
Introduction to Javascript

QITIAN LIAO

UNIVERSITY OF CALIFORNIA, BERKELEY

Contents

1	Introduction to Javascript	1
1.1	What is Javascript?	1
1.2	Console	1
1.3	Comments	1
1.4	Data Types	2
1.5	Arithmetic Operators	3
1.6	String Concatenation	3
1.7	Properties	4
1.8	Methods	4
1.9	Built-in Objects	5
1.10	Review	5
2	Variables	7
2.1	Variables	7
2.2	Create a Variable: var	7
2.3	Create a Variable: let	8
2.4	Create a Variable: const	8
2.5	Mathematical Assignment Operators	9
2.6	The Increment and Decrement Operator	9
2.7	String Concatenation with Variables	10
2.8	String Interpolation	10
2.9	typeof operator	10
2.10	Review Variables	11
3	Conditional Statements	12
3.1	What are Conditional Statements?	12
3.2	If Statement	12
3.3	If...Else Statements	12
3.4	Comparison Operators	13
3.5	Logical Operators	14
3.6	Truthy and Falsy	15
3.7	Truthy and Falsy Assignment	16
3.8	Ternary Operator	16
3.9	Else If Statements	17
3.10	The switch keyword	17
3.11	Review	18
4	Functions	20
4.1	What are Functions?	20
4.2	Function Declarations	21
4.3	Calling a Function	22
4.4	Parameters and Arguments	22
4.5	Default Parameters	23
4.6	Return	24
4.7	Helper Functions	25

4.8	Function Expressions	26
4.9	Arrow Functions	26
4.10	Concise Body Arrow Functions	27
4.11	Review Functions	28
5	Scope	30
5.1	Scope	30
5.2	Blocks and Scope	30
5.3	Global Scope	30
5.4	Block Scope	31
5.5	Scope Pollution	31
5.6	Practice Good Scoping	32
5.7	Review: Scope	33
6	Arrays	34
6.1	Arrays	34
6.2	Create an Array	34
6.3	Accessing Elements	35
6.4	Update Elements	35
6.5	Arrays with let and const	36
6.6	The .length property	36
6.7	The .push() Method	36
6.8	The .pop() Method	37
6.9	More Array Methods	37
6.9.1	The .shift() Method	38
6.9.2	The .unshift() Method	38
6.9.3	The .slice() Method	38
6.10	Arrays and Functions	38
6.11	Nested Arrays	39
6.12	Review Arrays	39
7	Loops	41
7.1	Loops	41
7.2	The For Loop	41
7.3	Looping in Reverse	42
7.4	Looping through Arrays	42
7.5	Nested Loops	43
7.6	The While Loop	43
7.7	Do...While Statements	44
7.8	The break Keyword	45
7.9	Review	45
8	Higher-Order Functions	46
8.1	Introduction	46
8.2	Functions as Data	46
8.3	Functions as Parameters	47
8.4	Review	48

9	Iterators	49
9.1	Introduction to Iterators	49
9.2	The .forEach() Method	49
9.3	The .map() Method	50
9.4	The .filter() Method	51
9.5	The .findIndex() Method	51
9.6	The .reduce() Method	52
9.7	Iterator Documentation	53
9.8	More Iterators	53
9.8.1	The .some() Method	53
9.8.2	The .every() Method	54
9.9	Review	54
10	Objects	55
10.1	Introduction to Objects	55
10.2	Creating Object Literals	55
10.3	Accessing Properties	56
10.4	Bracket Notation	56
10.5	Property Assignment	57
10.6	Methods	58
10.7	Nested Objects	58
10.8	Pass By Reference	59
10.9	Looping Through Objects	60
10.10	Review	61
11	Advanced Objects	63
11.1	Advanced Objects Introduction	63
11.2	The this Keyword	63
11.3	Arrow Functions and this	64
11.4	Privacy	64
11.5	Getters	65
11.6	Setters	66
11.7	Factory Functions	67
11.8	Property Value Shorthand	67
11.9	Destructured Assignment	68
11.10	Built-in Object Methods	69
11.10.1	The Object.keys() Method	69
11.10.2	The Object.entries() Method	69
11.10.3	The Object.assign() Method	70
11.11	Review	70
12	Classes	71
12.1	Introduction to Classes	71
12.2	Constructor	72
12.3	Instance	72
12.4	Methods	73
12.5	Method Calls	73
12.6	Inheritance	74

12.7 Static Methods	79
12.8 Review: Classes	80
13 Browser Compatibility and Transpilation	81
13.1 Introduction	81
13.2 caniuse.com	81
13.3 Why ES6?	81
13.4 Transpilation With Babel	82
13.5 npm init	82
13.6 Install Node Packages	83
13.7 .babelrc	83
13.8 Babel Source Lib	84
13.9 Build	85
13.10Review	85
14 Intermediate JavaScript Modules	87
14.1 Hello Modules	87
14.2 module.exports	87
14.3 require()	88
14.4 module.exports II	88
14.5 export default	89
14.6 import	89
14.7 Named Exports	90
14.8 Named Imports	90
14.9 Export Named Exports	90
14.10Import Named Imports	91
14.11Export as	91
14.12Import as	91
14.13Combining Export Statements	92
14.14Combining Import Statements	93
14.15Review	93
15 JavaScript Promises	94
15.1 Introduction	94
15.2 What is a Promise?	94
15.3 Constructing a Promise Object	95
15.4 The Node setTimeout() Function	96
15.5 Consuming Promises	96
15.6 The onFulfilled and onRejected Functions	97
15.7 Using catch() with Promises	98
15.8 Chaining Multiple Promises	99
15.9 Avoiding Common Mistakes	101
15.10Using Promise.all()	102
15.11Review	103
16 Async Await	104
16.1 Introduction	104
16.2 The async Keyword	104

16.3 The await Operator	105
16.4 Writing async Functions	105
16.5 Handling Dependent Promises	106
16.6 Handling Errors	107
16.7 Handling Independent Promises	108
16.8 Await Promise.all()	109
16.9 Review	109
17 Requests	111
17.1 Introduction to Requests	111

1 Introduction to Javascript

1.1 What is Javascript?

Last year, millions of learners from our community started with JavaScript. Why? JavaScript is primarily known as the language of most modern web browsers, and its early quirks gave it a bit of a bad reputation. However, the language has continued to evolve and improve. JavaScript is a powerful, flexible, and fast programming language now being used for increasingly complex web development and beyond!

Since JavaScript remains at the core of web development, it's often the first language learned by self-taught coders eager to learn and build. We're excited for what you'll be able to create with the JavaScript foundation you gain here. JavaScript powers the dynamic behavior on most websites, including this one.

In this lesson, you will learn introductory coding concepts including data types and built-in objects—essential knowledge for all aspiring developers. Make sure to take notes and pace yourself. This foundation will set you up for understanding the more complex concepts you'll encounter later.

1.2 Console

The console is a panel that displays important messages, like errors, for developers. Much of the work the computer does with our code is invisible to us by default. If we want to see things appear on our screen, we can print, or log, to our console directly.

In JavaScript, the `console` keyword refers to an object, a collection of data and actions, that we can use in our code. Keywords are words that are built into the JavaScript language, so the computer will recognize them and treats them specially.

One action, or method, that is built into the `console` object is the `.log()` method. When we write `console.log()` what we put inside the parentheses will get printed, or logged, to the console. It is going to be very useful for us to print values to the console, so we can see the work that we are doing.

```
console.log(5);
```

This example logs 5 to the console. The semicolon denotes the end of the line, or statement. Although in JavaScript your code will usually run as intended without a semicolon, we recommend learning the habit of ending each statement with a semicolon so you never leave one out in the few instances when they are required.

You'll see later on that we can use `console.log()` to print different kinds of data.

