loop.

A for loop with whitespace

```
for (let i = 0; i <= 100; i++)
{
document.write(i + “<br>”);
}
```

Next, here is an example of an `if` statement with good use of whitespace

if with whitespace

```
if (counter > 50)
{
alert(“stop it!”);
break;
}
```

The code already looks much more legible. By adding line breaks, you can make it even more readable and easier to follow.

Line breaks

A long JavaScript statement can be broken into multiple lines. Usually, a line longer than 80 characters makes reading and following code difficult. A line such as the following is quite long:

```
document.getElementById(“displaySizeLabel”).innerHTML = size;
/*Display screen size*/
```

A good place to break it off would be after an operator (*Chapter 4: JavaScript Operators* will cover operators in detail) like the next code snippet will show:

```
document.getElementById(“displaySizeLabel”).innerHTML =
size; /*Display screen size*/
```

A line break also aids in making your code more readable. I will refer to the `while` loop example again, and just add a bit of more line breaks in there. Let’s add line breaks to the above code examples

A while loop with line breaks and whitespace

```
var counter = 0;
while (counter <= 100)
{
document.write(counter + “<br>”);
counter++;
}
```

The above is a short example of what a small `while` loop can look like with a bit of whitespace and line breaks. Let’s add some line breaks to the `for` loop.

A for loop with line breaks and whitespace

```
for (let i = 0; i <= 100; i++)
{
document.write(i + “<br>”);
}
```

Next, here is an example of an `if` statement with good use of both whitespace and line breaks.

if with whitespace and line breaks

```
if (counter > 50)
{
alert(“stop it!”);
break;
}
```

Looks and reads much better than having a bunch of statements just grouped together without any spaces or line breaks.

JavaScript keywords

Keywords or reserved words are words in JavaScript that means something specific to JavaScript. These words cannot be used to name variables or functions. A list of JavaScript keywords follow:

abstract	await
arguments	boolean
break	case
byte	catch
char	const
class	continue
debugger	delete
default	do
double	enum
else	eval
export	FALSE
extends	final
finally	for
float	function
goto	implements
if	import
in	int
instanceof	interface
let	native
long	new
null	private

package	protected
public	short
return	static
super	synchronized
switch	this
throw	transient
throws	TRUE
try	var
typeof	void
volatile	with
while	yield

Table 3.1: *JavaScript keywords*

During the course of this book, we will cover these keywords, as this chapter is only an introduction.

Conclusion

In this chapter, welearnt about the ins-and-outs of JavaScript statements. We learned how to create statements, how a statement is formed, and we then learned about white space and line breaks. Lastly, we had a quick look at the available JavaScript keywords.

In the next chapter, we will learn about JavaScript operators. We explore the operator types, how to use operators to assign values. We then get a bit technical by comparing unary operators with binary operators.

Questions

1. What is the difference between a JavaScript statement and a code block?
2. What is the main advantage of adding white space to your code?
3. Why should you add line breaks in your code?
4. Name 4 JavaScript keywords.
5. What is the purpose of the semi-colon character?

CHAPTER 4

JavaScript Operators

Introduction

Operators enable us to work with different operands. These operands can be the name of an object which type we would like to identify. It can be two numbers we'd like to add, subtract, multiply, or divide.

There are many operators, each with its own function. This chapter explains what operators and operands are. It breaks them down into its categories and types for better understanding and gives practical examples on how to use the most common operators in JavaScript.

Structure

- What are operators?
    - Major operators
    - Operator sub-types
- Operator precedence

Objectives

- Understand what operators are
- Know what arithmetic operators there are
- Learn about comparison operators
- Understand logical operators
- Understand assignment operators
- Compare the different operator sub-types
- Learn about operator precedence

What are operators?

Operators can be explained as constructs which behave almost like functions but differ semantically or syntactically from ordinary functions. There are many types of operators, or even, categories and they will be cover shortly. There are comparison, arithmetic, and logical operators.

Some common operators include the following:

Operator type/category	**Operator(s)**
Arithmetic	`+ - * / %`
Comparison	`><>= <= == ===`
Logical	`&& ||`
Assignment	`=`

Table 4.1: *General operators*

Major operators

Major operators can be classified as any operator category that achieves a similar goal. Major operators would include arithmetic, assignment, comparison, logical, and type operators. I will highlight each major type one by one in the next few tables and headings.

Arithmetic operators

Arithmetic operators are mostly used to perform arithmetic on numbers. The available arithmetic operators in JavaScript are shown in *Table 4.2*:

Operator type	**Operator**	**Description**
Arithmetic	`+`	Addition
	`-`	Subtraction
	`*`	Multiplication
	`**`	Exponentiation
	`/`	Division
	`%`	Modulus (remainder division)
	`++`	Increment

	`--`	Decrement

Table 4.2: *Arithmetic operators*

Comparison operators

Comparison operators are used to determine equality or difference between values:

Operator type	**Operator**	**Description**
Comparison	`==`	Equal to
	`===`	Equal value and equal type
	`!=`	Not equal
	`!==`	Not equal value or not equal type
	`>`	Greater than
	`<`	Less than
	`>=`	Greater than or equal to
	`<=`	Less than or equal to
	`?`	Ternary operator

Table 4.3: *Conditional operators*

Logical operators

Logical operators are used to determine the logic between values. The available logical operators in JavaScript are shown in *Table 4.4*:

Operator type	**Operator**	**Description**
Logical	`&&`	Logical and
	`||`	Logical or
	`!`	Logical not

Table 4.4: *Logical operators*

Assignment operators

Assignment operators assign values to variables. The available assignment operators in JavaScript are shown in *Table 4.5*:

Operator	**Example**	**Equivalent**
`=`	`a = b`	a = b
`+=`	`a += b`	a = a + b
`-=`	`a -= b`	a = a – b
`*=`	`a *= b`	a = a * b
`/=`	`a /= b`	a = a / b
`%=`	`a %= b`	a = a % b
`**=`	`a **= b`	a = a ** b

Table 4.5: *Assignment operators*

Operator sub-types

Apart from having different functions as operators, there are different ways to formulate a statement with various operators. In most cases, you will need two operands (what operators are applied to) and an operator. Sometimes you may need three operands and two operators. The operator sub-types as I'd like to call them are listed below:

- Unary
- Binary
- Ternary

Let's explore them.

Unary operators

When an operator is unary, it means that the operator only has a single operand. The following sample shows some unary operators in action:

1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<html>
<body>
<h1>Unary Operators</h1>
<p>The following code demonstrates Unary operations on
```

```
operands</p>
<script>
let a = 1;
let b = 1;
let c = 5;
let d = 5;
a = -a;
alert( a ); // -1, unary negation was applied
b = +b;
alert( b ); // +1, unary addition was applied
c++;
alert( c ); // 6, increment was applied
d--;
alert( d ); // 4, decrement was applied
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 4.1.html`.
3. Double-click the file to open it in the default browser.

   Once loaded, the web page will produce a total of four alert boxes, one after the other. In the above example, the following answers were produced:

   `The variable named a has a value of -1. This is because we negated it, we made it negative.`
   `The variable named b has a value of 1. It hasn't changed, because it was already positive.`
   `The variable named c has a value of 6. We added one to its existing value of 5.`
   `The variable named d has a value of 4. We subtracted one from its existing value of 5.`

Binary operators

When an operator is binary, it has two operands. The following sample shows some binary operators in action:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Binary Operators</h1>
<p>The following code demonstrates Binary operations on
operands</p>
```

```
<script>
let a = 1;
let b = 2;
let c = 5;
let d = 2;
var e;
a = a + b;
alert( a ); // 3, the value of b is added to a
b = b - c;
alert( b ); // -1, the value of c is subtracted from b
e = c % d;
alert( e ); // 1, The remainder is 1
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 4.2.html`.
3. Double-click the file to open it in the default browser.

   Once loaded, the web page will produce a total of three alert boxes, one after the other. In the above example, the following answers were produced:

   ```
   The variable named a has a value of 3. This is because the
   value of b was added to a.
   The variable named b has a value of -1. The value of c is
   subtracted from b.
   The variable named e has a value of 1. The remainder is 1.
   ```

Ternary operators

A ternary operator is an operator that takes three operands. The following sample shows some binary operators in action:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Ternary Operators</h1>
<p>The following code demonstrates Ternary operations on
operands</p>
<script>
//Normal if condition
var age = 21;
var canEnterCollege;
if (age > 18) {
```

```
canEnterCollege = “Old enough”;
}
else {
canEnterCollege = “Too young”;
}
alert(canEnterCollege);
//Shortened by ternary conditional operator
var canEnterCollege = (age < 18) ? “Too young” : “Old
enough”;
alert(canEnterCollege);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 4.2.html`.
3. Double-click the file to open it in the default browser.

Once loaded, the web page will produce a total of two alert boxes, one after the other. In the above example, the following answers were produced.

The variable named `canEnterCollege` has a value of `21`. In the first control-flow statement (we will cover control-flow statements in *Chapter 13: Control Flow Statements*) we test if the variable `canEnterCollege` is greater than `18`. If it is, the alert will display `old enough`; `else`, another alert will show `Too young`.

In the last segment of the code, notice the use of both the `?` and `:`. The condition is written in brackets – in this case, age `< 18`. If the condition is true, the code next to the `?` (question-mark) will execute, if the condition is false (`age` is not less than 18), the code next to the `:` (colon) will fire.

Operator precedence

In the event that expression has more than one operator, the order of execution is defined by their priority order, that is, precedence. The next table shows the operator precedence in JavaScript:

Operator type	Individual operators
Grouping	`()`
Member access	`.`
Computed member access	`[ ]`
	`new ( )`

new (with argument list)	
Function call	`( )`
Optional chaining	`?.`
new (without argument list)	`new`
Postfix increment	`++`
Postfix decrement	`--`
Logical NOT	`!`
Bitwise NOT	`~`
Unary plus	`+`
Unary negation	`-`
Prefix increment	`++`
Prefix decrement	`--`
typeof	`typeof`
void	`void`
delete	`delete`
await	`await`
Exponentiation	`**`
Multiplication	`*`
Division	`/`
Remainder	`%`
Addition	`+`
Subtraction	`-`
Bitwise left shift	`<<`
Bitwise right shift	`>>`
Bitwise unsigned right shift	`>>>`
Less than	`<`
Less than or equal	`<=`
Greater than	`>`
Greater than or equal	`>=`

In	`in`
instanceof	`instanceof`
Equality	`==`
Inequality	`!=`
Strict equality	`===`
Strict inequality	`!==`
Bitwise AND	`&`
Bitwise XOR	`^`
Bitwise OR	`|`
Logical AND	`&&`
Logical OR	`||`
Conditional	`? :`
Assignment	`=`
	`+=`
	`-=`
	`**=`
	`*=`
	`/=`
	`%=`
	`<<=`
	`>>=`
	`>>>=`
	`&=`
	`^=`
	`|=`
yield	`yield`
yield*	`yield*`
Comma / Sequence	`,`

Table 4.6: *Operator precedence*

That’s not all

We have covered a lot of functions and a lot of different operators in this chapter thus far, but we can use operators to add strings together. Strings will be covered in *Chapter 12: Primitive Types*, and was mentioned in *Chapter 2: JavaScript Objects*.

Operators can be used to determine types of objects, and then there are bitwise operators. We will cover all of these as we go on.

Conclusion

In this chapter, we explored the world of JavaScript operators. We learned what operators are, and the different types and groups of operators there are in JavaScript. We had a little fun and did some practical examples on unary, binary and ternary operators. Lastly, we had a quick introduction into operator precedence.

In the next chapter, we will learn about JavaScript functions. JavaScript has many features, but the functions are where everything happens. This lesson explores the ins-and-outs of JavaScript functions.

Questions

1. What is the difference between unary, binary, and ternary operators?
2. Explain operator precedence.
3. Explain the use of the following operator: `%`
4. What is an assignment operator?
5. What are logical operators?

CHAPTER 5
JavaScript Functions

Introduction

Functions make things happen. We as programmers write functions to be used repeatedly throughout our code. It depends on each function where it can be used or when it should fire, this is where invocation comes in. We pass information to functions to produce similar yet different results.

In this chapter, we will focus on JavaScript functions. You will learn what they are and when they should be used. You will learn about parameters and setting default values. Lastly, we will touch on function expressions.

Structure

- What are functions?
- Function declaration
- Parameters
- Default values
- Function expressions

Objectives

- Understand functions
- Learn how to declare functions
- Learn about function parameters
- Understand function expressions

Functions

In layman’s terms, a function is a block of code that returns a certain value. It can be compared to a little subprogram inside a script or program that is invoked (or called) from another method or function. The calling method or function or object makes use of the value that gets returned by the invoked function. A function should always return a value. If no return value is specified, a value of undefined gets returned.

Functions in JavaScript are first-class objects meaning that they can have properties and methods. Any function consists of a declaration or expression, optional parameters, and default values.

Function declaration

By declaring a function, you tell the interpreter that the function should be stored in memory for later use. This will occur when the function is called or invoked from somewhere in the code. To declare a function, you should use code similar to the following:

```
function functionName(parameters) {
 // code block
}
```

The keyword `function` gets written first. This is to tell the interpreter that you are creating a function. The name follows the `function` keyword. It can be any name, except for a JavaScript keyword, and remember that it is case sensitive. Inside the brackets, you can include a list of parameters separated by commas. Here is a small example:

```
function showGreeting() {
 alert( ‘Good morning!’ );
}
```

In the code segment above, a function named `showGreeting` is created without any parameters, anddisplays an alertbox with the message `Good morning!` inside.

You have a function, now what?

Well, the function cannot run without being invoked (or called) from somewhere. When an invoking a function, you simply reference the function name and include its optional parameters. A function can be invoked from any of the following:

- When an event occurs

- When it’s invoked directly
- Self-Invoked

Event-invoked function example

Here is an example of how a function can be invoked from an event such as a button click, or mouse move:

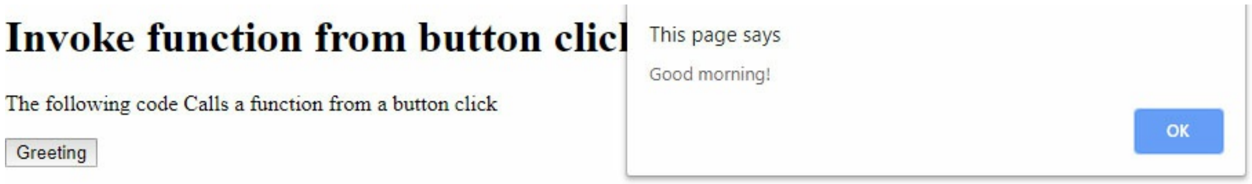
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Invoke function from button click</h1>
<p>The following code Calls a function from a button
click</p>
<script>
function showGreeting() {
 alert( ‘Good morning!’ );
}
</script>
<button onclick=”showGreeting()”>Greeting</button>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 5.1.html`.
3. Double-click the file to open it in the default browser.

As you will see, the page loads normally, and there is no alertbox popping up yet. This is because we have invoked the function from the button’s click event. If the button is not clicked, then the function will not get executed.

Click the button, and notice the message that pops up, as shown in the following screenshot:



Invoke function from button clicl

The following code Calls a function from a button click

This page says

Good morning!

Figure 5.1: *Function invoked from a button click*

Directly invoked function example

Here is an example of how a function can be invoked directly:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>Invoke function directly</h1>

   <p>The following code Calls a function just after the
   function is created</p>
   <script>
   function showGreeting() {
    alert( ‘Good morning!’ );
   }
   showGreeting();
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 5.2.html`.
3. Double-click the file to open it in the default browser.

Self-invoked function example

Here is an example of how a function can be invoked automatically. Meaning it is a function, but there is no need to add a separate call to it in order to invoke it, as it invokes itself:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>Self-Invoke function expression</h1>
   <p>The following code creates a function and invokes it
   automatically</p>
   <script>
   (function showGreeting(){
    alert( ‘Good morning!’ );
   })()
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 5.3.html`.

3. Double-click the file to open it in the default browser.

The function declaration looks a bit different here the reason for this is because we have now created a function expression (we will cover function expressions later in this chapter). The function named `showGreeting` is created and executed automatically without having to invoke it.

Parameters

A parameter or function argument is an extra piece of data that can be passed to a function. The function can then use and manipulate the parameter in any way it pleases. Another benefit of including parameters in functions is that you can pass different values as parameters, and the function will still work accordingly.

Here is an example of how you can send different values to a function by using its parameters:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Function Parameters</h1>
<p>The following code creates a function with parameters and
makes use of the parameters inside the function</p>
<script>
function showGreeting(from, message) { // arguments: from,
message
 alert(from + ‘: ‘ + message);
}
showGreeting(‘Ockert’, ‘Hello!’); // Outputs - Ockert:
Hello!
showGreeting(‘BpB’, “Hi!”); // Outputs - BPB: Hi!
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 5.4.html`.
3. Double-click the file to open it in the default browser.

A function named `showGreeting` is declared with two parameters, from and message. Inside the code block, an alert box is called with the values copied into the parameters. The first call to the `showGreeting` function includes:

```
Ockert
Hello!
```

The second call to the `showGreeting` function includes:

```
BpB
Hi!
```

When this script is run, you will see two alert boxes popping up after each other. The first alert contains the first input data; the second contains the last input data.

Default values

When a parameter is not provided, its value becomes undefined. For instance, if you do this:

```
function showGreeting(from, message) { // arguments: from,
message
 alert(from + ‘: ‘ + message);
}
showGreeting(‘Ockert’); // Outputs - Ockert: undefined
```

It is perfectly acceptable, and the interpreter will not produce an error. This is where default values come in. You can give a parameter a default value so that if a value is not supplied, it still has a set default value that can be used.

Here is small example:

```
function showGreeting(from, message = ‘Hey there!’) { //
arguments: from, message
 alert(from + ‘: ‘ + message);
}
showGreeting(‘Ockert’); // Outputs - Ockert: Hey there!
```

Function expressions

JavaScript functions can also be defined by using expression and is usually stored inside a variable. This variable can then be used as a function. Here is a small example:

```
var answer = function (operand1, operand2) {return operand1 *
operand2};
```

In *Chapter 12: Variables*, we will go into much greater detail on variables and functions.

Conclusion

In this chapter, welearnt JavaScript functions with some nice examples. We learned how Functions can improve our code, and how we can improve it further by using parameters and default values. A bit later in this book (in *Chapter 12: Variables*) we will come back to functions again and explore the world of function expressions.

In the next chapter, wewill dive into JavaScript classes. You will learn how to create JavaScript classes, properties and methods and Mixins. You will also have a look into the existing classes in JavaScript as well as inheritance.

Questions

1. Explain the term function.
2. Name the three ways in which a function can be invoked.
3. What is a parameter?
4. Give an example of a function with a default value.
5. Explain the term function expression.

SECTION II
The Power of JavaScript

Introduction

Section 2 dives right into the real power of JavaScript. Understanding what makes JavaScript tick is pivotal in creating extremely powerful websites and web applications. In this section, you will learn how to create classes, and then move on to prototypes, properties and promises, finally generators, and modules.

This section will have the following chapters:

1. Classes
2. Prototypes
3. Properties
4. Promises
5. Generators
6. Modules

CHAPTER 6

JavaScript Classes

Introduction

A JavaScript class creates the structure of its child objects. Although JavaScript is not a full-on **object-oriented programming** (**OOP**) language, it still contains many an OOP feature. Learning how to work with classes is essential to create proper reusable objects.

In this chapter, we will focus on JavaScript classes. You will learndifferent ways to create classes and how to instantiate child objects from classes. Properties and methods are next. You will learn how to create basic properties and methods to give your classes better structure and better adaptability. Lastly, you will get a quick introduction to inheritance. There is a lot of work, so let's get started!

Structure

- What are classes?
  - Creating a class
  - Class declarations
  - Class expressions
- Properties
  - Properties of properties
- Methods
  - Static methods
- Inheritance

Objectives

- Learn what classes are
- Understand the concepts of properties
- Understand the concepts of methods
- Learn what inheritance is

What are classes?

A class, in OOP is a template or blueprint from which objects can be created. A class in JavaScript is a type of function which can be created with a class declaration or a class expression.

Creating a class

There are essentially two ways to create a class in JavaScript; they are:

- Declaring a class
- Using a class expression

Let’s have a look at examples for each.

Class declarations

To declare a class, all you need to do is to use the class keyword and supply name of the class, with a constructor. A constructor is a special type of method where the object can be initialized, and it gets called automatically. You can also supply arguments (parameters) to the class here and initialize them as well. Initialization is the assignment (setting) of an initial value for an object or variable. Here is an example:

```
class Student {
 constructor(studentName, studentCourse) {
   this. studentName = studentName;
   this. studentCourse = studentCourse;
 }
}
```

In the above code, we created a class named `Student`is created. The constructor function gets two arguments supplied to it. These two arguments are `studentName` and `studentCourse`.

Let’s take it a step further.

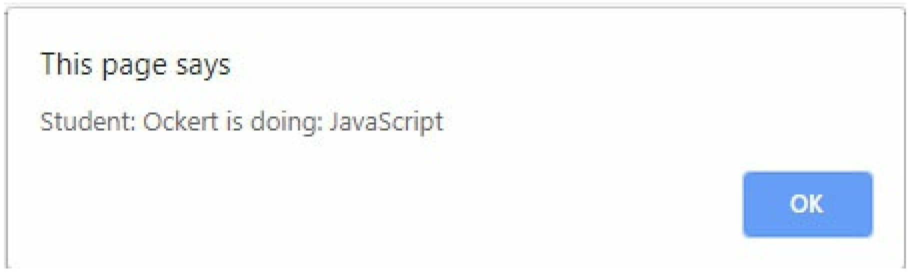
The following sample shows how to declare a class, initialize some of its objects, and make use of a class from code outside of the class:

1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<html>
<body>
<h1>Class declaration example</h1>
<p>Class Declaration example</p>
<script>
class Student
{
 constructor(studentName, studentCourse)
 {
this.studentName = studentName;
 this.studentCourse = studentCourse;
}
 showInfo()
{
alert( ‘Student: ‘ + this.studentName + ‘ is doing: ‘ +
this.studentCourse);
}
}
// Usage:
student = new Student(‘Ockert’, ‘JavaScript’);
student.showInfo();
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 6.1.html`.
3. Double-click the file to open it in the default browser.

An alert-box similar to the next screenshot will appear:


This page says

Student: Ockert is doing: JavaScript

Figure 6.1: *Class Declaration in action*

Let me explain what happens in the codewe have created the `Student` class and its constructor. Then, we initialize the `studentName` and `studentCourse` variables. Inside the `Student` class you created a function named `showInfo`. The `showInfo` function displays an alert box with the supplied `studentName` and `studentCourse`. Then, we created a `Student` object name student (notice that JavaScript is case sensitive). Inside the student object, you supply details such as your name (or mine in this case) and the course. Lastly, you invoke the `showInfo` function from the `student` object, and this displays the alert box shown in *Figure 6.1*.

Now that you know how to create a class let's see how to create class expressions to create classes.

Class expressions

Classes, just like functions, can be defined inside expressions, as explained in Chapter 5: Functions. Class expressions can be named or unnamed. Let's do an example on both.

The following sample shows an unnamed class expression example:

1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<html>
<body>
<h1>Unnamed Class expression example</h1>
<p>Unnamed Class expression example</p>
<script>
let Student = class {
 constructor(studentName, studentCourse) {
   this.studentName = studentName;
   this.studentCourse = studentCourse;
}
showInfo()
{
alert( ‘Student: ‘ + this.studentName + ‘ is doing: ‘ +
this.studentCourse);
}
};
// Usage:
var student = new Student(‘Ockert’, ‘JavaScript’);
student.showInfo();
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 6.2.html`.
3. Double-click the file to open it in the default browser.

OK, so what happens here?

You created an unnamed class by using an expression. Then you created a constructor, initialize the arguments and created the `showInfo` function to display the values input.

The following sample shows a named class expression example:

1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<html>
<body>
<h1>Named Class expression example</h1>
<p>Named Class expression example</p>
<script>
let student = class Student {
constructor(studentName, studentCourse) {
this.studentName = studentName;
this.studentCourse = studentCourse;
}
showInfo(){
alert( ‘Student: ‘ + this.studentName + ‘ is doing: ‘ +
this.studentCourse);
}
};
// Usage:
var newStudent = new student(‘Ockert’, ‘JavaScript’);
newStudent.showInfo();
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 6.3.html`.
3. Double-click the file to open it in the default browser.

As you can see, it is almost similar to the preceding two examples, but it is still different. You specify a name for the class in the class expression, create the constructor and then create a `newStudent` object which is based on the student object linking to the `Student` class.

Properties

A property describes an object. Any object has certain attributes that can be set and retrieved. Take for example a car object. There are numerous attributes that can be set for a car, these include:

- Make of car
- Model of car
- Transmission
- Fuel type
- Color
- Number of seats

Although there are different makes and models of cars, with different colors and transmission and fuel types, each car is still classified as a car. Why? Because the car object still gets made from the same car template or class.

We saw how to set properties in all the code examples so far in this chapter. These are known as instance properties. The one important thing to note there was that you have used the dot notation to set a property's value and to set its value. Let's take it a step further.

To access JavaScript properties, use the following syntax:

```
objectName.property     // student.studentName
```

OR

```
objectName[“property”]     // student[“studentName”]
```

OR

```
objectName[expression];     // newStudent = “studentName”;
student[newStudent]
```

You may be wondering how do I add new properties to an object? Well, it is quite simple! You simply give it a value. Here is an example:

```
student.studentCourse = “JavaScript”;
```

The left side of the equal sign contains the object, then its property. The right side of the equal sign contains the value that will be stored in the property.

Properties of properties

The most common attributes of a property are its value and its name. You access property by using its name, but that supplies the value requested. This is where it gets tricky! Some properties cannot be written to, this means that they are read-only and cannot be changed. Some properties cannot be looped over by using a loop. They must be enumerable. Some properties cannot be deleted.

These all depend on the attributes (or flags) of a certain property. Primarily there are three attributes (or flags) that can be set. They are:

- **w**ritable: Determines if the value of the property can be changed.
- **e**numerable: Can the property be listed in loops?
- **c**onfigurable: Allows property deletion, or property attributes to be modified.

In order to access the attributes for any property, you can use the `Object.getOwnPropertyDescriptor` method. You supply the object which you'd like to interrogate, then the property of the object. Here is a small exercise.

The following sample shows `Object.getOwnPropertyDescriptor` in action:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

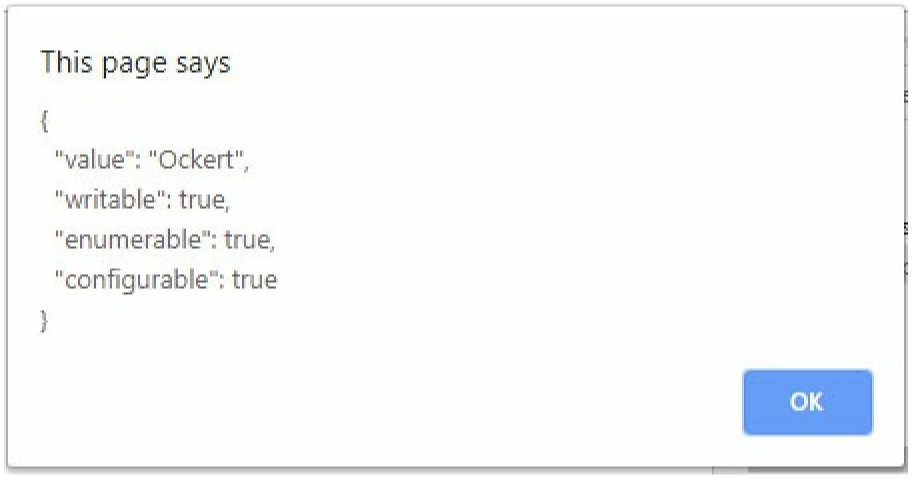
```
<html>
<body>
<h1>Object.getOwnPropertyDescriptor example</h1>
<p>Object.getOwnPropertyDescriptor example</p>
<script>
let student = class Student {
 constructor(studentName, studentCourse) {
  this.studentName = studentName;
  this.studentCourse = studentCourse;
 }
 showInfo()
 {
let descriptor = Object.getOwnPropertyDescriptor(newStudent,
‘studentName’);
alert(JSON.stringify(descriptor, null, 2));
}
};
```

```
// Usage:
var newStudent = new student('Ockert', 'JavaScript');
newStudent.showInfo();
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 6.4.html`.
3. Double-click the file to open it in the default browser.

In the above code segment, you created an object. Then in the `showInfo` method, you made use of the `getOwnPropertyDescriptor` method to investigate the attributes of the `studentName` property of the `newStudent` object.

Figure 6.2 shows the attributes of the `studentName` property:



```
This page says
{
  "value": "Ockert",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
```

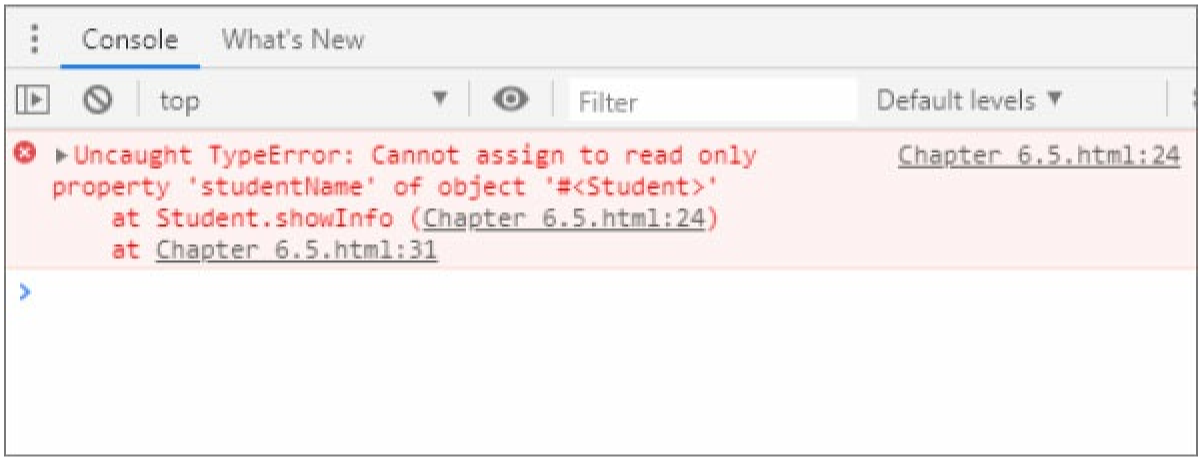
Figure 6.2: *studentName attributes*

To change these attributes, you should use the `defineProperty` method of the `Object` class.

The next example changes the writable attribute of the `studentName` property to false. This means that this property is now read-only, and cannot be changed from code:

```
Object.defineProperty(newStudent, "studentName", {
 writable: false
});
newStudent.studentName = "Test"; // Error: Cannot assign to read
only property 'studentName'
```

Figure 6.3 shows the error if code similar to the code segment above was run inside the browser:



```
▶Uncaught TypeError: Cannot assign to read only          Chapter 6.5.html:24
property 'studentName' of object '#<Student>'
    at Student.showInfo (Chapter 6.5.html:24)
    at Chapter 6.5.html:31
```

Figure 6.3: *defineProperty error*

In Chapter 8*:* Properties, we will go into much greater detail on getters and setters, flags and descriptors.

Methods

A method in any programming language means what an object can do. So, in the context of our continuous student object example, a student object can write an exam, pass or fail an exam, skip class, be late for class or not pay his or her school fees.

In the properties section, I have used a car object as an example. If you were to think of a car's methods, they might be along the lines of:

1. Start
2. Accelerate
3. Stop
4. Turn
5. Reverse

Even though most cars can perform the same functions, how they do it differs and matters. A *Porsche 911 Turbo* can accelerate much faster than *Renault Kwid* for example.

In all the exercises we have done in this chapter, we have made use of a method named `showInfo`. You simply add the method to the class and give it the logic it needs. I will change the `showInfo` function to something a bit different now in this next example.

The following sample shows how to create a method and make use of it through code:

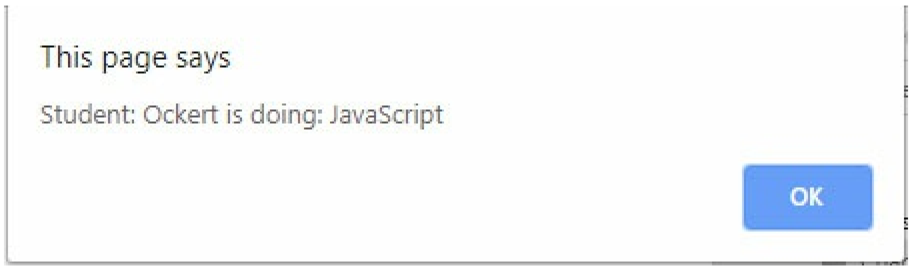
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>Method example</h1>
   <p>Method example</p>
   <script>
   let student = class Student {
    constructor(studentName, studentCourse) {
      this.studentName = studentName;
      this.studentCourse = studentCourse;
   }
   AllInfo()
    {
   return ‘Student: ‘ + this.studentName + ‘ is doing: ‘ +
   this.studentCourse;
    }
   };
   // Usage:
   var newStudent = new student(‘Ockert’,
   ‘JavaScript’).AllInfo();
   alert(newStudent);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 6.5.html`.
3. Double-click the file to open it in the default browser.

You create the class with its properties (as we have done all along). Then you created a method named `AllInfo`. The `AllInfo` method returns a string containing the student’s name, the student’s course, and some text to form a nice sentence.

You then created a new student object with the name of `newStudent`, gave it initialization values, and immediately invoked the `AllInfo` function. Lastly, the alert box displays the results, as shown in *Figure 6.4*:


This page says

Student: Ockert is doing: JavaScript

Figure 6.4: *AllInfo method*

Let’s move on to static methods.

Static methods

Static methods differ slightly from ordinary methods. Static methods are created in the class itself, and not on the object making use of the class (the prototype). This means that you can’t call a static method on an object, but you can on the class.

The following sample shows an example of a static method:

1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<html>
<body>
<h1>Method example</h1>
<p>Method example</p>
<script>
let student = class Student {
 constructor(studentName, studentCourse) {
   this.studentName = studentName;
   this.studentCourse = studentCourse;
 }
Static AllInfo()
{
return ‘Student: ‘ + this.studentName + ‘ is doing: ‘ +
this.studentCourse;
 }
};
// Usage:
var newStudent = new student(‘Ockert’,
‘JavaScript’).AllInfo();
alert(newStudent);
</script>
</body>
```

```
</html>
```

2. Save the file with an HTML extension, for example `Chapter 6.6.html`.
3. Double-click the file to open it in the default browser.

This will produce an error because the `AllInfo` method is static and can only be accessed from within the class.

Inheritance

Objects inherit features from other objects. Earlier in this chapter, I have spoken about the car object. I have explained that each car has four wheels and other common features, but what about trucks or buses?

Although trucks usually have more than four wheels, they still form part of the vehicle family. Buses may have four wheels, but they have many more seats than cars and trucks; still they form part of the vehicle family.

Another example would be that you inherit features from your parents. You may have your mother’s hair and eyes, and you may have your dad’s personality, but you are still unique and still you. This process is known as inheritance.

In Chapter 7*:* Prototypes, we will explore inheritance much further as well as use Mixins to use methods from different classes without the need of inheritance.

Conclusion

In this chapter, welearnt JavaScript classes and how to make use of them. We learned whyclassesareimportant, and how we can add properties and methods to JavaScript classes. We created a lot of examples in this chapter because that is the easiest way to learn anything! Practice makes perfect.

In the next chapter, we will deals with prototypes. Although we have touched on the subject in this chapter, we will dive deeper into them. We will also learn the importance of inheritance and Mixins and primitives.

Questions

1. Explain the term class.

2. Name the two ways in which a class can be created.
3. What is the difference between a method and a static method?
4. Name the three attributes of a property.
5. Explain the term inheritance.

CHAPTER 7

JavaScript Prototypes

Introduction

Prototypes in JavaScript enable us to make our code a bit more Object Oriented. JavaScript prototypes help us construct objects nicely. Yes, classes do the same thing, but prototypes address several shortcomings of ordinary JavaScript classes.

There are different prototypes, and we will investigate a few of them. The fun part of this chapter starts with inheritance and Mixins where we can borrow properties and methods from other JavaScript classes

Structure

- What are prototypes?
- Prototypal inheritance
- Mixins

Objectives

- Learn the different types of prototypes
- Understand the need for prototypes
- Understand prototypal inheritance
- Understand Mixins

What are prototypes?

Prototypes in JavaScript enable us to make our code a bit more object-oriented. JavaScript prototypes help us construct objects nicely. Yes, classes do the same thing, but prototypes address several shortcomings of ordinary JavaScript classes.

There are two prototypes in JavaScript, namely prototype and [[Prototype]]:

- `prototype`: The prototype is assigned as a property of any function you create in JavaScript, as explained in *Chapter 6: JavaScript Classes*.
- `[[Prototype]]`: Each object that is accessed by the running context has a `[[Prototype]]` property, which gets read when a certain property on the object being read, is not available.

OK, this doesn't really make sense now, does it? No.

Let's have a closer look.

We have learned in Chapter 6: JavaScript Classes, that you can create objects by using a constructor function. Let's have a look at the below example:

```
function Student(studentFirstName, studentLastName) {
this.studentFirstName = studentFirstName,
this.studentLastName = studentLastName,
this.studentFullName = function() {
return this.studentFirstName + " " + this.studentLastName;
}
}
var student1 = new Student("Ockert", "du Preez");
var student2 = new Student("BpB", "Publications");
console.log(student1)
console.log(student2)
```

The function `Student` is the constructor function for a `Student` object. It includes the first name and last name parameters, or in this case, properties. `studentFullName` is a function combining the first name and last name strings.

Two `Student` objects are then created: `student1` and `student2`, and the results get displayed in the console window.

Now, by looking at this code, nothing seems to be out of place. It is not. It creates two copies of the `Student` constructor function, each for `student1` and `student2`. This means that every object created by using the `Student` constructor function will have its own copy of methods and properties. This is where the problem comes in. The `studentFirstName` method in the constructor. It wastes memory by having two instances of it, and it does the same thing in each separate object.

We fix this by creating prototypes.

Every time a function is created, the JavaScript engine automatically adds a prototype property to it. This is known as a prototype object, and it includes its own constructor by default. The constructor points back to the function on which the prototype object is a property. You access the function's prototype property using `functionName.prototype`.

Figure 7.1 below shows that the Student constructor function contains a prototype property that points to the prototype object. This prototype object contains a constructor property that points back to the Student constructor function:

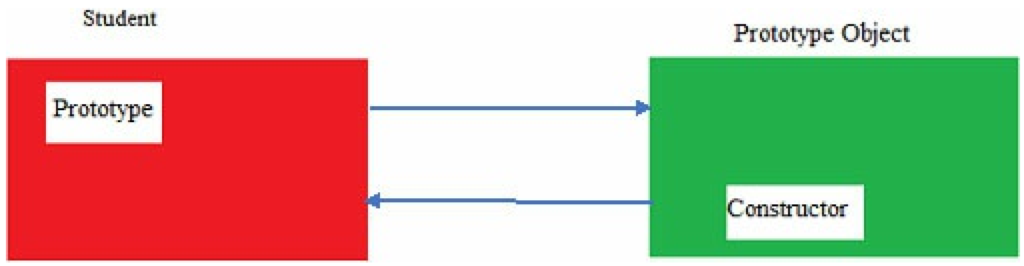


Figure 7.1: *Student constructor function*

Let's do a quick example

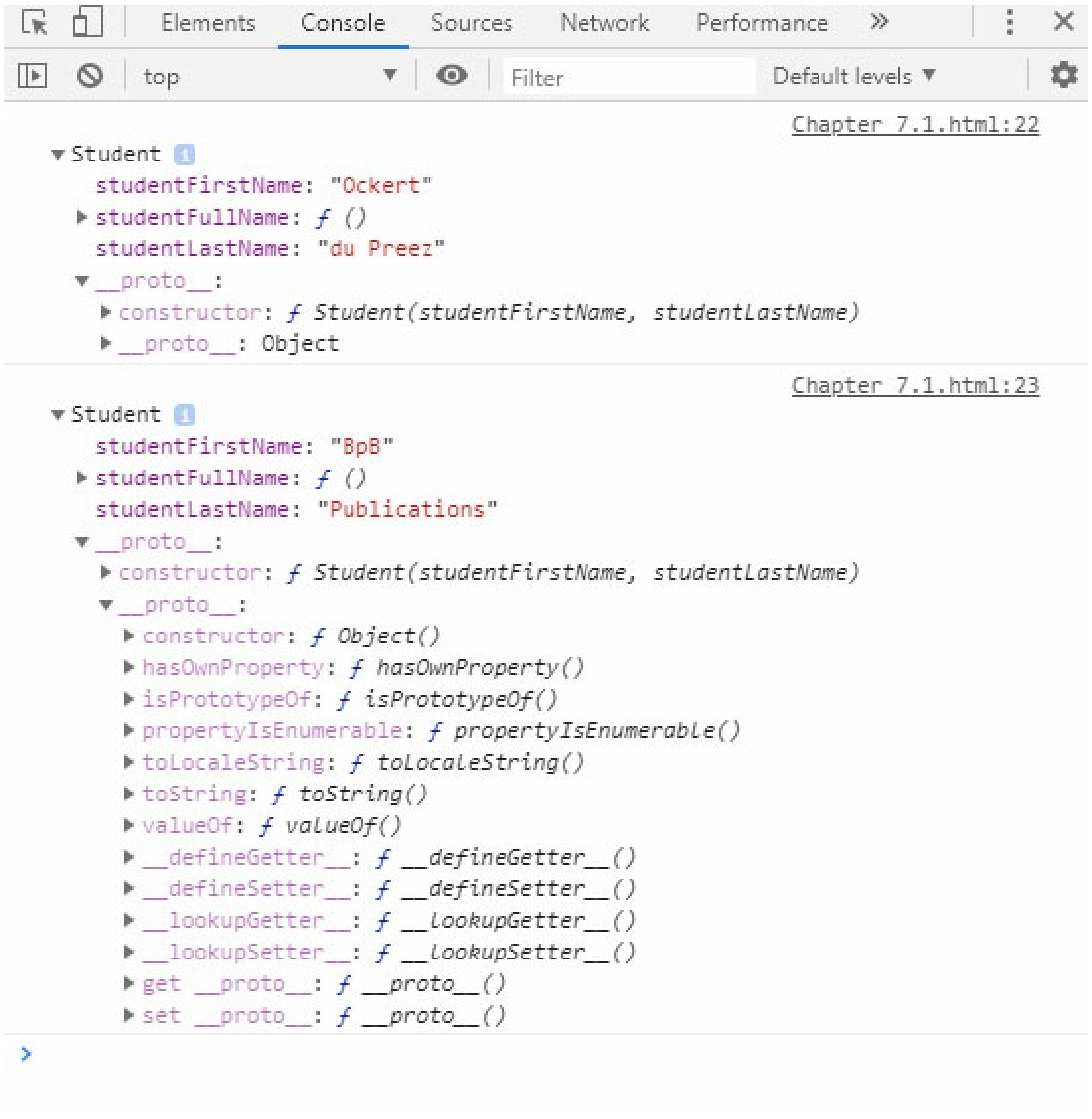
The following sample makes use of a function to construct `Student` objects:

1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<html>
<body>
<h1>Prototype object example</h1>
<p>Prototype object example</p>
<script>
function Student(studentFirstName, studentLastName) {
this.studentFirstName = studentFirstName,
this.studentLastName = studentLastName,
this.studentFullName = function() {
return this.studentFirstName + “ “ + this.studentLastName;
}
}
var student1 = new Student(“Ockert”, “du Preez”);
var student2 = new Student(“BpB”, “Publications”);
// Usage:
console.log(student1)
console.log(student2)
</script>
</body>
```

```
</html>
```

2. Save the file with an HTML extension, for example `Chapter 7.1.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window. The Console window will look like *Figure 7.2* below:



```
                                                    Chapter 7.1.html:22
▼Student
   studentFirstName: "Ockert"
  ▶studentFullName: ƒ ()
   studentLastName: "du Preez"
  ▼__proto__:
    ▶constructor: ƒ Student(studentFirstName, studentLastName)
    ▶__proto__: Object
                                                    Chapter 7.1.html:23
▼Student
   studentFirstName: "BpB"
  ▶studentFullName: ƒ ()
   studentLastName: "Publications"
  ▼__proto__:
    ▶constructor: ƒ Student(studentFirstName, studentLastName)
    ▼__proto__:
      ▶constructor: ƒ Object()
      ▶hasOwnProperty: ƒ hasOwnProperty()
      ▶isPrototypeOf: ƒ isPrototypeOf()
      ▶propertyIsEnumerable: ƒ propertyIsEnumerable()
      ▶toLocaleString: ƒ toLocaleString()
      ▶toString: ƒ toString()
      ▶valueOf: ƒ valueOf()
      ▶__defineGetter__: ƒ __defineGetter__()
      ▶__defineSetter__: ƒ __defineSetter__()
      ▶__lookupGetter__: ƒ __lookupGetter__()
      ▶__lookupSetter__: ƒ __lookupSetter__()
      ▶get __proto__: ƒ __proto__()
      ▶set __proto__: ƒ __proto__()
>
```

***Figure 7.2:** Student objects*

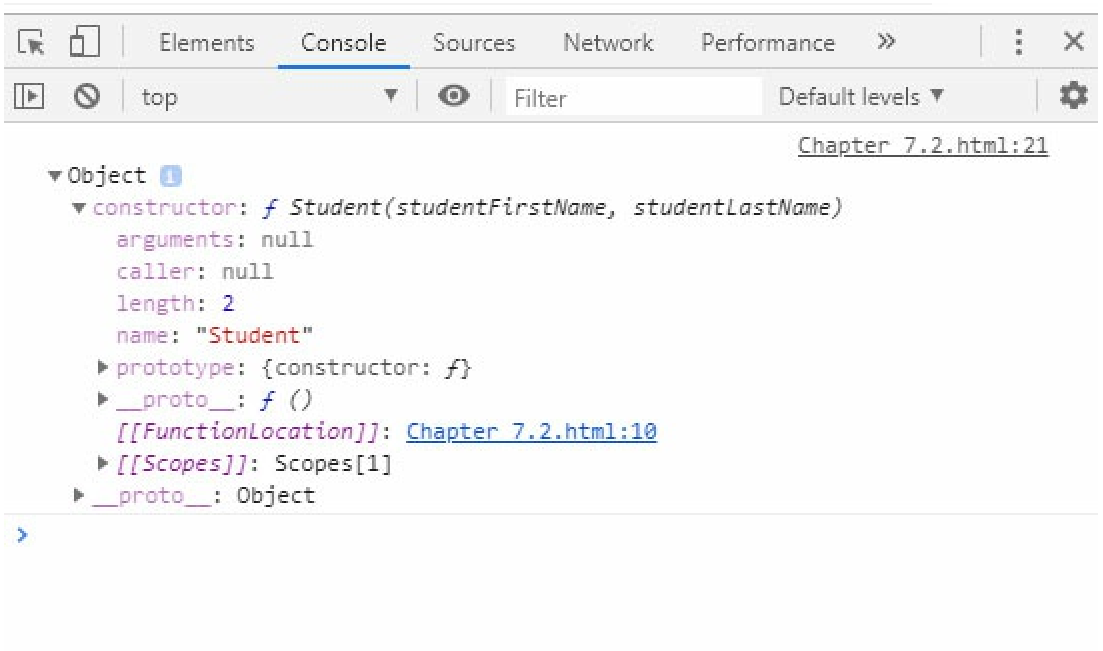
5. Edit your code in the previous exercise(`Chapter 7.1.html`) to resemble the following. Add only the bolded statement:

```
<html>
<body>
<h1>Prototype object example</h1>
```

```
<p>Prototype object example</p>
<script>
function Student(studentFirstName, studentLastName) {
this.studentFirstName = studentFirstName,
this.studentLastName = studentLastName,
this.studentFullName = function() {
return this.studentFirstName + " " + this.studentLastName;
}
}
var student1 = new Student("Ockert", "du Preez");
var student2 = new Student("BpB", "Publications");
console.log(Student.prototype)
</script>
</body>
</html>
```

6. Save the file with an HTML extension, for example `Chapter 7.2.html`.
7. Double-click the file to open it in the default browser.

If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window. The Console window will look like *Figure 7.3* below:



```
                                                          Chapter 7.2.html:21
▼Object
  ▼constructor: ƒ Student(studentFirstName, studentLastName)
     arguments: null
     caller: null
     length: 2
     name: "Student"
   ▶prototype: {constructor: ƒ}
   ▶__proto__: ƒ ()
     [[FunctionLocation]]: Chapter 7.2.html:10
   ▶[[Scopes]]: Scopes[1]
  ▶__proto__: Object
>
```

Figure 7.3: *Prototype object*

The prototype property of the function is an object that has two properties:

- `constructor property`: This points to the `Student` function.
- `__proto__ property`: This points to the prototype object of the

constructor function.

Adding properties to a prototype object

By attaching properties and methods to the prototype object, you enable all the objects created using the constructor function to share them.

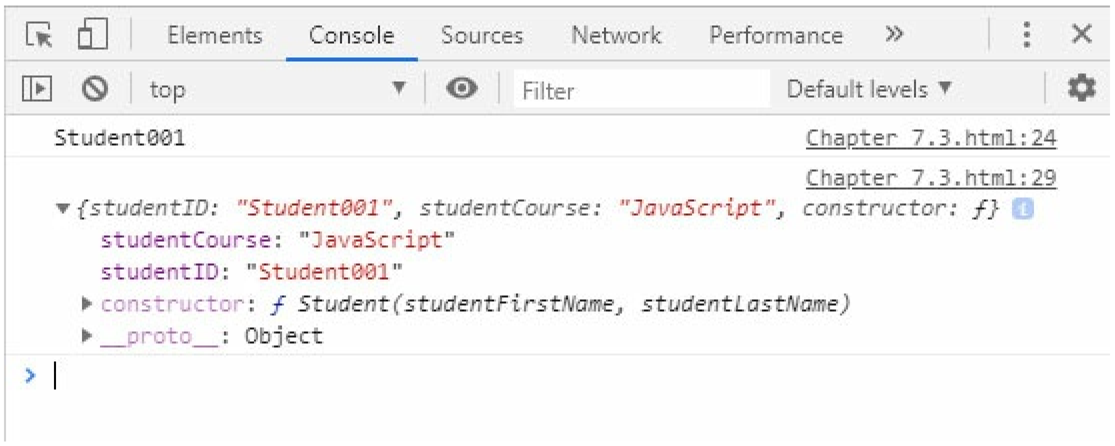
You attach properties and methods to the prototype object in the next exercise.

The following sample demonstrates how to add properties to the prototype object

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Add properties to prototype object example</h1>
<p>Add properties to prototype object example</p>
<script>
function Student(studentFirstName, studentLastName) {
this.studentFirstName = studentFirstName,
this.studentLastName = studentLastName,
this.studentFullName = function() {
return this.studentFirstName + “ “ + this.studentLastName;
}
}
var student1 = new Student(“Ockert”, “du Preez”);
var student2 = new Student(“BpB”, “Publications”);
//Method 1 - dot notation
Student.prototype.studentID = ‘Student001’
console.log(Student.prototype.studentID)
//Method 2 - square brackets
Student.prototype[“studentCourse”] = ‘JavaScript’
console.log(Student.prototype)
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 7.3.html`.
3. Double-click the file to open it in the default browser
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window. The Console window will look similar to *Figure 7.4* below:



```
Student001                                              Chapter 7.3.html:24
                                                        Chapter 7.3.html:29
▼{studentID: "Student001", studentCourse: "JavaScript", constructor: ƒ}
    studentCourse: "JavaScript"
    studentID: "Student001"
  ▶constructor: ƒ Student(studentFirstName, studentLastName)
  ▶__proto__: Object
>
```

Figure 7.4: *Properties added to prototype object*

If you look closely, you will notice that I have made use of two different techniques to attach a property to the prototype object.

The first method I have used was to make use of dot notation to add the `studentID` property and value to the `Student` prototype. The next method I used was square brackets. I made use of the square bracket notation technique to attach the `studentCourse` and value to the `Student` prototype object.

Prototypal inheritance

Let's say, for examplewe have various students. Do you really want to create a separate class for each student, or a separate implementation of function prototypes? All `Student` objects contain similar properties and methods. `Car` objects contain similar methods and properties than other cars.

Now, what if we simply want to take an existing object and extend it? The answer is prototypal inheritance. Inheritance means when you inherit features (methods or properties) from other prototypes. The benefit of inheritance is that after you have inherited the object features, you can extend it, and this saves a lot of work when creating objects and prototypes because you do not have to recreate similar objects.

Let's look at an example.

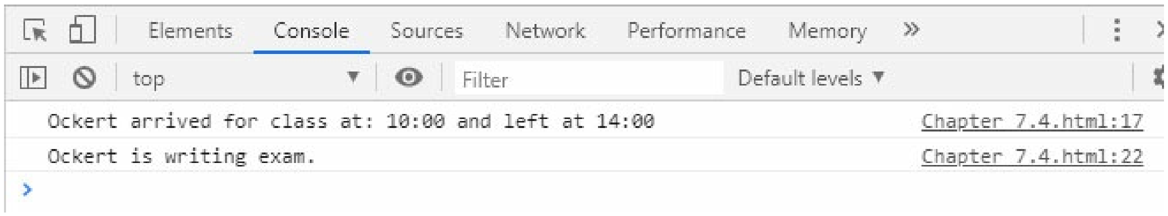
The following exercise demonstrates creating a prototype or a class:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
```

```
<body>
<h1>prototypal inheritance example</h1>
<p>prototypal inheritance example</p>
<script>
function Student (studentFirstName, studentLastName) {
this.studentFirstName = studentFirstName,
this.studentLastName = studentLastName
}
Student.prototype.attendClass = function (timeIn, timeOut) {
console.log(this.studentFirstName + ' arrived for class at:
' + timeIn + ' and left at ' + timeOut)
}
Student.prototype.writeExam = function () {
 console.log(this.studentFirstName + ' is writing exam.')
}
const student1 = new Student('Ockert', 'du Preez');
student1.attendClass('10:00', '14:00');
student1.writeExam()
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 7.4.html`.
3. Double-click the file to open it in the default browser
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window. The Console window will look similar to *Figure 7.5* below:



```
Ockert arrived for class at: 10:00 and left at 14:00      Chapter 7.4.html:17
Ockert is writing exam.                                   Chapter 7.4.html:22
>
```

Figure 7.5: *Prototype object*

In the above exercise, you made use of a function to create a `Student` and included the `studentFirstName` and `studentLastName` parameters. You then added two methods to the `Student` prototype. One method is `attendClass` and the last method is `writeExam`. Both functions log a string to the browser's console window, as shown in *Figure 7.5* above.

Based on the `Student` prototype we have created, we sit with a problem. If we want to add a programming student, or a web design student, or a literacy student to the project, we'd basically have to duplicate this code for each of the classes or prototypes. So, what now? You will have to inherit from the `Student` class:

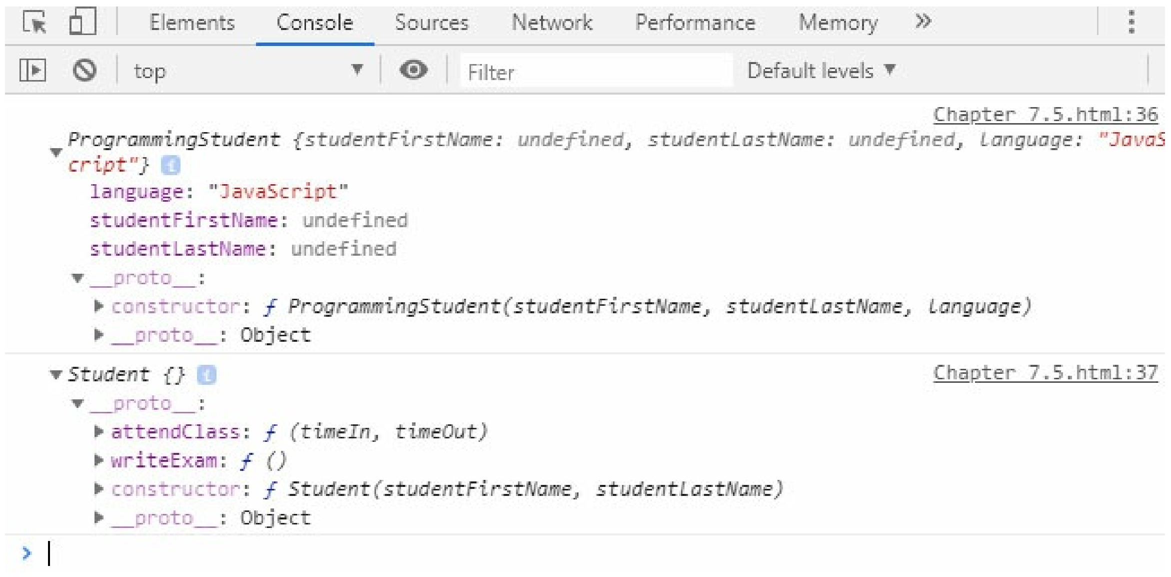
Let’s look at an example.

The following exercise demonstrates prototypal inheritance:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>prototypal inheritance example</h1>
   <p>prototypal inheritance example</p>
   <script>
   function Student (studentFirstName, studentLastName) {
   this.studentFirstName = studentFirstName,
   this.studentLastName = studentLastName
   }
   Student.prototype.attendClass = function (timeIn, timeOut) {
    console.log(this.studentFirstName + ‘ arrived for class at:
    ‘ + timeIn + ‘ and left at ‘ + timeOut)
   }
   Student.prototype.writeExam = function () {
    console.log(this.studentFirstName + ‘ is writing exam.’)
   }
   function ProgrammingStudent(studentFirstName,
   studentLastName, language) {
    Student.call(this)
   this.language = language
   }
   const progStudent = new ProgrammingStudent(‘Ockert’, ‘du
   Preez’, ‘JavaScript’)
   ProgrammingStudent.prototype =
   Object.create(Student.prototype)
   console.log(progStudent)
   console.log(ProgrammingStudent.prototype)
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 7.5.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window. The Console window will look like *Figure 7.6* below:



```
                                                                    Chapter 7.5.html:36
▼ProgrammingStudent {studentFirstName: undefined, studentLastName: undefined, language: "JavaS
 cript"}
   language: "JavaScript"
   studentFirstName: undefined
   studentLastName: undefined
  ▼__proto__:
    ▶constructor: ƒ ProgrammingStudent(studentFirstName, studentLastName, language)
    ▶__proto__: Object
▼Student {}                                                         Chapter 7.5.html:37
  ▼__proto__:
    ▶attendClass: ƒ (timeIn, timeOut)
    ▶writeExam: ƒ ()
    ▶constructor: ƒ Student(studentFirstName, studentLastName)
    ▶__proto__: Object
> |
```

Figure 7.6: *Prototypal inheritance*

As you can see, the `ProgrammingStudent` class was created without the `attendClass` and `writeExam` methods, but now contains them as well. This is because the `ProgrammingStudent` class' prototype has inherited its methods and properties from the `Student` class' prototype on this line:

```
ProgrammingStudent.prototype = Object.create(Student.prototype)
```

Mixins

A limiting factor of JavaScript is that we can only inherit from a single object because an object can only have one [[Prototype]]. Another limiting factor of JavaScript is that a class can extend only one other class. What if we want to combine features of two or more different classes? Or, what if we want to mix-up different classes?

The answer is *Mixins*.

A Mixin is a class that contains methods which can be used by other classes without having to inherit from it.

In the next exercise you will create a Mixin with its methods, then link this Mixin into an existing class.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   ```

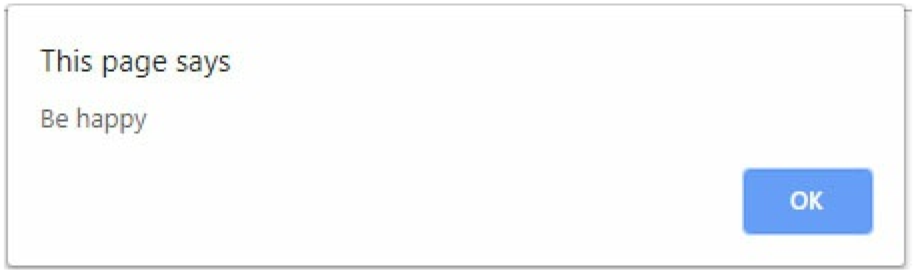
```
<body>
<h1>Mixin example</h1>
<p>Mixin example</p>
<script>
// mixin
let HumanMixin = {
 cry() {
  alert('Stop crying');
 },
 laugh() {
  alert('Be happy');
 }
};
function Student (studentFirstName, studentLastName) {
this.studentFirstName = studentFirstName,
this.studentLastName = studentLastName
}

Student.prototype.attendClass = function (timeIn, timeOut) {
 console.log(this.studentFirstName + ' arrived for class at:
 ' + timeIn + ' and left at ' + timeOut)
}

Student.prototype.writeExam = function () {
 console.log(this.studentFirstName + ' is writing exam.')
}

function ProgrammingStudent(studentFirstName,
studentLastName, language) {
 Student.call(this)
 this.language = language
}
ProgrammingStudent.prototype =
Object.create(Student.prototype)
// copy the methods from HumanMixin
Object.assign(ProgrammingStudent.prototype, HumanMixin);
const progStudent = new ProgrammingStudent('Ockert', 'du
Preez', 'JavaScript').laugh();
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 7.6.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window. An alert box like *Figure 7.7* below will be displayed:


This page says

Be happy

Figure 7.7: *Mixin method*

Some of the code may look familiar. I try to keep the exercises as easy as possible and let each exercise build upon the previous. Let's interrogate the code.

You created a Mixin named `HumanMixin` containing two methods: `cry()` and `laugh()`. You create a `Student`class, and then make use of inheritance to create a `ProgrammingStudent` class based on the `Student` class. This means that the `ProgrammingStudent` now contains its own methods and properties as well as those inherited from the `Student` class.

With the following code, you use the assign method of `Object` to copy the Mixins methods to the desired class. Lastly, you simply made use of the laugh method, which now also exists in the `ProgrammingStudent` class:

```
Object.assign(ProgrammingStudent.prototype, HumanMixin);
const progStudent = new ProgrammingStudent(‘Ockert’, ‘du Preez’,
‘JavaScript’).laugh();
```

Conclusion

In this chapter, we learnt JavaScript prototypes. To fully make use of JavaScript's power, you need to know its object-oriented features such as prototypes. Prototypes let you extend JavaScript classes with the help of inheritance and Mixins.

In the next chapter, we willdive into JavaScript properties. You will learn how to create properties properly. We will move on to creating getters and setters for properties and more involved features such as property flags and property descriptors.

Questions

1. Explain the term prototype.

2. What are some benefits of using prototypes?
3. Explain the term prototypal inheritance.
4. Why would you make use of Mixins?
5. How do you attach properties to prototypes?

CHAPTER 8
JavaScript Properties

Introduction

Properties in JavaScript enable you to store and set various values during the course of your application's lifetime. By storing information in properties provides a better structure to your applications

In this lesson you will create properties and learn how to set each property's attributes to be writeable or read-only. You will learn about the differences between object properties and accessor properties. Finally, you will learn the ins-and-outs of getters and setters.

Structure

- What are properties?
- Properties of properties
- Getters and setters
- Adding tricks to getters and setters

Objectives

- Learn what properties are
- Understand getters and setters
- Create properties with limitations

What are properties?

In *Chapter 6: JavaScript Classes*, we had a quick overview of properties. Let's refresh our minds a bit. A property describes an object. Any object has certain attributes that can be set and retrieved. Take, for example, a car object. There are numerous attributes that can be set for a car; these include:

- Make of car
- Model of car
- Transmission
- Fuel type
- Color
- Number of seats

Although there are different makes and models of cars, with different colors and transmission and fuel types, each car is still classified as a car. Why? Because the car object still gets made from the same car template or class.

In the code examples we have done in this chapter thus far, we have seen how to set properties. These are known as *instance properties*. The one important thing to note there was that you had used the dot notation to set a property's value and to set its value. Let's take it a step further.

To access JavaScript properties, use the following syntax:

```
objectName.property     // student.studentName
```

OR

```
objectName[“property”]     // student[“studentName”]
```

OR

```
objectName[expression];     // newStudent = “studentName”;
student[newStudent]
```

You may be wondering how do I add new properties to an object. Well, it is quite simple! You simply give it a value. Here is an example:

```
student.studentCourse = “JavaScript”;
```

You start with the instance of the class, then the property that needs to be set, and after the assignment operator (the equal sign) you supply the new value for the property.

Properties of properties

The most common attributes of a property are its value and its name. You access property by using its name, but that supplies the value requested. This is where it gets tricky! Some properties cannot be written to, this means that they are read-only and cannot be changed. Some properties cannot be looped over by using a loop. They must be enumerable. Some properties cannot be deleted.

These all depend on the attributes (or flags) of a certain property. Primarily there are three attributes (or flags) that can be set. They are:

- **writable**: Determines if the value of the property can be changed.
- **enumerable**: Can the property be listed in loops?
- **configurable**: Allows property deletion, or property attributes to be modified.

In order to access the attributes for any property, you can use the `Object.getOwnPropertyDescriptor` method. You supply the object which you'd like to interrogate, then the property of the object. Here is a small exercise.

The following sample shows `Object.getOwnPropertyDescriptor` in action:

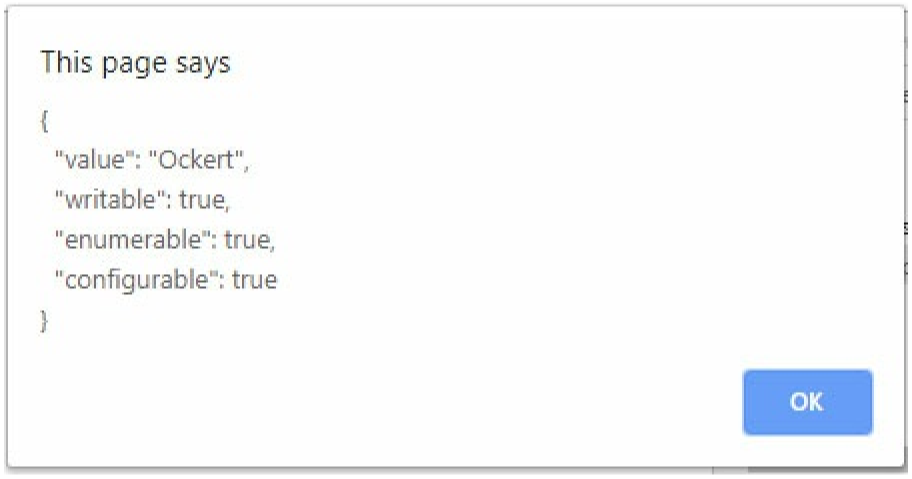
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Object.getOwnPropertyDescriptor example</h1>
<p>Object.getOwnPropertyDescriptor example</p>
<script>
let student = class Student {
 constructor(studentName, studentCourse) {
  this.studentName = studentName;
  this.studentCourse = studentCourse;
 }
 showInfo()
 {
let descriptor = Object.getOwnPropertyDescriptor(newStudent,
‘studentName’);
alert(JSON.stringify(descriptor, null, 2));
 }
};
// Usage:
var newStudent = new student(‘Ockert’, ‘JavaScript’);
newStudent.showInfo();
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 8.1.html`.
3. Double-click the file to open it in the default browser.

In the above code segment, you created an object. Then in the `showInfo` method, you made use of the `getOwnPropertyDescriptor` method to investigate the attributes of the `studentName` property of the `newStudent` object.

Figure 8.1 shows the attributes of the `studentName` property:



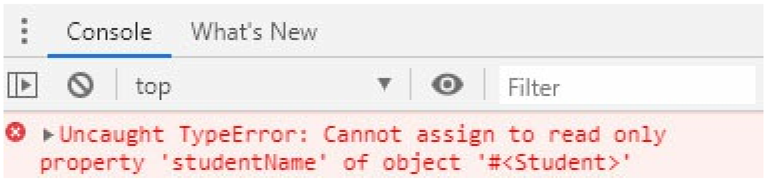
```
This page says
{
  "value": "Ockert",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
```

Figure 8.1: *studentName attributes*

To change these attributes, you should use the `defineProperty` method of the `Object` class. The next example changes the writable attribute of the `studentName` property to false. This means that this property is now read-only and cannot be changed from code:

```
Object.defineProperty(newStudent, “studentName”, {
 writable: false
});
newStudent.studentName = “Test”; // Error: Cannot assign to read
only property ‘studentName’
```

Figure 8.2 shows the error if code similar to the code segment above was run inside the browser:



```
Uncaught TypeError: Cannot assign to read only
property 'studentName' of object '#<Student>'
```

Figure 8.2: *defineProperty error*

Now that we have a proper understanding of properties, let’s move on to creating getters and setters for our properties.

Getters and setters

So far, in this book, I have mostly dealt with data properties. There is another type of property called an accessor property. This means that these types of properties are actually functions that can get and set values but are seen as regular properties from the outside.

Accessor properties contain *getter* and *setter* methods, as shown next:

```
let object = {
 get propertyName() {
  //Code executed on getting object.propertyName
 },
 set propertyName(value) {
  // Code executed on setting object.propertyName = value
 }
};
```

In the object declaration above, you will notice a `get` and a `set`. Get will execute when the object’s property is requested. For example: `c = Object.Color`. The get function will return the value of the object’s `Color` property to the variable named `c`.

Set is used to give a property value. For example: `Object.Color = Red`. The `Color` property will be set to `Red` via the use of the setter.

Let’s do a small exercise so that you can see how easy it is to make use of getters and setters.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>Property Getters and Setters example</h1>
   <p>Property Getters and Setters example</p>
   <script>
   let Student = {
    studentName: “Ockert”,
    studentSurname: “du Preez”,
    get studentFullName() {
     return ‘${this.studentName} ${this.studentSurname}’;
   ```

```
},
set studentFullName(value) {
 [this.studentName, this.studentSurname] = value.split(" ");
}
};
Student.studentFullName = "BpB Publications";
alert(Student.studentName); //BPB
alert(Student.studentFullName); //BpB Publications
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 8.3.html`.
3. Double-click the file to open it in the default browser.

With the use of the setter, you set the `studentFullName` property to `BPB Publications`. If there was no property setter but a property letter, then you might have gotten an error. With the use of the property getter, you got access to the property's current value. When opened, two alertboxes will be displayed. One showing `BpB`, and the next showing `BPB Publications`.

Figure 8.3 shows a combination of the given alertboxes based on the above example:

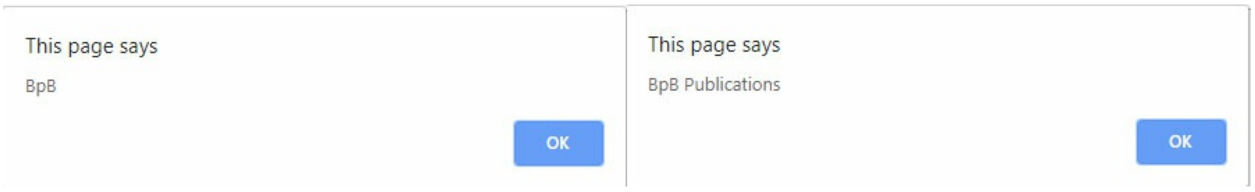

This page says
BpB
This page says
BpB Publications

Figure 8.3: *Getter and setter properties*

Just as with property flags, explained earlier in this chapter, there are descriptors for accessors. Let's have a look at them.

Accessor descriptors differ slightly from descriptors those for data properties. Accessor properties have no writable descriptor but include get and set functions. Accessor descriptors have the following:

- **get**: Works when a property is read.
- **set**: Called when the property is set.
- **enumerable**: Can the property be listed in loops?
- **configurable**: Allows property deletion, or property attributes to be modified.

Let’s do a small exercise so that you can see how easy it is to make use of getters and setters.

1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<html>
<body>
<h1>Accessor Descriptors example</h1>
<p>Accessor Descriptors example</p>
<script>
let Student = {
studentName: “Ockert”,
 studentSurname: “du Preez”,
};
Object.defineProperty(Student, ‘studentFullName’, {
 get() {
 return ‘${this. studentName} ${this. studentSurname}’;
 },
 set(value) {
  [this. studentName, this. studentSurname] = value.split(“
  “);
 }
});
alert(Student.studentFullName); //Ockert du Preez
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 8.4.html`.
3. Double-click the file to open it in the default browser/

In the above exercise, I have made use of the `defineProperty` method of `Object`. I included it get and set methods, and there you go, another property added.

Adding tricks to getters and setters

By tricks I mean include logic to see if a supplied value is of the correct type or the value is in the correct range. We can include logic to check the length of strings or check a certain integer’s value. By adding logic to getters and setters, we basically create a wrapper over real property values.

Let’s quickly do an example.

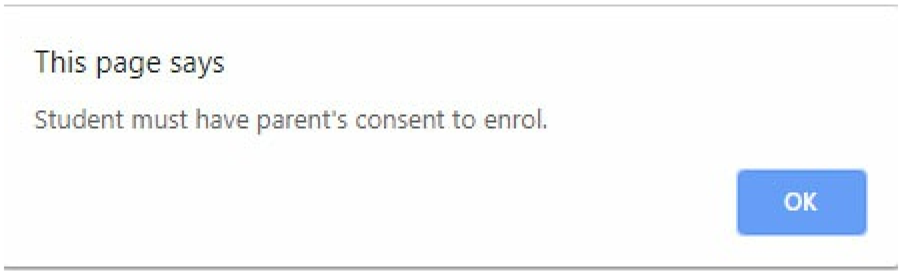
1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<html>
<body>
<h1> Age Wrapper example</h1>
<p> Age Wrapper example</p>
<script>
let Student = {
 studentName: “Ockert”,
 studentSurname: “du Preez”,
 get studentAge() {
return this._age;
},
 set studentAge(value) {
  if (value < 18) {
  alert(“Student must have parent’s consent to enrol.”);
  return;
 }
this._age = value;
 }
};
Student.studentAge = 17;
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 8.5.html`.
3. Double-click the file to open it in the default browser.

I have created a getter and setter to store the `Student` object’s `age` and to retrieve it. You will notice that I have included an `if` check inside the setter to ensure that the student’s `age` is higher than or equal to 18. When you set the `studentAge` property to a value less than `18`, an alert box displays stating that the student needs the parent’s consent in order to enroll for a course.

Figure 8.4 below shows the resulting alert box:


This page says

Student must have parent's consent to enrol.

Figure 8.4: Age test inside property setter

If however the correct input is supplied to the `studentAge` property, in this case, age above 17, there will be no alert box as the value is acceptable.

Conclusion

In this chapter, we learntexplained JavaScript properties. Knowing how to create properties is crucial in building proper apps in JavaScript. You have learned how to create properties and the different types of properties you get. You have then learned how to create flags and descriptors for your properties. Lastly, you have created a complex property that checks for valid input.

In the next chapter, we willdive into JavaScript promises. You will learn what promises are, how to chain them and get a bit complicated with the Promise methods, asynchronous and synchronous calls and microtasks.

Questions

1. Explain the term property.
2. What is the difference between accessor properties and object properties?
3. What is the purpose of getters and setters?
4. Explain property attributes.
5. Give an example of how you would create a property that can produce an error when a wrong value is supplied to it.

CHAPTER 9

JavaScript Promises

Introduction

A nice feature of the JavaScript language is that it's able to do tasks asynchronously. Synchronous tasks can block other needed tasks, thus delaying instructions. Being able to do tasks independently and not to worry about when they will finish is very helpful. It is also quite easy to set up.