1.3 Comments

Programming is often highly collaborative. In addition, our own code can quickly become difficult to understand when we return to it. For these reasons, it's often useful to leave notes in our code for other developers or ourselves.

As we write JavaScript, we can write comments in our code that the computer will ignore as our program runs. These comments exist just for human readers. Comments can explain what the code is doing, leave

instructions for developers using the code, or add any other useful annotations. There are two types of code comments in JavaScript:

1. A *single line comment* will comment out a single line and is denoted with two forward slashes `//` preceding it.

```
// Prints 5 to the console
console.log(5);
```

You can also use a single line comment to comment after a line of code:

```
console.log(5); // Prints 5
```

2. A *multi-line comment* will comment out multiple lines and is denoted with `/*` to begin the comment, and `*/` to end the comment.

```
/*
This is all commented
console.log(10);
None of this is going to run!
console.log(99);
*/
```

You can also use this syntax to comment something out in the middle of a line of code:

```
console.log(/*IGNORED!*/ 5); // Still just prints 5
```

1.4 Data Types

Data types are the classifications we give to the different kinds of data that we use in programming. In JavaScript, there are seven fundamental data types:

1. *Number*: Any number, including numbers with decimals: `4`, `8`, `1516`, `23.42`.
2. *String*: Any grouping of characters on your keyboard (letters, numbers, spaces, symbols, etc.) surrounded by single quotes: `'...'` or double quotes `"..."`. Though we prefer single quotes. Some people like to think of string as a fancy word for text. If there is a single quote character, `'`, in our string, we can use double quotes around the string to make sure character prints.
3. *Boolean*: This data type only has two possible values— either `true` or `false` (without quotes). It's helpful to think of booleans as on and off switches or as the answers to a “yes” or “no” question.
4. *Null*: This data type represents the intentional absence of a value, and is represented by the keyword `null` (without quotes).
5. *Undefined*: This data type is denoted by the keyword `undefined` (without quotes). It also represents the absence of a value though it has a different use than `null`.
6. *Symbol*: A newer feature to the language, symbols are unique identifiers, useful in more complex coding. No need to worry about these for now.
7. *Object*: Collections of related data.

The first 6 of those types are considered *primitive data types*. They are the most basic data types in the language. *Objects* are more complex, and you will learn much more about them as you progress through

JavaScript. At first, seven types may not seem like that many, but soon you will observe the world opens with possibilities once you start leveraging each one. As you learn more about objects, you'll be able to create complex collections of data. But before we do that, let us first get comfortable with strings and numbers!

```
console.log("Location of The Metropolitan: 2301 Durant Ave, Berkeley");  
console.log(40);
```

In the example above, we first printed a string. Our string is not just a single word; it includes both capital and lowercase letters, spaces, and punctuation. Next, we printed the number 40, notice we did not use quotes.

1.5 Arithmetic Operators

Basic arithmetic often comes in handy when programming. An *operator* is a character that performs a task in our code. JavaScript has several built-in *arithmetic operators*, that allow us to perform mathematical calculations on numbers. These include the following operators and their corresponding symbols:

1. Add: `+`
2. Subtract: `-`
3. Multiply: `*`
4. Divide: `/`
5. Remainder: `%`

The first four work how you might guess:

```
console.log(3 + 4); // Prints 7  
console.log(5 - 1); // Prints 4  
console.log(4 * 2); // Prints 8  
console.log(9 / 3); // Prints 3
```

Note that when we `console.log()` the computer will evaluate the expression inside the parentheses and print that result to the console. If we wanted to print the characters `3 + 4`, we would wrap them in quotes and print them as a string.

```
console.log(11 % 3); // Prints 2  
console.log(12 % 3); // Prints 0
```

The remainder operator, sometimes called *modulo*, returns the number that remains after the right-hand number divides into the left-hand number as many times as it evenly can: `11 % 3` equals 2 because 3 fits into 11 three times, leaving 2 as the remainder.

1.6 String Concatenation

Operators are not just for numbers! When a `+` operator is used on two strings, it appends the right string to the left string:

```
console.log("hi" + "ya"); // Prints "hiya"  
console.log("wo" + "ah"); // Prints "woah"  
console.log("I love to " + "code."); // Prints "I love to code."
```

This process of appending one string to another is called *concatenation*. Notice in the third example we had to make sure to include a space at the end of the first string. The computer will join the strings exactly, so we needed to make sure to include the space we wanted between the two strings.

```
console.log("front " + "space"); // Prints "front space"
console.log("back" + " space"); // Prints "back space"
console.log("no" + "space"); // Prints "nospace"
console.log("middle" + " " + "space"); // Prints "middle space"
```

Just like with regular math, we can combine, or chain, our operations to get a final result:

```
console.log("One" + ", " + "two" + ", " + "three!");
// Prints "One, two, three!"
```

1.7 Properties

When you introduce a new piece of data into a JavaScript program, the browser saves it as an instance of the data type. Every string instance has a property called `length` that stores the number of characters in that string. You can retrieve property information by appending the string with a period and the property name:

```
console.log("Hello".length); // Prints 5
```

The `.` is another operator! We call it the dot operator. In the example above, the value saved to the `length` property is retrieved from the instance of the string, `'Hello'`. The program prints `5` to the console, because `Hello` has five characters in it.

1.8 Methods

Remember that methods are actions we can perform. JavaScript provides a number of string methods. We *call*, or use, these methods by appending an instance with:

- a period (the dot operator)
- the name of the method
- opening and closing parentheses

E.g. `'example string'.methodName()`.

The syntax looks familiar. When we use `console.log()` we're calling the `.log()` method on the `console` object. Let us see `console.log()` and some real string methods in action!

```
console.log("hello".toUpperCase()); // Prints "HELLO"
console.log("Hey".startsWith("H")); // Prints true
```

Let's look at each of the lines above:

- On the first line, the `.toUpperCase()` method is called on the string instance `'hello'`. The result is logged to the console. This method returns a string in all capital letters: `'HELLO'`.
- On the second line, the `.startsWith()` method is called on the string instance `'Hey'`. This method also accepts the character `'H'` as an input, or argument, between the parentheses. Since the string `'Hey'` does start with the letter `'H'`, the method returns the boolean `true`.

You can find a list of built-in string methods in the JavaScript documentation. Developers use documentation as a reference tool. It describes JavaScript's keywords, methods, and syntax.

1.9 Built-in Objects

In addition to `console`, there are other objects built into JavaScript. Down the line, you'll build your own objects, but for now these "built-in" objects are full of useful functionality. For example, if you wanted to perform more complex mathematical operations than arithmetic, JavaScript has the built-in `Math` object.

The great thing about objects is that they have methods! Let's call the `.random()` method from the built-in `Math` object:

```
console.log(Math.random()); // Prints a random number between 0 and 1
```

In the example above, we called the `.random()` method by appending the object name with the dot operator, the name of the method, and opening and closing parentheses. This method returns a random number between 0 and 1.

To generate a random number between 0 and 50, we could multiply this result by 50, like so:

```
Math.random() * 50;
```

The example above will likely evaluate to a decimal. To ensure the answer is a whole number, we can take advantage of another useful `Math` method called `Math.floor()`.

`Math.floor()` takes a decimal number, and rounds down to the nearest whole number. You can use `Math.floor()` to round down a random number like this:

```
Math.floor(Math.random() * 50);
```

In this case:

1. `Math.random` generates a random number between 0 and 1.
2. We then multiply that number by `50`, so now we have a number between 0 and 50.
3. Then, `Math.floor()` rounds the number down to the nearest whole number.