In this lesson, you will learn how to create promises to enable more `async` features on your pages. We will also have a look at callbacks and how to chain multiple promises.

Structure

- JavaScript Promises description
- Synchronous versus asynchronous methods
- Chaining Promises
- Callbacks and Promisification
- Microtasks and macrotasks
- Promise methods

Objectives

- Learn what Promises are
- Understand the difference between synchronous and asynchronous methods
- Understand the difference between macrotasks and microtasks
- Learn how to make multiple promises work together

JavaScript Promises description

A dictionary explains the noun *promise* as the following: A declaration or assurance that one will do something or that a particular thing will happen. So, what happens when a promise is made between two parties?

A promise gives one party the assurance that something will be done. Whether the task will be done by the party that has made the promise, or someone else is immaterial. The assurance is given, based on which the other party can plan something.

Promises can either be kept or broken. This is where different routes develop, and different scenarios should be catered for. When a promise is kept the one-party expects something out of that promise. This allows the party to make use of the promise’s output for its further actions or plans. When a promise is broken, one-party should know why the promise was not kept. Once the party knows the reason for the promise not being kept, the party can plan what to do next or how to handle it.

Upon making a promise, both parties have an assurance that a task will get done and they can decide a course of action when the promise is kept, or when the promise is broken.

The party that has made the promise may even never respond at all. What happens then?

In such cases, one party should keep a time threshold. For example: if the promising party doesn’t respond in a given time, the promised party may realize that the promise may not be kept, due to unforeseen circumstances. Even then if the promising party responds later than what it was supposed to, it doesn’t matter, because the promised party may have made alternate plans already.

JavaScript promises

A JavaScript promise works similar to an ordinary promise, as explained above. You can associate handlers with an asynchronous action’s success value or failure reason. Instead of immediately returning the success or failure value, the asynchronous method returns a promise to supply the appropriate value eventually.

A Promise has three states:

- **pending**: This is the promise’s initial state. The promise is neither fulfilled nor rejected (the other two states).
- **resolved**: This means that the promise’s operation has completed successfully.
- **rejected**: This means that the promise’s operation has failed.

JavaScript promises can either be fulfilled with a value or rejected with a reason - an error. Depending on which of these options happens, the associated handlers queued up by a promise’s then method is called. We will do some examples now.

The following sample shows a resolved promise in action

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

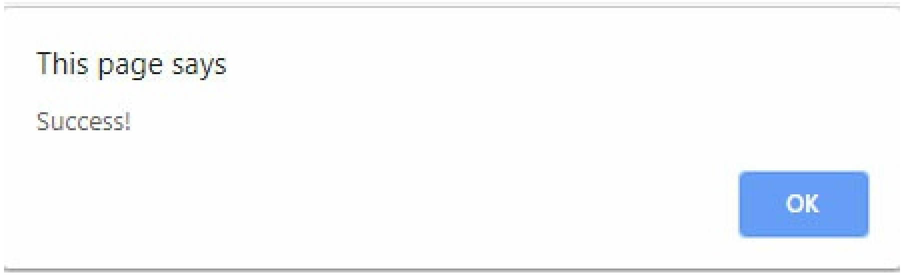
   ```
   <html>
   <body>
   <h1>Promise resolve example</h1>
   <p>Promise resolve example</p>
   <script>
   let promise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve(“Success!”), 5000); //5000
    milliseconds = 5 seconds
   });
   promise.then(
    result => alert(result), //Displays “Success!” after 5
    seconds, if successful
    error => alert(error) // doesn’t run
   );
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 9.1.html`.
3. Double-click the file to open it in the default browser.

In the above code segment, you created a new `Promise` with two arguments – resolve and reject. Inside the Promise, you make use of the built-in JavaScript timer function named `setTimeout` to start a timer. After 5 seconds, it should display a message of `Success!`.

When the `Promise` has completed its task, it will either return a reject message or a resolved message. In this case, an alert box with the message of

`Success!`will pop-up. This looks like *Figure 9.1* underneath:


This page says

Success!

Figure 9.1: *Resolved Promise*

Let’s do an example of a rejected promise next.

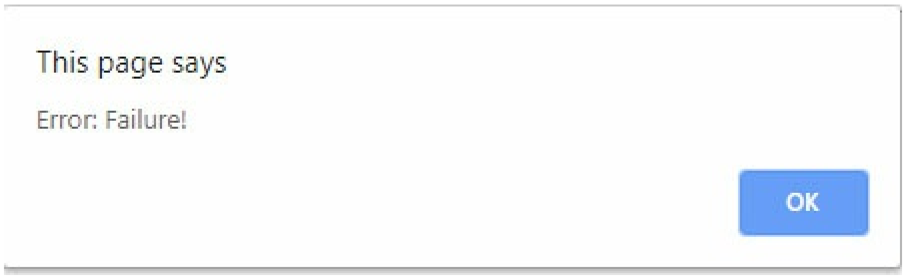
The following sample shows a rejected promise in action:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Promise resolve example</h1>
<p>Promise resolve example</p>
<script>
let promise = new Promise(function(resolve, reject) {
 setTimeout(() => reject(new Error(“Failure!”)), 5000);
 //5000 milliseconds = 5 seconds
});
promise.then(
 result => alert(result), // doesn’t run
 error => alert(error) //Displays “Error: Failure!” after 5
 seconds
);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 9.2.html`.
3. Double-click the file to open it in the default browser.

When the `Promise` has completed its task, it will either return a reject message or a resolvedmessage. In this case an alert box with the message of `Error: Failure!` shows. This looks like *Figure 9.2* underneath:


This page says

Error: Failure!

Figure 9.2: *Rejected Promise*

I have mentioned asynchronous and synchronous methods and tasks earlier, but what are they? Let's go into greater detail.

Synchronous versus asynchronous methods

Let's assume you start your very own cupcake business. It is relatively small, so you have direct interaction with your clients. They will order a cupcake or two, you will bake the number of cupcakes, and then you will deliver the cupcakes. This is synchronous behavior because it is an ask-and-get scenario.

Imagine further that your cupcakes are really taking off and you get busier, and there are even talks of franchising! You cannot be at different places at the same time. You cannot serve different customers at the same time. You need help. This is where iasynchronicity comes in. Asynchronocity allows you to put orders on a waiting list and complete them when and if you can.

When JavaScript does a synchronous task, the browser stops the loading (or rendering) of the page so that the code can execute. This blocks the page from finishing to load until the code execution completes.

Asynchronously acts in the opposite way than synchronous. When JavaScript does an asynchronous task, the code is processed in parallel to the rest of the page content. This means that the page doesn't stop loading, and it doesn't slow down the loading of the page.

Let's do an exampleto demonstrate the differences between synchronous and asynchronous tasks.

The following sample shows synchronous calls in action:

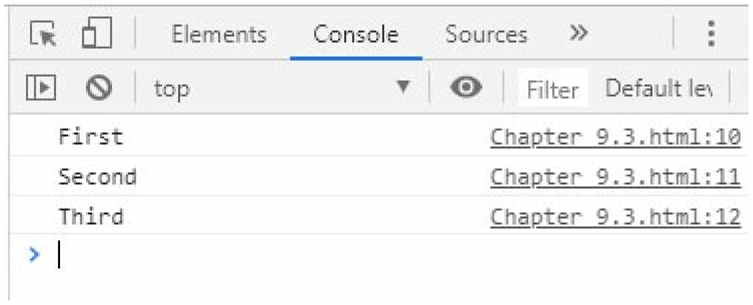
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
```

```
<h1>Synchronous methods example</h1>
<p>Synchronous methods example</p>
<script>
console.log("First");
console.log("Second");
console.log("Third");
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 9.3.html`.
3. Double-click the file to open it in the default browser. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The instructions will happen one at a time and one after the other. In the event of one of the instructions are taking too long, the next instructions will wait until its completion to be run. Your console window should resemble *Figure 9.3* below:



```
First                                    Chapter 9.3.html:10
Second                                   Chapter 9.3.html:11
Third                                    Chapter 9.3.html:12
>
```

Figure 9.3: *Synchronous calls in action*

The following sample shows asynchronous calls in action:

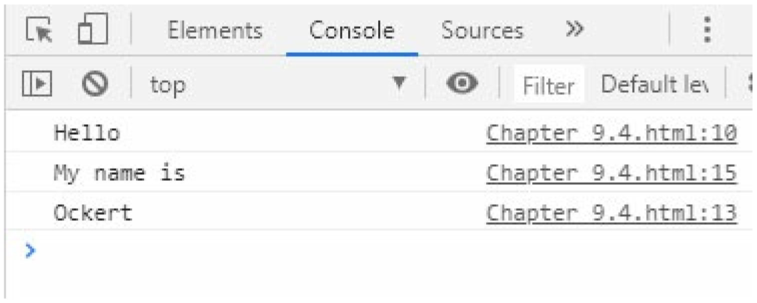
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Asynchronous methods example</h1>
<p>Asynchronous methods example</p>
<script>
console.log("Hello")
setTimeout(function()
{
console.log("Ockert")
}, 3000);
console.log("My name is ")
```

```
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 9.4.html`.
3. Double-click the file to open it in the default browser. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The first statement to be run would be the statement logging `Hello` into the console window. It will start the `Timer`, but still continue onto the next statement logging `My name`. When the timer has completed, it will log my name. As you can see, it didn't wait for the timer to completeits task in order to continue. Your console window should resemble *Figure 9.4* below:



```
Hello          Chapter 9.4.html:10
My name is     Chapter 9.4.html:15
Ockert         Chapter 9.4.html:13
```

Figure 9.4: *Asynchronous calls in action*

Now that you have a better understanding of asynchronous calls let's shift our focus back to promises.

Chaining Promises

What if we take asynchronous tasks a bit further? How? Well, imagine you have a sequence of asynchronous tasks to be done one after another. In other words, you want to call `method_2` when `method_1` completes, and when `method_2` completes, you then want to call `method_3`, and so on. This is where chaining comes in.

You literally chain as many then ables needed in your promise by using the keyword `.then`. The following sample shows how to chain promises:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
```

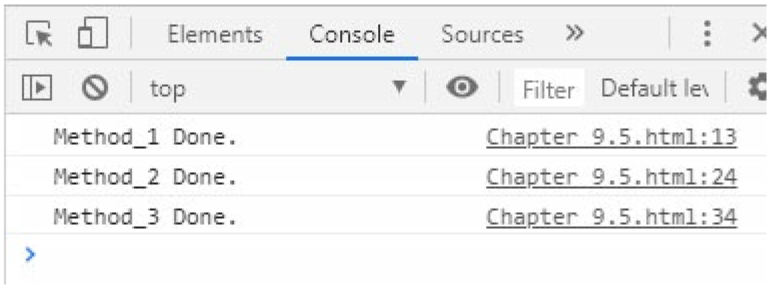
```
<h1>Chaining promises example</h1>
<p>Chaining Promises example</p>
<script>
var Method_1 = function() {
var promise = new Promise(function(resolve, reject){
  setTimeout(function() {
console.log('Method_1 Done.');
  resolve({string: '1'});
  }, 3000);
});
return promise;
};
var Method_2 = function(strTemp) {
var promise = new Promise(function(resolve, reject){
setTimeout(function() {
console.log('Method_2 Done.');
resolve({newString: strTemp.string + '2'});
}, 3000);
});
return promise;
};
var Method_3 = function(strTemp) {
var promise = new Promise(function(resolve, reject){
setTimeout(function() {
console.log('Method_3 Done.');
resolve({endString: strTemp.newString});
}, 3000);
});
 return promise;
};
Method_1()
.then(Method_2)
.then(Method_3);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 9.5.html`.
3. Double-click the file to open it in the default browser. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

There are three asynchronous methods in the example above. You will also notice that each method contains a promise with a resolve argument. This is the trick. When the asynchronous method completes successfully, it sends the resolve argument to the next asynchronous method. The argument contains a simple string value that is passed on to the next method.

This information is then not really used, but the string grows continuously. In the event of something wrong happens, the resolve argument will fail, and the promise would also fail.

Each method contains a timer that prints a string of text after three seconds. If all went well, your console window should resemble *Figure 9.5* underneath:



```
Method_1 Done.                Chapter 9.5.html:13
Method_2 Done.                Chapter 9.5.html:24
Method_3 Done.                Chapter 9.5.html:34
>
```

Figure 9.5: *Chaining Promises*

Continuing with asynchronous tasks and methods, those of you who are a bit more advanced or experienced, may be surprised that I haven't mentioned callbacks yet, well I try to do things systematically, especially for newbies.

Callbacks and Promisification

I will do callbacks now, but I will then talk about Promisification as well because they tie into one another.

Callbacks

A callback is simply a function that executes after another function has finished executing. It sounds easy enough, so let's look at an easy but complex example.

The following sample creates a callback:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Callback example</h1>
<p>Callback example</p>
<script>
 function DisplayName(name, surname, callback){
 document.write('Welcome ${name + surname}!' +"<br>");
callback();
}
function EnjoyMessage(){
document.write('Enjoy your stay.');
}
DisplayName('Ockert',' du Preez', EnjoyMessage);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 9.6.html`.
3. Double-click the file to open it in the default browser.

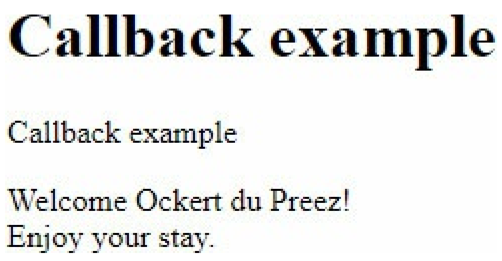
In the above example, you have created a function named `DisplayName`. This function accepts three arguments: name, surname and callback. The next line you use `document.write` to write text to the page; in this case, it will be my name. Then callback is called.

The next function you created is the function named `EnjoyMessage`. This function simply writes more text to the web page document. The question,, however, is when does it do that?

Well, the last line of code you invoke the `DisplayName` function. You feed it my name and surname, and then you feed it the `EnjoyMessage` function. The `EnjoyMessage` function is the callback function, and it will run as soon as the other function has completed.

See, it is not too difficult!

Your web browser should resemble *Figure 9.6*shown below:



```
Callback example

Callback example

Welcome Ockert du Preez!
Enjoy your stay.
```

Figure 9.6: *Callback in action*

Now that we know more about callbacks let's have a look into Promisification.

Promisification

Promisification is the process of converting a function that accepts a callback into a function that returns a promise.

The following sample shows a quick way to promisify a callback function:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Promisification example</h1>
<p>Promisification example</p>
<script>
function promiseDisplayName(name, surname) {
  return new Promise((resolve, reject) =>
 DisplayName(name, surname, (err, result) =>
   // If there’s an error, reject; otherwise resolve
   err ? reject(err) : resolve(result)
)
);
}
//Invoke it like:

promiseDisplayName(‘Ockert’, ‘ Du Preez’).then(result => {
//resolved
}).catch(err => {
// error
});
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 9.7.html`.
3. Double-click the file to open it in the default browser.

Callbacks and promises have their uses. There are some tasks that can be easily promisified.

Microtasks and macrotasks

A microtask is simply a short function (hence the name) that executes after its creation function exits. Promises in JavaScript make use of microtasks in order to run their callbacks.

A macrotask is any JavaScript code which is run by event callbacks, starting of a program or firing intervals.

Allow me to explain, and there are mainly two differences.

The first difference is, when a macrotask exits, the JavaScript event loop checks whether the macrotask is returning control to any other code. If it does not, the event loop proceeds to run all the microtasks present in the microtask queue.

The second difference is, when a microtask adds more microtasks to its queue by using the `queueMicrotask()` method, the new microtasks execute before the next task is run. This is because the event loop will keep calling microtasks until there are none left in the queue.

Let's see this in a practical example.

The following sample shows how to chain promises:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

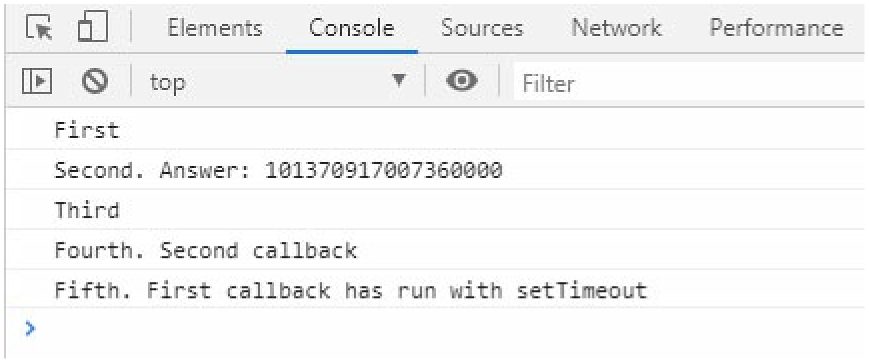
   ```
   <html>
   <body>
   <h1>Microtasks versus Macrotasks example</h1>
   <p>Microtasks versus Macrotasks example</p>
   <script>
   let firstCallback = () => console.log("Fifth. First callback
   has run with setTimeout");
   let secondCallback = () => console.log("Fourth. Second
   callback");
   let doTasks = () => {
    let result = 1;
    queueMicrotask(secondCallback);
    for (let i = 5; i <= 20; i++) {
     result *= i;
    }
    return result;
   };
   console.log("First");
   setTimeout(firstCallback, 0);
   console.log('Second. Answer: ${doTasks()}');
   console.log("Third");
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 9.8.html`.
3. Double-click the file to open it in the default browser. If the browser is

Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The above code sample uses `queueMicrotask()` to schedule a microtask. The microtask isn't processed upon function exit, but instead when the main program exits. The `doTasks()` function calls `queueMicrotask()`, yet the microtask being called doesn't fire until the entire program exists, because that's when the task exists and there's nothing else on the execution stack. Notice the order in which the tasks get executed.

Your screen would resemble *Figure 9.7* shown below:



```
First
Second. Answer: 101370917007360000
Third
Fourth. Second callback
Fifth. First callback has run with setTimeout
```

Figure 9.7: *Macrotasks versus microtasks*

Because we made use of the `queueMicrotask` method, the order in which the tasks got executed differs from how we entered them.

Promise methods

The `Promise` class contains a few static methods in the `Promise` class. They are:

- `Promise.all`
- `Promise.allSettled`
- `any`
- `Promise.race`
- `Promise.resolve/reject`

I have covered resolve and reject here this whole chapter, so let me concentrate on the others for a while.

Promise.all

When you need to run many promises in parallel and wait until all of them are ready, the all method should be used. `Promise.all` takes an array of promises and returns a new promise. This new promise is resolvedonly when all listed promises are settled. The array of their results becomes its result.

For example, the `Promise.all` method below is resolved after 3 seconds, and then its result is gets stored as an `array [1, 2, 3]`:

```
Promise.all([
 new Promise(resolve => setTimeout(() => resolve(1), 3000)), //
 1
 new Promise(resolve => setTimeout(() => resolve(2), 2000)), //
 2
 new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
]).then(alert);
```

Although the first promise takes the longest, it still comes first in the array.

Promise.allSettled

The problem with `Promise.all` is that it rejects if any promise rejects. It needs all the results in order to continue. What if you'd like to fetch information, but if one request fails, you still want the other successful information. This is where `Promise.allSettled` comes in handy. Let's have a look at the next code segment:

```
const prom_1 = Promise.resolve(5);
const prom_2 = new Promise((resolve, reject) =>
setTimeout(reject, 100, 'fail'));
const promArray = [prom_1, prom_2];
Promise.allSettled(promArray).then((results) =>
results.forEach((result) =>
console.log(result.status)));
```

We will get the following output:

```
fulfilled
rejected
```

Even though there was a rejected, it continued to go through all the promises and didn't immediately reject the whole array of results.

Promise.any

`Promise.any()` takes an array of `Promise` objects and, as soon as one of the promises in the array fulfills, returns a single promise that resolves with the value from that promise.

Promise.race

Waits for the first settled promise and gets its result. So whichever promise is fastest to be fulfilled, becomes the result. The others are then ignored.

Conclusion

In this chapter, we learnt about JavaScript Promises. We learned how to create them and the real need for Promises in today's web-oriented world. We learned how to combine multiple microtasks for one promise, thus changing them and we learned about the various methods of the `Promise` JavaScript class.

In the next chapter, we will dives into JavaScript generators and iterators. You will learn what generators and iterators are, how to make use of them properly.

Questions

1. Name 4 `Promise` methods.
2. What is the difference between macrotasks and microtasks?
3. What is the difference between calls backs and Promisification?
4. What is the difference between synchronous and asynchronous tasks?
5. Define the term: JavaScript `Promise`.

CHAPTER 10

JavaScript Generators and Iterators

Introduction

Being able to loop or iterate through lists of information, whether it be a sequence of characters, an array of numbers or names, or custom lists, is vital for any web application. The trick, however, is to know which option to use for which occasion.

This lesson will teach you what generators, iterators, and iterables are with practical exercises and guidance. You will also learn the differences between async iterators and regular iterators, and the same with generators.

Structure

- What are JavaScript generators?
- Advantages of generators
- Iterators
- Iterables
- Asynciterators vs.async generators

Objectives

- Learn what generators are
- Learn what iterators are
- Learn what iterables are
- Understand the advantages of generators
- Compare regular iterators with async iterators
- Compare regular generators with async generators

What are JavaScript generators?

In *Chapter 5: JavaScript Functions*, we introduced you to functions. You have learned that each JavaScript function has a beginning and an end. Once a function starts, it continues until all the lines of code inside it, has executed, whether these lines are macrotasks or microtasks.

Functions can also be exited depending on a certain condition by using the `return` keyword. The `return` keyword exits the function and continues onto the next statement or function call in the script.

Generator functions are similar yet different. Generator functions return a generator object which holds the entire generator iterable that can be iterated using `next()` method or a `for`of loop. You can assign this object to a variable and then keep track of where you are in the current generator function.

Yes, you can also end a generator function with a `return` statement, or the generator function can run until it ends, but you can also include a `yield` statement which allows you to pause the execution of the function temporarily.

Another difference between ordinary functions and generator functions is that whereas a normal JavaScript function can only return one value, a generator function can return multiple values with the help of `yield`. This provides a continuous stream of values that get returned.

Let's do a quick example. The following sample demonstrates the difference between a normal looping function and a generator function:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

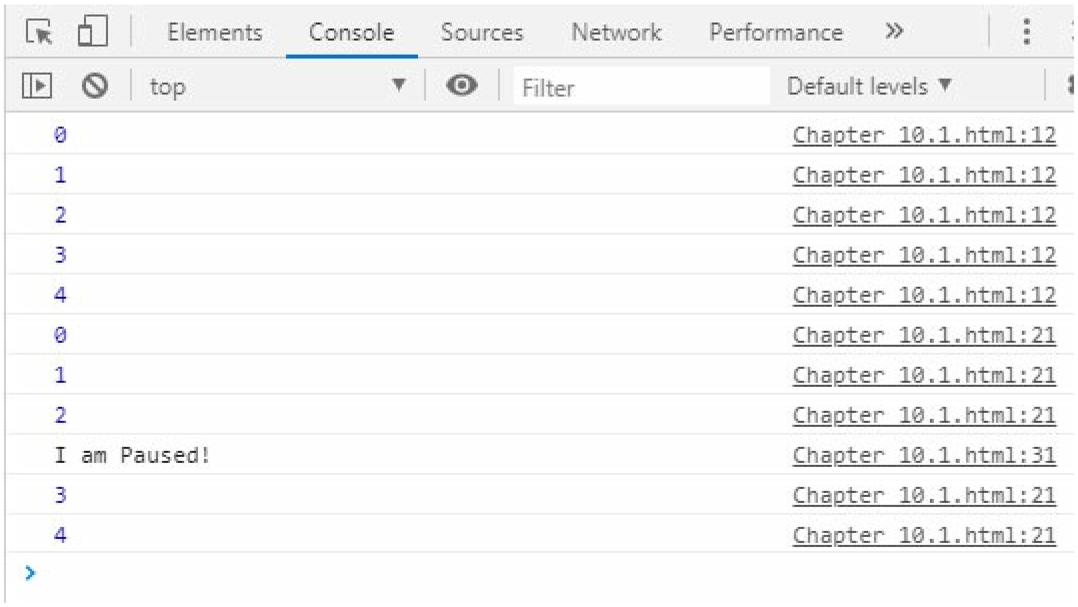
```
<html>
<body>
<h1>Quick Generator Function example</h1>
<p>Quick Generator Function example</p>
<script>
//Normal loop
for (let i = 0; i < 5; i += 1) {
 console.log(i);
}
//Returns immediately 0, 1, 2, 3, 4
//Generator Loop
function * genFunctionForLoop(_number) {
 for (let i = 0; i < _number; i += 1) {
    yield console.log(i);
  }
}
const genLoop = genFunctionForLoop(5);
genLoop.next(); // 0
genLoop.next(); // 1
genLoop.next(); // 2
console.log("I am Paused!");
genLoop.next(); // 3
genLoop.next(); // 4
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 10.1.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

You will see that the initial `for` loop immediately prints `0to5` to the console window, without any interruption. You then created a generator function by identifying it with a `*`. Inside it, you made use of the `yield` keyword to return the next value.

You set the generator function equal to a variable so that you can keep track of each iteration. Notice that I have included a string `I am paused` inside the loop. What happened? The generator function looped, pause after the third iteration, wrote the string: `I am paused` to the console window, and then it continued with the loop.

Your console window should resemble *Figure 10.1*shown below:



```
0    Chapter 10.1.html:12
1    Chapter 10.1.html:12
2    Chapter 10.1.html:12
3    Chapter 10.1.html:12
4    Chapter 10.1.html:12
0    Chapter 10.1.html:21
1    Chapter 10.1.html:21
2    Chapter 10.1.html:21
I am Paused!    Chapter 10.1.html:31
3    Chapter 10.1.html:21
4    Chapter 10.1.html:21
>
```

Figure 10.1: *Example of a generator function*

I mentioned earlier that you need to make use of an asterisk (*) symbol to create a generator function, and I have shown you with the first code segment in this chapter, but here are a few more quick ways to create a generator function:

```
function * generator () {}
function* generator () {}
function *generator () {}
let generator = function * () {}
let generator = function* () {}
let generator = function *() {}
```

With each `next()` call inside a generator function, the `yield` expression returns an object containing two parameters, they are:

- value: The returned value.
- done: It Can the generator be executed further or not? When done returns true, the generator function has finished its run.

A bit more detail on them follows.

Advantages of generators

There are two main advantages of JavaScript Generators, they are:

- Lazy evaluation

- Memory efficiency

Let see them in detail.

Lazy evaluation

This delays the evaluation of an expression until its value is needed. This means that if the value is not needed, it will not exist. The value is calculated on demand.

Let's do a quick example. The following sample demonstrates lazy evaluation:

1. Open a text editor of your choice and enter the following HTML and JavaScript code

    ```
    <html>
    <body>
    <h1>Lazy Evaluation example</h1>
    <p>Lazy Evaluation example</p>
    <script>
    function * genPowerFunction(number, power) {
     let start = number;
     while(true) {
      yield Math.pow(start, power);
      start++;
     }
    }
    const genPower = genPowerFunction(5, 2);
    console.log(genPower.next()); //25 = 5*5
    //console.log(genPower.next()); // 36 = 6*6
    //console.log(genPower.next()); //49 = 7*7
    //console.log(genPower.next()); // 64 = 8*8
    </script>
    </body>
    </html>
    ```

2. Save the file with an HTML extension, for example `Chapter 10.2.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The `genPowerFunction` generator function gives the series of the number raised to a power. For example, a power series of 5 raised to 2 would be 25(5^2), or 6 raised to `2` would be 36 (6^2). When we create the variable object `genPower = genPowerFunction(5, 2)`; we only create the generator object. None of the values has been computed. Now, only if and when we call `genPower.next()`, 25 would be computed and returned.

Memory efficiency

Generators are memory-efficient because of lazy evaluation. Unlike with normal functions, only the needed values are generated, and execution is deferred until needed.

Iterators

A JavaScript iterator is an object that defines a sequence and can return value when it terminates.

An iterator implements the iterator protocol by having a `next()` method which returns an object with the same two properties as a generator. An iterator object can be iterated explicitly by calling `next()`.

They are:

- **value**: The value is the next value in the sequence
- **done**: It is True if the last value in the sequence has already been consumed.

Now for a quick exercise!

The following sample creates a range iterator:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

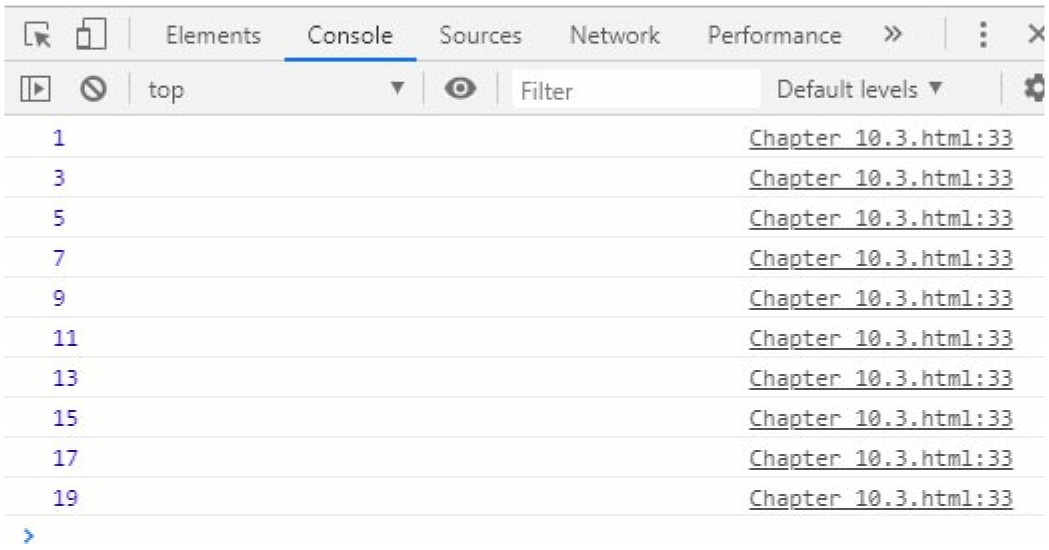
```
<html>
<body>
<h1>Iterator example</h1>
<p>Iterator example</p>
<script>
function iteratorRange(iStart = 0, iEnd = Infinity, iStep =
1) {
  let next = iStart;
  let Count = 0;
```

```
    const iterator = {
    next: function() {
  let result;
  if (next < iEnd) {
       result = { value: next, done: false }
       next += iStep;
       Count++;
       return result;
  }
  return { value: Count, done: true }
  }
  };
  return iterator;
  }
  let iterate = iteratorRange(1, 20, 2);
  let result = iterate.next();
  while (!result.done) {
   console.log(result.value); // 1 3 5 7 9 11 13 15 17 19
   result = iterate.next();
  }
  </script>
  </body>
  </html>
```

2. Save the file with an HTML extension, for example `Chapter 10.3.html`.
3. Double-click the file to open it in the default browser
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The above example creates a simple range iterator that defines a sequence of integers from a starting number to an ending number spaced step apart.

Your screen should resemble *Figure 10.2*shown below:



```
1     Chapter 10.3.html:33
3     Chapter 10.3.html:33
5     Chapter 10.3.html:33
7     Chapter 10.3.html:33
9     Chapter 10.3.html:33
11    Chapter 10.3.html:33
13    Chapter 10.3.html:33
15    Chapter 10.3.html:33
17    Chapter 10.3.html:33
19    Chapter 10.3.html:33
>
```

Figure 10.2: *Range iterator*

The range starts at 1, ends at 20, and does every second number.

Iterables

Any object that defines its iteration behavior is iterable. This means any object that can be looped over is iterable. Built-in types such as the following are iterable:

- Arrays
- TypedArrays
- Strings
- Maps
- Sets
- Arguments
- DOM elements

You can also create your own iterables. Here is a small example.

Now for a quick exercise!The following sample creates a range iterator:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

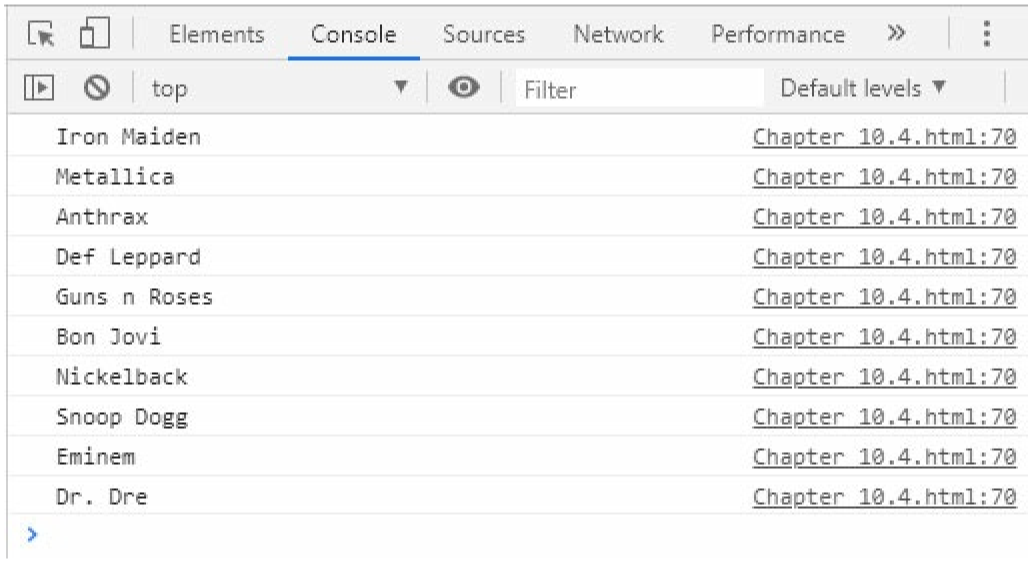
```
<html>
<body>
```

```
<h1>Custom Iterator example</h1>
<p>Custom Iterator example</p>
<script>
const iterableMusicCollection = {
allBands: {
 Metal: [
  ‘Iron Maiden’,
  ‘Metallica’,
  ‘Anthrax’
  ],
  Rock: [
  ‘Def Leppard’,
  ‘Guns n Roses’,
  ‘Bon Jovi’,
  ‘Nickelback’
  ],
 Rap: [
  ‘Snoop Dogg’,
  ‘Eminem’,
  ‘Dr. Dre’
  ],
 },
 [Symbol.iterator]() {
  const genres = Object.values(this.allBands);
  let BandIndex = 0;
  let GenreIndex = 0;
  return {
 next() {
  const bands = genres[GenreIndex];
  const BandsComplete = !(BandIndex < bands.length);
  if (BandsComplete) {
  GenreIndex++;
  BandIndex = 0;
 }
  const GenreComplete = !(GenreIndex < genres.length);
  if (GenreComplete) {
   return {
     value: undefined,
    done: true
   };
 }
 return {
  value: genres[GenreIndex][BandIndex++],
   done: false
  }
}
};
}
};
for (const band of iterableMusicCollection) {
 console.log(band);
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 10.4.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The code looks more complicated than what it actually is. Here is what happens: You create an array containing a list of your favorite bands within specific genres. You then create the iterator Symbol and tells it where it should get its information. You then do a few very important checks. These checks to see if the bands in the list for each genre are complete, and then to see if all the genres have been looped through. This is important so that the iterator knows when to stop; else it would just keep on iterating. Lastly, you print all the bands from each genre into the console.

Your screen should look similar to *Figure 10.3*, shown below:



```
Iron Maiden                                Chapter 10.4.html:70
Metallica                                  Chapter 10.4.html:70
Anthrax                                    Chapter 10.4.html:70
Def Leppard                                Chapter 10.4.html:70
Guns n Roses                               Chapter 10.4.html:70
Bon Jovi                                   Chapter 10.4.html:70
Nickelback                                 Chapter 10.4.html:70
Snoop Dogg                                 Chapter 10.4.html:70
Eminem                                     Chapter 10.4.html:70
Dr. Dre                                    Chapter 10.4.html:70
>
```

Figure 10.3

My music taste might not be in everybody's taste, but you are still welcome to play around with your own favorite musicians.

Async iterators vs.async generators

In *Chapter 9: JavaScript Promises*, I have talked about asynchronous and synchronous tasks and methods. Here is a little refresher.

Let’s assume you start your very own cupcake business. It is relatively small, so you have direct interaction with your clients. They will order a cupcake or two, you will bake the number of cupcakes, and then you will deliver the cupcakes. This is synchronous behavior because it is an ask-and-get scenario.

Imagine further that your cupcakes are really taking off and you get busier, and there are even talks of franchising! You cannot be at different places at the same time. You cannot serve different customers at the same time. You need help. This is where asynchronicity comes in. Asynchronocity allows you to put orders on a waiting list and complete them when and if you can.

In JavaScript, doing a synchronous task means that the browser must halt the rendering of the page in order to complete the execution of JavaScript code. This blocks the page from rendering completely until the code execution completes.

When doing tasks asynchronously, JavaScript code is processed in parallel to the rest of the page content. Asynchronous tasks will not slow down the rest of the page and minimize the impact of loading external JavaScript files on the page rendering process.

Async iterators

Async iterators iterate over asynchronous data on-demand. There are a few syntactic differences between regular iterators and async iterators. A regular iterator looks like the following code segment:

```
let iRange = {
 start: 1,
 finish: 5,
 [Symbol.iterator]() {
  return {
   current: this.start,
   end: this.finish,
   next() {
    if (this.current <= this.end) {
     return { done: false, value: this.current++ };
    } else {
     return { done: true };
```

```
      }
    }
  };
 }
};
for(let val of iRange) {
 alert(val);
}
```

Code segment 1: Regular iterator

The iterator will start at 1 and continue to 5. While the current value is within range, it will increment, and once it is outside of the range, it will notify the iterator that it cannot iterate anymore.

```
Now, let’s have a look at the same example, but as an async
iterator:
let iRange = {
 start: 1,
 finish: 5,
 [Symbol.asyncIterator]() {
  return {
   current: this.start,
   end: this.finish,
   async next() {
    await new Promise(resolve => setTimeout(resolve, 1000));
    if (this.current <= this.end) {
     return { done: false, value: this.current++ };
    } else {
     return { done: true };
    }
   }
  };
 }
};
(async () => {
 for await (let val of iRange) {
  alert(val);
 }
})()
```

Code segment 2: Async iterator

You can perhaps already see the subtle differences between regular iterators and async iterators. In case you cannot, here are a few differences:

- Async iterators make use of `Symbol.asyncIterator` and regular iterators make use of `Symbol.iterator`.
- Async `next()` methods must return promises. Promises were discussed in *Chapter 9: JavaScript Promises*.
- The `await` should be used to loop through async iterable objects.

Let's have a look at async generators next.

Async generators

The main difference between regular generators and async generators is that an async generator's `next()` function returns a promise.

Let's have a quick look:

```
function* quickSequence(begin, end) {
 for (let i = begin; i <= end; i++) {
  yield i;
 }
}
for(let val of quickSequence(1, 5)) {
 alert(val);
}
```

Code segment 3: Regular generator function

The above generator creates a sequence between 1 and 5. Now, let's have a look at the same function with async capabilities added:

```
async function* quickSequence(begin, end) {
 for (let i = begin; i <= end; i++) {
  await new Promise(resolve => setTimeout(resolve, 1000));
  yield i;
 }
}
(async () => {
 let gen = quickSequence(1, 5);
 for await (let val of gen) {
  alert(val);
 }
})();
```

Code sample 4: Async generator function

The async generator returns a promise, and it makes use of `await` to `yield` another number.

Conclusion

This chapter was a bit tougher than the rest! We learned how generators and Iterators work. We also learned what iterables are and why we need iterables and generators. Lastly, we determined the differences between async iterators and regular iterators, as well as generators.

In the next chapter, is the last chapter in The Power of JavaScript section of this book. We will learn about what modules are and how they affect our code clarity.

Questions

1. Explain the term generator.
2. What is the difference between iterators and generators?
3. Name two advantages of generators.
4. Explain the term iterable
5. What does `yield` do?

CHAPTER 11
Modules

Introduction

Modules allow you to move your logic out of your web page logic. We can import functions from modules, and we can export functions from modules, making it very user-friendly.

In this lesson, you will learn what modules are. You will see how easy it is to export and import modular functions. Then we will get a little complicated with a discussion on CORS and what problems it may cause, as well as solutions to CORS errors. Finally, you will learn about bundling your modules.

Structure

- What are the modules?
- Exporting modules
- Importing modules
- CORS
- Dynamic import
- Nomodule
- Bundlers

Objectives

- Create modules
- Export module functions
- Import module functions in JavaScript and HTML
- Know the benefits of modules

- Understand CORS and how to fix CORS problems
- Understand what Bundlers are

What are the modules?

In *Chapter 1: Overview of the Power of JavaScript and Its Purpose*, we discussed the history and origin of the JavaScript language. Fast-forward a decade or two and the language have grown tremendously and the need for it even more.

Where JavaScript was used to provide interactivity and a few scripting tasks, it is now used much more. Simply put: without JavaScript, there is no webpage in 2019.

In older days, when developers needed similar functionality in their pages, they simply copied and pasted the scripts to each page or created JavaScript libraries. Organizing large and common scripts were problematic and difficult to do. Luckily, today there exists a mechanism for organizing large scripts, these are called modules.

Modules

A module is a file containing JavaScript code, allowing you to divide your web application into logical pieces. Modules are self-contained with distinct and specific functionality. This allows developers to remove them or add new ones, without disrupting the whole application.

Benefits of modules

Benefits of using manifests include the following:

- **Reusability**: Having scripts or functions that are reusable saves a lot of time. You do not have to edit the same code at different places; you do not have to make the same changes at different places. The code in a module is reusable - you only have to change it inside the module when needed, and the code can be used at different places easily.
- **Maintainability**: A module is self-contained. Modules lessen dependencies on parts of the script’s code because updating modules are decoupled from the other pieces of code.

- **Avoid namespace pollution**: Variables outside the scope of top-level functions are global. This causes namespace pollution, where unrelated code shares global variables. Modules enable you to avoid this by creating a private space for variables.

Working with modules

Modules can call functions of one module from another one and interchange functionality by using the export and import keywords.

Exporting modules

The export keyword identifies variables and functions you want to expose to other modules. The best way to explain how to export module functions is to do it, so let’s do an exercise!

The following sample creates a small module with two functions that can be exported and used by other modules:

1. Open a text editor of your choice and enter the following JavaScript code:

   ```
   export function sayHello(name) {
    alert(‘Hello, ${name}!’);
   }
   export function sayBye(name) {
    alert(‘Bye, ${name}!’);
   }
   ```

2. Save the file with a `.mjs` extension, for example `greet.mjs`.

By saving the file with an`.mjs` extension, you indicate that there is no other code in it, thus making it a module. The `.mjs` extension helps to keepyour JavaScript modules and JavaScript classes more organized. This module contains simple two functions that simply produces an alert box with aspecific greeting.

The most important thing to note here is the little word: `export`. This allows the function to be used from other modules.

We will continue with this example as we go on with this chapter, so, leave the file as is for now.

Importing modules

The `import` keyword enables you to import functionality from other modules. Let’s see how to import functions from an existing module into another module:

1. Open a text editor of your choice and enter the following JavaScript code:

   ```
   import {sayHello} from ‘./greet.mjs’;
   import {sayBye} from ‘./greet.mjs’;
   alert(sayHello); // function…
   sayHello(‘Ockert’); // Hello, Ockert!
   alert(sayBye); // function…
   sayBye(‘Ockert’); // Bye, Ockert!
   ```

2. Save the file with a `.mjs` extension, for example,`main.mjs`.

As you can see, it is quite easy to import functions from other modules. All you need is to use the `import` keyword to import the desired function from the module in question. So, in the code above, you import the `sayHello` and `sayBye` functions from the module named greet.js

After the functions have been imported, they can be used as ordinary functions. Let’s now see how we can import a function into an HTML web page.

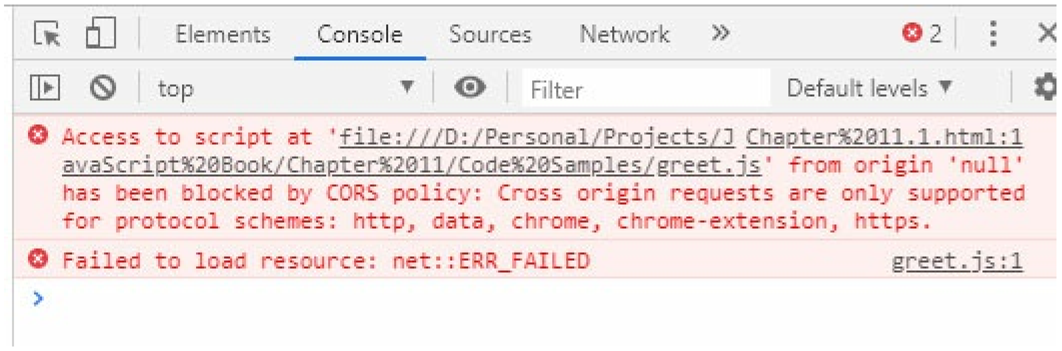
The following sample demonstrates how to import a module into your web pages:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>Import module functions example</h1>
   <p>Import module functions example</p>
   <script type=”module”>
   import {sayHello} from ‘./greet.mjs’;
   import {sayBye} from ‘./greet.mjs’;
   document.body.innerHTML = sayHello(‘Ockert’);
   //document.body.innerHTML = sayBye(‘Ockert’);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 11.1.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

We identify the script type as a module, and then we import the functions. Now, this may or may not work, as shown in *Figure 11.1* below. Why? Because of CORS:



```
Access to script at 'file:///D:/Personal/Projects/JavaScript%20Book/Chapter%2011/Code%20Samples/greet.js' from origin 'null' has been blocked by CORS policy: Cross origin requests are only supported for protocol schemes: http, data, chrome, chrome-extension, https.    Chapter%2011.1.html:1
Failed to load resource: net::ERR_FAILED    greet.js:1
```

Figure 11.1: *Error in the console window*

Another way to import a module is next.

The following sample demonstrates how to import a module into your web pages:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Import module functions example</h1>
<p>Import module functions example</p>
<script type="module" src="greet.mjs"></script>
<script>
document.body.innerHTML = sayHello('Ockert');
document.body.innerHTML = sayBye('Ockert');
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 11.2.html`.
3. Double-click the file to open it in the default browser

4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window

The source of the script file gets set to the module in question. This is a bit quicker and legible. Both web pages may throw errors because of **Cross-Origin Resource Sharing** (**CORS**). Let me explain what it is.

Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (**CORS**) makes use of additional HTTP headers to inform browsers to give the web application access to selected resources from a different origin. This usually occurs when a web application requests a resource that has a different domain or protocol or port, from its own.

CORS adds new HTTP headers to describe which origins are permitted to read information from a web browser. For HTTP request methods other than GET and POST, that could cause side-effects on server data, and browsers should *preflight* the request. This solicits supported methods from the server via the `HTTP OPTIONS` request method.

Here is how it works.

Suppose a user visits the following website: http://www.javascriptforgurus.comand a page on the site attempt a cross-origin request to fetch some data from http://service.javascriptforgurus.com. A CORS-compatible browser will attempt to make a cross-origin request to `service.javascriptforgurus.com` in the following way.

The CORS-compatible browser sends an `OPTIONS` request with an `Origin HTT`P header to http://service.javascriptforgurus.com containing the domain that served the parent page:

http://www.javascriptforgurus.com

The server at http://service.javascriptforgurus.com may respond with any of the following:

- An Access-Control-Allow-Origin header showing which origin sites are allowed. For example:
    - Access-Control-Allow-Origin: http://www.javascriptforgurus.com
- An Access-Control-Allow-Origin header with a wildcard that allows all

domains. For example:

  - Access-Control-Allow-Origin: `*`

- An error page

CORS preflight request

Earlier I mentioned that browsers should *preflight* requests. When performing cross-domain requests, CORS-compatible browsers insert *preflight* requests that determine whether they have permission to perform the action. Here is a small example:

```
OPTIONS /
Host: service.javascriptforgurus.com
Origin: http://www.javascriptforgurus.com
```

If service.javascriptforgurus.com accepts the action, it responds with the following headers:

```
Access-Control-Allow-Origin: http://www.javascriptforgurus.com
Access-Control-Allow-Methods: PUT, DELETE
```

Another example follows. This example demonstrates a client asking a server if it allows `DELETE` requests, before sending the `DELETE` preflight request:

```
OPTIONS http://www.javascriptforgurus.com
Access-Control-Request-Method: DELETE
Access-Control-Request-Headers: origin, x-requested-with
Origin: http://www.javascriptforgurus.com
```

And here follows the server's potential response:

```
HTTP/1.1 204 No Content
Connection: keep-alive
Access-Control-Allow-Origin: http://www.javascriptforgurus.com
Access-Control-Allow-Methods: POST, GET, OPTIONS, DELETE
Access-Control-Max-Age: 86400
```

Because the server responded with an `Access-Control-Allow-Methods` response header, the server allows the request.

We're still sitting with the CORS problem. What do I mean? Well, the exercise that you have done still throws CORS errors, as shown in *Figure 11.1*. How do we fix this?

Well, there are plenty of ways to bypass CORS errors; here are a few ways that might work in Google Chrome:

- Moesif Origin &CORS Changer
- Command Prompt
- Out of blink CORS

Moesif Origin & CORS Changer

The Moesif Origin &CORS Changer Google Chrome plugin enables you to send cross-domain requests and override request origin and CORS headers without receiving errors.

Use the next few steps to install the Moesif Origin & CORS Changer plugin:

1. Navigate to: https://chrome.google.com/webstore/detail/moesif-orign-cors-changer/digfbfaphojjndkpccljibejjbppifbc?hl=en-US
2. Click on the **`Add to Chrome`** button to add this extension to Chrome. This is shown in *Figure 11.2*:

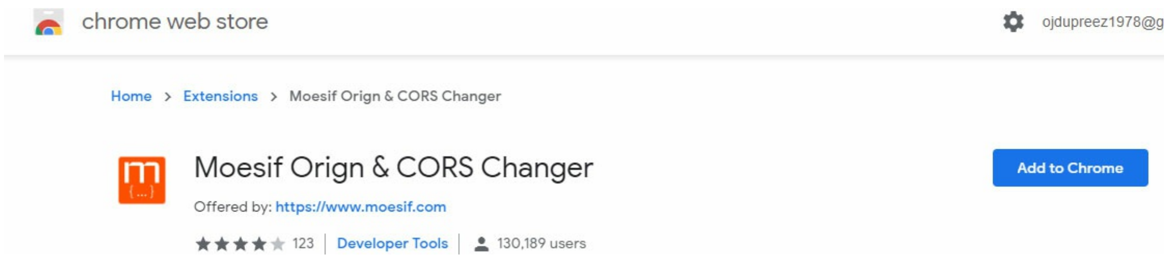
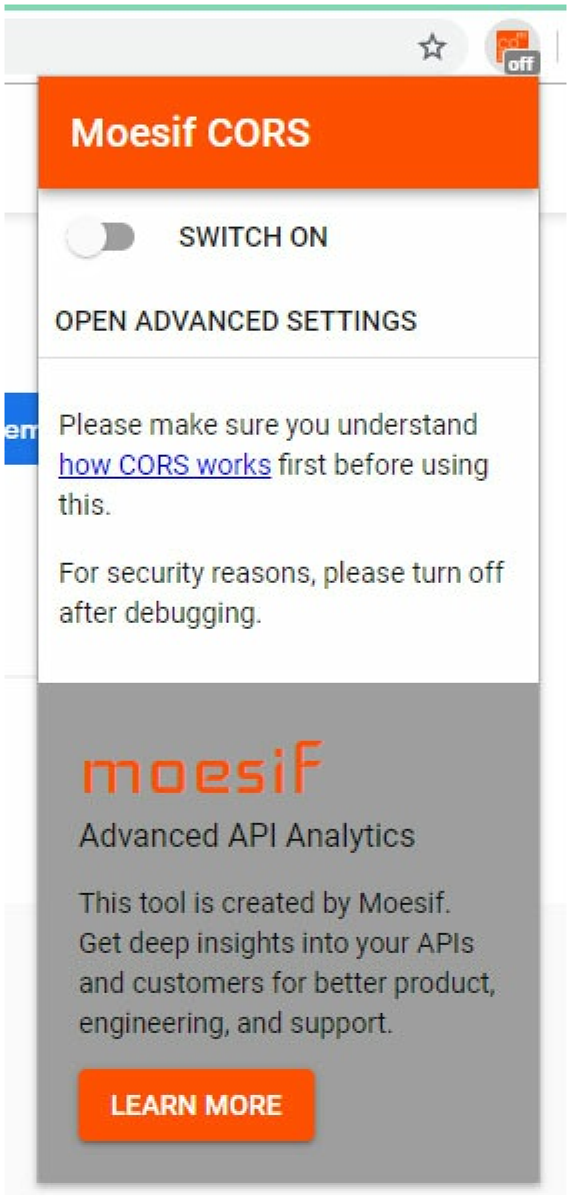

chrome web store
ojdupreez1978@g
Home > Extensions > Moesif Orign & CORS Changer
Moesif Orign & CORS Changer
Offered by: https://www.moesif.com
123 | Developer Tools | 130,189 users
Add to Chrome

Figure 11.2: *Moesif Origin & CORS changer (as found on the chrome web store)*

3. Restart your browser, if necessary.
4. You will notice a small icon next to your address bar. This is the extension. Click on it, the next screen appears.



Please make sure you understand how CORS works first before using this.

For security reasons, please turn off after debugging.

This tool is created by Moesif. Get deep insights into your APIs and customers for better product, engineering, and support.

Figure 11.3: *Moesif CORS settings when clicked upon*

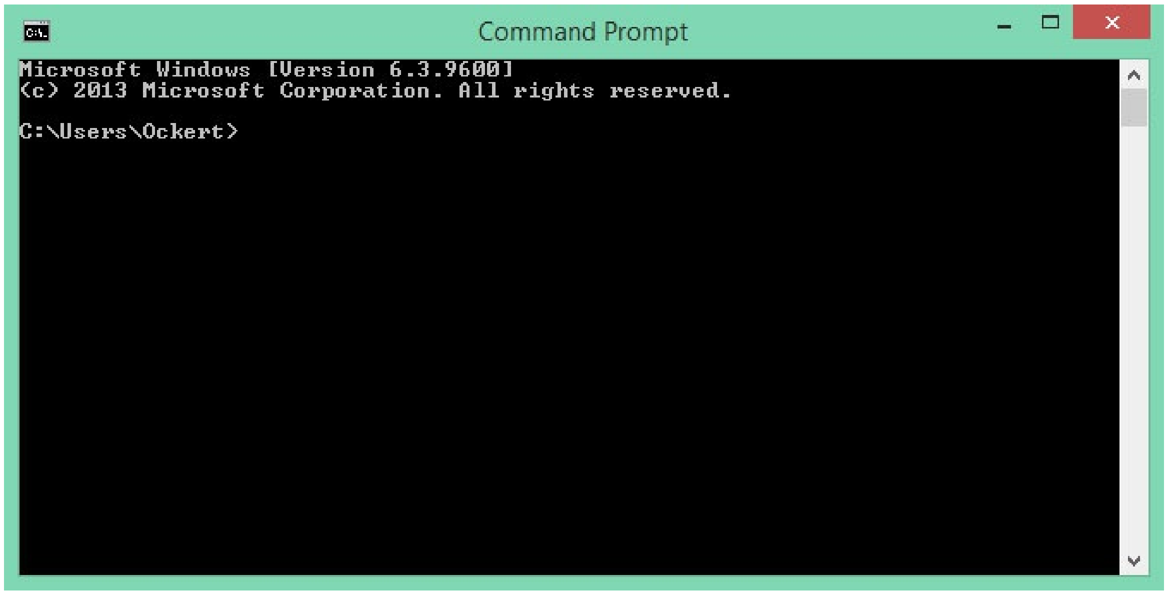
5. Click on the **`SWITCH ON`** button, top switch this extension on.
6. This should do the trick.

Command Prompt

Command Prompt executes entered commands such as automate tasks via scripts and batch files, perform advanced administrative functions, and troubleshoot Windows issues.

The next few steps open Google Chrome in disabled security mode:

1. Close all Google Chrome instances.
2. Press your Windows button on your keyboard.
3. Type `cmd`. Press *Enter*.
4. The Command Prompt window will appear, as shown in *Figure 11.4*:



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Ockert>
```

Figure 11.4: *Command Prompt*

5. Enter the following command, and press enter:

   ```
   CD\Program Files (x86)\Google\Chrome\Application
   ```

   This means change directory (folder) to the location of the Chrome application file.
6. Type in the following command, and press *Enter*. The screen is in *Figure 11.5*:

   ```
   Chrome.exe --disable-web-security
   ```

   OR

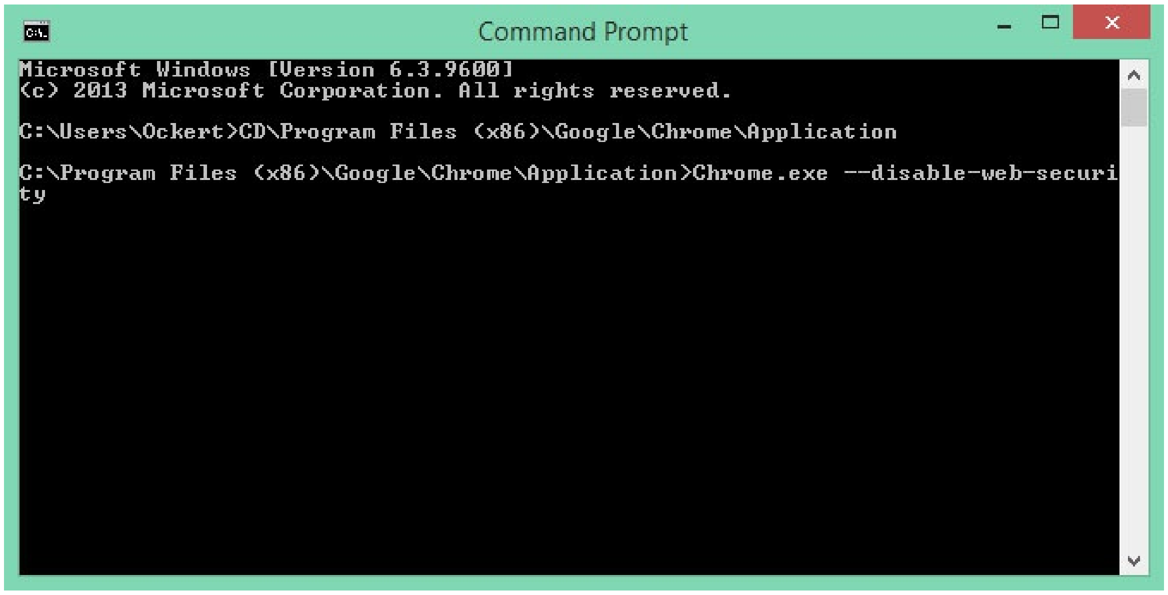
   ```
   chrome.exe --user-data-dir="C://Chrome dev session" --
   disable-web-security
   ```

   OR

   ```
   chrome.exe --disable-web-security --disable-gpu --user-data-
   dir=~C://Chrome dev session
   ```

7. This launches Chrome with disabled security, which may not be a

completely good idea:



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Ockert>CD\Program Files (x86)\Google\Chrome\Application

C:\Program Files (x86)\Google\Chrome\Application>Chrome.exe --disable-web-securi
ty
```

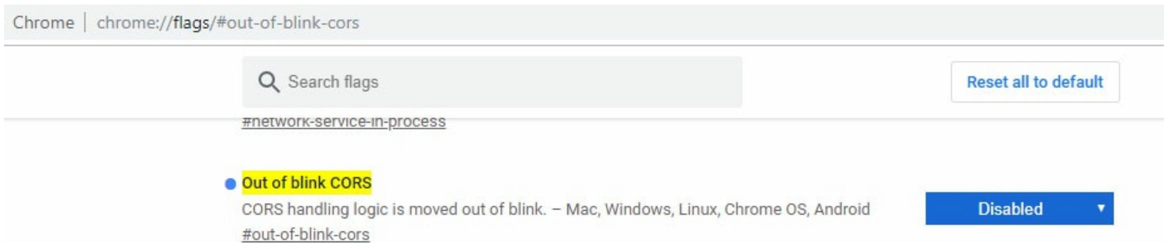
Figure 11.5: *Command Prompt in action*

Now, let us have a look at the Out of Blink CORS setting.

Out of blink CORS

Out of blink, CORS is Chrome setting which you could use to enable or disable CORS checks. Use the next few steps to disable Out of blink CORS:

1. Open your Chrome browser (if it is not already opened)
2. Navigate to: chrome://flags/#out-of-blink-cors
3. Next to the **`Out of blink CORS`** setting, select the dropdown arrow.
4. Select **`Disabled`** from the list, as shown in *Figure 11.6*:


Chrome | chrome://flags/#out-of-blink-cors
Search flags
Reset all to default
#network-service-in-process

Out of blink CORS
CORS handling logic is moved out of blink. – Mac, Windows, Linux, Chrome OS, Android
#out-of-blink-cors
Disabled

Figure 11.6: *Out of blink CORS*

5. Restart Chrome for changes to take effect.

What if I am testing locally and still get the CORS errors?

Sadly, there is not much you can do. You will run into CORS errors because of the JavaScript module security requirements. The only thing you can do then in this situation is to do your testing through a server.

Dynamic import

You can dynamically load or import modules only when needed, instead of having to load everything up front. This increases performance greatly. You call `import()` as a function, then pass the path to the module as a parameter. This returns a promise that fulfils with a module object.

The following sample demonstrates how to import a module dynamically into your web pages:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Dynamic Import example</h1>
<p>Dynamic Import example</p>
<script>
import('greet.mjs')
.then((module) => {
document.body.innerHTML = sayHello('Ockert');
 });
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 11.3.html`.

This imports the functions inside the module dynamically.

Nomodule

With web development, it is always good practice to be backwards compatible or at least be aware of newer browser features compared to older browsers. Older browsers may not understand the module type, so it is a good idea to make use of the nomodule attributes to inform users that whatever they are trying to make use of may not be available.

The following sample demonstrates how the nomodule attribute works.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>nomodule functions example</h1>
   <p>nomodule functions example</p>
   <script type=”module”>
    alert(“Works!”);
   </script>
   <script nomodule>
    alert(“For Older Browsers.”);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 11.4.html`.

Older browsers will ignore the unknown script type, and then display a message for older browsers.

The following sample demonstrates another way to make use of the nomodule attribute:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>nomodule functions example</h1>
   <p>nomodule functions example</p>
   <script type=”module” src=”main.mjs”></script>
   <script nomodule src=”old.js”></script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 11.5.html`.

If a modern browser is used, all the modules can be interpreted and used. However, when an older browser is used, it makes use of an older JavaScript file.

Bundlers

In large websites, it is always easier to make use of bundlers such as webpack, Rollup, or Parcel to bundle your scripts.

Bundlers achieve the following:

- Analyze a module’s dependencies, which include imports and imports of imports.
- Build one singular file with all modules.
- Removes unreachable code.
- Removes unused exports
- Minifies file. This removes spaces and replaces variables with shorter names

I have mentioned the three most popular bundlers (webpack, Rollup, Parcel), let me quickly tell you what they’re about.

Quick notes on Bundlers

- **webpack**: webpack is a static module bundler for modern JavaScript applications. It builds a dependency graph which maps every module your project needs and generates one or more bundles.webpack can be found here:

  https://webpack.js.org/concepts/
- **rollup.js**: Rollup compiles small pieces of code into something larger, such as a library or application. Rollup can be found here:

  https://rollupjs.org/guide/en/
- **parcel.js**: Parcel is a web application bundler that offers fast performance and requires zero configurations. Find Parcel here:

  https://parceljs.org/getting_started.html

Smaller websites may not have to use bundlers but remember: the more imports there are, the more bottlenecks may appear, and the more confusion of where a certain function is or not.

Conclusion

In this chapter, we learned what modules are and why and how you should use them. You had a look at its benefits, and what happens when older browsers do not understand modules. You learned about CORS and some tools available, especially in the developing phase. You have learned about the various bundler tools available.

In the next chapter, we will kick off the *Using JavaScript Productively* section by explaining variables, data types and scope.

Questions

1. Give an example of how to export a module function.
2. Give an example of how to import a module function.
3. Explain what makes dynamic importing so different.
4. What is the nomodule attribute used for?
5. Explain the term CORS.

SECTION III

Using JavaScript Productively

Introduction

Section 3 dives right into the real power of JavaScript. Understanding what makes JavaScript tick is pivotal in creating extremely powerful websites and web applications. In this section, you will learn how to create classes, and then move on to prototypes, properties and promises, finally generators and modules

This section will have the following chapters:

1. Variables
2. Control Flow Statements
3. Loops
4. Code Quality

CHAPTER 12

Variables

Introduction

A variable is a named container for a piece of information. The container is in memory and holds temporary values.

In this lesson, you will learn what variables are. You will learn how to declare variables, and most importantly, you will learn about the various primitive data types in the JavaScript language. This lesson is jam-packed with exercises, so let’s get started!

Structure

- Declaring variables
- Scope
- Primitive types

Objectives

- Learn how to declare variables.
- Understand variable scope
- Create and use the various built-in primitives:
    - String
    - Boolean
    - Number
    - BigInt
    - Symbol
    - Null
    - undefined

Declaring variables

You create variables by using the `var` keyword. This process creates a named memory location ready to store values.

You create a variable the following way:

```
var progStudent;
```

The variable is created and currently contains no value. Later in this chapter, I will introduce you to the undefined primitive type.

We have the variable, now what? We need to store a value inside it. We do so by assigning it a value, as shown next:

```
progStudent = “Ockert du Preez”;
```

In an assignment you specify the variable name on the left-hand side, then use the assignment operator which is the equal sign and then supply the value you’d like to store in that variable. Here is another example of two:

```
var Age;
Age = 41;
var Grades;
Grades = 75.5;
```

OR

```
var progStudent = “Ockert du Preez”, Age = 41, Grades = 75.5;
```

OR

```
var progStudent = “Ockert du Preez”,
Age = 41,
Grades = 75.5;
```

Variables can have any name, but there are some rules that you need to abide by.

Basic rules for naming variables

Some rules for naming objects are as follows:

- Object names can contain letters, digits, underscores, and even dollar signs.
- Object names can begin with a letter or with `$` or `_`.
- Object names are case sensitive, `varname` is different from `varName` and `VarName`.

- JavaScript keywords cannot be used as names.

The last thing I'd like to mention that creating variables is the scope of variables, as explained next.

Variable scope

The scope of a variable means where the variable is known. To explain it simpler. At home, you are known to your close family. At work, you are known to your co-workers. In a strange country, you may not be known at all or known by very few.

JavaScript has two types of scope:

- Local
- Global

Local Scope versus Global scope

A local variable will be visible only within a function where it is defined. A global variable has a global scope which means it can be defined anywhere in your JavaScript code.

Primitive types

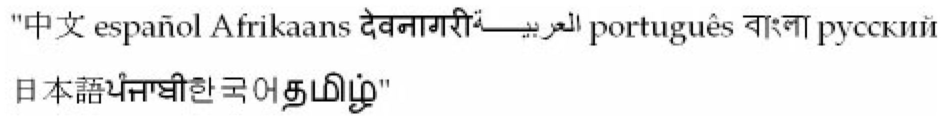
In Chapter 2: JavaScript Objects, I briefly spoke about Primitives. Here is a quick refresher. Primitive values are values that do not have properties or methods. A primitive data type is simply data that has a primitive value. There are actually seven types of primitive data types in JavaScript, and they are:

- string
- number
- BigInt
- null
- undefined
- boolean
- Symbol

Everything else is an object. In *Chapter 2: JavaScript Objects*, I have explained the ins and outs of JavaScript objects. Let's now have a look at all the primitives.

String

The JavaScript string data type represents textual data. Examples of strings include:

- 'Ockert'
- "BpB"
- 
  "中文 español Afrikaans देवनागरी العربية português বাংলা русский 日本語ਪੰਜਾਬੀ한국어தமிழ்"
- "1"

Unlike many other programming languages, there is no separate data type for a single character. JavaScript strings are immutable, which means that once a string is created, it cannot be modified. You can make another string which is based on an operation on the original string with stringfunctions such as, `String.concat()` which joins two strings together and `String.substr()` which allows you to take out part of the string.

I will go into string functions a bit later.

Now, how do I identify or set a string?

Quotes

You will notice at the beginning of this topic that I have placed all the strings inside quotes. But, if you look closer, you will notice that I have made use of two different types of quotes, the single quote and double quote. Another way to identify a string is by making use of backticks.

Single quotes (') and double quotes (") are essentially the same, the only thing to remember is that you cannot start with a single quote and end the string with a double quote for example, or vice versa. Keep some sort of standard as well by making all your strings look the same. It just makes it easier to identify them in future.

Backticks

Backticks (‘)allow you to embed expressions into a string, by enclosing it with `${…}`. We have been doing examples of this throughout this book, but let’s refresh our memory with a small exercise.

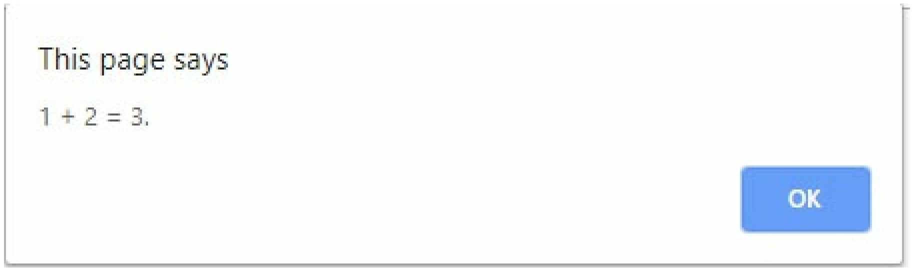
The following sample demonstrates the use of backticks your web pages:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<html>
<body>
<h1>Backticks example</h1>
<p>Backticks example</p>
<script>
 function sum(a, b) {
 return a + b;
}
alert(‘1 + 2 = ${sum(1, 2)}.’); // 1 + 2 = 3.
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 12.1.html`.
3. Double-click the file to open it in the default browser.

Here we simply create a very basic function that adds two objects, and then display it inside an alertbox, as shown in the next screenshot:


This page says

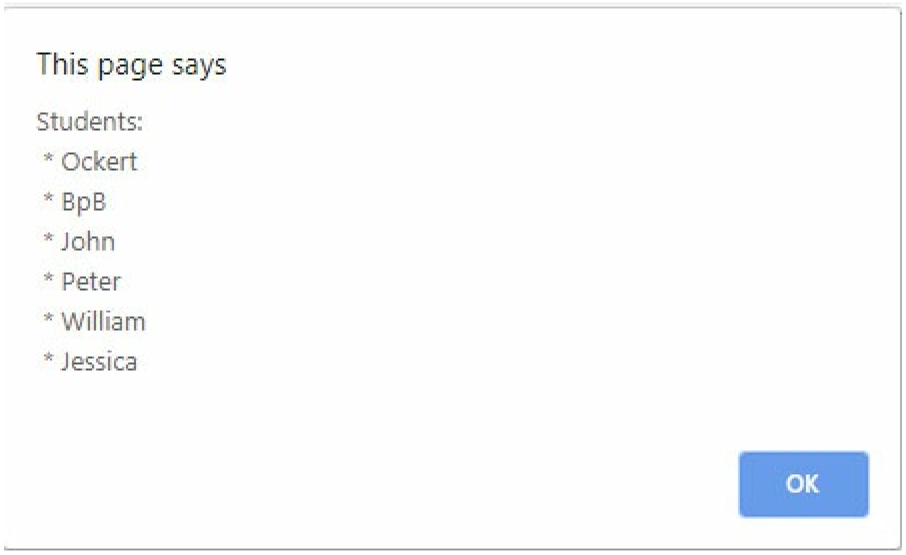
1 + 2 = 3.

Figure 12.1: *Backticks*

One feature I really love about backticks is the ability for one string to span multiple lines. If I were to do the following with a single quote or a double quote, the JavaScript compiler would throw an error.

```
let students = ‘Students:
  * Ockert
  * BpB
  * John
  * Peter
* William
  * Jessica
';
alert(students);
```

Figure 12.2 shows what the above code, when executed, will look like:



```
This page says
Students:
 * Ockert
 * BpB
 * John
 * Peter
 * William
 * Jessica
```

Figure 12.2: *Backticks for multiple line strings*

You can, however, still create multiline strings with single and double-quotes. You do this by inserting the newline character, written as `\n`, inside the string. The next line of code produces the exact same result as the multiline quote surrounded by backticks shown earlier.

```
let students = "Students:\n * Ockert\n * BpB\n * John\n * Peter\n
* William\n* Jessica";
alert(students);
```

The special character `\n` (newline) is inserted in the string at the place or places where the string must be split into a new line.

Speaking of special characters, let's have a more detailed look into the next.

Special characters

A special character is a character that is not a letter, number, symbol, or punctuation mark. *Table 12.1* shows some common special characters:

Character	**Description**
`\n`	New line
`\r`	Carriage return. Usually, a combination of `\n` and `\r` is used to create a new line in text files
`\’`	Single quote
`\”`	Double quote
`\\`	Backslash
`\t`	Tab
`\b, \f, \v`	Backspace, form feed, a vertical tab. Mostly kept for compatibility
`\xXX`	Unicode characters. For example: ‘`\x6F`’ is the same as ‘`o`’
`\uXXXX`	Unicode symbol. For example,`\u00AE` – is the same as the Registered sign `®`.

Table 12.1: *Special characters*

Notice that all special characters must start with the backslash. In this sense, it is known as an escape character.

String properties

A primitive value such as `Ockert du Preez`” is not an object, so it cannot have properties or methods. JavaScript, however, makes methods and properties available to primitive values because these values are treated as objects while executing methods and properties.