If you wanted to see the number printed to the terminal, you would still need to use a `console.log()` statement:

```
console.log(Math.floor(Math.random() * 50));  
// Prints a random whole number between 0 and 50
```

To see all of the properties and methods on the `Math` object, take a look at the documentation [here](#). To see all of the properties and methods on the `Number` object, take a look at the documentation [here](#).

1.10 Review

Let us take one more glance at the concepts we just learned:

- Data is printed, or logged, to the console, a panel that displays messages, with `console.log()`.

- We can write single-line comments with `//` and multi-line comments between `/*` and `*/`.
- There are 7 fundamental data types in JavaScript: strings, numbers, booleans, null, undefined, symbol, and object.
- Numbers are any number without quotes: `23.8879`
- Strings are characters wrapped in single or double quotes: `'Sample String'`
- The built-in arithmetic operators include `+`, `-`, `*`, `/`, and `%`.
- Objects, including instances of data types, can have properties, stored information. The properties are denoted with a `.` after the name of the object, for example: `'Hello'.length`.
- Objects, including instances of data types, can have methods which perform actions. Methods are called by appending the object or instance with a period, the method name, and parentheses. For example: `'hello'.toUpperCase()`.
- We can access properties and methods by using the `.`, dot operator.
- Built-in objects, including `Math`, are collections of methods and properties that JavaScript provides.

2 Variables

2.1 Variables

In programming, a *variable* is a container for a value. You can think of variables as little containers for information that live in a computer's memory. Information stored in variables, such as a username, account number, or even personalized greeting can then be found in memory.

Variables also provide a way of labeling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves.

In short, variables label and store data in memory. There are only a few things you can do with variables:

1. Create a variable with a descriptive name.
2. Store or update information stored in a variable.
3. Reference or “get” information stored in a variable.

It is important to distinguish that variables are not values; they contain values and represent them with a name. Later, we will cover how to use the `var`, `let`, and `const` keywords to create variables.

2.2 Create a Variable: `var`

There were a lot of changes introduced in the ES6 version of JavaScript in 2015. One of the biggest changes was two new keywords, `let` and `const`, to create, or declare, variables. Prior to the ES6, programmers could only use the `var` keyword to declare variables.

```
var myName = "Arya";  
console.log(myName); // Output: Arya
```

Let's consider the example above:

1. `var`, short for variable, is a JavaScript *keyword* that creates, or *declares*, a new variable.
2. `myName` is the variable's name. Capitalizing in this way is a standard convention in JavaScript called *camel casing*. In camel casing you group words into one, the first word is lowercase, then every word that follows will have its first letter uppercased. (e.g. camelCaseEverything).
3. `=` is the assignment operator. It assigns the value (`'Arya'`) to the variable (`myName`).
4. `'Arya'` is the value assigned (`=`) to the variable `myName`. You can also say that the `myName` variable is initialized with a value of `'Arya'`.
5. After the variable is declared, the string value `'Arya'` is printed to the console by referencing the variable name: `console.log(myName)`.

There are a few general rules for naming variables:

- Variable names cannot start with numbers.
- Variable names are case sensitive, so `myName` and `myname` would be different variables. It is bad practice to create two variables that have the same name using different cases.

- Variable names cannot be the same as *keywords*. For a comprehensive list of keywords check out MDN's keyword documentation.

Later, we will learn why ES6's `let` and `const` are the preferred variable keywords by many programmers. Because there is still a ton of code written prior to ES6, it is helpful to be familiar with the pre-ES6 `var` keyword. To learn more about `var` and the quirks associated with it, check out the MDN var documentation.

2.3 Create a Variable: `let`

As mentioned before, the `let` keyword was introduced in ES6. The `let` keyword signals that the variable can be reassigned a different value. Take a look at the example:

```
let meal = "Enchiladas";
console.log(meal); // Output: Enchiladas
meal = "Burrito";
console.log(meal); // Output: Burrito
```

Another concept that we should be aware of when using `let` (and even `var`) is that we can declare a variable without assigning the variable a value. In such a case, the variable will be automatically initialized with a value of `undefined`:

```
let price;
console.log(price); // Output: undefined
price = 350;
console.log(price); // Output: 350
```

Notice in the example above:

- If we don't assign a value to a variable declared using the `let` keyword, it automatically has a value of `undefined`.
- We can reassign the value of the variable.

2.4 Create a Variable: `const`

The `const` keyword was also introduced in ES6, and is short for the word constant. Just like with `var` and `let` you can store any value in a `const` variable. The way you declare a `const` variable and assign a value to it follows the same structure as `let` and `var`. Take a look at the following example:

```
const myName = "Gilberto";
console.log(myName); // Output: Gilberto
```

However, a `const` variable cannot be reassigned because it is constant. If you try to reassign a `const` variable, you'll get a `TypeError`.

Constant variables must be assigned a value when declared. If you try to declare a `const` variable without a value, you'll get a `SyntaxError`.

If you're trying to decide between which keyword to use, `let` or `const`, think about whether you'll need to reassign the variable later on. If you do need to reassign the variable use `let`, otherwise, use `const`.

2.5 Mathematical Assignment Operators

Let us consider how we can use variables and math operators to calculate new values and assign them to a variable. Check out the example below:

```
let w = 4;
w = w + 1;
console.log(w); // Output: 5
```

In the example above, we created the variable `w` with the number `4` assigned to it. The following line, `w = w + 1`, increases the value of `w` from `4` to `5`.

Another way we could have reassigned `w` after performing some mathematical operation on it is to use built-in *mathematical assignment operators*. We could re-write the code above to be:

```
let w = 4;
w += 1;
console.log(w); // Output: 5
```

In the second example, we used the `+=` assignment operator to reassign `w`. We're performing the mathematical operation of the first operator `+` using the number to the right, then reassigning `w` to the computed value.

We also have access to other mathematical assignment operators: `-=`, `*=`, and `/=` which work in a similar fashion.

```
let x = 20;
x -= 5; // Can be written as x = x - 5
console.log(x); // Output: 15

let y = 50;
y *= 2; // Can be written as y = y * 2
console.log(y); // Output: 100

let z = 8;
z /= 2; // Can be written as z = z / 2
console.log(z); // Output: 4
```

2.6 The Increment and Decrement Operator

Other mathematical assignment operators include the increment operator (`++`) and decrement operator (`--`). The increment operator will increase the value of the variable by 1. The decrement operator will decrease the value of the variable by 1. For example:

```
let a = 10;
a++;
console.log(a); // Output: 11

let b = 20;
b--;
console.log(b); // Output: 19
```

Just like the previous mathematical assignment operators (`+=`, `-=`, `*=`, `/=`), the variable's value is updated and assigned as the new value of that variable.

2.7 String Concatenation with Variables

Before we assigned strings to variables. Now, let us go over how to connect, or concatenate, strings in variables. The `+` operator can be used to combine two string values even if those values are being stored in variables:

```
let myPet = "armadillo";
console.log("I own a pet " + myPet + "."); // Output: "I own a pet armadillo."
```

In the example above, we assigned the value `'armadillo'` to the `myPet` variable. On the second line, the `+` operator is used to combine three strings: `'I own a pet'`, the value saved to `myPet`, and `'.'`. We log the result of this concatenation to the console as:

```
I own a pet armadillo.
```

2.8 String Interpolation

In the ES6 version of JavaScript, we can insert, or *interpolate*, variables into strings using *template literals*. Check out the following example where a template literal is used to log strings together: `

```
const myPet = "armadillo";
console.log(`I own a pet ${myPet}.`); // Output: I own a pet armadillo.
```

Notice that:

- a template literal is wrapped by backticks ``` (this key is usually located on the top of your keyboard, left of the `1` key).
- Inside the template literal, you'll see a placeholder, `${myPet}`. The value of `myPet` is inserted into the template literal.
- When we interpolate ``I own a pet ${myPet}.``, the output we print is the string: `'I own a pet armadillo.'`

One of the biggest benefits to using template literals is the readability of the code. Using template literals, you can more easily tell what the new string will be. You also do not have to worry about escaping double quotes or single quotes.

2.9 typeof operator

While writing code, it can be useful to keep track of the data types of the variables in your program. If you need to check the data type of a variable's value, you can use the `typeof` operator. The `typeof` operator checks the value to its right and *returns*, or passes back, a string of the data type.

```
const unknown1 = "foo";
console.log(typeof unknown1); // Output: string

const unknown2 = 10;
console.log(typeof unknown2); // Output: number

const unknown3 = true;
console.log(typeof unknown3); // Output: boolean
```

Let us break down the first example. Since the value `unknown1` is `'foo'`, a string, `typeof unknown1` will return `'string'`.

2.10 Review Variables

Variables is a powerful concept you will use in all your future programming endeavors. Let us take one more glance at the concepts we just learned:

- Variables hold reusable data in a program and associate it with a name.
- Variables are stored in memory.
- The `var` keyword is used in pre-ES6 versions of JS.
- `let` is the preferred way to declare a variable when it can be reassigned, and `const` is the preferred way to declare a variable with a constant value.
- Variables that have not been initialized store the primitive data type `undefined`.
- Mathematical assignment operators make it easy to calculate a new value and assign it to the same variable.
- The `+` operator is used to concatenate strings including string values held in variables.
- In ES6, template literals use backticks ``` and `${}` to interpolate values into a string.
- The `typeof` keyword returns the data type (as a string) of a value.

3 Conditional Statements

3.1 What are Conditional Statements?

In life, we make decisions based on circumstances. Think of an everyday decision as mundane as falling asleep — if we are tired, we go to bed, otherwise, we wake up and start our day. These if-else decisions can be modeled in code by creating *conditional statements*. A conditional statement checks a specific condition(s) and performs a task based on the condition(s).

Now we will explore how programs make decisions by evaluating conditions and introduce logic into our code. We will cover the following concepts:

- `if`, `else if`, and `else` statements
- comparison operators
- logical operators
- truthy vs falsy values
- ternary operators
- `switch` statement

3.2 If Statement

We often perform a task based on a condition. For example, if the weather is nice today, then we will go outside. If the alarm clock rings, then we will shut it off. If we are tired, then we will go to sleep.

In programming, we can also perform a task based on a condition using an `if` statement:

```
if (true) {  
  console.log("This message will print!");  
}  
// Prints: This message will print!
```

Notice in the example above, we have an `if` statement. The `if` statement is composed of:

- The `if` keyword followed by a set of parentheses `()` which is followed by a *code block*, or *block statement*, indicated by a set of curly braces `{}`.
- Inside the parentheses `()`, a condition is provided that evaluates to `true` or `false`.
- If the condition evaluates to `true`, the code inside the curly braces `{}` runs, or *executes*.
- If the condition evaluates to `false`, the block will not execute.

3.3 If...Else Statements

In the previous chapter, we used an `if` statement that checked a condition to decide whether or not to run a block of code. In many cases, we'll have code we want to run if our condition evaluates to `false`.

If we wanted to add some default behavior to the `if` statement, we can add an `else` statement to run a block of code when the condition evaluates to `false`. Take a look at the inclusion of an `else` statement:

```

if (false) {
  console.log("The code in this block will not run.");
} else {
  console.log("But the code in this block will!");
}

// Prints: But the code in this block will!

```

An `else` statement must be paired with an `if` statement, and together they are referred to as an `if...else` statement. In the example above, the `else` statement:

- Uses the `else` keyword following the code block of an `if` statement.
- Has a code block that is wrapped by a set of curly braces `{}`.
- The code inside the `else` statement code block will execute when the `if` statement's condition evaluates to false.

`if...else` statements allow us to automate solutions to yes-or-no questions, also known as binary decisions.

3.4 Comparison Operators

When writing conditional statements, sometimes we need to use different types of operators to compare values. These operators are called *comparison operators*. Here is a list of some handy comparison operators and their syntax:

- Less than: `<`
- Greater than: `>`
- Less than or equal to: `<=`
- Greater than or equal to: `>=`
- Is equal to: `===`
- Is not equal to: `!==`

Comparison operators compare the value on the left with the value on the right. For instance:

```
10 < 12 // Evaluates to true
```

It can be helpful to think of comparison statements as questions. When the answer is “yes”, the statement evaluates to `true`, and when the answer is “no”, the statement evaluates to `false`. The code above would be asking: is 10 less than 12? Yes! So `10 < 12` evaluates to `true`.

We can also use comparison operators on different data types like strings:

```
"apples" === "oranges" // false
```

In the example above, we're using the identity operator (`===`) to check if the string `'apples'` is the same as the string `'oranges'`. Since the two strings are not the same, the comparison statement evaluates to `false`.

All comparison statements evaluate to either true or false and are made up of:

- Two values that will be compared.

- An operator that separates the values and compares them accordingly (`>`, `<`, `<=`, `>=`, `===`, `!==`).

The difference between `==` and `===` is that `===` is known as Identity / strict equality, meaning that the data types also must be equal.

```
console.log(1 == true) // output: true
console.log(1 === true) // output: false
```

3.5 Logical Operators

Working with conditionals means that we will be using booleans, `true` or `false` values. In JavaScript, there are operators that work with boolean values known as *logical operators*. We can use logical operators to add more sophisticated logic to our conditionals. There are three logical operators:

- the and operator (`&&`)
- the or operator (`||`)
- the not operator, otherwise known as the bang operator (`!`)

When we use the `&&` operator, we are checking that two things are true:

```
if (stopLight === "green" && pedestrians === 0) {
  console.log("Go!");
} else {
  console.log("Stop");
}
```

When using the `&&` operator, both conditions must evaluate to `true` for the entire condition to evaluate to `true` and execute. Otherwise, if either condition is `false`, the `&&` condition will evaluate to `false` and the `else` block will execute.

If we only care about either condition being true, we can use the `||` operator:

```
if (day === "Saturday" || day === "Sunday") {
  console.log("Enjoy the weekend!");
} else {
  console.log("Do some work.");
}
```

When using the `||` operator, only one of the conditions must evaluate to `true` for the overall statement to evaluate to `true`. In the code example above, if either `day === 'Saturday'` or `day === 'Sunday'` evaluates to `true` the `if`'s condition will evaluate to `true` and its code block will execute. If the first condition in an `||` statement evaluates to `true`, the second condition will not even be checked. Only if `day === 'Saturday'` evaluates to `false` will `day === 'Sunday'` be evaluated. The code in the `else` statement above will execute only if both comparisons evaluate to `false`.

The `!` *not operator reverses*, or *negates*, the value of a boolean:

```
let excited = true;
console.log(!excited); // Prints false

let sleepy = false;
console.log(!sleepy); // Prints true
```

Essentially, the `!` operator will either take a `true` value and pass back `false`, or it will take a `false` value and pass back `true`.

Logical operators are often used in conditional statements to add another layer of logic to our code.

3.6 Truthy and Falsy

Let us consider how non-boolean data types, like strings or numbers, are evaluated when checked inside a condition.

Sometimes, you will want to check if a variable exists and you will not necessarily want it to equal a specific value — you'll only check to see if the variable has been assigned a value. Here is an example:

```
let myVariable = "I Exist!";

if (myVariable) {
  console.log(myVariable)
} else {
  console.log("The variable does not exist.")
}
```

The code block in the `if` statement will run because `myVariable` has a *truthy* value; even though the value of `myVariable` is not explicitly the value `true`, when used in a boolean or conditional context, it evaluates to `true` because it has been assigned a non-falsy value.