The length property returns the number of characters in a given string. The following sample demonstrates the length string property:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>String length Property</h1>
<p>String length Property</p>
<script>
   var str = new String( “Ockert du Preez” );
   document.write(“str.length is:” + str.length);
</script>
</body>
```

```
</html>
```

2. Save the file with an HTML extension, for example `Chapter 12.2.html`.
3. Double-click the file to open it in the default browser.

The answer will be `15` because the length property includes spaces.

String methods

The next table lists the standard string methods:

Method	**Description**
`charAt()`	Gets character at specified index.
`charCodeAt()`	Gets Unicode of the character at a specified index.
`concat()`	Joins two or more strings.
`endsWith()`	Determines if a string ends with a specified substring.
`fromCharCode()`	Converts Unicode to characters.
`includes()`	Determines if a string contains the specified substring.
`indexOf()`	Gets the index of the first occurrence of a specified value in a string.
`lastIndexOf()`	Gets the index of the last occurrence of a specified value in a string.
`localeCompare()`	Compares two strings.
`match()`	Matches a string against a regular expression.
`repeat()`	Repeats a string.
`replace()`	Replaces a string or pattern inside a string with another string.
`search()`	Searches a string against a regular expression.
`slice()`	Extracts a portion of a string.
`split()`	Splits a string into substrings.
`startsWith()`	Determines if a string begins with a specified substring.
`substr()`	Extracts the part of a string.
`substring()`	Extracts the part of a string.
`toLocaleLowerCase()`	Converts a string to lowercase.
`toLocaleUpperCase()`	Converts a string to uppercase.

`toLowerCase()`	Converts a string to lowercase.
`toString()`	Creates a string representing the specified string object.
`toUpperCase()`	Converts a string to uppercase.
`trim()`	Removes whitespace from the start and end of a string.
`valueOf()`	Returns the primitive value.

Table 12.2: *String methods*

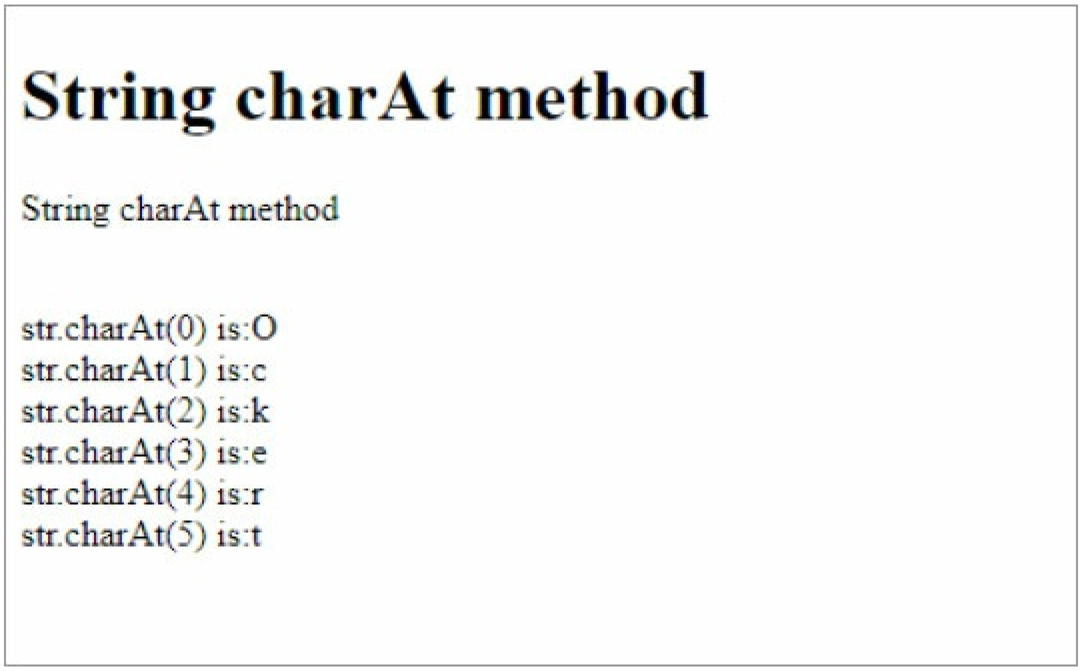
The following sample demonstrates the use of the `charAt` string method:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>String charAt method</h1>
<p>String charAt method</p>
<script>
    var str = new String( “Ockert du Preez” );
    document.writeln(“<br />str.charAt(0) is:” +
    str.charAt(0));
    document.writeln(“<br />str.charAt(1) is:” +
    str.charAt(1));
    document.writeln(“<br />str.charAt(2) is:” +
    str.charAt(2));
    document.writeln(“<br />str.charAt(3) is:” +
    str.charAt(3));
    document.writeln(“<br />str.charAt(4) is:” +
    str.charAt(4));
    document.writeln(“<br />str.charAt(5) is:” +
    str.charAt(5));
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 12.3.html`.
3. Double-click the file to open it in the default browser.

A string gets created with my name and surname. Then I obtain the character at the specified index with the `charAt` method. For instance: 0 is the first character of the string. Your web page should resemble *Figure 12.3*, shown below:



```
String charAt method

String charAt method


str.charAt(0) is:O
str.charAt(1) is:c
str.charAt(2) is:k
str.charAt(3) is:e
str.charAt(4) is:r
str.charAt(5) is:t
```

Figure 12.3: *charAt example*

There are so many methods, and one book cannot explain them all. Let's move on to the next primitive type, Number.

Number

The Number type is a double-precision 64-bit binary value. It can hold floating-point numbers orintegers between -(2^{53} -1) and (2^{53} -1). The number type also has three special values: +Infinity, -Infinity, and NaN (not-a-number). The constants `Number.MAX_VALUE` and `Number.MIN_VALUE` can be used to determine the largest or smallest value within +Infinity and -Infinity. You can also check if a number is in the double-precision floating-point range by using `Number.isSafeInteger()` or `Number.MAX_SAFE_INTEGER` and `Number.MIN_SAFE_INTEGER`.

Number properties

The number ofprimitive types has the following properties:

Property	**Description**
`EPSILON`	Smallest interval between 2 representable numbers
`MAX_SAFE_INTEGER`	Maximum safe integer

`MAX_VALUE`	Largest positive representable number
`MIN_SAFE_INTEGER`	Minimum safe integer
`MIN_VALUE`	The positive number closest to zero
`NaN`	Not a Number
`NEGATIVE_INFINITY`	Negative infinity
`POSITIVE_INFINITY`	Positive infinity

Table 12.3: *Number properties*

Now that we have been acquainted with the properties of the number type let’s have alook at its methods.

Number methods

The number of primitive types has the following methods:

Method	**Description**
`isFinite`	Is the value a finite number?
`isInteger`	Is the value an integer (whole number)?
`isNaN`	Is the value not a number?
`isSafeInteger`	Is the value an integer between -(253 -1) and (253 -1)
`parseFloat`	Parses an argument to a floating-point number
`parseInt`	Parses an argument to an integer number
`prototype.toExponential`	String in Exponential notation
`prototype.toFixed`	Fixed-point notation
`prototype.toLocaleString`	language sensitive representation of the number
`prototype.toPrecision`	Represents a number to a specified precision in fixed-point or exponential notation
`prototype.toSource`	object literal of the Number object
`prototype.toString`	String representation of the number
`prototype.valueOf`	Returns primitive value of object

Table 12.4: *Number methods*

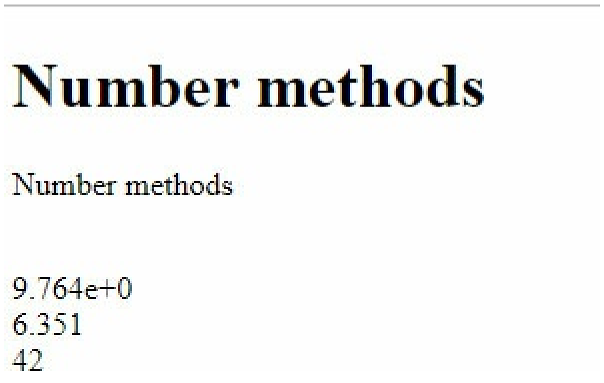
Let’s do a quick exercise!The following sample demonstrates the use of a few numbers ofmethods:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>Number methods</h1>
<p>Number methods</p>
<script>
   var num = 9.76386;
var n = num.toExponential(3);
document.writeln(“<br />” + n);
var x = 6.351;
x.toFixed(2);
document.writeln(“<br />” + x);
var numObj = new Number(42);
var nn = numObj.valueOf();
document.writeln(“<br />” + nn);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 12.4.html`.
3. Double-click the file to open it in the default browser.

As you can see, it is quite easy working with the methods of numbers. In the above exercise, you display the exponential value of a number, get a fixed value of another number and display the value of a third number. *Figure 12.4* shows the result:



```
Number methods

Number methods

9.764e+0
6.351
42
```

Figure 12.4: *Number methods*

Speaking of numbers, let's have a quick peek into the `BigInt` primitive type.

BigInt

You may be wondering about the limits of the number primitive. Just a quick refresher: it is between -(2^{53} -1) and (2^{53} -1). Now, what if you have a larger number? This is where `BigInt` comes in. `BigInt` represents whole numbers larger than (2^{53} -1). It is quite easy to create a `BigInt`, all you need to do is to add an `n` to the integer literal, or by calling the function `BigInt()`.

Let's do a quick exercise!The following sample demonstrates how to create BigInts:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>Creating BigInts</h1>
   <p>Creating BigInts</p>
   <script>
   const num1 = 9007199254740991n;
   const num2= BigInt(9007199254740991);
   const num3 = BigInt("9007199254740991");
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 12.5.html`.
3. Double-click the file to open it in the default browser.

Notice the `n` I appended after the long number when I created the `num1` object. `num2` is created with the `BigInt` function and `num3` as well, but a string is supplied for `num3` which gets converted into a `BigInt` number.

Null versus undefined

Null means nothing. It is the intentional absence of value. Undefined means that the variable has been created, but it hasn't been assigned a value.

A quick example may help; let's see. The following sample demonstrates how to create BigInts:

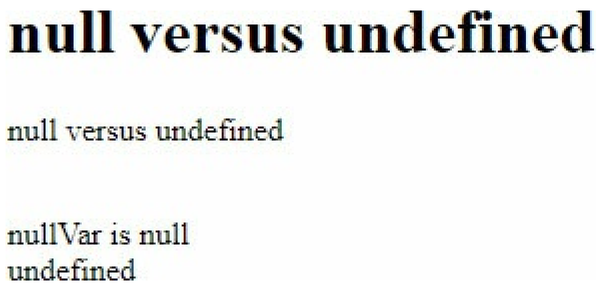
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>null versus undefined</h1>
<p>null versus undefined</p>
<script>
var nullVar = null;
if (nullVar)
document.writeln("<br />nullVar is not null");
else
document.writeln("<br />nullVar is null");
var undefinedVar;
document.writeln("<br />" + undefinedVar);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 12.6.html`.
3. Double-click the file to open it in the default browser.

Here I created a variable named `nullVar` and initialized it to null. This means that there is no value, so the `if` statement checking the value of the `nullVar` variable will produce a written line stating that `nullVar` is null.

I then created another variable named `undefinedVar` and didn't give it a value. The line that will get written will state that `undefinedVar`'s value is undefined. Why? Because the object is created but isn't populated with any value. *Figure 12.5* shows the code in action:



```
null versus undefined

null versus undefined

nullVar is null
undefined
```

Figure 12.5: *Null versus undefined*

Now that we know how to store numeric and string values as well as complicated values such as null and undefined, we can move on to a simpler primitive named Boolean.

Boolean

A Boolean value is a value that can either be true or false. If it is not true, it is false and vice versa. Booleans are quite commonly used in binary options such as male or female, or cold and hot, and so on. Boolean contains no methods and only has a length property.

A quick way to see a Boolean in action is to do another quick exercise.

The following sample demonstrates how to create and use Booleans:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>Boolean example</h1>
   <p>Boolean example</p>
   <script>
   var varYES = true;
   var varNO = false;
   if(varYES)
   {
     alert("This code executes");
   }
   if(varNO)
   {
     alert("This code does not execute");
   }
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 12.7.html`.
3. Double-click the file to open it in the default browser.

As you can see, I have made use of another if statement here, actually two `if` statements. I will go into greater detail on if statements in *Chapter 13: Control Flow Statements*. These if statements check to see what value the passed object contains. If it contains a true value, the code inside its block will be executed, as shown in *Figure 12.6*:

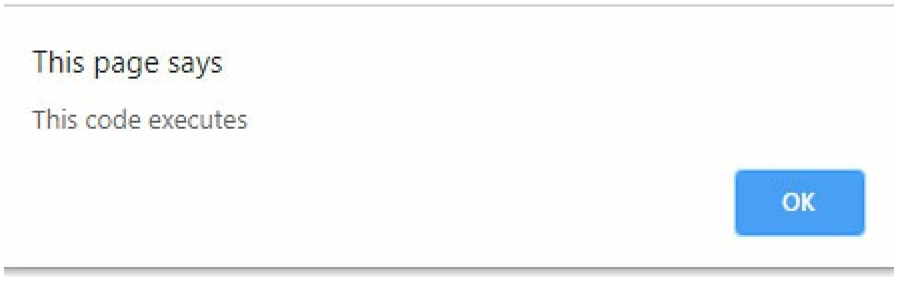

This page says
This code executes

Figure 12.6: *Boolean test*

The last primitive I'd like to talk about is the Symbol primitive

Symbol

A symbol represents a unique identifier, and it is guaranteed to be unique.

Here is a short example. The following sample demonstrates how to create and use Symbols:

1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<!doctype html>
<body>
<h1>Symbol example</h1>
<p>Symbol example</p>
<script>
let student = {
 name: “Ockert”
};
let id = Symbol(“id”);
student[id] = 123;
alert( student[id] );
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 12.8.html`.
3. Double-click the file to open it in the default browser

When run, the IDsymbol is assigned to the student object, and it will produce `123`.

Conclusion

In this chapter, we learntabout data types and variables. Without variables, you cannot store any information or manipulate information. Most importantly, you have learned which data type is correct for which use.

In the next chapter, we willdig into control-flow statements. Although you may have seen them during this book, in the further chapter, we will explain all the ins and outs of how to include logic in your scripts with the use of control-flow stamens.

Questions

1. What is a variable?
2. What is the difference between local and global scope?
3. Name the seven primitive data types
4. Name three-string functions.
5. What is a symbol?

CHAPTER 13

Control Flow Statements

Introduction

Controlflow statements dictate the flow of your script based on conditions or small little tests it performs. These tests can include testing for the time of day, or the user's age, or whether a certain piece of information was supplied.

In this lesson, you will learn what controlflow statements are, and which statements are best suited for your particular needs. You will learn how to make use of the `if` statement, `else…if` statement and `else` to execute code depending on a situation. You will then learn how the `switch` statement and the conditional operator can save you time. Lastly, you will learn how to nest statements

Structure

- Control flow statements definition
- if
- else…if
- else
- switch
- nested statements
- Conditional operator '`?`'

Objectives

- Understand how the control-flow statements work.
- Make use of the `if` statement to test for conditions.
- Make use of the `else…if` statement to test for multiple conditions.
- Make use of the `else` statement when a condition is false.

- Understand how the `switch` statement works.
- Understand how to nest `if` statements and `switch` statements.
- Know how to use the conditional operator.

Control-flow statements definition

Statements get executed one line at a time. The browser's interpreter starts executing statements from top to bottom. This is known as the control flow of the script or program. Very often, the control flow of the script can be changed depending on certain conditions. These conditionals are unknown values or situations that we, as developers need to test for.

A good example will be if the website requires a password to be entered. What happens if there is no text entered, or the password entered is too short? Based on these situations, a different branch of code should be executed. What if the password entered was wrong? Yet another branch of code should get executed. What if the password entered was correct? The branch that then gets executed would allow further user entry into the website.

How do we do this?The answer is simple. Controlflow statements such as `if`, `if…else`, `else`, `switch`, and the conditional operator. Let's have a look at each of them.

if statement

What would you do if you win the lottery? I know what I'd do! I'd pay off all my debt, buy a nice car. Retire and go live somewhere off the grid and in the middle of nowhere with my wife and kid. Bear with me; I'm trying to make a point.

It becomes night; what do you do? You switch on lights, and when it becomes dawn, you start switching off lights. It rains, and you're outside. If you have an umbrella, you take it out and use it. If you don't have an umbrella, you run for shelter.

Notice the little two-letter keyword I keep using in my analogies?

The `if` statement works the same way in a script. It is just a different platform. There are a few things you need to test for namely:

- User interaction

- Certain events or situations in your script
- Computer events.

To explain it better. You may need to check the state of a variable in order to complete a task. A user, for example, cannot continue to the site if certain information hasn't been supplied. Or, you may be busy with a countdown timer, and you need to check if the time has expired continuously. Or, you would need to determine if a certain area on a picture has been clicked upon.

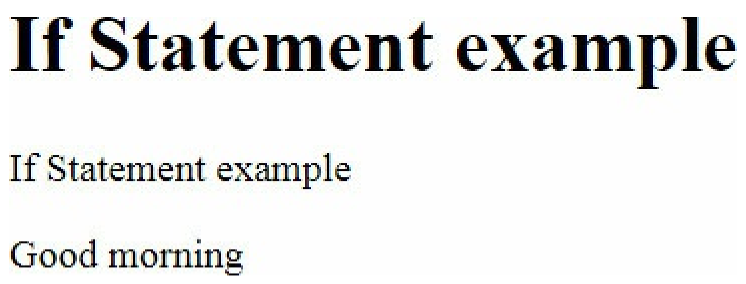
I think you get the picture by now, so let's quickly do an exercise. Exercises demonstrate concepts better. The following sample shows you how to create a basic `if` statement:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>If Statement example</h1>
   <p>If Statement example</p>
   <script>
   var time = new Date().getHours();
   if (time < 12) {
    greeting = “Good morning”;
   }
   document.write(greeting);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 13.1.html`.
3. Double-click the file to open it in the default browser.

This is perhaps the most basic demonstration of an `if` statement in action. A variable named `time` gets created. The `time` is a `Date` object. Because we are making it a date object, we can access `Date` properties such as `getHours`. The `getHours` method gets the current hour of the day. If the current hour is less than 12, then another variable named `greeting` gets populated with the text: `Good morning`. Lastly, the value of the greeting variable gets written to the page. If the time is later than 12, the greeting will be undefined. *Figure 13.1* shows what it will look like when the hour is less than 12:



If Statement example

If Statement example

Good morning

Figure 13.1: *if statement in action*

There may be times when one `if` statement simply isn't enough. In comes the `else…if` statement.

else…if statement

The `else…if` statement is used to check for multiple conditions. If the original condition doesn't exist, the flow jumps to the next `else…if` statement, if that condition doesn't exist, it continues to the next, and so on. If none of the conditions exists, nothing happens, the code in all the `if` and `else…if` statements simply will not be executed.

The following sample adds an `else…if` statement:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>If Statement example</h1>
<p>If Statement example</p>
<script>
var time = new Date().getHours();
if (time < 12) {
 greeting = “Good morning”;
}
else if (time < 18) {
 greeting = “Good afternoon”;
}
document.write(greeting);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 13.2.html`.

3. Double-click the file to open it in the default browser.

I have included an `else…if` statement in the previous exercise. If time is less than 18, then a greeting will contain the message `Good afternoon`.

Earlier I mentioned that when no `else…if`condition exists, nothing will happen. Well, we can make something happen!

else statement

The `else`statement is used to execute when a certain condition doesn't exist. So, to get back to my very first analogy of me winning the lottery. If I do not win the lottery, I must continue to work and cannot retire. In the above two exercises if the time is neither less than 12 nor 18 nothing currently happens.

Let's change that!The following sample adds an `else` statement:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>If Statement example</h1>
   <p>If Statement example</p>
   <script>
   var time = new Date().getHours();
   if (time < 12) {
    greeting = “Good morning”;
   }
   else if (time < 18) {
    greeting = “Good afternoon”;
   }
   else {
    greeting = “Good evening”;
   }
   document.write(greeting);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 13.3.html`.
3. Double-click the file to open it in the default browser.

When the time is neither less than `12`nor `18` the greeting will be `Good` `evening`.

switch statement

In JavaScript, you can make use of a `switch` statement instead of having multiple if and else if statements. The `switch` statement provides a more descriptive way to check a value against multiple variants.

The following sample shows how to create a `switch` statement:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>Switch Statement example</h1>
<p>Switch Statement example</p>
<script>
var objDay = new Date();
switch (objDay.getDay())
 {
  case 0:
console.log(“Sunday”);
break;
 case 1:
console.log(“Monday”);
break;
  case 2:
  console.log(“Tuesday”);
 break;
 case 3:
console.log(“Wednesday”);
break;
  case 4:
console.log(“Thursday”);
break;
case 5:
console.log(“Friday”);
break;
case 6:
console.log(“Saturday”);
break;
default:
console.log(“Choose a proper day value”);
break;
}
```

```
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 13.4.html`.
3. Double-click the file to open it in the default browser.

It looks much more complicated than what it actually is! You create a new `Date` object named `objDay`. Then the `switch` statement starts by testing what day is it currently. It achieves this via the use of the `getDay` date method.

Depending on which day it currently is, that day will get logged to the console window. In my case, it is a `Monday`. As you can see, day `0` is on a `Sunday` and day `6` is on a `Saturday`.

If you have even keener eyes, you will notice the little word break; in between each case statement. This is to indicate that the current branch's code has finished, and the code should not continue through the other cases.

Lastly, the default constructs put in there is a default value if none of the cases is met.

Let's do another example by changing the above one quickly:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>Switch Statement example</h1>
<p>Switch Statement example</p>
<script>
var objDay = new Date();
switch (objDay.getDay())
 {
 case 0:
 case 5:
 case 6:
console.log(“Camping Weekend!”);
 break;
 case 1:
 case 2:
  case 4:
   console.log(“Work”);
 break;
case 3:
 console.log(“Off”);
break;
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 13.5.html`.
3. Double-click the file to open it in the default browser.

There are some subtle differences. You may notice that case `0` and `5` do not end with a break statement. Why? Because case `0`, `5` and `6` all do the same thing, which is `Camping weekend`. This is called grouping and can save a lot of time instead of having to have repetitive code. The same holds true for case 1, 2 and 4 where I will be at work. Case 3 (`Wednesday`) I will be off, for example.

Now that we understand the basics of these statements, let’s get a bit more complicated!

Nested statements

Something that is nested means that it is placed in a similar thing. OK. That doesn’t tell me much! A nested `if` statement is an `if` statement within another `if` statement. A nested `switch` statement is a switch statement that is inside another `switch` statement.

Let’s see a nested `if` statement in action:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>

<body>
<h1>Nested If Statement example</h1>
<p>nested If Statement example</p>
<script>
 var studentAge = 15;
 if( studentAge < 18 )
 {
  document.write(“Under the age of 18!<br>”);
  document.write(“Needs parent’s consent to enroll. “ );
 }
 else
 {
  if (studentAge >= 18 && studentAge <= 60 )
  {
    document.write("Eligible to enroll<br>");
document.write("Please continue to application page." );
}
else
{
document.write("Older than most students here.<br>");
document.write("Would you prefer a softer chair and
different time slot?" );
}
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 13.6.html`.
3. Double-click the file to open it in the default browser.

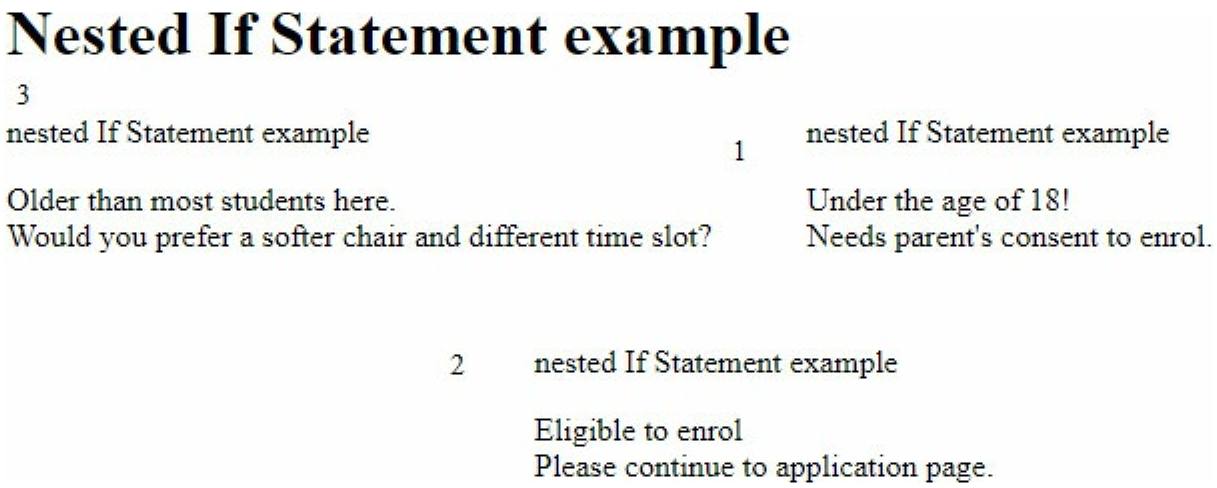
As you can see, including whitespace in your code helps to see the flow of the script. A variable named `studentAge` gets created and assigned a value of `15`. The outer if statement tests if `studentAge` is less than `18`. If the `studentAge` is less than `18` (which it is currently at `15`), then the page will inform the user that the age is under `18` and parents would need to give the student consent to enrol.

Here is where the fun starts.

In the `else` statement (meaning the `studentAge` variable's value is `18` or higher) I have put another `if` statement. This is nested. The nested if statement checks to see if `studentAge` is `18` or higher and smaller or equal to `60`. If both these conditions are met, i.e. `studentAge` is `18` or higher and less than or equal to `60`, the page will notify the user that he or she is eligible to enrol.

If `studentAge` is higher than `60`, the nested `if` statement's `else` clause kicks in and informs the user that he or she may be older than the rest and asks if a softer chair will be needed.

All possible results are shown below in *Figure 13.2*:


Nested If Statement example

3

nested If Statement example

Older than most students here.
Would you prefer a softer chair and different time slot?

1

nested If Statement example

Under the age of 18!
Needs parent's consent to enrol.

2

nested If Statement example

Eligible to enrol
Please continue to application page.

Figure 13. 2: *Nested if statement results*

Let’s go crazy and look at the nested `switch` statement, next

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>Nested Switch Statement example</h1>
<p>Nested Switch Statement example</p>
<script>
var Stops = 30;
var Location = ‘South Africa’;
switch(Location)
{
 case ‘South Africa’:
 switch(Stops)
 {
  case 10:
document.write(‘Visiting 10 places in 10 days’);
break;
 case 20:
 document.write(‘Visiting 20 places in 20 days’);
 break;
case 30:
  document.write(‘Visiting 30 places in 30 days’);
   break;
 }
 break;
case ‘India’:
switch(Stops)
 {
  case 5:
   document.write(‘Visiting 5 places in 5 days’);
   break;
  case 15:
   document.write(‘Visiting 15 places in 15 days’);
  break;
   case 25:
  document.write(‘Visiting 25 places in 25 days’);
 break;
}
 break;
 case ‘America’:
switch(Stops)
 {
  case 7:
 document.write(‘Visiting 7 places in 7 days’);
   break;
 case 14:
  document.write(‘Visiting 14 places in 14 days’);
  break;
 case 21:
   document.write(‘Visiting 21 places in 21 days’);
   break;
 }
 break;
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 13.7.html`.
3. Double-click the file to open it in the default browser.

A lot of typing…

What happens here is that it is a mini-tourist site, where potential tourists can choose which country to visit along with the number of places to visit in the selected country.

There are three countries: `America`, `India` and `South Africa`. Each with their unique amount of Stops at each place. So, when choosing South Africa, you have the option of 10, 20 and 30 stops, whereas if you choose India, you will have 5, 15, and 25 stops.

Conditional operator ‘?’

In Chapter 4: JavaScript Operators, I spoke about the ternary operator. The ternary operator is also known as a conditional operator. The operator is ternary because it accepts three operands in order to function.

The three operands accepted by the ternary operator is:

- The condition
- True expression
- False expression

A small example of what it looks like follows:

```
Condition ? Value if true : Value if false
```

The condition is what gets tested. When the condition is true, the code after the question mark executes, if the condition is false, the code after the colon is executed.

Let’s see a nested if statement in action:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>Conditional statement example</h1>
   <p>Conditional statement example</p>
   <script>
    var studentAge = 19;
    var greeting = (studentAge < 18) ? “Not Allowed” :
    “Welcome!”;
     document.write(greeting);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 13.8.html`.
3. Double-click the file to open it in the default browser.

The value of the `studentAge` variable is tested and thus becomes the condition. If `studentAge` is less than `18`, `Not Allowed` will be printed on the page, else, as is the current case because `studentAge` is `19`, `Welcome!` will be printed on the web page.

Conclusion

In this chapter, we had lots of practical exercises demonstrating the use of all the various controlflow statements. You learned that most of a script just depends on the logic and the conditions that need to be tested for. You saw the differences between the `switch` statement and the `if` statements and the conditional operator.

In the next chapter, we will learnabout the various loops in JavaScript.

Questions

1. Why would you use a `switch` statement instead of an `else…if` statement?
2. What is a ternary operator?
3. Why would you use the `else` clause?
4. Explain the term condition.

CHAPTER 14

Loops

Introduction

You simply cannot avoid loops in any programming environment or language. JavaScript has a nice variety of loops, including counter loops and conditional loops. JavaScript also includes features to exit loops early or to skip iterations.

In this lesson, you will learn what loops exist in JavaScript and their differences in syntax and inner workings. You will do a lot of practical exercises because it is always easier to learn practically.

Structure

- JavaScript loops
- for
- for…in
- for…of
- while
- do…while
- Exiting loops
    - The break statement
    - break with labels
- The continue statement

Objectives

- Identify the different loops in JavaScript.
- Make use of the `for` loop to loop a certain number of times.

- Make use of the `for…in`loop to loop through object properties.
- Make use of the `for…of`loop through arrays, strings, Maps and NodeLists.
- Make use of the `while` loop to loop based on certain conditions.
- Understand the difference between the while and `do…while` statements.
- Learn how to exit loops.
- Learn how to skip iterations in loops.

JavaScript loops

A loop by definition is something that repeats. In the context of JavaScript, a loop is a block of code that repeats until a certain condition becomes true or while a certain condition exists. There are also loops that can iterate over similar items (in *Chapter 10: JavaScript Generators and Iterators*, I explained iterables and iteration) and loops that execute a code block a specified number of times.

The following loops are available in JavaScript:

- for
  - for…in
  - for…of
- while
  - do…while

Let’s have a look at them one by one.

for

The `for` loop executes a block of code a given number of times. Its syntax is as follows:

```
for (statement1; statement2; statement3)
{
 // code block that will be executed
}
```

The loop is broken up in three parts. The first part is the initialization of the loop. This sets the starting point of the loop. The second part is where the condition comes in. here you supply the condition that needs to be evaluated in order for the loop to execute. The third and final part is the incrementation or decrementation of the loop and gets executed with each iteration of the loop.

Here is a more practical example. The next sample shows a `for` loop in action:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>for loop example</h1>
   <p>for loop example</p>
   <script>
   document.write(“Forward:</br>”);
   for (i = 0; i < 10; i++)
   {
   document.write(i);
   }
   </script>
   </body>
   </html>
   ```

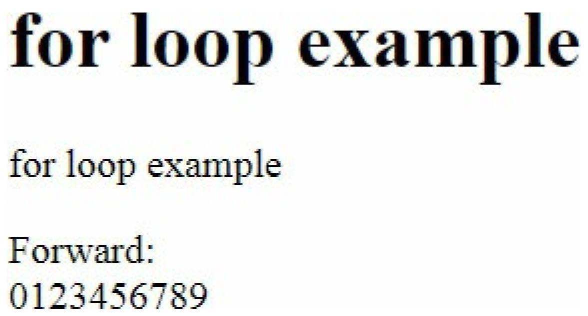
2. Save the file with an HTML extension, for example `Chapter 14.1.html.`
3. Double-click the file to open it in the default browser.

In this exercise you printed the word `Forward` on the web page, then I created a `for`a loop. The variable `i` is 0, so this means that the loop counter will start counting at `0`. The condition is next. The condition simply states that `i` is less than `10`. This means that `i` is anywhere between `0` and `9`. Not at the same time, though. The loop will loop `10` times, from `0` to `9`, thus repeating its code block. The last part of the `for` loop increments the i variable. The first time `i` is equal to `0`, the next time `i` is `1`, then `i` is `2` and so on.

Now, this is where it gets interesting!

The code block that is executed is writing the current value of `i` to the page. The first time `i` is `0`, so it will print `0`, then `i` is `1`, then `2` and so on, until `i` becomes greater than or equal to `10`.

The result of this loop looks like *Figure 14.1*, as shown below:


for loop example

for loop example

Forward:
0123456789

Figure 14.1: *Incrementing for loop*

Let’s loop backwards!

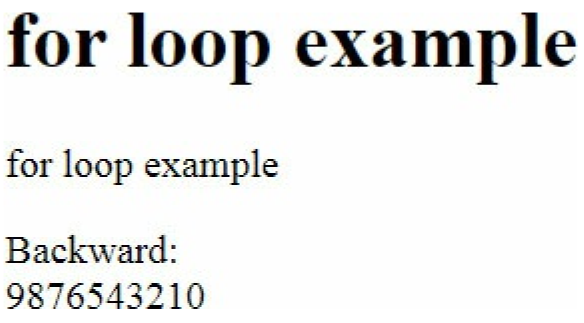
Here is a more practical example. The next sample shows a `for` loop looping backwards.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>for loop example</h1>
   <p>for loop example</p>
   <script>
   document.write(“Backward:</br>”);
   for (i = 9; i > -1; i--)
   {
   document.write(i);
   }
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 14.2.html`.
3. Double-click the file to open it in the default browser.

The `for` loops starts at `9`. Then you test to make sure that `i` is greater than minus `1`, lastly the loop decrements with the decrement operator. Operators were discussed in *Chapter 4: JavaScript Operators*. The end result looks like the next screenshot:



for loop example

for loop example

Backward:
9876543210

***Figure 14.2:** Backward for loop*

The `for` loops are also quite commonly used in conjunction with arrays. Arrays will be discussed in *Chapter 16: Arrays*, so more on that a bit later. The `for` loop has a few variants, let’s examine them.

for…in

The `for…in`statement loops through the properties of an object. Its syntax is as follows:

```
for (variable in object)
{
Statement(s)
}
```

As you can see, this `for` loop looks quite different than the one we discussed earlier. A variable in this context is a temporary place holder for the current property. The object in this context is the object which gets looped over. Let’s do a small exercise

Here is a more practical example. The next sample shows a `for…in`loop in action.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

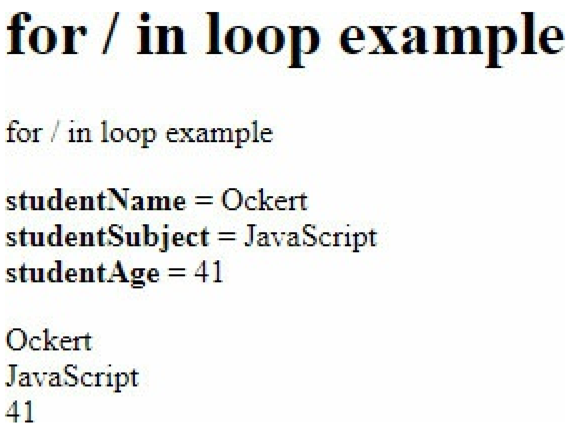
```
<!doctype html>
<body>
<h1>for…in loop example</h1>
<p>for…in loop example</p>
<script>
var student = {studentName: “Ockert”, studentSubject:
“JavaScript”, studentAge: 41};
var output = “”;
var stu;
for (stu in student)
```

```
{
output += student[stu] + “</br>”;
document.write(“<b>” + ‘${stu}’ + “</b> = “ +
‘${student[stu]}’ + “</br>”);
}
document.write(“<p>” + output);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 14.3.html`.
3. Double-click the file to open it in the default browser.

This exercise has two examples in one. Let’s break it down;an object named `student` is created. Its properties include `studentName`, `studentSubject`, and `studentAge`. We created two variables named `output` and `stu`. Then, the `for…in`loop begins. The `stu` variable acts as a placeholder for the properties inside the `student` object. This loop will loop three times because the student object has three properties. The output variable gets the current property value added to it. The += operator means to add the new value to the existing value, as explained in *Chapter 4: JavaScript Operators*. The next line writes in bold the property name, and its setting with the use of backticks, as explained in *Chapter 12: Variables*. Finally, the value of the output variable gets written to the page as well

The final output looks like *Figure 14.3*:


```
for / in loop example

for / in loop example

studentName = Ockert
studentSubject = JavaScript
studentAge = 41

Ockert
JavaScript
41
```

Figure 14.3: *for…in loop*

The next and last `for`the statement I’d like to talk about is the `for…of` statement.

for…of

The `for…of` loops over iterable data structures such as Arrays, Strings, NodeLists and Maps. Its syntax is as follows:

```
for (variable of iterable)
{
 Statement(s)
}
```

It looks similar to the `for…in`loop, except for the fact that it loops through something else. It loops through an iterable. In *Chapter 10: JavaScript Generators and Iterators* you have learned that an iterable is a thing which can be repeated, so in this case, the variable loops through each part of the iterable until it is done. To give a better explanation, let's do an exercise!

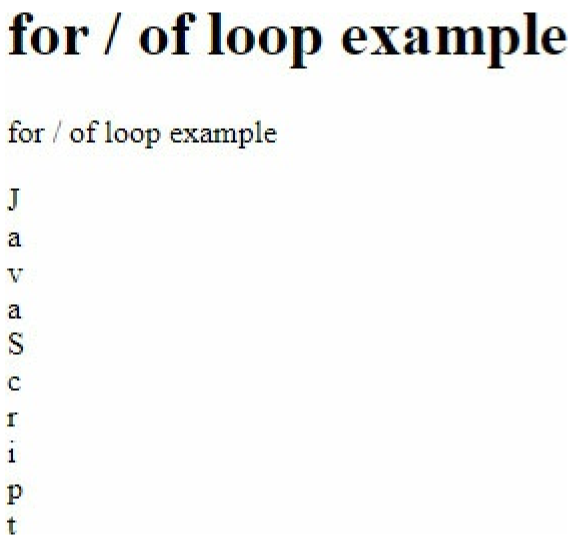
The next sample shows a `for…of`loop in action.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>for…of loop example</h1>
   <p>for…of loop example</p>
   <script>
   var subject = 'JavaScript';
   var s;
   for (s of subject)
   {
   document.write(s + "<br >");
   }
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 14.4.html`.
3. Double-click the file to open it in the default browser.

First, a variable named `subject` is created and gets assigned a value of `JavaScript`. Another variable named `s` is created. The `for…of`loop starts. The variable `s` is equal to a character in the string `JavaScript` which is stored inside the `subject` variable. It gets written to the page, as shown in *Figure 14.4*:



```
for / of loop example

for / of loop example

J
a
v
a
S
c
r
i
p
t
```

Figure 14.4: *for…of loop in action*

Let’s do another example. This example will loop through an array which will be covered in greater detail in Chapter 16: Arrays.

The next sample shows a `for…of`loop looping through an array.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

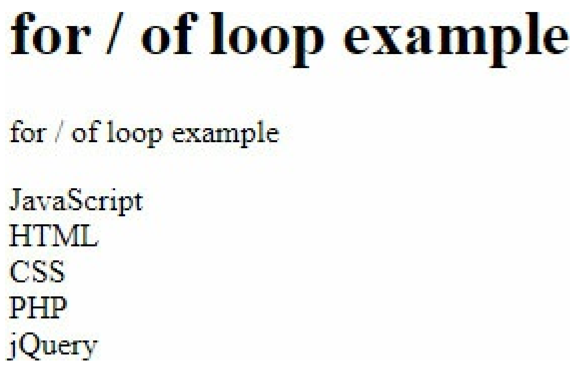
```
<!doctype html>
<body>
<h1>for…of loop example</h1>
<p>for…of loop example</p>
<script>
var arraySubjects = [‘JavaScript’, ‘HTML’, ‘CSS’, ‘PHP’,
‘jQuery’];
var sub;
for (sub of arraySubjects) {
 document.write(sub + “<br >”);
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 14.5.html`.
3. Double-click the file to open it in the default browser.