So which values are *falsy*— or evaluate to `false` when checked as a condition? The list of falsy values includes:

- `0`
- Empty strings like `''` or `' '`
- `null` which represent when there is no value at all
- `undefined` which represent when a declared variable lacks a value
- `NaN`, or Not a Number

Here's an example with numbers:

```
let numberOfApples = 0;

if (numberOfApples){
  console.log("Let us eat apples!");
} else {
  console.log("No apples left!");
}

// Prints 'No apples left!'
```

The condition evaluates to `false` because the value of the `numberOfApples` is `0`. Since `0` is a falsy value, the code block in the `else` statement will run.

3.7 Truthy and Falsy Assignment

Truthy and falsy evaluations open a world of short-hand possibilities. Say you have a website and want to take a user's username to make a personalized greeting. Sometimes, the user does not have an account, making the `username` variable falsy. The code below checks if `username` is defined and assigns a default string if it is not:

```
let defaultName;
if (username) {
  defaultName = username;
} else {
  defaultName = "Stranger";
}
```

If you combine your knowledge of logical operators you can use a short-hand for the code above. In a boolean condition, JavaScript assigns the truthy value to a variable if you use the `||` operator in your assignment:

```
let defaultName = username || "Stranger";
```

Because `||` or statements check the left-hand condition first, the variable `defaultName` will be assigned the actual value of `username` if it is truthy, and it will be assigned the value of `'Stranger'` if `username` is falsy. This concept is also referred to as *short-circuit evaluation*.

3.8 Ternary Operator

In the spirit of using short-hand syntax, we can use a *ternary operator* to simplify an `if...else` statement. Take a look at the `if...else` statement example:

```
let isNightTime = true;

if (isNightTime) {
  console.log("Turn on the lights!");
} else {
  console.log("Turn off the lights!");
}
```

We can use a *ternary operator* to perform the same functionality:

```
isNightTime ? console.log("Turn on the lights!") : console.log("Turn off the lights!");
```

In the example above:

- The condition, `isNightTime`, is provided before the `?`.
- Two expressions follow the `?` and are separated by a colon `:`.
- If the condition evaluates to `true`, the first expression executes.
- If the condition evaluates to `false`, the second expression executes.

Like `if...else` statements, ternary operators can be used for conditions which evaluate to `true` or `false`.

3.9 Else If Statements

We can add more conditions to our `if...else` with an `else if` statement. The `else if` statement allows for more than two possible outcomes. You can add as many `else if` statements as you would like, to make more complex conditionals!

The `else if` statement always comes after the `if` statement and before the `else` statement. The `else if` statement also takes a condition. Let us take a look at the syntax:

```
let stopLight = "yellow";

if (stopLight === "red") {
  console.log("Stop!");
} else if (stopLight === "yellow") {
  console.log("Slow down.");
} else if (stopLight === "green") {
  console.log("Go!");
} else {
  console.log("Caution, unknown!");
}
```

The `else if` statements allow you to have multiple possible outcomes. `if/else if/else` statements are read from top to bottom, so the first condition that evaluates to `true` from the top to bottom is the block that gets executed.

In the example above, since `stopLight === 'red'` evaluates to false and `stopLight === 'yellow'` evaluates to `true`, the code inside the first `else if` statement is executed. The rest of the conditions are not evaluated. If none of the conditions evaluated to true, then the code in the `else` statement would have executed.

3.10 The switch keyword

`else if` statements are a great tool if we need to check multiple conditions. In programming, we often find ourselves needing to check multiple values and handling each of them differently. For example:

```
let groceryItem = "papaya";

if (groceryItem === "tomato") {
  console.log("Tomatoes are $0.49");
} else if (groceryItem === "papaya"){
  console.log("Papayas are $1.29");
} else {
  console.log("Invalid item");
}
```

In the code above, we have a series of conditions checking for a value that matches a `groceryItem` variable. Our code works fine, but imagine if we needed to check 100 different values. Having to write that many `else if` statements sounds like a pain.

A `switch` statement provides an alternative syntax that is easier to read and write. A `switch` statement looks like this:

```

let groceryItem = "papaya";
switch (groceryItem) {
  case "tomato":
    console.log("Tomatoes are $0.49");
    break;
  case "lime":
    console.log("Limes are $1.49");
    break;
  case "papaya":
    console.log("Papayas are $1.29");
    break;
  default:
    console.log("Invalid item");
    break;
}
// Prints "Papayas are $1.29"

```

- The `switch` keyword initiates the statement and is followed by `(...)`, which contains the value that each `case` will compare. In the example, the value or expression of the `switch` statement is `groceryItem`.
- Inside the block, `{ ... }`, there are multiple `cases`. The `case` keyword checks if the expression matches the specified value that comes after it. The value following the first `case` is `'tomato'`. If the value of `groceryItem` equalled `'tomato'`, that `case`'s `console.log()` would run.
- The value of `groceryItem` is `'papaya'`, so the third `case` runs— `Papayas are $1.29` is logged to the console.
- The `break` keyword tells the computer to exit the block and not execute any more code or check any other cases inside the code block. Note: Without the `break` keyword at the end of each case, the program would execute the code for all matching cases and the default code as well. This behavior is different from `if/else` conditional statements which execute only one block of code.
- At the end of each `switch` statement, there is a `default` statement. If none of the `cases` are true, then the code in the `default` statement will run.

3.11 Review

Here are some of the major concepts for conditionals:

- An `if` statement checks a condition and will execute a task if that condition evaluates to `true`.
- `if...else` statements make binary decisions and execute different code blocks based on a provided condition.
- We can add more conditions using `else if` statements.
- Comparison operators, including `<`, `>`, `<=`, `>=`, `===`, and `!==` can compare two values.
- The logical and operator, `&&`, or “and”, checks if both provided expressions are truthy.
- The logical operator `||`, or “or”, checks if either provided expression is truthy.
- The bang operator, `!`, switches the truthiness and falsiness of a value.
- The ternary operator is shorthand to simplify concise `if...else` statements.

- A `switch` statement can be used to simplify the process of writing multiple `else if` statements. The `break` keyword stops the remaining `cases` from being checked and executed in a `switch` statement.

4 Functions

4.1 What are Functions?

When first learning how to calculate the area of a rectangle, there is a sequence of steps to calculate the correct answer:

1. Measure the width of the rectangle.
2. Measure the height of the rectangle.
3. Multiply the width and height of the rectangle.

With practice, you can calculate the area of the rectangle without being instructed with these three steps every time. We can calculate the area of one rectangle with the following code:

```
const width = 10;
const height = 6;
const area = width * height;
console.log(area); // Output: 60
```

Imagine being asked to calculate the area of three different rectangles:

```
// Area of the first rectangle
const width1 = 10;
const height1 = 6;
const area1 = width1 * height1;

// Area of the second rectangle
const width2 = 4;
const height2 = 9;
const area2 = width2 * height2;

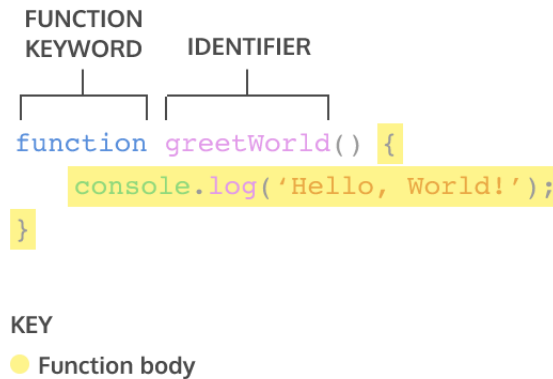
// Area of the third rectangle
const width3 = 10;
const height3 = 10;
const area3 = width3 * height3;
```

In programming, we often use code to perform a specific task multiple times. Instead of rewriting the same code, we can group a block of code together and associate it with one task, then we can reuse that block of code whenever we need to perform the task again. We achieve this by creating a *function*. A function is a reusable block of code that groups together a sequence of statements to perform a specific task.

Now, you will learn how to create and use functions, and how they can be used to create clearer and more concise code.

4.2 Function Declarations

In JavaScript, there are many ways to create a function. One way to create a function is by using a *function declaration*. Just like how a variable declaration binds a value to a variable name, a function declaration binds a function to a name, or an *identifier*. Take a look at the anatomy of a function declaration below:



A function declaration consists of:

- The `function` keyword.
- The name of the function, or its identifier, followed by parentheses.
- A function body, or the block of statements required to perform a specific task, enclosed in the function's curly brackets, `{ }`.

A function declaration is a function that is bound to an identifier, or name. In the next chapter, we will go over how to run the code inside the function body. We should also be aware of the *hoisting* feature in JavaScript which allows access to function declarations before they're defined. Take a look at example of hoisting:

```
console.log(greetWorld()); // Output: Hello, World!

function greetWorld() {
    console.log("Hello, World!");
}
```

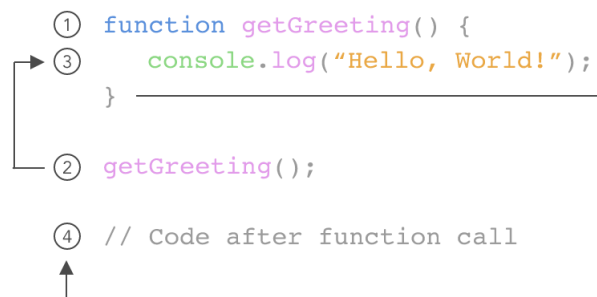
Notice how hoisting allowed `greetWorld()` to be called before the `greetWorld()` function was defined! Since hoisting is not considered good practice, we simply want you to be aware of this feature.

4.3 Calling a Function

As we saw in the previous chapter, a function declaration binds a function to an identifier. However, a function declaration does not ask the code inside the function body to run, it just declares the existence of the function. The code inside a function body runs, or *executes*, only when the function is *called*. To call a function in your code, you type the function name followed by parentheses.

IDENTIFIER
└───┬───┘
greetWorld();

This *function call* executes the function body, or all of the statements between the curly braces in the function declaration.



We can call the same function as many times as needed.

4.4 Parameters and Arguments

So far, the functions we have created execute a task without an input. However, some functions can take inputs and use the inputs to perform a task. When declaring a function, we can specify its *parameters*. Parameters allow functions to accept input(s) and perform a task using the input(s). We use parameters as placeholders for information that will be passed to the function when it is called.

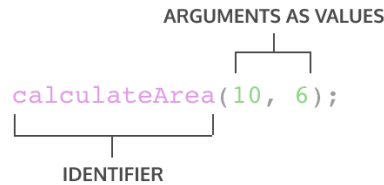
Let's observe how to specify parameters in our function declaration:

PARAMETERS
└───┬───┘
function calculateArea(width, height) {
 console.log(width * height);
}
└──┬──┘ └──┬──┘
PARAMETERS ARE TREATED LIKE
VARIABLES WITHIN A FUNCTION

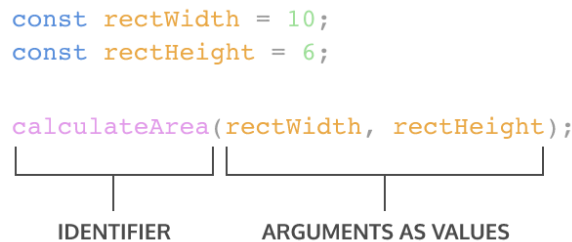
In the diagram above, `calculateArea()`, computes the area of a rectangle, based on two inputs, `width` and `height`. The parameters are specified between the parenthesis as `width` and `height`, and inside the function body, they act just like regular variables. `width` and `height` act as placeholders for values that

will be multiplied together.

When calling a function that has parameters, we specify the values in the parentheses that follow the function name. The values that are passed to the function when it is called are called *arguments*. Arguments can be passed to the function as values or variables.



In the function call above, the number `10` is passed as the `width` and `6` is passed as `height`. Notice that the order in which arguments are passed and assigned follows the order that the parameters are declared.



The variables `rectWidth` and `rectHeight` are initialized with the values for the height and width of a rectangle before being used in the function call.

By using parameters, `calculateArea()` can be reused to compute the area of any rectangle! Functions are a powerful tool in computer programming so let's practice creating and calling functions with parameters.

4.5 Default Parameters

One of the features added in ES6 is the ability to use *default parameters*. Default parameters allow parameters to have a predetermined value in case there is no argument passed into the function or if the argument is `undefined` when called.

Take a look at the code snippet below that uses a default parameter:

```
function greeting (name = "stranger") {  
  console.log(`Hello, ${name}!`)  
}  
  
greeting("Nick") // Output: Hello, Nick!  
greeting()       // Output: Hello, stranger!
```

- In the example above, we used the `=` operator to assign the parameter `name` a default value of `'stranger'`. This is useful to have in case we ever want to include a non-personalized default greeting!

- When the code calls `greeting('Nick')` the value of the argument is passed in and, `'Nick'`, will override the default parameter of `'stranger'` to log `'Hello, Nick!'` to the console.
- When there is not an argument passed into `greeting()`, the default value of `'stranger'` is used, and `'Hello, stranger!'` is logged to the console.

By using a default parameter, we account for situations when an argument isn't passed into a function that is expecting an argument.

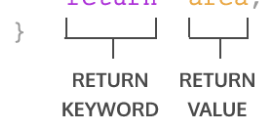
4.6 Return

When a function is called, the computer will run through the function's code and evaluate the result of calling the function. By default that resulting value is `undefined`.

```
function rectangleArea(width, height) {
  let area = width * height;
}
console.log(rectangleArea(5, 7)) // Prints undefined
```

In the code example, we defined our function to calculate the `area` of a `width` and `height` parameter. Then `rectangleArea()` is invoked with the arguments `5` and `7`. But when we went to print the results we got `undefined`. The computer did calculate the area as `35`, but we did not capture it. But we can capture with the keyword `return`.

```
function calculateArea(width, height) {
  const area = width * height;
  return area;
}
```



To pass back information from the function call, we use a return statement. To create a return statement, we use the `return` keyword followed by the value that we wish to return. Like we saw above, if the value is omitted, `undefined` is returned instead.

When a `return` statement is used in a function body, the execution of the function is stopped and the code that follows it will not be executed. Look at the example below:

```
function rectangleArea(width, height) {
  if (width < 0 || height < 0) {
    return "You need positive integers to calculate area!";
  }
  return width * height;
}
```

If an argument for `width` or `height` is less than `0`, then `rectangleArea()` will return the string `'You need positive integers to calculate area!'`. The second return statement `width * height` will not run.

The `return` keyword is powerful because it allows functions to produce an output. We can then save the output to a variable for later use.

4.7 Helper Functions

We can also use the return value of a function inside another function. These functions being called within another function are often referred to as *helper functions*. Since each function is carrying out a specific task, it makes our code easier to read and debug if necessary.

If we wanted to define a function that converts the temperature from Celsius to Fahrenheit, we could write two functions like:

```
function multiplyByNineFifths(number) {  
    return number * (9/5);  
};  
  
function getFahrenheit(celsius) {  
    return multiplyByNineFifths(celsius) + 32;  
};  
  
getFahrenheit(15); // Returns 59
```

In the example above:

- `getFahrenheit()` is called and `15` is passed as an argument.
- The code block inside of `getFahrenheit()` calls `multiplyByNineFifths()` and passes `15` as an argument.
- `multiplyByNineFifths()` takes the argument of `15` for the `number` parameter.
- The code block inside of `multiplyByNineFifths()` function multiplies `15` by `(9/5)`, which evaluates to `27`.
- `27` is returned back to the function call in `getFahrenheit()`.
- `getFahrenheit()` continues to execute. It adds `32` to `27`, which evaluates to `59`.
- Finally, `59` is returned back to the function call `getFahrenheit(15)`.