The `arraySubjects` is what is known as an array. An array stored more than one value, so in a sense, anarray is a group of variables with the same name.

The array gets supplied with five strings, and all five strings are now stored in one place, the array.

A variable sub gets created. As with the `for…in`example, and the previous `for…of`example, this variable will be used to obtain the current value in the array (or group of values). Then, we simply write the value on the page, as shown in *Figure 14.5*:



```
for / of loop example

for / of loop example

JavaScript
HTML
CSS
PHP
jQuery
```

Figure 14.5: *for…of loop looping through the array*

Let’s move on to conditional loops.

while

The `while` loop executes a block of code as long as the given condition is true. Its syntax is as follows:

```
while (condition)
{
Statements(s)
}
```

This is simple. The condition can be testing if a value is false, or if a numeric value is less than 100, for example. While the condition exists, the statements would repeat. This called a conditional loop because unlike a `for` loop where you specify the start and end parameters, with a `while` loop, you may not know how many times it will execute.

Let’s do a proper example.

The next sample demonstrates the power of a conditional loop:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>while loop example</h1>
<p>while loop example</p>
<script>
let total = 0;
while (total < 100) {
 let newVal = + prompt("Enter a number", '');
 total += newVal;
}
alert( 'Total: ' + total );
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 14.6.html`.
3. Double-click the file to open it in the default browser.

In the above exercise, you create a variable named `total` and give it a value of `0`. The `while` loop checks to see if the `total` variable is less than `100`. This means that if the `total` variable becomes `100` or more, the loop should exit.

Inside the loop, another variable named `newVal` accepts the value of whatever was entered in the prompt box, and then this gets added to the `total` variable.

Very important here is that the variable `total` is the one whose value gets tested, if the value doesn't change somewhere in the loop, the loop will be infinite and will never end because total will never become `100` or more for example.

When this page is opened, you will see a box that looks like *Figure 14.6*, where the numbers should be entered. Now, you can enter `1` every time, and it will keep adding 1 until it reaches `100`. Or, you can enter `50` and `50`, or even `100` directly.

If you had entered `1` every time it wouldloop `100` times;if you entered `50` and `50` again, it wouldloop twice, if you enter `100` immediately, it willloop just once. To test this, refresh the page after you have reached `100` and try again with a different combination of numbers:

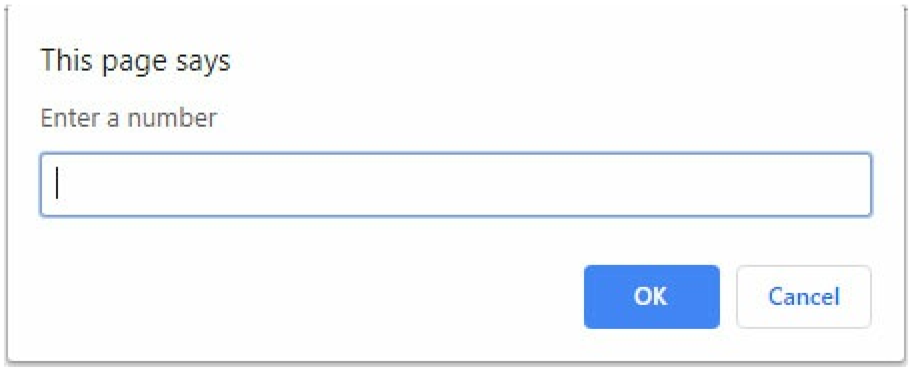


Figure 14.6: *while loop in action*

The `do…while`loop (which I will cover next) is almost identical to the `while` loop, let’s have a look

do…while

The `do…while`loop is also a conditional loop as the while loop you did earlier. There is just one main difference. Whereas with the while loop the condition was tested at the beginning of the loop, with a `do…while`loop, the condition is tested at the end of the loop. This means that the loop will always execute at least once. Its syntax looks like the following:

```
do
{
 Statement(s)
} while (condition);
```

The next sample demonstrates a `do…while`loop.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <!doctype html>
   <body>
   <h1>while loop example</h1>
   <p>while loop example</p>
   <script>
   let i = 0; //change to 6
   do
   {
   i += 1;
   document.write(i);
   } while (i < 5);
   </script>
   ```

```
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 14.7.html`.
3. Double-click the file to open it in the default browser.

This code will simply output `12345` of the page when run. Now since the condition gets evaluated at the end of the loop, this can be troublesome. For instance, if you were to change the value of `1`to `6` instead of `0` before the loop, the loop will still execute once and produce `7`.

Exiting loops

There may be times that you need to exit a loop early. In these cases, you should use the break statement. There are two ways in which you can use the break statement in loops.

- break
- break to label

Let’s see how they work.

break

You can exit a loop by including the `break` statement at the point where you want to exit. This is usually based on a condition that has become true.

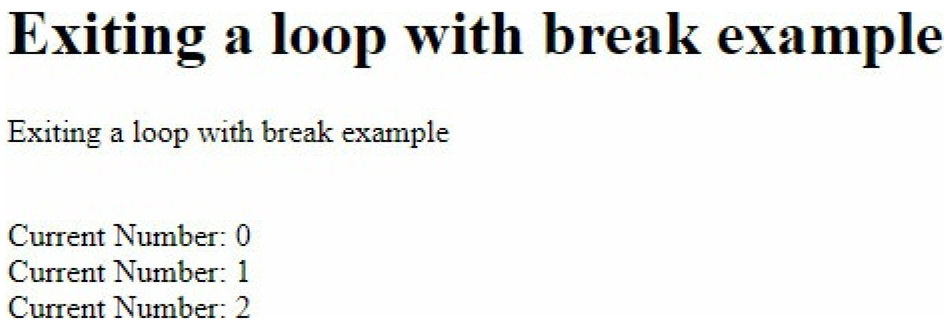
The next sample demonstrates how to exit a loop early.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>Exiting a loop with break example</h1>
<p>Exiting a loop with break example</p>
<script>
var num = “”;
var i = 0;
while (i < 5)
{
num += “<br>Current Number: “ + i;
i++;
if (i === 3)
{
break;
}
}
document.write(num);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example,`Chapter 14.8.html`
3. Double-click the file to open it in the default browser

The loop is supposed to loop from 0 to 5, but the `if` statement tested if `i` is `3`. If it is, then the loop should exit denoted by the `break` statement. *Figure 14.7* displays the half-completed loop.



```
Exiting a loop with break example

Exiting a loop with break example

Current Number: 0
Current Number: 1
Current Number: 2
```

Figure 14.7: *The loop exited early*

Let's add a twist.

break with label

As you saw, the `break` statement causes the loop to exit early. You can specify where to start executing code by including a label and exiting to that label. You simply add a labelled statement and add the code afterwards

Let's do an example.

The next sample demonstrates how to exit a loop early with a label.

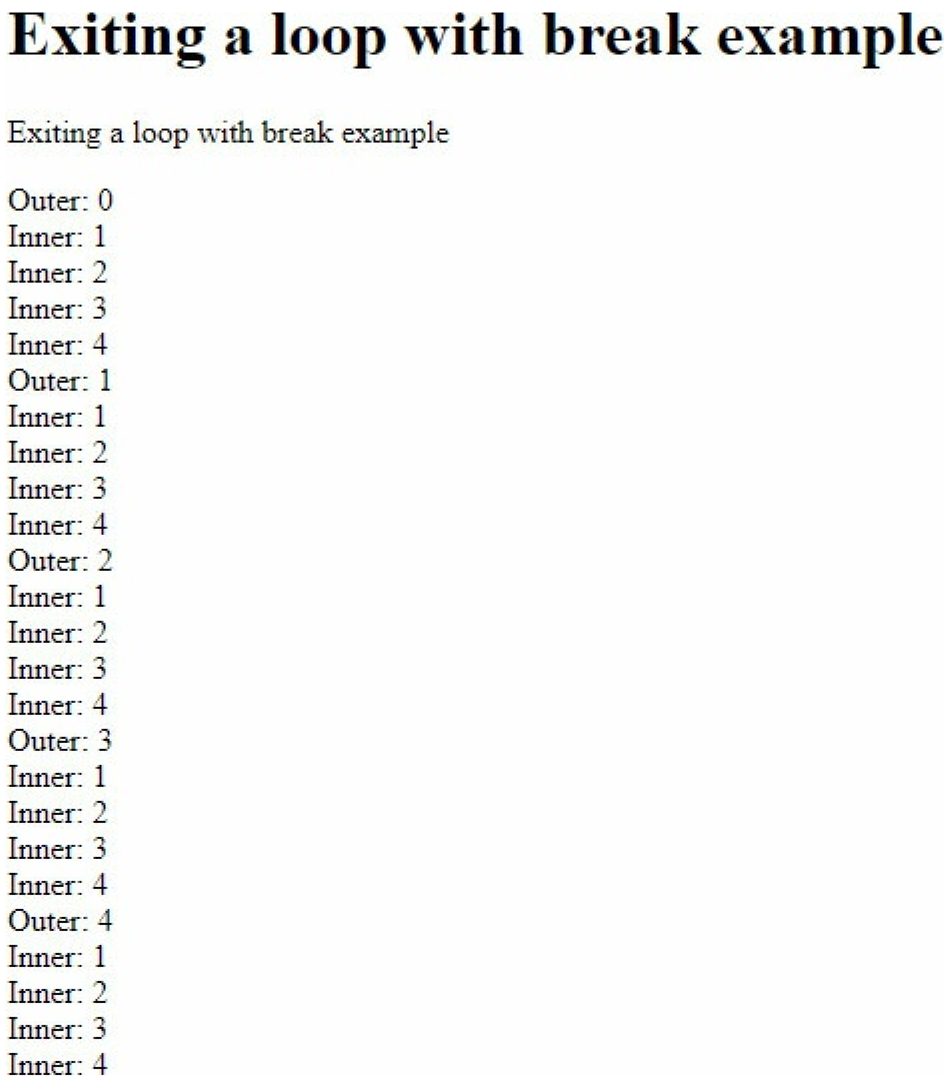
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
```

```
<body>
<h1>Exiting a loop with break example</h1>
<p>Exiting a loop with break example</p>
<script>
let i = 0;
let j = 0;
Repeat: while (true)
{
document.write(‘Outer: ‘ + i + ‘</br>’);
i += 1;
j = 1;
while (true)
{
document.write(‘Inner: ‘ + j + ‘</br>’);
j += 1;
if (i === 5 && j === 5)
{
break Repeat;
}
else if (j === 5)
{
break;
}
}
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 14.9.html`.
3. Double-click the file to open it in the default browser.

When the variable `i`and `j`both equal `5`, the execution will jump to the outer loop labelled`Repeat`. It will repeat this process until `j` is equal to `5`. *Figure 14.8* shows the result:



Exiting a loop with break example

```
Exiting a loop with break example

Outer: 0
Inner: 1
Inner: 2
Inner: 3
Inner: 4
Outer: 1
Inner: 1
Inner: 2
Inner: 3
Inner: 4
Outer: 2
Inner: 1
Inner: 2
Inner: 3
Inner: 4
Outer: 3
Inner: 1
Inner: 2
Inner: 3
Inner: 4
Outer: 4
Inner: 1
Inner: 2
Inner: 3
Inner: 4
```

Figure 14.8: *break with a label*

Now that you know how to exit loops, let's have a look at how to continue loops.

The continue statement

The continue statement restarts a `while`, `do…while`, or `for` loop.

The next sample demonstrates the use of the `continue` statement in a loop:

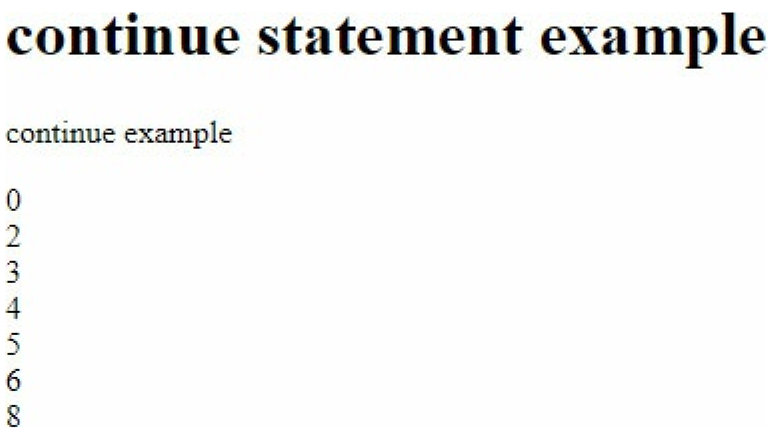
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!doctype html>
<body>
<h1>continue statement example</h1>
```

```
<p>continue example</p>
<script>
for (let i = 0; i < 9; i++)
{
if (i === 1)
continue;
else if (i === 7)
continue;
document.write(i + '</br>' );
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 14.10.html`.
3. Double-click the file to open it in the default browser.

The loop is supposed to loop from `0` to `9`. It does, but it skips a loop when `i` is `1` and when `i` is `7` because of the continue statement. *Figure 14.9* shows the result:



```
continue example

0
2
3
4
5
6
8
```

Figure 14.9: *continue statement in action*

As you can see the numbers `1` and `7` are skipped.

Conclusion

In this chapter, we learnt how to use loops. You saw how you coulduse the `for` loop to repeat tasks for a certain number of times. You saw that the `for…in`and `for…of`loops provide more power to the ordinary `for` loop where you can easily iterate through strings, arrays, and object properties. You learned about the `while` statement and how it is different from the various for loops. Lastly, you learned how to quickly exit loops and continue to different iterations with the help of labelled statements and the break and `continue` keywords.

The next chapteris the last chapter in the *Using JavaScript Productively* section. You will learn about the importance of code quality and how to improve your sloppy code.

Questions

1. What is the difference between the `for` loop and the `while` loop?
2. Give the syntax for a basic `for` loop.
3. Give the syntax for a basic `while` loop.
4. How can you exit a loop?
5. What is the `continue` statement used for?

CHAPTER 15
Code Quality

Introduction

Any written code will always have errors. The trick is to write code in such a way that it makesspotting problematic code easier.

In this lesson, you will learn how to debug JavaScript code with the built-in tools JavaScript provides. You will also learn how to write neat code so that you can read it easier

Structure

- Debugging
- Coding style
- Comments

Objectives

- Understand what debugging means.
- Identify the types of errors and fix them.
- Make use of the console object's debugging features.
- Make use of the `debugger` keyword to pause script execution.
- Make use of `try` and `catch` blocks to catch errors.
- Learn how to write tidy code.
- Know how to use comments in JavaScript.

Debugging

Show me a script without any errors, and I'll show you the *Loch Ness* monster. Whilst surfing a website, it may seem perfect, and by all means, it is, but there used to be a plethora of errors, and maybe there still are. Errors happen. Mistakes happen. It is part of programming. The trick, however, is how you handle them.

There are mainly two types of errors in JavaScript; they are:

- **Syntax errors**: Syntax errors happen whilst typing code. When a person types fast, he or she doesn't always registera mistake until later. These can be difficult to find, as you may have to scan the lines one by one unless you have a proper debugging tool at hand.
- **Logical errors**: Logical errors give developers nightmares. These usually occur when a certain result is expected, but something different happens. For example: Say you want to turn the background blue, and you write code for it, but the background ends up red. That is a logical error.
- **Runtime errors**: A runtime error happens while a user is busy using the page. Let's say the user enters the wronginformation or supplies information in the wrong format. These are runtime errors, and you should be able to handle them.

In order to fix any of these types of errors, you need tools. Remember, JavaScript doesn't have a fully-fledged **IDE** (**Integrated Development Environment**) which comes with debugging tools and features, so there aren't too many options available.

JavaScript debugging language features

The JavaScript language includes two debugging features; they are:

- The console window
- The debugger keyword

Let's get into more details.

The console window

The console window is more powerful than what you may realize! You can not only output important content to it, but you can time certain procedures seeing how long they will take. This can be a long loop or a complex statement. After all, speed is crucial in web applications.

console.log

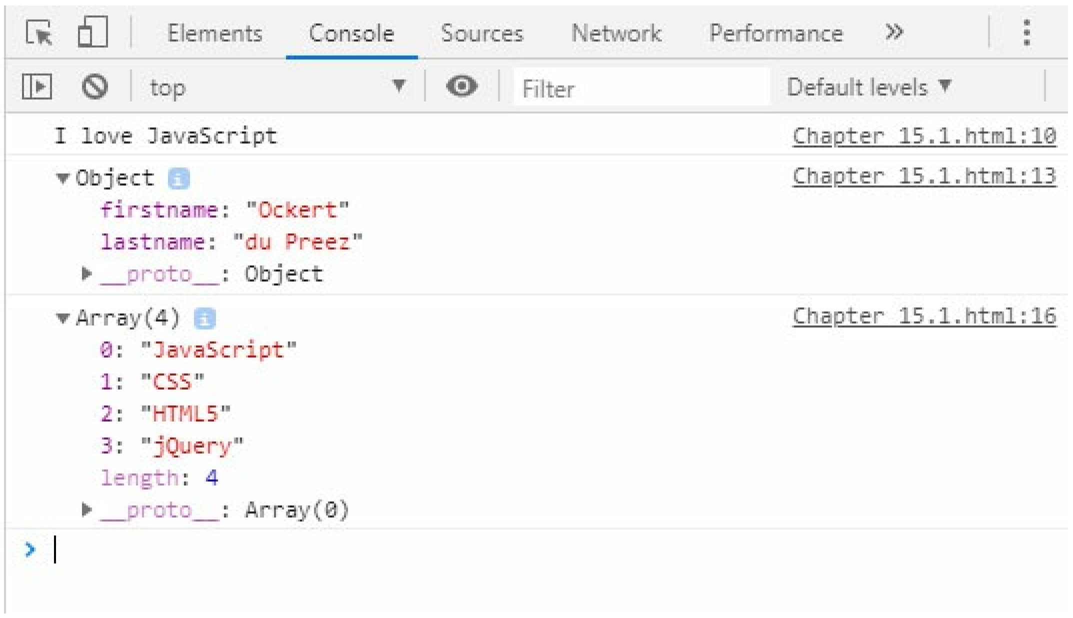
The function of console.log is to output information to the browser's console window. I have been demonstrating various code output to the console window in practically all the lessons thus far, but, just in case you have forgotten, let's do a small exercise demonstrating the use of `console.log`.

- Open a text editor of your choice and enter the following HTML and JavaScript code:

  ```
  <html>
  <body>
  <h1>console.log Output</h1>
  <p>The following code demonstrates the use of
  console.log</p>
  <script>
  console.log(“I love JavaScript”);
  var myObj = { firstname : “Ockert”, lastname : “du Preez” };
  console.log(myObj);
  var myArr = [“JavaScript”, “CSS”, “HTML5”, “jQuery” ];
  console.log(myArr);
  </script>
  </body>
  </html>
  ```

- Save the file with an HTML extension, for example `Chapter 15.1.html`.
- Double-click the file to open it in the default browser.
- Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

When the console window is opened, you will see the three different types of content written to it. First, you have the string: `I love JavaScript!`, then the object displaying my first name and last name. Lastly, the content of the array is also printed. *Figure 15.1* shows the result:



```
I love JavaScript                                  Chapter 15.1.html:10
▼Object                                            Chapter 15.1.html:13
   firstname: "Ockert"
   lastname: "du Preez"
  ▶__proto__: Object
▼Array(4)                                          Chapter 15.1.html:16
   0: "JavaScript"
   1: "CSS"
   2: "HTML5"
   3: "jQuery"
   length: 4
  ▶__proto__: Array(0)
```

Figure 15.1: *console.log*

Another useful feature of the console window is the `console.time` keywords

console.time and console.timeEnd keywords

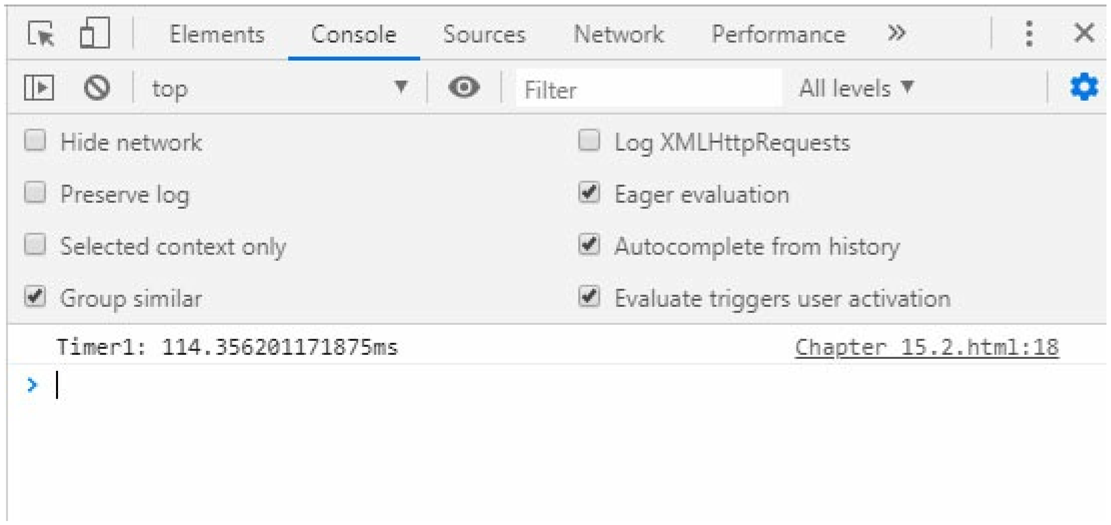
Sometimes you need to ensure that your code runs fast enough on every browser. For this, you can use the `time,` and `timeEnd` methods of the `console`object start the timer andstop the built-in timer respectively. Let's do a quick exercise!

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>console timing features</h1>
<p>The following code demonstrates the use of console.time
and console.timeEnd</p>
<script>
console.time('Timer1');
var items = [];
for(var i = 0; i < 500000; i++){
    items.push({index: i});
}
console.timeEnd('Timer1');
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 15.2.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

The output I got is displayed in *Figure 15.2*:



Hide network [ ] · Log XMLHttpRequests [ ]
Preserve log [ ] · Eager evaluation [✓]
Selected context only [ ] · Autocomplete from history [✓]
Group similar [✓] · Evaluate triggers user activation [✓]

```
Timer1: 114.356201171875ms                     Chapter 15.2.html:18
```

Figure 15.2: *console timing features*

The console informs us that it took `114`milliseconds to complete all the iterations of the `for` loop. You specify where the timer should start and where it should end.

More console methods

The `console` object includes many other methods to help with debugging. They are listed in the following table:

Method	**Description**
`assert()`	Writes an error message to the console if the assertion is false
`clear()`	Clears the console
`count()`	Logs the number of times that the call to count() has been called
`error()`	Outputs an error message to the console
`group()`	Creates a new inline group in the console.

`groupCollapsed()`	Creates a new inline group in the console.
`groupEnd()`	Exits the current inline group in the console
`info()`	Outputs an informational message to the console
`table()`	Displays tabular data as a table
`trace()`	Outputs a stack trace to the console
`warn()`	Outputs a warning message to the console

Table 15.1: *Console methods*

Let’s do an exercise!

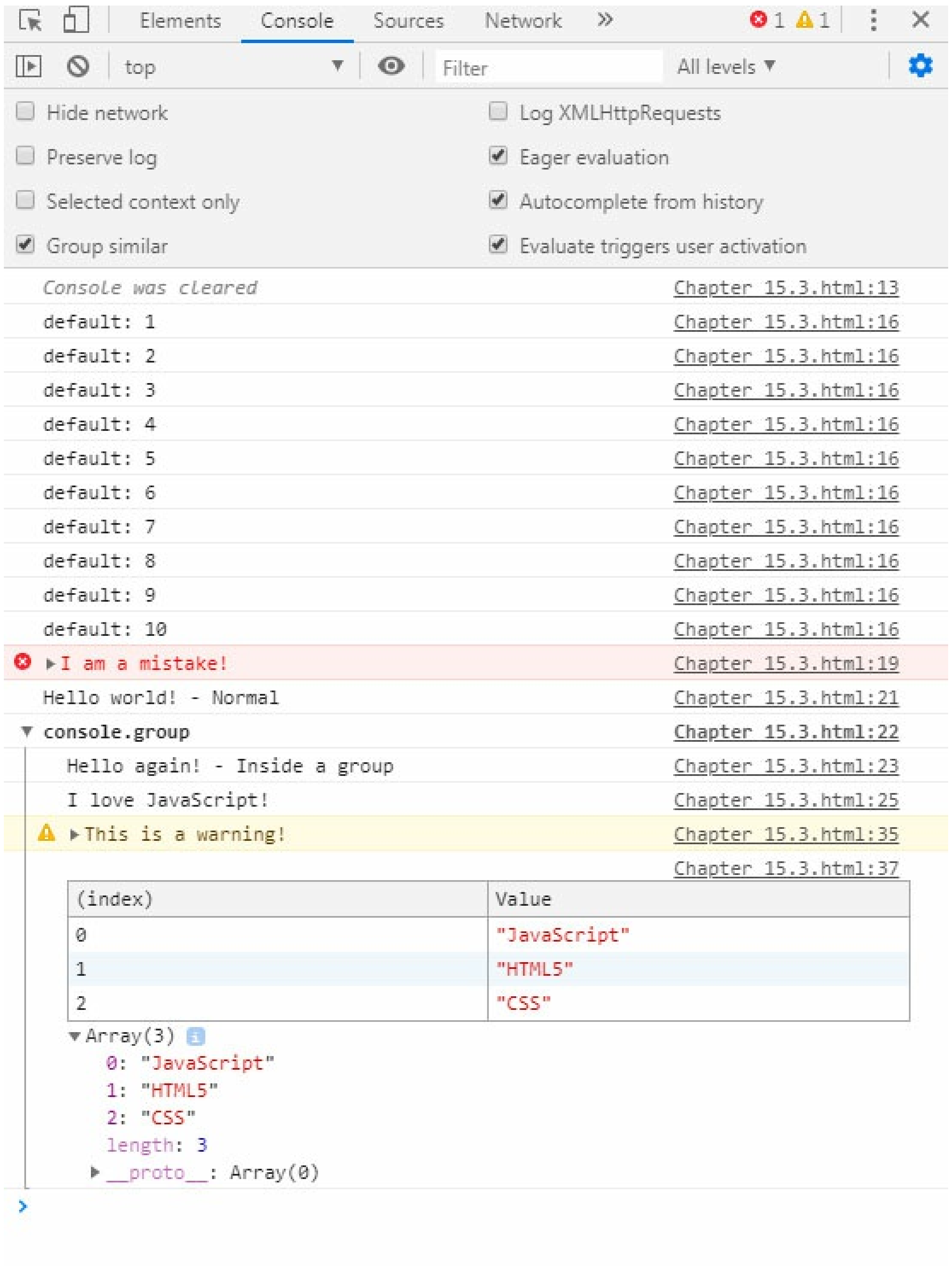
There are a few more console methods, as the next exercise will demonstrate:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1> more console methods </h1>
<p>The following code demonstrates the use of more console
methods</p>
<script>
var myObj = { firstname : “Ockert”, lastname : “du Preez” };
console.assert(document.getElementById(“demo”), myObj);
console.clear();
for (i = 0; i < 10; i++) {
 console.count();
}
console.error(“I am a mistake!”);
console.log(“Hello world! - Normal”);
console.group();
console.log(“Hello again! - Inside a group”);
console.info(“I love JavaScript!”);
function myFunction() {
 anotherFunction();
}
function anotherFunction() {
 console.trace();
}
console.warn(“This is a warning!”);
console.table([“JavaScript”, “HTML5”, “CSS”]);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 15.3.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

The output of the above code is shown in the following screenshot:



```
☐ Hide network                  ☐ Log XMLHttpRequests
☐ Preserve log                  ☑ Eager evaluation
☐ Selected context only         ☑ Autocomplete from history
☑ Group similar                 ☑ Evaluate triggers user activation
```

```
Console was cleared                          Chapter 15.3.html:13
default: 1                                   Chapter 15.3.html:16
default: 2                                   Chapter 15.3.html:16
default: 3                                   Chapter 15.3.html:16
default: 4                                   Chapter 15.3.html:16
default: 5                                   Chapter 15.3.html:16
default: 6                                   Chapter 15.3.html:16
default: 7                                   Chapter 15.3.html:16
default: 8                                   Chapter 15.3.html:16
default: 9                                   Chapter 15.3.html:16
default: 10                                  Chapter 15.3.html:16
▶I am a mistake!                             Chapter 15.3.html:19
Hello world! - Normal                        Chapter 15.3.html:21
▼ console.group                              Chapter 15.3.html:22
    Hello again! - Inside a group            Chapter 15.3.html:23
    I love JavaScript!                       Chapter 15.3.html:25
    ▶This is a warning!                      Chapter 15.3.html:35
                                             Chapter 15.3.html:37
```

(index)	Value
0	"JavaScript"
1	"HTML5"
2	"CSS"

```
▼Array(3)
    0: "JavaScript"
    1: "HTML5"
    2: "CSS"
    length: 3
  ▶__proto__: Array(0)
>
```

Figure 15.3: *More console methods*

As you can see, the console is much more powerful than what is thought. With the use of the `assert`, warn, and error methods, you can quickly interrogate errors, and with the grouping and table methods, you can organize the output of your data.

The following JavaScript debugging language feature is the debugger keyword which will be explained next.

The debugger keyword

The JavaScript `debugger` statement stops the execution of code and calls the browser debugging function. This is the same as setting a breakpoint in code. For those of you not familiar with the term breakpoint, a breakpoint allows the debugger to stop execution so that you can interrogate variable values before and while the code executes.

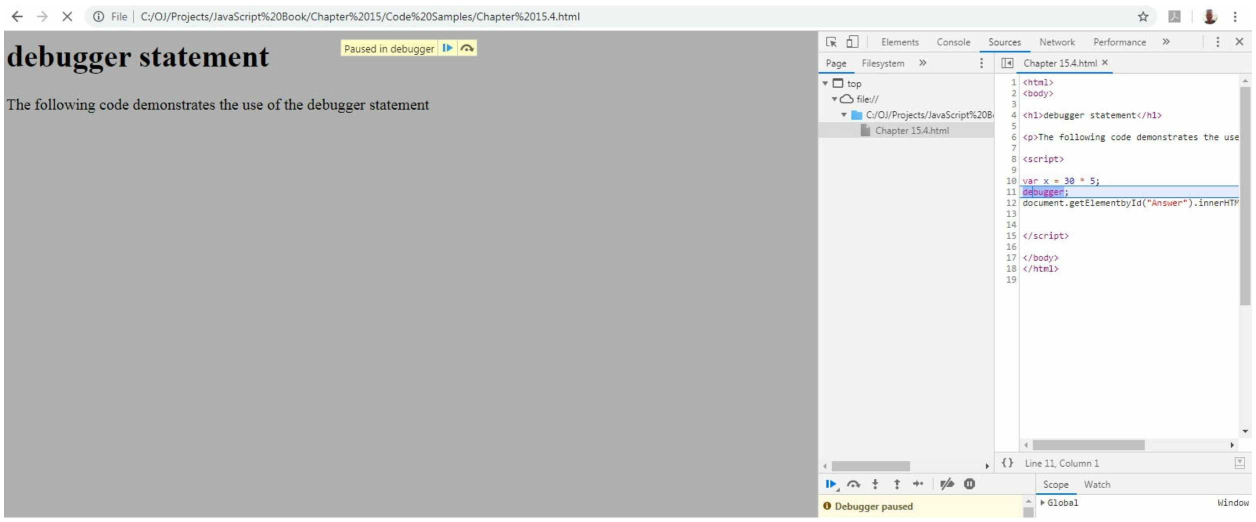
The next little exercise demonstrates the use of the `debugger` keyword:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>debugger statement</h1>
   <p>The following code demonstrates the use of the debugger
   statement</p>
   <script>
   var x = 30 * 5;
   debugger;
   document.getElementbyId(“Answer”).innerHTML = x;
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 15.4.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

The paused browser window is displayed:



debugger statement

The following code demonstrates the use of the debugger statement

```
<html>
<body>

<h1>debugger statement</h1>

<p>The following code demonstrates the use

<script>

var x = 30 * 5;
debugger;
document.getElementbyId("Answer").innerHTM


</script>

</body>
</html>
```

Figure 15.4: *Code paused in the Debugger*

As you can see, in debugging mode, the screen turns grey and highlights the debugger line. Also, notice the small yellow window in the browser window. It states that the code is paused in the debugger and provides two buttons. Resume execution continues code execution until the next breakpoint, if present. Step over the next function goes to the next line (in this case).

If the **`Resume`** is pressed, the next line will be underlined as an error because there is no Answer field present in the document.

Let's see how to deal with runtime errors.

try and catch blocks

The `try…catch` statement handles the errors that can occur while running code. You put the potential breaking code inside the `try` block, and the debugger will try and execute it. In the event of an error, the code inside the `catch` block will fire. This prevents the page from breaking and the browser to produce an error.

You may optionally add a `finally` statement after the `catch` block, which will execute irrespective of an error.

The next exercise demonstrates the use of the `try…catch`statement.

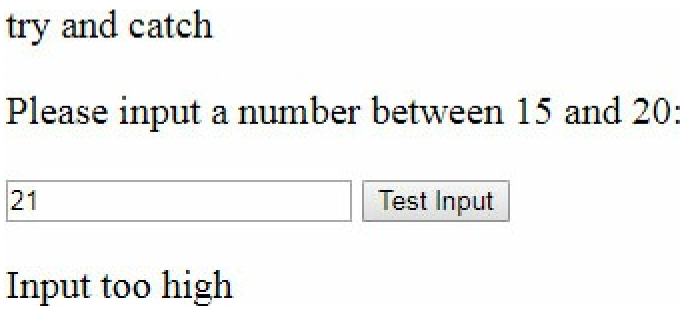
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!DOCTYPE html>
<html>
```

```
<body>
<p>try and catch</p>
<p>Please input a number between 15 and 20:</p>
<input id=”userInput” type=”text”>
<button type=”button” onclick=”Check()”>Test Input</button>
<p id=”output”></p>
<script>
function Check() {
 var message, x;
 message = document.getElementById(“output”);
 message.innerHTML = “”;
 x = document.getElementById(“userInput”).value;
 try {
  if(x == “”) throw “is Empty”;
  if(isNaN(x)) throw “not a number”;
  if(x > 20) throw “too high”;
  if(x < 15) throw “too low”;
 }
 catch(err) {
   message.innerHTML = “Input “ + err;
 }
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 15.5.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

An example of the output is shown below:


try and catch

Please input a number between 15 and 20:

21

Input too high

Figure 15.5: *try and catch example*

The `try` block is used to determine what is input by the user into the `userInput` field. If the number is between `15` and `20,` nothing will happen; in other words, no error will be produced. If, however, there is no value input, a word instead of a number, or a value higher than `20`, as is the case in *Figure 15.5*, or a value less than `15` an error will be thrown, and it will be handled inside the `catch` block.

Coding style

Each programming language has a set of *unwritten* rules that developers should follow. OK, not totally unwritten, but rules that are followed, standards if you may, to improve the readability of code. This is necessary because scripts can become very, very long and you can get lost quite quickly in the plethora of bad variable and object names and messy conditional statements and loops.

Variable and object names

There are mainly two naming conventions for variable names, namely: camel case and pascal case. They differ in the way they are capitalized, as shown next:

- camelCase
- PascalCase

Throughout this book, I have tried to keep the camelCase naming convention.

It is also quite common to have global variables in UPPERCASE.

Variable names can contain letters, numbers, the dollar sign, and the underscore. A few examples follow:

```
full_Name
student1
student_Name$
```

Remember though that you should always maintain the same naming convention throughout your script. Variables were discussed in *Chapter 12: Variables*.

Operators and indentation

The common rule with operators is always to have spaces between all the operands. Compare the following:

```
Result=varibale1+variable2
```

versus

```
Result = varibale1 + variable2
```

Spaces make the code much more legible. Operators were discussed in Chapter 4: JavaScriptOperators.

Talking about spaces, when you have controlflow statements (*Chapter 13: Control Flow Statements*) in your code or make use of functions (*Chapter 5: JavaScript Functions*) you absolutely should make use of spaces to improve the flow of your code. Compare the following code segments:

```
var time = new Date().getHours();
if (time < 12) {
greeting = “Good morning”;
}
else if (time < 18) {
greeting = “Good afternoon”;
}
else {
greeting = “Good evening”;
}
document.write(greeting);
```

versus

```
var time = new Date().getHours();
if (time < 12) {
 greeting = “Good morning”;
}
else if (time < 18) {
 greeting = “Good afternoon”;
}
else {
 greeting = “Good evening”;
}
document.write(greeting);
```

The segment with the spaces is much easier to read. This may be a very simple example, but when nested control-flow statements come into play, you can get confused very quickly.

Rules for objects

Common practices for object (*Chapter 2: JavaScript Objects*) definitions are as follows:

- The opening bracket should be placed on the same line as the object name.
- Use a colon and one space between a property name and its value.
- Use quotes around string values only, unless the numeric value should be a string.
- No comma is needed after the last property-value pair.
- The closing bracket should be placed on a new line, without a leading space.
- End an object definition with a semicolon.