We can use functions to section off small bits of logic or tasks, then use them when we need to. Writing helper functions can help take large and difficult tasks and break them into smaller and more manageable tasks.

4.8 Function Expressions

Another way to define a function is to use a *function expression*. To define a function inside an expression, we can use the `function` keyword. In a function expression, the function name is usually omitted. A function with no name is called an *anonymous function*. A function expression is often stored in a variable in order to refer to it. Consider the following function expression:

```
const calculateArea = function(width, height) {  
  const area = width * height;  
  return area;  
};
```

To declare a function expression:

1. Declare a variable to make the variable's name be the name, or identifier, of your function. Since the release of ES6, it is common practice to use `const` as the keyword to declare the variable.
2. Assign as that variable's value an anonymous function created by using the `function` keyword followed by a set of parentheses with possible parameters. Then a set of curly braces that contain the function body.

To invoke a function expression, write the name of the variable in which the function is stored followed by parentheses enclosing any arguments being passed into the function.

```
variableName(argument1, argument2)
```

Unlike function declarations, function expressions are not hoisted so they cannot be called before they are defined.

4.9 Arrow Functions

ES6 introduced *arrow function syntax*, a shorter way to write functions by using the special “fat arrow” `() =>` notation.

Arrow functions remove the need to type out the keyword `function` every time you need to create a function. Instead, you first include the parameters inside the `()` and then add an arrow `=>` that points to the function body surrounded in `{ }` like this:

```
const rectangleArea = (width, height) => {  
  let area = width * height;  
  return area;  
};
```

It is important to be familiar with the multiple ways of writing functions because you will come across each of these when reading other JavaScript code.

4.10 Concise Body Arrow Functions

JavaScript also provides several ways to refactor arrow function syntax. The most condensed form of the function is known as *concise body*. We will explore a few of these techniques below:

1. Functions that take only a single parameter do not need that parameter to be enclosed in parentheses. However, if a function takes zero or multiple parameters, parentheses are required.

ZERO PARAMETERS

```
const functionName = () => {};
```

ONE PARAMETER

```
const functionName = paramOne => {};
```

TWO OR MORE PARAMETERS

```
const functionName = (paramOne, paramTwo) => {};
```

2. A function body composed of a single-line block does not need curly braces. Without the curly braces, whatever that line evaluates will be automatically returned. The contents of the block should immediately follow the arrow `=>` and the `return` keyword can be removed. This is referred to as *implicit return*.

SINGLE-LINE BLOCK

```
const sumNumbers = number => number + number;
```

MULTI-LINE BLOCK

```
const sumNumbers = number => {  
  const sum = number + number;  
  return sum; } — RETURN STATEMENT  
};
```

So if we have a function:

```
const squareNum = (num) => {  
  return num * num;  
};
```

We can refactor the function to:

```
const squareNum = num => num * num;
```

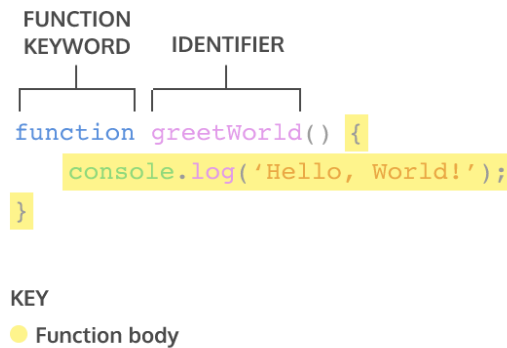
Notice the following changes:

- The parentheses around `num` have been removed, since it has a single parameter.
- The curly braces `{ }` have been removed since the function consists of a single-line block.
- The `return` keyword has been removed since the function consists of a single-line block.

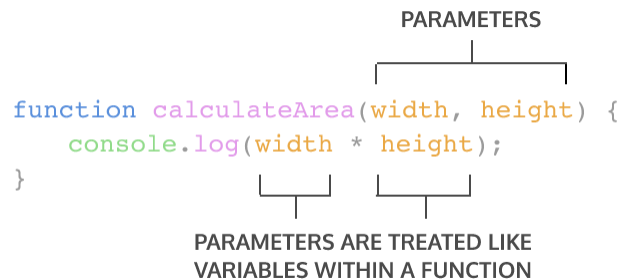
4.11 Review Functions

In this we covered some important concepts about functions:

- A *function* is a reusable block of code that groups together a sequence of statements to perform a specific task.
- A *function declaration*:



- A parameter is a named variable inside a function's block which will be assigned the value of the argument passed in when the function is invoked:



- To *call* a function in your code:



- ES6 introduces new ways of handling arbitrary parameters through *default parameters* which allow us to assign a default value to a parameter in case no argument is passed into the function.
- To return a value from a function, we use a *return statement*.
- To define a function using *function expressions*:

```

const calculateArea = function(width, height) {
  const area = width * height;
  return area;
};

```

- To define a function using *arrow function notation*:

```

const calculateArea = (width, height) => {
  const area = width * height;
  return area;
};

```

- Function definition can be made concise using concise arrow notation:

SINGLE-LINE BLOCK

```
const sumNumbers = number => number + number;
```

MULTI-LINE BLOCK

```
const sumNumbers = number => {
  const sum = number + number;
  return sum; } — RETURN STATEMENT

```

It is good to be aware of the differences between function expressions, arrow functions, and function declarations. As you program more in JavaScript, you will see a wide variety of how these function types are used.

5 Scope

5.1 Scope

An important idea in programming is *scope*. Scope defines where variables can be accessed or referenced. While some variables can be accessed from anywhere within a program, other variables may only be available in a specific context.

You can think of scope like the view of the night sky from your window. Everyone who lives on the planet Earth is in the global scope of the stars. The stars are accessible *globally*. Meanwhile, if you live in a city, you may see the city skyline or the river. The skyline and river are only accessible *locally* in your city, but you can still see the stars that are available globally.

Over the next few chapters, we will explore how scope relates to variables and learn best practices for variable declaration.

5.2 Blocks and Scope

Before we talk more about scope, we first need to talk about *blocks*. We have seen blocks used before in functions and `if` statements. A block is the code found inside a set of curly braces `{ }`. Blocks help us group one or more statements together and serve as an important structural marker for our code.

A block of code could be a function, like this:

```
const logSkyColor = () => {  
  let color = "blue";  
  console.log(color); // blue  
};
```

Notice that the function body is actually a block of code. Observe the block in an `if` statement:

```
if (dusk) {  
  let color = "pink";  
  console.log(color); // pink  
};
```

In the next few chapters, we will see how blocks define the scope of variables.

5.3 Global Scope

Scope is the context in which our variables are declared. We think about scope in relation to blocks because variables can exist either outside of or within these blocks.

In *global scope*, variables are declared outside of blocks. These variables are called *global variables*. Because global variables are not bound inside a block, they can be accessed by any code in the program, including code in blocks.

Let us take a look at an example of global scope:

```
const color = "blue"

const returnSkyColor = () => {
  return color; // blue
};

console.log(returnSkyColor()); // blue
```

- Even though the `color` variable is defined outside of the block, it can be accessed in the function block, giving it global scope.
- In turn, `color` can be accessed within the `returnSkyColor` function block.

5.4 Block Scope

The next context we will cover is *block scope*. When a variable is defined inside a block, it is only accessible to the code within the curly braces `{}`. We say that variable has *block scope* because it is *only* accessible to the lines of code within that block.