Here is a small example making use of all the above rules:

```
var student = {
 firstName: “Ockert”,
 lastName: “du Preez”,
 age: 41,
 course: “JavaScript”
};
```

It is very easy to see what the object consists of.

Comments

Comments are used to describe what the code is doing. Trust me; you don’t realize how important this is until it is too late. Imagine working on a complex problem, and after a while, you figured it out. Imagine further that you didn’t add comments to describe the code and that you come back to the code 6 months later to do an update. It can be extremely hard to remember what you have done and where you have done it.

Another important fact to remember is that you may be working in a team environment. Comments can help you understand another team member’s code and vice versa.

There are two types of comments:

- Single line comments
- Multi-line comments

Let’s have a closer look.

Single line comments

A single line comment can be placed anywhere on a line, for example:
```
var objstudent; //Declare student object here
```

In the above code, the comment starts with two forward slashes and is placed after an object declaration.

Multi-line comment

A multi-line comment can span multiple lines, for example:

```
/*This conditional statement tests the time
The result that will be shown depends on the time of day
Created by: OJ du Preez
2019 */
var time = new Date().getHours();
if (time < 12) {
 greeting = “Good morning”;
}
else if (time < 18) {
 greeting = “Good afternoon”;
}
else {
 greeting = “Good evening”;
}
document.write(greeting);
```

The code between the forward slashes and asterisks describe what the code does and who developed it.

Comments do not get displayed on the web page; they are solely for informational purposes.

Conclusion

This was the last chapter in the *Using JavaScript Productively* section. In this chapter, we learnt the importance of writing good code and making use of all the debugging and coding tools at your disposal. The better your code is structured, the better it is to navigate and to debug.

In the next chapter, we will kick off the *Advanced Concepts* section and will cover arrays in JavaScript. Although you may have seen the term, or even used them in code inside this book, you do not know all the intricacies and how to use them productively. That is what you will do next.

Questions

1. Name the two most common types of errors in JavaScript
2. What do the `console.time` and `console.timeEnd` keywords do?
3. How do you pause code in the debugger?
4. Why would you use `try…catch`?
5. Name the two types of comments in JavaScript.

SECTION IV

Advanced Concepts

Introduction

Section 4 explores some advanced concepts in JavaScript. Chapter 16 deals with arrays. You will learn how to create and use arrays productively. Chapter 17 explores the world of regular expressions and demonstrates how easy it is to search for text in strings and to limit text input. Chapter 18 takes functions a huge step further where it explains functional programming concepts such as partial applications and currying.

This section will have the following chapters:

1. Arrays
2. Regular Expressions
3. Partials and Currying.

CHAPTER 16

JavaScript Arrays

Introduction

Arrays give you the ability to store more than one value inside a named container. You access elements of an array by using the array's index.

In this lesson, you will learn how to add and remove elements from arrays, how to access items, and how to iterate over arrays.

Structure

- Arrays
    - Creating arrays
        - Adding items to an array
        - Removing items from arrays
    - Multidimensional arrays

Objectives

- Understand what arrays are and what you can do with them.
- Know the various ways of adding elements to the array.
- Know the various ways of removing items from an array.
- Know what Multidimensional arrays are.

Arrays

There are occasions when an ordinary object is simply not enough. What if I have a list of information such as results that I'd like to store? One object cannot store multiple values in this case. An array is a list of values that are organized. You may remember that I have mentioned the term numerous times thus far in this book, but I never really explained it extensively.

Creating arrays

There are essentially two ways to create arrays:

- `let arr = new Array();`
- `let arr = [];`

Although the latter is more preferred, I will explain both practically

Method 1

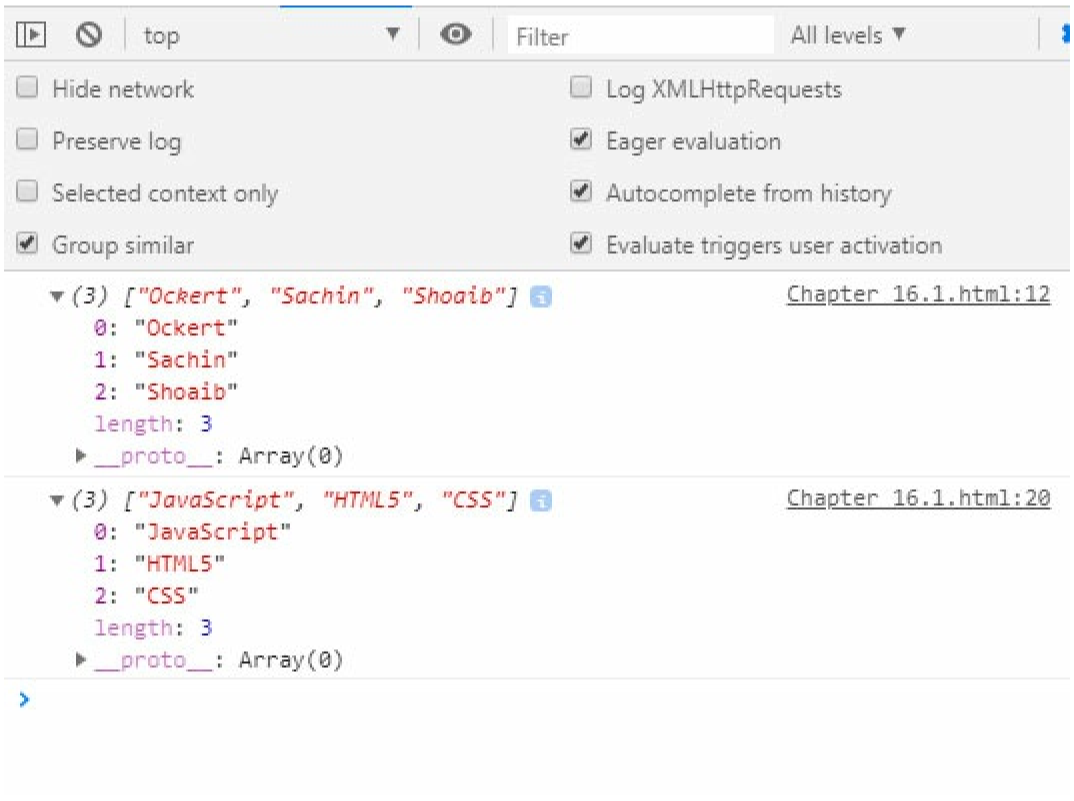
The first method makes use of the `Array` keyword to create arrays. There is an easy way and a hard way, as you will shortly see in the following exercise:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>new Array</h1>
   <p>new Array</p>
   <script>
   let arrStudents = new Array("Ockert", "Sachin", "Shoaib");
   //Easy
   console.log(arrStudents);
   let arrSubjects = new Array(2); //More Complicated
   arrSubjects[0] = 'JavaScript';
   arrSubjects[1] = 'HTML5';
   arrSubjects[2] = 'CSS';
   console.log(arrSubjects);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 16.1.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

The result is displayed in the following screenshot:



Hide network
Preserve log
Selected context only
✓ Group similar
Log XMLHttpRequests
✓ Eager evaluation
✓ Autocomplete from history
✓ Evaluate triggers user activation

```
▼(3) ["Ockert", "Sachin", "Shoaib"]                Chapter 16.1.html:12
    0: "Ockert"
    1: "Sachin"
    2: "Shoaib"
    length: 3
  ▶__proto__: Array(0)
▼(3) ["JavaScript", "HTML5", "CSS"]                Chapter 16.1.html:20
    0: "JavaScript"
    1: "HTML5"
    2: "CSS"
    length: 3
  ▶__proto__: Array(0)
>
```

Figure 16.1: *Creating an array with new Array*

As expected, the two arrays are output nicely into the console window. The easier way in the above code was a one-liner. The array is created and instantiated at the same time. The more complicated way not only is a bit more complicated, but it is a bit more work as well. That is why developers shy away from it.

The more complicated way required a two-step process for the array to have values. The array is created with a specified number of elements as supplied within the brackets. The problem here is that the array's values are still undefined. This is where you have to physically supply a value to each element in the array by using the array's index. This can get quite tedious if you have many elements inside the array.

Method 2

This one may be familiar already, or at least you may have seen it before in this book. Let's quickly create the same arrays as above in a preferred way.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>new Array</h1>
<p>new Array</p>
<script>
let arrStudents = [“Ockert”, “Sachin”, “Shoaib”]; //Easy
console.log(arrStudents);
let arrSubjects = [‘JavaScript’, ‘HTML5’, ‘CSS’];
console.log(arrSubjects);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 16.2.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

The results look the same, as shown in *Figure 16.1*. As you can see, it is a lot less code.

We have an array, now what?

Well, the object of an array is to know which elements are where and you should be able to access them; else, what’s the purpose? With the above arrays, let’s say you want to only retrieve the second element from the first array, and the third element from the second array. In other words: `Sachin` and `CSS`. In the next exercise, you will do just that!

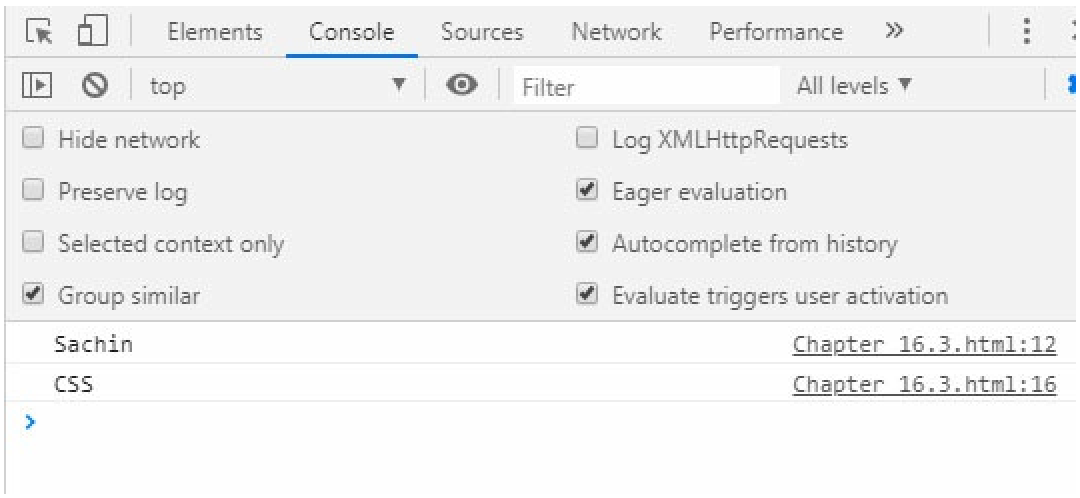
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>new Array</h1>
<p>new Array</p>
<script>
let arrStudents = [“Ockert”, “Sachin”, “Shoaib”]; //Easy
console.log(arrStudents[1]);
let arrSubjects = [‘JavaScript’, ‘HTML5’, ‘CSS’];
console.log(arrSubjects[2]);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 16.3.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

I have only made two changes to the original code in above code. You will notice that I have included square brackets after the array name and included a number. To get the second element of the first array, I used the number `1`. To get the third element in the second array, I used the number `2`. This may be confusing, but remember computers start counting at `0`, so the first element is `0`.

Figure 16.2 shows the result of the element extraction of both arrays:



```
Sachin                                          Chapter 16.3.html:12
CSS                                             Chapter 16.3.html:16
```

Figure 16.2: *Obtaining specific elements in arrays*

The console window shows `Sachin` and `CSS`, as expected. Play around with the other elements and don't be scared to add more elements to the initial arrays.

The beauty of an array is that you can manipulate its elements by adding more elements to it and removing elements from it. You are also able to insert elements at specific locations and remove elements from specific locations.

Adding items to an array

Arrays are surprisingly versatile when adding items to it. You can add elements to the beginning of an array, to the end of an array and to the middle of an array.

- **Using push to add items to the end of an array**

  In order to add elements to the end of an array, you need to use the push method, as demonstrated below:

  ```
  let arrStudents = [“Ockert”, “Sachin”, “Shoaib”]; //Easy
  arrStudents.push(“John”);
  ```

  John is added to the back of the array
- **Using unshift to add items to the beginning of an array**

  Make use of unshift to add items to the beginning of an array. Here is a small example:

  ```
  let arrStudents = [“Ockert”, “Sachin”, “Shoaib”];
  arrStudents.unshift(“Sam”);
  ```

  Sam is now at the beginning.
- **Using splice to add items into the middle of an array**

  To add items into the middle of an array (or anywhere other than the beginning or end of an array), you need to make use of the splice `array` method. This is shown below:

  ```
  let arrStudents = [“Ockert”, “Sachin”, “Shoaib”];
  arrStudents.splice(2, 0, “Peter”);
  ```

  Here `Peter` is placed in the third position of the array. One thing about the splice method is that you can remove items as well, but in this case, you supply 0, because you do not want to remove an item, you simply want to insert.

To bring all the adding methods together, let’s do a quick exercise:

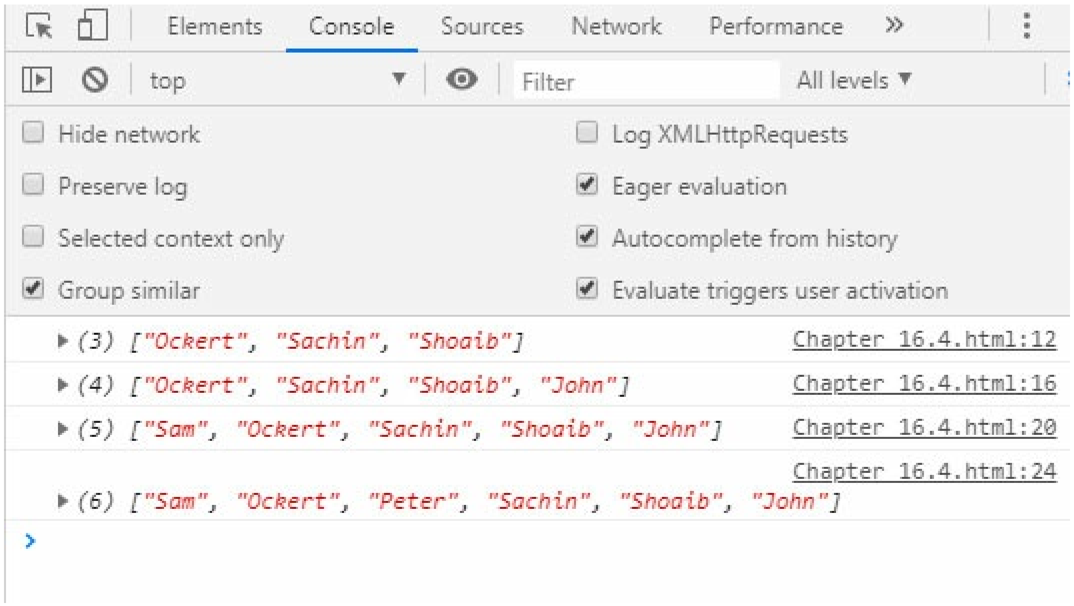
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>Adding Array elements</h1>
   <p>Adding Array elements</p>
   <script>
   let arrStudents = [“Ockert”, “Sachin”, “Shoaib”];
   ```

```
console.log(arrStudents);
arrStudents.push(“John”);
console.log(arrStudents);
arrStudents.unshift(“Sam”);
console.log(arrStudents);
arrStudents.splice(2, 0, “Peter”);
console.log(arrStudents);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 16.4.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

The output is shown below:



Hide network
Preserve log
Selected context only
✓ Group similar
Log XMLHttpRequests
✓ Eager evaluation
✓ Autocomplete from history
✓ Evaluate triggers user activation

```
▸ (3) ["Ockert", "Sachin", "Shoaib"]                       Chapter 16.4.html:12
▸ (4) ["Ockert", "Sachin", "Shoaib", "John"]               Chapter 16.4.html:16
▸ (5) ["Sam", "Ockert", "Sachin", "Shoaib", "John"]        Chapter 16.4.html:20
                                                           Chapter 16.4.html:24
▸ (6) ["Sam", "Ockert", "Peter", "Sachin", "Shoaib", "John"]
>
```

Figure 16.3: *Adding array elements*

The initial array contained three items. John was added to the back, then Sam was added to the beginning of the array. Finally, Peter was added to the middle. Removing items work similarly, let’s have a look

Removing items from arrays

Arrays are just as versatile when it comes to removing items from them. You can remove items from the beginning of an array and at the end. You can also remove items from the middle of an array.

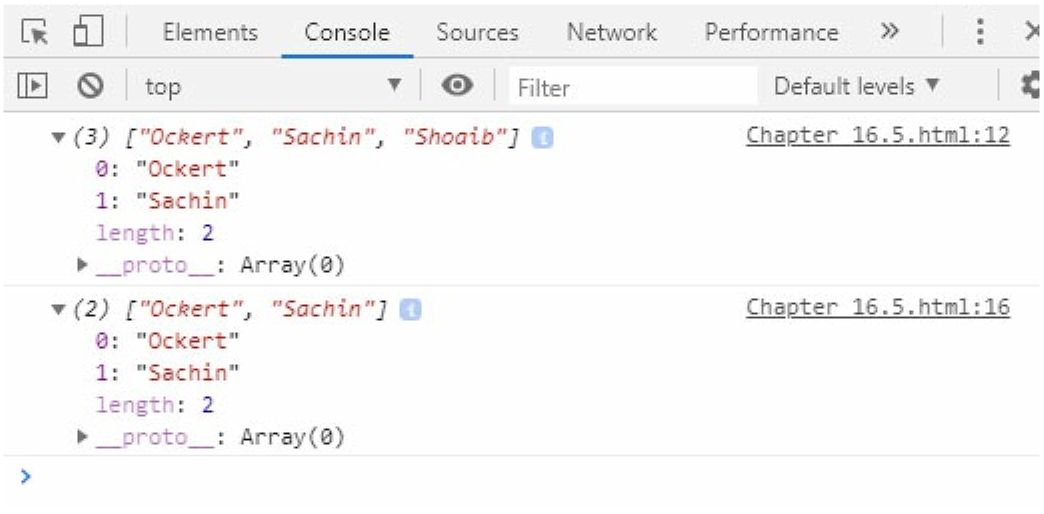
In order to remove an item from the end of an array, you need to make use of the `pop()` method. Let's do this practically:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>Removing Array elements</h1>
   <p> Removing Array elements</p>
   <script>
   let arrStudents = [“Ockert”, “Sachin”, “Shoaib”];
   console.log(arrStudents);
   arrStudents.pop();
   console.log(arrStudents);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 16.5.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

The resulting array is shown in the following screenshot:



```
▼(3) ["Ockert", "Sachin", "Shoaib"]                    Chapter 16.5.html:12
    0: "Ockert"
    1: "Sachin"
    length: 2
  ▶__proto__: Array(0)
▼(2) ["Ockert", "Sachin"]                              Chapter 16.5.html:16
    0: "Ockert"
    1: "Sachin"
    length: 2
  ▶__proto__: Array(0)
>
```

Figure 16.4: *Removing items at the end of an array*

The initial array contained three elements, and now only two. This is because of the `pop()` method removing the last array item (`Shoaib`). Let's now remove items from the front, and middle.

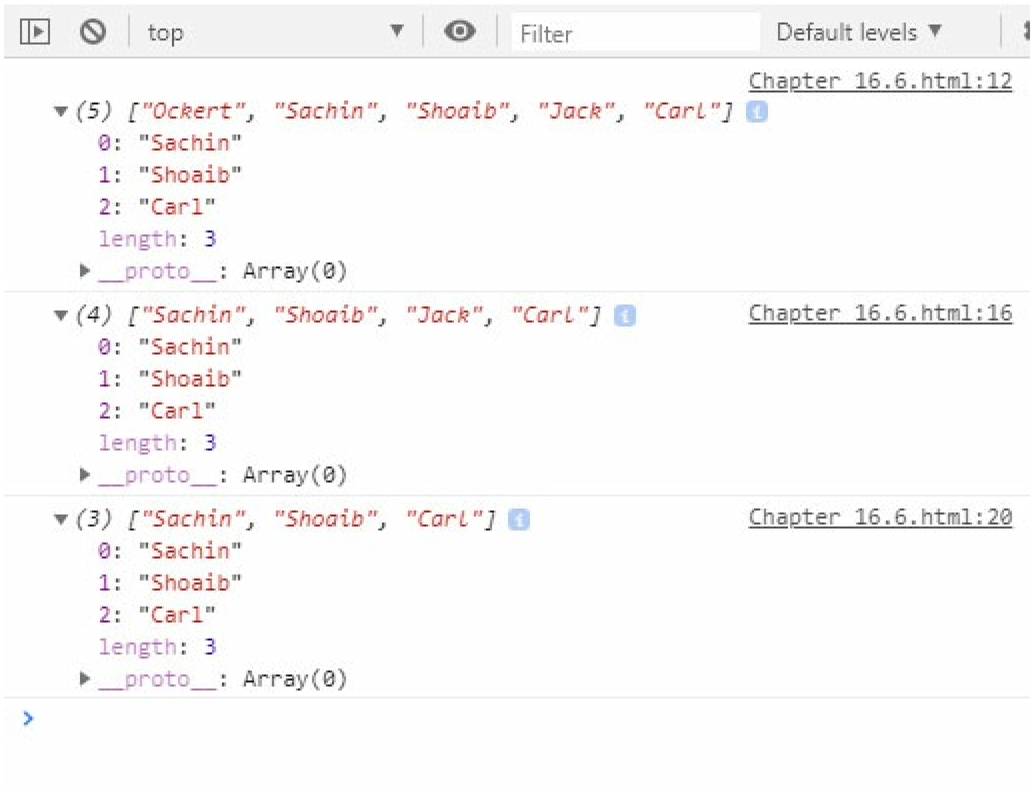
To remove items from the front of an array make use of the shift method and use splice to remove items from the middle:

1. Open a text editor of your choice and enter the following HTML and JavaScript code

   ```
   <html>
   <body>
   <h1>Adding Array elements</h1>
   <p>Adding Array elements</p>
   <script>
   let arrStudents = [“Ockert”, “Sachin”, “Shoaib”, “Jack”,
   “Carl”];
   console.log(arrStudents);
   arrStudents.shift();
   console.log(arrStudents);
   arrStudents.splice(2, 1)
   console.log(arrStudents);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 16.6.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

Figure 16.5 shows the output:



```
                                                         Chapter 16.6.html:12
▼(5) ["Ockert", "Sachin", "Shoaib", "Jack", "Carl"]
    0: "Sachin"
    1: "Shoaib"
    2: "Carl"
    length: 3
  ▶__proto__: Array(0)
▼(4) ["Sachin", "Shoaib", "Jack", "Carl"]              Chapter 16.6.html:16
    0: "Sachin"
    1: "Shoaib"
    2: "Carl"
    length: 3
  ▶__proto__: Array(0)
▼(3) ["Sachin", "Shoaib", "Carl"]                      Chapter 16.6.html:20
    0: "Sachin"
    1: "Shoaib"
    2: "Carl"
    length: 3
  ▶__proto__: Array(0)
>
```

Figure 16.5: *Removing items from the front and middle of an array*

The array initially contains five items. By calling the shift method, the first item (`Ockert`) is removed. The item `Jack` is also removed because of the `splice` method. In the `splice()` method you specified the index to remove, and how many indexes should be removed.

Multidimensional arrays

We have been working with single-dimensional arrays throughout this chapter, but now I'd like to talk about multidimensional arrays. JavaScript doesn't directly support them, but it can hold array elements inside other array elements, thus creating an object similar to a multidimensional array.

Now, what is the difference between a single dimension array and a multi-dimension array?

To explain this, or to at least have more clarity, let's quickly do an exercise:

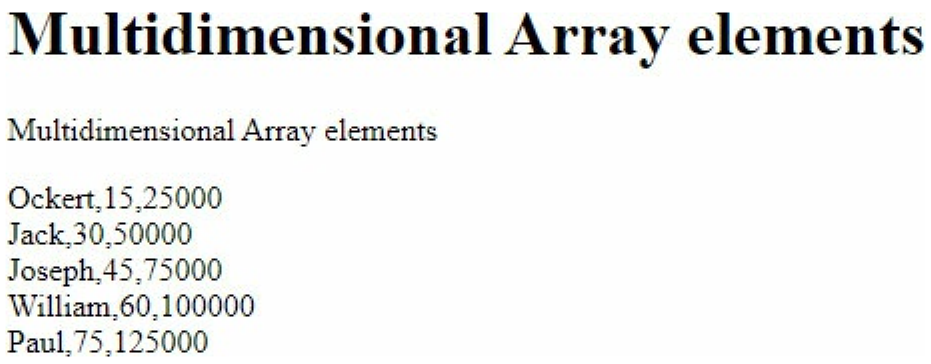
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   ```

```
<body>
<h1>Multidimensional Array</h1>
<p>Multidimensional Array</p>
<script>
var emp1 = [“Ockert”, 15, 25000];
var emp2 = [“Jack”, 30, 50000];
var emp3 = [“Joseph”, 45, 75000];
var emp4 = [“William”, 60, 100000];
var emp5 = [“Paul”, 75, 125000];
var salaries = [emp1, emp2, emp3, emp4, emp5];
for(var i = 0; i < salaries.length; i++) {
  document.write(salaries[i] + “</br>”);
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 16.7.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

The resulting text that gets written to the page is shown below:



```
Multidimensional Array elements

Multidimensional Array elements

Ockert,15,25000
Jack,30,50000
Joseph,45,75000
William,60,100000
Paul,75,125000
```

Figure 16.6: *Multidimensional array*

Five arrays are created. Each with a string value and two numeric values. Another array is then created, and it contains the five arrays, thus becoming multidimensional. Finally, the array gets iterated with a `for` loop, and its contents get printed onto the page, as shown above.

Conclusion

In this chapter, we learnt how to use arrays. You learned how to remove items from an array as well as add items into an array. You also learned what multidimensional arrays are.

In the next chapter, we will cover regular expressions in JavaScript.

Questions

1. Define the term array.
2. What does the shift method do?
3. Give an example of how to use splice to remove an array item.
4. What is a multidimensional array?
5. What does the pop method do?

CHAPTER 17
Regular Expressions

Introduction

Searching text in strings can be difficult, especially if you only know a portion of a phrase within the string. Luckily JavaScript includes regular expressions. Regular expressions help you identify parts of the text with a combination of special characters and patterns.

In this lesson, you will learn how to use regular expressions to identify certain pieces of text.

Structure

- Regular expression definition
- Regular expression modifiers/flags
- Regular expression patterns

Objectives

- Know what regular expressions are.
- Know how to use modifiers/flags to improve search results.
- Make use of patterns to search for text or phrases.

Regular expression definition

Regular expressions in JavaScript are patterns used to match various character combinations in strings. This sequence of characters forms a search pattern which can be used to search for text inside strings as well as replace text in strings. It can be a single character or a more complex pattern.

A regular expression can be constructed in either of the following two ways:

- Using a regular expression literal
- Calling the constructor function of the `RegExp`object

Let's get into more details.

Using a regular expression literal

A regular expression literal consists of a pattern enclosed between slashes, for example:

```
let exp = /ab+c/;
```

Regular expression literals help the regular expression to be compiled upon the loading of the script. If a regular expression remains the same, using regular expression literals can improve performance.

Calling the constructor function of the RegExp object

You call the constructor function of the `RegExp` object by using the keyword new (as explained in *Chapter 2: JavaScript Objects*), for example:

```
let exp = new RegExp('ab+c');
```

Going this route is best when you know the regular expression pattern may change, or you don't know the pattern and are getting it from another source.

Regular expression patterns can consist of simple characters, such as `/xyz/`, or a combination of simple and special characters, such as `/xy*z/` or `/Student Number (\d+)\.\d*/`. I will get into details of the pattern a little bit later. Now, let's have a look at regular expression modifiers.

Regular expression modifiers/flags

All regular expressions can contain modifiers or flags that can affect the search.JavaScript has 6 flags, as shown in the table below:

Modifier	**Description**
`i`	Case insensitive search
`g`	Searches for all matches
`m`	Multiline mode

`s`	Enables dotall mode
`u`	Enables full Unicode support
`y`	Sticky mode. Search at the exact position

Table 17.1: *Regular expression flags*

There is obviously still a lot to learn about regular expressions, but I feel let's do a quick exercise to fit everything we have learned so far together.

This exercise makes use of the above flags and the string match method (string functions were covered in *Chapter 12: Variables*) to find some text using a regular expression.

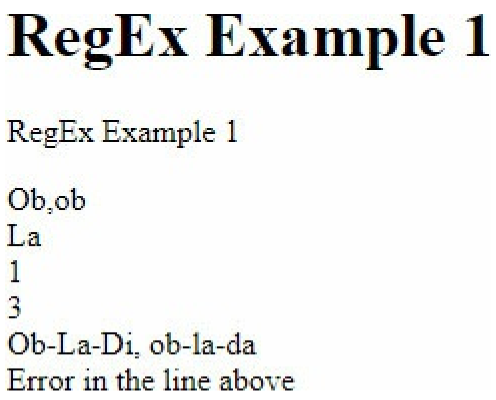
1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>RegEx Example 1</h1>
<p>RegEx Example 1</p>
<script>
let str = "Ob-La-Di, ob-la-da";
document.write( str.match(/Ob/gi) + "</br>" ) ;
let result = str.match(/La/i);
document.write( result[0] + "</br>") ;
document.write( result.length + "</br>") ;
// Details:
document.write( result.index + "</br>") ;
document.write( result.input + "</br>") ;
try
{
let matches = str.match(/JavaScript/);
if (!matches.length) {
}
}
catch(err) {
document.write("Error in the line above");
}
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 17.1.html`.

3. Double-click the file to open it in the default browser.

The output of the above code is shown in the following screenshot:



```
RegEx Example 1

RegEx Example 1

Ob,ob
La
1
3
Ob-La-Di, ob-la-da
Error in the line above
```

Figure 17.1: *string.match regular expression*

I start off by creating a string containing `Ob-La-Di, ob-la-da`. An old Beatles song that was released in 1968, still a classic. Anyway, then I used the match method of string to search for the term `Ob`. Because I supplied the g and i modifiers, it searched for all instances of `Ob` irrespective of the case in the string.

Next,I created variable named results that will the result of the regular expression searching for `La`. Notice that I didn't include the `g` flag here, so only the first instance will be found. Its index is 3 because `L` starts at the fourth character in the original string. Then the entire string is printed because that is the regular expression's input or source from which to search.

Lastly, I included a `try` and `catch` block to demonstrate what you should do when there are no matches for the regular expression.

Regular expression patterns

As I mentioned earlier, regular expression patterns can consist of simple characters, such as `/xyz/`, or a combination of simple and special characters, such as `/xy*z/` or `/Student Number (\d+)\.\d*/`. The exact sequence of the provided character or characters must occur in the string being tested (in the exact order).

Regular expression patterns can also consist of special characters, which serve as placeholders. This is necessary when the search for a match needs something more than a direct match, such as finding one or more o's, or finding white space in a string.

Below is a list of all the special characters that you can use in your regular expressions. After the table, you will do a nice little exercise using some of them, seeing the fact that you have already seen a simple character regular expression in action in the previous example:

Character	**Description**
\	When a backslash precedes a non-special character, it indicates that the next character is special and must not to be interpreted literally. When a backslash precedes a special character, it indicates that the next character is not special and must be interpreted literally
^	Matches the beginning of input.
$	Matches the end of input.
*	Matches the preceding expression 0 or more times.
+	Matches the preceding expression 1 or more times.
?	Matches the preceding expression 0 or 1 time.
.	Matches any single character except the newline character
(x)	Matches x and remembers the match.
(?:x)	Matches x but does not remember the match.
x(?=y)	Matches x only if x is followed by y.
x(?!y)	Matches x only if x is not followed by y.
(?<=y)x	Matchesxonly ifxis preceded byy.
(?<!y)x	Matchesxonly ifxis not preceded byy.
x|y	Matches x, or y (when there is no match for x).
{n}	Matches n occurrences of the expression.
{n,}	Matches at least n occurrences of the preceding expression.
{n,m}	Matches at least n and at most m occurrences of the preceding expression.
[xyz]	Matches any one of the characters in the brackets.
[^xyz]	Matches anything not enclosed in the brackets.
[\b]	Matches a backspace.
\b	Matches the position between a word character followed by a non-

	word character.
`\B`	Matches the following: • Before the first character of the string. • After the last character of the string. • Between two-word characters. • Between two non-word characters. • Empty string.
`\cX`	Matches a control character in a string.
`\d`	Matches a digit character.
`\D`	Matches a non-digit character.
`\f`	Matches a form feed.
`\n`	Matches a line feed
`\r`	Matches a carriage return.
`\s`	Matches a white space character.
`\S`	Matches a character other than white space.
`\t`	Matches a tab.
`\v`	Matches a vertical tab.
`\w`	Matches any alphanumeric character.
`\W`	Matches any non-word character.
`\0`	Matches a NULL character.
`\xhh`	Matches the character with the code `hh` (two hexadecimal digits).
`\uhhhh`	Matches the character with the code `hhhh` (four hexadecimal digits).

Table 17.2: *Special character patterns*

Let's do a quick exercise!

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>RegEx Example 2</h1>
<p>RegEx Example 2</p>
<script>
```

```
var Num = /^\d{10}$/;
console.log(Num.test('7795483645'));
var Dt = /^(\d{1,2}-){2}\d{2}(\d{2})?$/;
console.log(Dt.test('01-01-1990'));
console.log(Dt.test('01-01-90'));
var Txt = /^(.{3}\.){3}.{3}$/;
console.log(Txt.test('abc.def.123.456'));
console.log(Txt.test('abc.def.ghi.jkl'));
</script>
</body>
</html>
```

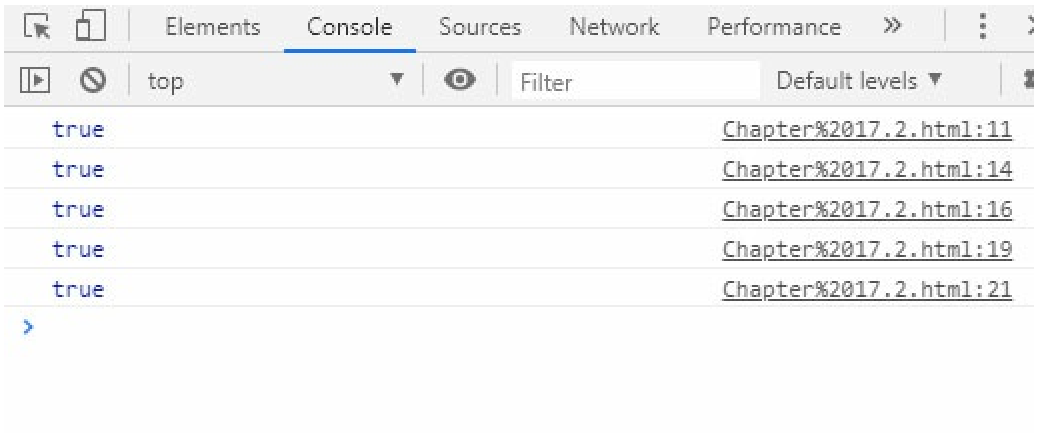
2. Save the file with an HTML extension, for example `Chapter 17.2.html`.
3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift* + *J* or *F12* in Google Chrome to open the console window.

This looks seriously complicated! It isn't. Let me break it down.

- The variable `Num` contains a regular expression pattern.
    - In this pattern, you ensure that the match spans the whole string by including`^` at the beginning and the `$` at the end of the string.
    - The `\d` in the expression matches digits.
    - `{10}` matches the previous expression exactly 10 times.
- The variable `Dt` also starts with `^` and ends with `$` to look at the whole string.
    - The first open bracket starts the first sub-expression.
    - `\d{1,2}` matches 1 digit but can have at most 2 digits.
    - A hyphen character gets matched.
    - The closing bracket ends the first sub-expression.
    - `{2}` matches the first sub-expression 2 times.
- The variable `Txt` is only allowed to have string input which consists of four three-letter groups, each separated by dots.

In each of the expressions, the test method is used. This will return either true or false, depending on whether the expression will be successful.

The result is shown in the following screenshot:



```
true                                              Chapter%2017.2.html:11
true                                              Chapter%2017.2.html:14
true                                              Chapter%2017.2.html:16
true                                              Chapter%2017.2.html:19
true                                              Chapter%2017.2.html:21
>
```

Figure 17.2: *Regular expressions in action*

All the tests are successful.

Let’s do another exercise just to make sure you understand the concepts.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

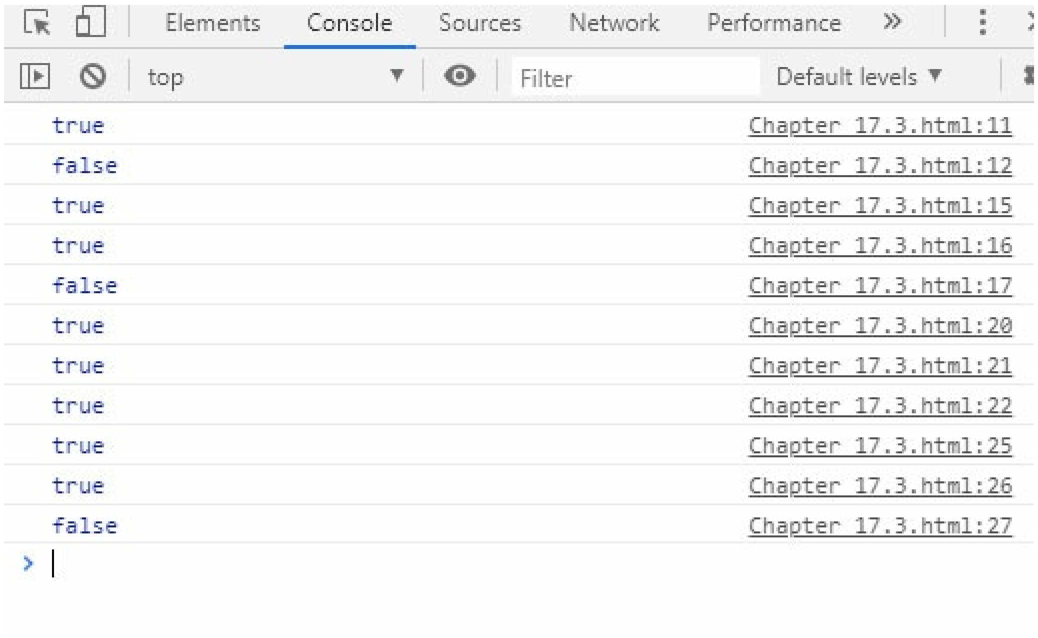
```
<html>
<body>
<h1>RegEx Example 3</h1>
<p>RegEx Example 3</p>
<script>
var regex1 = /lo{2}t/;
console.log(regex1.test(‘loot’));
console.log(regex1.test(‘lot’));
var regex2 = /[hm]ail/;
console.log(regex2.test(‘hail’));
console.log(regex2.test(‘mail’));
console.log(regex2.test(‘nail’));
var regex3 = /\d+/;
console.log(regex3.test(‘7’));
console.log(regex3.test(‘77845’));
console.log(regex3.test(‘77777723’));
var regex4 = /(green|red) apple/;
console.log(regex4.test(‘green apple’));
console.log(regex4.test(‘red apple’));
console.log(regex4.test(‘orange apple’));
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter`

`17.3.html`.

3. Double-click the file to open it in the default browser.
4. Press *Ctrl* + *Shift*+ *J* or *F12* in Google Chrome to open the console window.

Figure 17.3 shows the results of the tests:



```
true                                   Chapter 17.3.html:11
false                                  Chapter 17.3.html:12
true                                   Chapter 17.3.html:15
true                                   Chapter 17.3.html:16
false                                  Chapter 17.3.html:17
true                                   Chapter 17.3.html:20
true                                   Chapter 17.3.html:21
true                                   Chapter 17.3.html:22
true                                   Chapter 17.3.html:25
true                                   Chapter 17.3.html:26
false                                  Chapter 17.3.html:27
>
```

Figure 17.3: *More regular expression tests*

Let's have a look and see what gets tested here. The first object, `regex1` tests for two words: `loot` and `lot`. Loot succeeds, but a lot does not. Why? Well because you have indicated that you need 2 characters before the ending t.

The object `regex2` succeeds when testing for `hail` and `mail`, but not for `nail`. This is because the `n` character wasn't specified in the regular expression pattern. Only `h` and `m` were specified.

All tests for `regex3` succeed because all of them contain at least one number.

`regexp4` tests for either a `green apple` or a `red apple`. When supplied with the text `orange apple` it failed.

Conclusion

In this chapter, we have learntwhat regular expressions are and how to use them productively when searching for text or validating text. It is an important skill to know, but luckily it is not too complicated once you get the hang of the modifiers and patterns.

The next chapter is the last chapter in the *Advanced Concepts* section. You will revisit functions for a while and then dive into currying and partials.

Questions

1. Define the term regular expression.
2. Name two ways regular expressions can be constructed.
3. Name three regular expression flags.

CHAPTER 18

Partials and Currying

Introduction

JavaScript functions can be made much easier and be customized to make use of specific arguments when needed.

In this lesson, you will learn the difference between normal JavaScript functions, partials, and currying.

Structure

- Partials
- Currying

Objectives

- Understand the partial application functional programming concept.
- Understand the currying functional programming concept.

Partials

Partials or partial applications fix the value of a few of a function’s arguments without really fully evaluating the function. This produces another function of smaller arguments which binds values to one or more of those arguments as the chain of function progressed.

JavaScript contains a built-in method named bind that can work on functions with any number of arguments and can bind an arbitrary number of parameters.

To invoke the bind method, make use of the following syntax:

```
function.prototype.bind(thisValue, [argument1], [argument2], …)
```

The bind method turns the function into a new function whose implicit this parameter is this value and whose initial arguments are always as given.

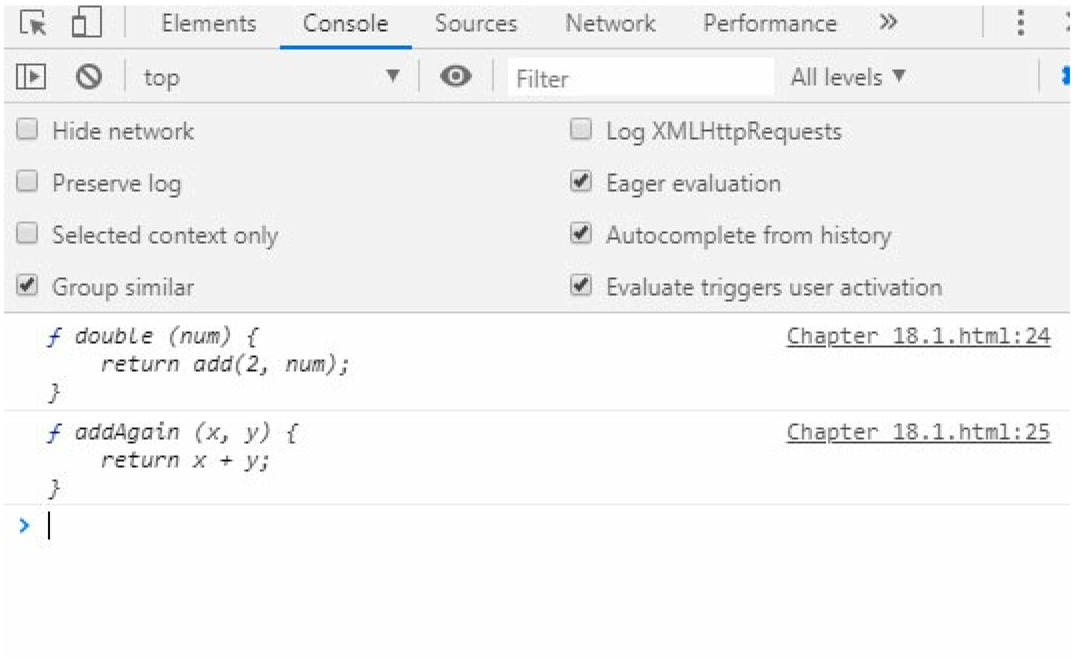
To better understand the concept of the partial, let’s do a practical exercise and dissect it piece by piece, as usual.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <body>
   <h1>Partials Example</h1>
   <p>Partials Example</p>
   <script>
   function add (x, y) {
     return x + y;
   }
   function double (num) {
     return add(2, num);
   }
   function addAgain (x, y) {
     return x + y;
   }
   var doubleAgain = addAgain.bind(null, 2);
   console.log(double);
   console.log(doubleAgain);
   </script>
   </body>
   </html>
   ```

2. Save the file with an HTML extension, for example `Chapter 18.1.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The output of the above code is shown in the following screenshot:



Hide network [ ]
Preserve log [ ]
Selected context only [ ]
Group similar [✓]
Log XMLHttpRequests [ ]
Eager evaluation [✓]
Autocomplete from history [✓]
Evaluate triggers user activation [✓]

```
ƒ double (num) {                                   Chapter 18.1.html:24
    return add(2, num);
}
ƒ addAgain (x, y) {                                Chapter 18.1.html:25
    return x + y;
}
>
```

Figure 18.1: *Partials example binding*

Function `add` is the normal way in JavaScript in calling a function and using it. The function `addAgain` demonstrates binding in action. As you can see, not all parameters have values supplied, but the JavaScript compiler doesn't complain at all.

Let's do a more complex example.

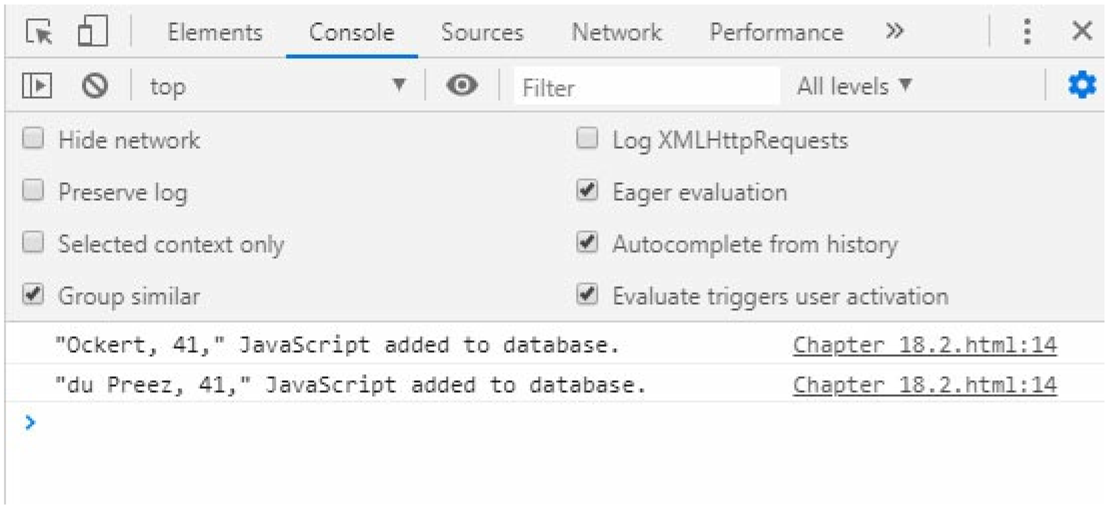
1. Open a text editor of your choice and enter the following HTML and JavaScript code

```
<html>
<body>
<h1>Partials Example</h1>
<p>Partials Example</p>
<script>
function addStudent(fullName, age, course) {
var message = '"'+ fullName + ', ' + age + '," ' +
   course + ' added to database.';
console.log(message);
}
var student_1 = addStudent.bind(null, "Ockert");
student_1(41, "JavaScript");
var student_2 = addStudent.bind(null, "du Preez", 41,
"JavaScript");
student_2();
</script>
```

```
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 18.2.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The output is shown in the following screenshot:



Hide network [ ]
Preserve log [ ]
Selected context only [ ]
Group similar [✓]
Log XMLHttpRequests [ ]
Eager evaluation [✓]
Autocomplete from history [✓]
Evaluate triggers user activation [✓]

```
"Ockert, 41," JavaScript added to database.      Chapter 18.2.html:14
"du Preez, 41," JavaScript added to database.    Chapter 18.2.html:14
>
```

Figure 18.2: *Complex function.bind example*

The original function `addStudent` contains three parameters: `fullName`, `age`, and `course`. Whatever is fed to this function gets printed in the console window with a nice message. The `student_1` variable gets bound to the `addStudent` function by only supplying one value. On the next line, the other two parameter values are supplied to the bound function. The`student_2` is also bound to the `addStudent` function, but an extra parameter is supplied.

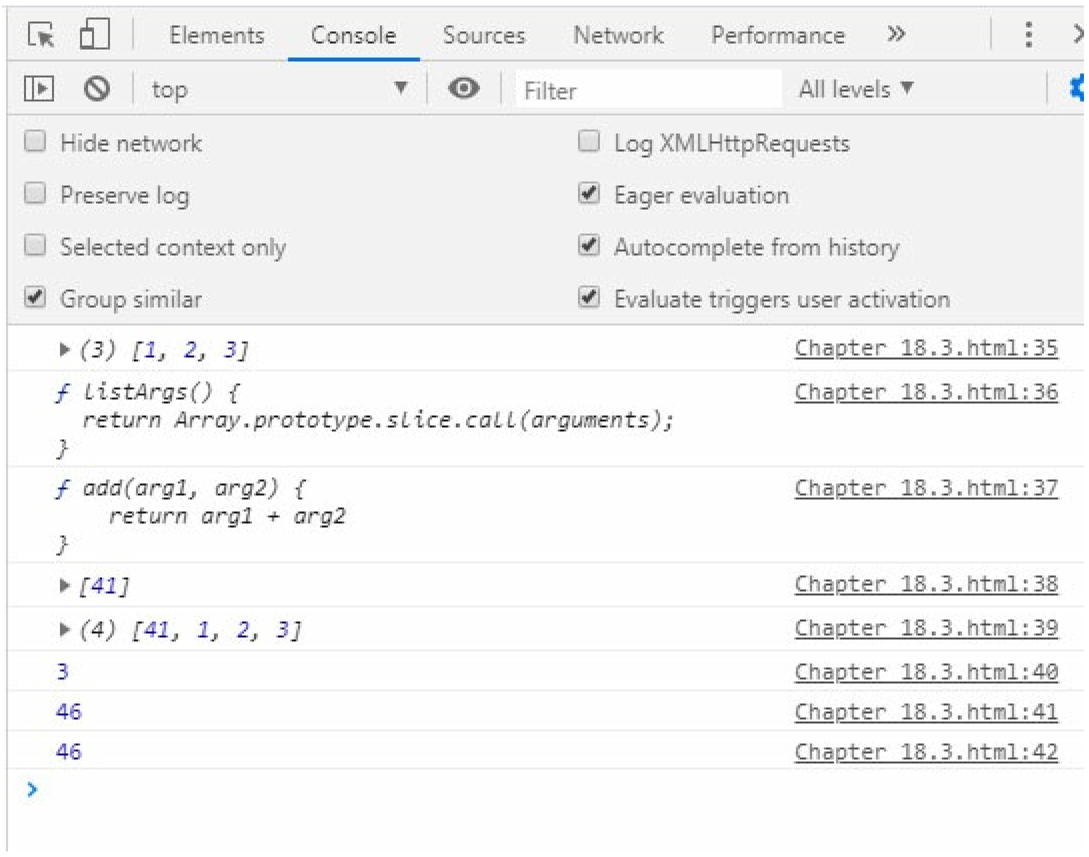
Let's do an even more complex example.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Partials Example</h1>
<p>Partials Example</p>
<script>
function listArgs() {
 return Array.prototype.slice.call(arguments);
}
function add(arg1, arg2) {
  return arg1 + arg2
}
var list_1 = listArgs(1, 2, 3);
var result_1 = add(1, 2);
var list_2 = listArgs.bind(null, 41);
var list_3 = add.bind(null, 41);
var list_4 = list_2();
var list_5 = list_2(1, 2, 3);
var result_2 = list_3(5);
var result_3 = list_3(5, 10);
console.log(list_1);
console.log(list_2);
console.log(list_3);
console.log(list_4);
console.log(list_5);
console.log(result_1);
console.log(result_2);
console.log(result_3);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 18.3.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The output is shown in the following screenshot:



```
☐ Hide network                ☐ Log XMLHttpRequests
☐ Preserve log                ✓ Eager evaluation
☐ Selected context only       ✓ Autocomplete from history
✓ Group similar               ✓ Evaluate triggers user activation
```

```
▶ (3) [1, 2, 3]                                        Chapter 18.3.html:35
ƒ listArgs() {                                         Chapter 18.3.html:36
  return Array.prototype.slice.call(arguments);
}
ƒ add(arg1, arg2) {                                    Chapter 18.3.html:37
    return arg1 + arg2
}
▶ [41]                                                 Chapter 18.3.html:38
▶ (4) [41, 1, 2, 3]                                    Chapter 18.3.html:39
3                                                      Chapter 18.3.html:40
46                                                     Chapter 18.3.html:41
46                                                     Chapter 18.3.html:42
>
```

Figure 18.3: *Arrays and function binding*

Currying

Currying transforms a function with multiple arguments into a sequence of nesting functions. It keeps returning a new function that expects the current argument inline until all the arguments are exhausted.

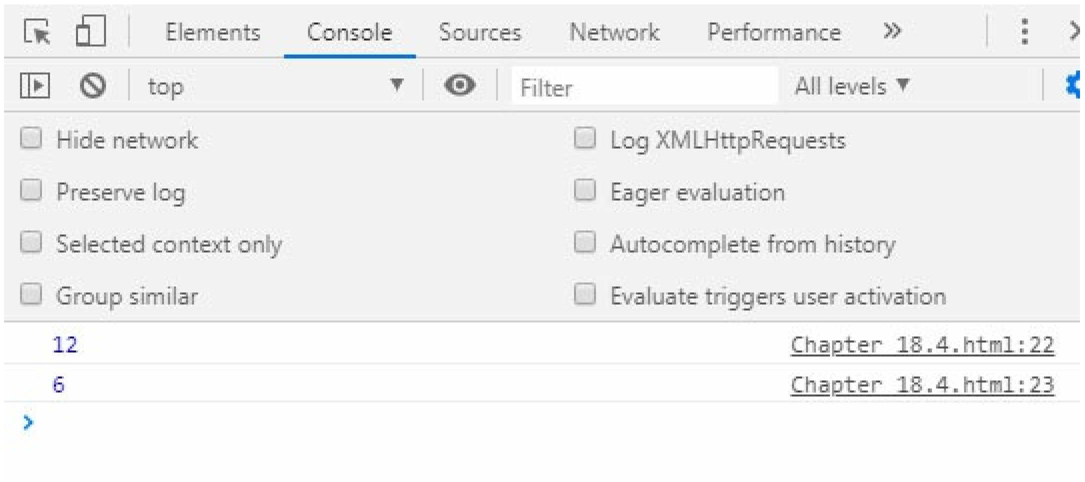
Here is a small example.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Currying Example</h1>
<p>Currying Example</p>
<script>
function add_1(a, b, c) {
  return a + b + c;
}
function add_2(a) {
    return (b) => {
     return (c) => {
      return a + b + c
}
}
}
console.log(add_1(2,4,6));
console.log(add_2(1)(2)(3))
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 18.4.html`.
3. Double-click the file to open it in the default browser.
4. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window.

The output of the above code is shown in the following screenshot:



```
12                                  Chapter 18.4.html:22
6                                   Chapter 18.4.html:23
>
```

Figure 18.4: *Currying example*

The first function, `add_1,` is an ordinary JavaScript function and it gets output to the console. The function `adds_2` is an example of currying in action. It splits the function into more functions, each accepting an argument. In this case, there are 3 arguments, so there will be 3 inline functions.

Conclusion

In this last chapter in the *AdvancedConcepts* section, we have learnt how to make use of functional programming features in JavaScript. You learned how to make use of partial applications to manipulate a function’s arguments according to your needs. Lastly, you saw how currying makes use of inline functions.

In the next final chapter, we will introduce you to various languages and language variants based on JavaScript such as AJAX, jQuery, and JSON.

Questions

1. Define the termpartials.
2. What is currying?

SECTION V

JAVASCRIPT AND FRIENDS

CHAPTER 19

JavaScript and Other Languages

Introduction

JavaScript has been around for a long time and will be even longer. Like any programming language, it has influenced a lot of different languages.

In this lesson, you will learn how to use JavaScript with AJAX, jQuery, and JSON. Later on in the lesson, you will have a quick introduction into languages influenced by JavaScript such as TypeScript, ActionScript, and CoffeeScript.

Structure

- jQuery and JavaScript
- JSON and JavaScript
- AJAX and JavaScript
- Languages based on or influenced by JavaScript

Objectives

- Understand what jQuery is and how to use it.
- Understand what JSON and AJAX are and how to use them in JavaScript.
- See how ActionScript, CoffeeScript, and TypeScript work.

Other languages

Throughout this book, you have been using JavaScript with HTML because HTML is the foundation for any web page. if you can recall, in *Chapter 1: Overview of the Power of JavaScript and Its Purpose*, I have spoken about the history of JavaScript and why it became such a strong language that developers simply cannot work without.

Another language such as **CSS** (**Cascading Style Sheets**) also comes to mind as a language that any webpage simply cannot do without. But, there are many languages out there. There are numerous scripting languages and numerous variants of languages. Each with their own flavor and purpose.

jQuery and JavaScript

jQuery was founded by *John Resig* in 2006.jQuery is a cross-platform JavaScript library. It is a collection of JavaScript code packaged together that can be accessed quicker and easier than JavaScript. Its main use is to simplify HTML DOM tree traversal and manipulation, plus event handling, CSS animation, and AJAX (I will speak of AJAX a bit later). jQuery also provides capabilities to create plug-ins on top of the JavaScript library.

Some important differences between JavaScript and jQuery are included in the table below:

JavaScript	**jQuery**
Weakly typed dynamic language.	Rich and lightweight JavaScript library.
Write all coding from scratch.	Contains pre-written functions.
Developers have to consider browser compatibility when developing.	jQuery is cross-browser compatible.
No additional plugins needed to run JavaScript.	jQuery library script link must be included on the web page.
Long lines of code.	Less coding.
Difficult to learn.	Relatively easier to learn.

Table 19.1: *JavaScript versus jQuery*

Let’s quickly have a look at the differences in coding by doing a few exercises.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

   ```
   <html>
   <head>
   <script src=
   “https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.m
   ```

```
</script>
</head>
<body>
<h1>jQuery versus JavaScript 1</h1>
<p>jQuery versus JavaScript 1</p>
<input id="button1" type="button"value="click Me!" />
<script>
   $('#button1').click(function() {
    alert("BpB");
   });
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 19.1.html`.
3. Double-click the file to open it in the default browser.

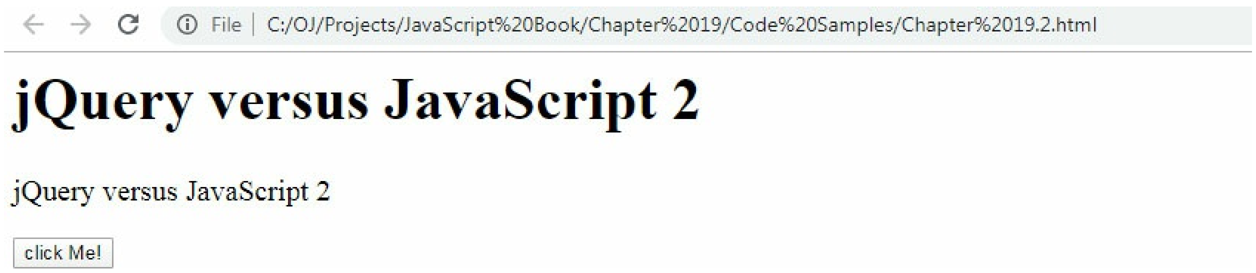
The first part of this exercise demonstrated the jQuery way of doing things; now, let's do the same in pure JavaScript.

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>jQuery versus JavaScript 2</h1>
<p>jQuery versus JavaScript 2</p>
<input id="button1" type="button"value="click Me!"/>
<script>
   document.getElementById("button1").addEventListener('clic
   function(){
    alert("BpB");
   });
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 19.2.html`.
3. Double-click the file to open it in the default browser.

Both the above exercises do the same thing. The first one is done in jQuery, and the second one is done with JavaScript. By opening either one of the above exercises, you will see a screen similar to *Figure 19.1* underneath:


C:/OJ/Projects/JavaScript%20Book/Chapter%2019/Code%20Samples/Chapter%2019.2.html

jQuery versus JavaScript 2

jQuery versus JavaScript 2

Figure 19.1: *jQuery versus JavaScript*

Let’s have a look at the code. Considering that this is just a small example demonstrating the differences between JavaScript and jQuery, both exercises have relatively little code. Let me explain the crux.

In the previous first program, you wrote the following:

```
<head>
<script src=
“https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js
</script>
</head>
```

This code imports a jQuery library into your page. Because you put it into the head section of your HTML document, it is part of the complete page, and the methods inside the library get loaded before anything else. In JavaScript, you do not need to include a special library to do the task at hand (in this example).

You then created a normal input button and did the following:

```
<script>
    $(‘#button1’).click(function() {
     alert(“BpB”);
    });
</script>
```

This attaches a click event handler to the button which shows an alert box with the text `BpB`. Very simple and very streamlined.

In the previous second program, you didn’t have to import a library. You also created a button input and did the following:

```
<script>
    document.getElementById(“button1”).addEventListener(‘click’,
    function(){
     alert(“BpB”);
    });
</script>
```

You first had to get the object named `button1` through the `getElementById` method. After that, you had to manually attach an event handler to the button by adding a listener which listens for events. This also then shows an alert box with the text `BpB`

JSON and JavaScript

JSON stands for **JavaScript Object Notation**. It is not actually a different programming language at all, but many a newbie confuse JSON and JavaScript. JSON is a lightweight format for storing and transporting data. It is self-describing and is used mostly to send data from a server to a web page.

If you look at the name, you will see JavaScript Object Notation, now what is the real difference between JavaScript objects and JSON? Let me answer it.

JavaScript objects were covered in depth in *Chapter 2: JavaScript Objects*, but here are a few highlights.

JavaScript objects

An object is something that has properties and methods. Properties describe an object. Methods are the actions which that object can take. Basically, everything can be an object in JavaScript. All JavaScript values, except primitives, are objects. Before we go into detail about Built-in objects, let's explore primitives quickly.

The Object data type

Objects are variables too, but unlike with primitives, objects can store more than one value in it. Objects store keyed collections of data and complex entities. The information inside an object is stored as key-value pairs. The key is usually the property name. The value is the property's value, for example:

```
var objStudent = {
Name : “Ockert”,
Course : “JavaScript”,
Year : 2019
};
```

In the above code segment, a new JavaScript object called `objStudent` is created. `objStudent` contains three properties, namely: `Name`, `Course`, and `Year`.

Creating JavaScript objects

Now that we know a bit more about objects let's have a look at how to create them. Objects can be created using any of the following ways:

- Object literal syntax
- The `new` keyword
- `Object.create()`
- `Object.assign()`
- ES6 classes

Let me explain them one by one.

Object literal syntax

This method is quite easy to understand and quick to do. Here is an example:

```
var objCar = {
make : “BMW”,
model : “i8 Roadster”,
price : 2329300
};
```

This example may look almost familiar. If you look closely at the previous code segment in the Object data type section, you will notice that both code examples have the same similarities. The keyword var creates the object. The curly braces contain the properties and values separated with a colon, and each property is separated with a comma.

The object created now is named `objCar`. It contains the properties make, model and price. To make use of the `objCar` object, you can have a statement like:

```
console.log(objCar); //Outputs {make: “BMW”, model: “i8
Roadster”, price: “2329300”}
console.log(objCar.make); //Outputs BMW
console.log(objCar.model); //Outputs i8 Roadster
console.log(objCar.price); //Outputs 2329300
```

The ‘new’ keyword

Using the `new` keyword to create objects resembles the way objects are created in class-based programming languages, like C# or Java. There are 2 ways to use the `new` keyword to create objects; they are:

- **Using ‘new’ with in-built Object constructor functions**: A constructor, in class-based object-oriented programming, is a special type of subroutine called to create an object. A constructor prepares the new object for use by initializing the object’s members. To create a new object, you must use the ‘`new`’ keyword with the `Object()` constructor, like this:

  ```
  var house = new Object();
  ```

  This creates and initializes a new object named `house`. We can add properties to this object by using code like the following:

  ```
  house.bedrooms = 4;
  house.bathrooms = 2;
  house.garages = 2;
  house.hasSwimmingPool = false;
  house.hasOutsideWalls = true;
  ```

  As you can see, creating properties for an object, which was created using the `Object()` constructor, can become quite a tedious task, and involves a lot of typing. The house object we created above has a bedrooms property, bathrooms, garages, as well as Boolean properties indicating the presence of a swimming pool, or outside walls.

  What if we had to create hundreds of house objects? Let’s see how we can make this code a bit shorter by using a user-defined constructor function.

- **Using ‘new’ with user-defined constructor functions**: When creating a user-defined constructor function, you basically create a template for the object’s properties and methods. You create the function once, and just feed the necessary properties to it, instead of manually entering each property and each setting as shown earlier in the Using `new` with in-built Object constructor functions section.

  Let’s create the user-defined constructor using the house object as an example again:

  ```
  function House(_bed, _bath, _garage, _pool, _walls) {
  ```

```
  this.bedRooms = _bed;
  this.bathRooms = _bath;
  this.garages = _garage;
  this.hasSwimmingPool = _pool;
  this.hasOutsideWalls = _walls;
}
```

The name of the constructor function is `House`. Inside the brackets there are five parameters: (`_bed`, `_bath`, `_garage`, `_pool`, and `_walls`. These will be used to pass information to the House function. Inside the constructor function, each property gets set to its own parameter. Remember, this is simply a template, there are no values yet.

Now that we have the constructor function, it is easy to create objects based on it. Here is how to:

```
var houseOne = new House(4, 2, 2, false, true);
var houseTwo = new House(3, 1, 1, true, false);
```

The `houseOne` variable above is a new `House` object. It has 4 bedrooms, 2 bathrooms, 2 garages, no swimming pool and it has outside walls. The `houseTwo` variable, in contrast, has 3 bedrooms, 1 bathroom, 1 garage, a swimming pool and it doesn't have outside walls.

Creating JSON objects

JSON objects are written inside curly braces and can contain multiple name/value pairs, as shown below:

```
{“firstName”:”Ockert”, “lastName”:”du Preez”}
```

You can take it a step further and include more `firstName`and `lastName` to one `Student` object, for example:

```
{
“Student”:[
 {“firstName”:”Ockert”, “lastName”:”du Preez”},
 {“firstName”:”William”, “lastName”:”Shakespeare”},
 {“firstName”:”Stephen”, “lastName”:”King”}
]
}
```

Rules for valid JSON

The following are all allowed in JSON format

- Empty objects and empty arrays.
- Strings may contain any Unicode characters.
- Null is a valid JSON value.
- All object properties should always be double-quoted.
- Number values must be in decimal format.
- No trailing commas are allowed on arrays.

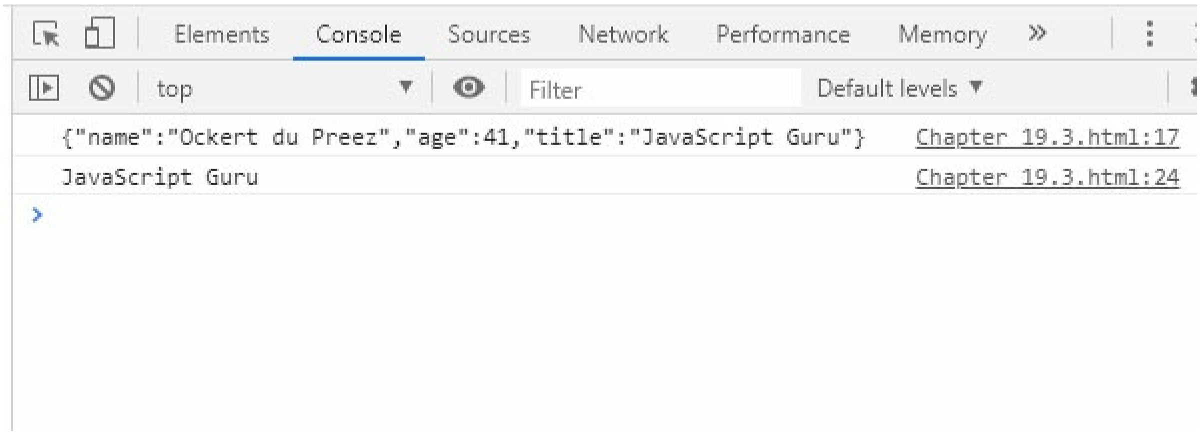
Let’s do a quick exercise on encoding JSON and decoding JSON through JavaScript:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<html>
<body>
<h1>Encoding and Decoding JSON</h1>
<p>Encoding and Decoding JSON</p>
<script>
const encodeData = {
 name: “Ockert du Preez”,
 age: 41,
 title: “JavaScript Guru”
}
const encodeStr = JSON.stringify(encodeData);
console.log(encodeStr);
const decodeStr = ‘{
“name”:”Ockert du Preez”,
“age”:41,
“title”:”JavaScript Guru”
}’;
const decodeData = JSON.parse(decodeStr);
console.log(decodeData.title);
</script>
</body>
</html>
```

2. Save the file with an HTML extension, for example `Chapter 19.3.html`.
3. Double-click the file to open it in the default browser. If the browser is Google Chrome, press *Ctrl* + *Shift* + *J* to open the console window:

The output in the Console window is shown next



```
{"name":"Ockert du Preez","age":41,"title":"JavaScript Guru"}    Chapter 19.3.html:17
JavaScript Guru                                                  Chapter 19.3.html:24
```

Figure 19.2: *Encoding and decoding JSON*

Let me dissect what happens here.

First, a JavaScript object called `encodeData` is created and given values such as name, age and title. Then, a variable named `encodeString` makes use of the `stringify` method in JSON to convert the JavaScript object to a JSON object.

Second is basically the opposite. You supplied a JSON object to the parse method of JSON to convert it to an ordinary JavaScript object.

AJAX and JavaScript

AJAX is not a programming language. AJAX stands for Asynchronous JavaScript and XML and makes use of:

- A browser built-in `XMLHttpRequest` object that requests data from a web server.
- JavaScript and HTML DOM that displays or uses the data.

The real power of AJAX is that it allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes, meaning it is possible to update parts of a web page, without the need of reloading the whole page.

Here’s a breakdown on how AJAX works. An event occurs on a web page:

- JavaScript creates an `XMLHttpRequest` object.
- The `XMLHttpRequest` object sends a request to a web server.
- The server processes the request.

- The server sends a response back to the web page.
- JavaScript read the response.
- JavaScript performs the appropriate action.

The next little exercise, you will do demonstrates how to use AJAX with JavaScript and another scripting language called **PHP**. PHP is a server scripting language, so keep in mind that the PHP file must exist on a server for this exercise to work completely:

1. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<?php
if(isset($_GET[“studentFirstName”]) &&
isset($_GET[“studentLastName”])) {
  $fname = htmlspecialchars($_GET[“studentFirstName”]);
  $lname = htmlspecialchars($_GET[“studentLastName”]);
  $studentFullName = $studentFirstName.“ ”.$studentLastName;

  echo “Hello, $studentFullName! Welcome!”;
} else {
  echo “Hi there, stranger! Welcome!”;
}
?>
```

2. Save the file with an HTML extension, for example `greet.php`.
3. Open a text editor of your choice and enter the following HTML and JavaScript code:

```
<!DOCTYPE html>
<head>
<script>
function showStudentName() {
 var request = new XMLHttpRequest();
  request.open(“GET”, “greet.php?
  studentFirstName=Ockert&studentLastName=du Preez”);
request.onreadystatechange = function() {
    if(this.readyState === 4 && this.status === 200) {
   document.getElementById(“result”).innerHTML =
   this.responseText;
   }
};
 request.send();
}
</script>
```

```
</head>
<body>
<h1>AJAX Example</h1>
<p>AJAX Example</p>
<div id=”result”>
<p>Result will be displayed here</p>
</div>
<button type=”button” onclick=”showStudentName()”>Show Full
Name</button>
</body>
</html>
```

4. Save the file with an HTML extension, for example `Chapter 19.4.html`.
5. Double-click the file to open it in the default browser. As mentioned, if the PHP file is saved on the server and everything is set up properly, you will not get errors when loading this document.

What happened here? The `greet.php` is a PHP file that exists on a server. It tests to see if the `studentFirstName` and `studentLastname` fields contain information. If they do, their values get stored inside two PHP variable objects `fname` and `lname`, respectively.

The `studentFullName` PHP variable consists of a combination of the `fname` and `lname` variables separated by a space character. If values were supplied, a message greeting the student by his/her full nameswould appear; else, a message greeting a stranger will appear.

In the file `Chapter19.4html`you created a function named `showStudentName`. The object is function is to call the logic inside the `greet.php` file to combine the given first name and last name. If the function is successful, it will display the full name inside the `div` named result.

Languages based on or influenced by JavaScript

The following languages are a few of the many languages that were influenced in some way by JavaScript:

- ActionScript
- AtScript
- CoffeeScript
- Dart

- JScript .NET
- LiveScript
- Objective-J
- Opa
- QML
- Raku
- TypeScript

I’m quickly going to explain some of these.

ActionScript

ActionScript is an object-oriented programming language developed by *Macromedia Inc*. Here is a small example on how to create an ActionScript function:

1. Open a text editor of your choice and enter the following code

   ```
   package{
    import flash.display.Sprite;
    public class Main extends Sprite{
     public function Main(){
      showGreeting();
     }
     function showGreeting():void {
      trace(“Hello ActionScript”);
     }
    }
   }
   ```

2. Save the file with an HTML extension, for example `actionscript_ex.as`.

Quite simple and easy to follow.

CoffeeScript

CoffeeScript compiles to JavaScript. CoffeeScript adds syntactic sugar inspired by Ruby, Python and Haskell to enhance JavaScript’s readability.

Let’s assume you have the following function in Coffeescript:

```
add =(a,b) ->
 c=a+b
 console.log “Answer is: “+c
```

It’s equivalent in JavaScript is:

```
(function() {
 var add;
 add = function(a, b) {
  var c;
  c = a + b;
  return console.log(“Answer is: “ + c);
 };
}).call(this);
```

Although CoffeeScript makes use of less code, it can be a bit more difficult to follow. Both functions essentially do the same which is to add two numbers and provide an answer.

TypeScript

The command-line TypeScript compiler can be installed as a Node.js package, by navigating to the following URL:

https://www.typescriptlang.org/index.html#download-links

Here’s a small example on how to create a class in TypeScript:

1. Open a text editor of your choice and enter the following code:

```
class Car {
 make: String;
 model: String;
 doors: Number;
 isAutomatic: Boolean;
 constructor(make: make, model: String, doors: Number,
 isAutomatic: Boolean) {
  this.make = make;
  this.model = model;
  this.doors = doors;
  this.isAutomatic = isAutomatic;
 }
 make(): void {
console.log(‘Car Make and model: ${this.make},
${this.model}. Number of doors: ${this.doors}’ );
}
}
let newCar = new Car(‘Lamborghini’, ‘Aventador’, 2, false);
```

```
newCar.make();
```

2. Save the file with an HTML extension, for example `Car.ts`.

Because TypeScript can extend JavaScript features, this looks quite similar to an ordinary JavaScript class.

Conclusion

In this chapter, we saw how to use JavaScript with other variants and languages influenced by it. jQuery, AJAX, and JSON are topics that could fill books on their own. You have also seen what other languages look like and how they compare to JavaScript.

Questions

1. What is jQuery?
2. What is JSON?
3. What is AJAX?
4. Name two languages influenced by JavaScript
5. Is JavaScript still difficult?