Variables that are declared with block scope are known as *local variables* because they are only available to the code that is part of the same block.

Block scope works like this:

```
const logSkyColor = () => {
  let color = "blue";
  console.log(color); // blue
};

logSkyColor(); // blue
console.log(color); // ReferenceError
```

We will notice:

- We define a function `logSkyColor()`.
- Within the function, the `color` variable is only available within the curly braces of the function.
- If we try to log the same variable outside the function, throws a `ReferenceError`.

5.5 Scope Pollution

It may seem like a great idea to always make your variables accessible, but having too many global variables can cause problems in a program.

When you declare global variables, they go to the *global namespace*. The global namespace allows the variables to be accessible from anywhere in the program. These variables remain there until the program finishes which means our global namespace can fill up really quickly.

Scope pollution is when we have too many global variables that exist in the global namespace, or when we reuse variables across different scopes. Scope pollution makes it difficult to keep track of our different

variables and sets us up for potential accidents. For example, globally scoped variables can collide with other variables that are more locally scoped, causing unexpected behavior in our code.

Let us look at an example of scope pollution in practice so we know how to avoid it:

```
let num = 50;

const logNum = () => {
  num = 100; // Take note of this line of code
  console.log(num);
};

logNum(); // Prints 100
console.log(num); // Prints 100
```

We will notice:

- We have a variable `num`.
- Inside the function body of `logNum()`, we want to declare a new variable but forgot to use the `let` keyword.
- When we call `logNum()`, `num` gets reassigned to `100`.
- The reassignment inside `logNum()` affects the global variable `num`.
- Even though the reassignment is allowed and we will not get an error, if we decided to use `num` later, we'll unknowingly use the new value of `num`.

While it is important to know what global scope is, it is best practice to not define variables in the global scope.

5.6 Practice Good Scoping

Given the challenges with global variables and scope pollution, we should follow best practices for scoping our variables as tightly as possible using block scope.

Tightly scoping your variables will greatly improve your code in several ways:

- It will make your code more legible since the blocks will organize your code into discrete sections.
- It makes your code more understandable since it clarifies which variables are associated with different parts of the program rather than having to keep track of them line after line!
- It is easier to maintain your code, since your code will be modular.
- It will save memory in your code because it will cease to exist after the block finishes running.

Here is another example of how to use block scope, as defined within an `if` block:

```
const logSkyColor = () => {
  const dusk = true;
  let color = "blue";
  if (dusk) {
    let color = "pink";
    console.log(color); // pink
  }
}
```

```
}  
  console.log(color); // blue  
};  
  
console.log(color); // ReferenceError
```

Here, we will notice:

- We create a variable `dusk` inside the `logSkyColor()` function.
- After the `if` statement, we define a new code block with the `{}` braces. Here we assign a new value to the variable `color` if the `if` statement is truthy.
- Within the `if` block, the `color` variable holds the value `'pink'`, though outside the `if` block, in the function body, the `color` variable holds the value `'blue'`.
- While we use block scope, we still pollute our namespace by reusing the same variable name twice. A better practice would be to rename the variable inside the block.

Block scope is a powerful tool in JavaScript, since it allows us to define variables with precision, and not pollute the global namespace. If a variable does not need to exist outside a block, then it should not!

5.7 Review: Scope

In this chapter, you learned about scope and how it impacts the accessibility of different variables. Let us review the following terms:

- **Scope** is the idea in programming that some variables are accessible/inaccessible from other parts of the program.
- **Blocks** are statements that exist within curly braces `{}`.
- **Global scope** refers to the context within which variables are accessible to every part of the program.
- **Global variables** are variables that exist within global scope.
- **Block scope** refers to the context within which variables that are accessible only within the block they are defined.
- **Local variables** are variables that exist within block scope.
- **Global namespace** is the space in our code that contains globally scoped information.
- **Scope pollution** is when too many variables exist in a namespace or variable names are reused.

6 Arrays

6.1 Arrays

Organizing and storing data is a foundational concept of programming. One way we organize data in real life is by making lists. Let us make one here:

New Year's Resolutions:

1. Keep a journal
2. Take a falconry class
3. Learn to juggle

Let us now write this list in JavaScript, as an array:

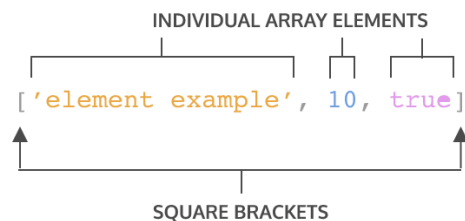
```
let newYearsResolutions = ["Keep a journal", "Take a falconry class", "Learn to juggle"];
```

Arrays are JavaScript's way of making lists. Arrays can store any data types (including strings, numbers, and booleans). Like lists, arrays are ordered, meaning each item has a numbered position. Here is an array of the concepts we'll cover:

```
let concepts = ["creating arrays", "array structures", "array manipulation"];
```

6.2 Create an Array

One way we can create an array is to use an *array literal*. An array literal creates an array by wrapping items in square brackets `[]`. Remember from the previous exercise, arrays can store any data type — we can have an array that holds all the same data types or an array that holds different data types.



Let us take a closer look at the syntax in the array example:

- The array is represented by the square brackets `[]` and the content inside.
- Each content item inside an array is called an *element*.
- There are three different elements inside the array. Each element inside the array is a different data type.

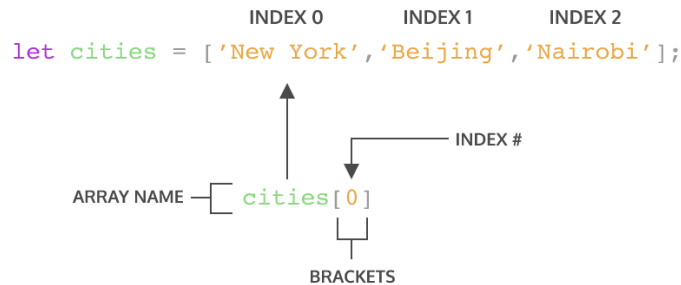
We can also save an array to a variable. You may have noticed we did this in the previous chapter:

```
let newYearsResolutions = ['Keep a journal', 'Take a falconry class', 'Learn to juggle'];
```


6.3 Accessing Elements

Each element in an array has a numbered position known as its *index*. We can access individual items using their index, which is similar to referencing an item in a list based on the item's position.

Arrays in JavaScript are *zero-indexed*, meaning the positions start counting from **0** rather than **1**. Therefore, the first item in an array will be at position **0**. Let us see how we could access an element in an array:



In the code snippet above:

- `cities` is an array that has three elements.
- We are using bracket notation, `[]` with the index after the name of the array to access the element.
- `cities[0]` will access the element at index **0** in the array `cities`. You can think of `cities[0]` as accessing the space in memory that holds the string `'New York'`.

You can also access individual characters in a string using bracket notation and the index. For instance, you can write:

```
const hello = "Hello World";  
console.log(hello[6]); // Output: W
```

The console will display **W** since it is the character that is at index **6**.

If you try to access an index that is beyond the last element, console will return `undefined`.

6.4 Update Elements

In the previous exercise, you learned how to access elements inside an array or a string by using an index. Once you have access to an element in an array, you can update its value.

```
let seasons = ["Winter", "Spring", "Summer", "Fall"];  
  
seasons[3] = "Autumn";  
console.log(seasons);  
//Output: ["Winter", "Spring", "Summer", "Autumn"]
```

In the example above, the `seasons` array contained the names of the four seasons. However, we decided that we preferred to say `'Autumn'` instead of `'Fall'`. The line, `seasons[3] = 'Autumn';` tells our program to change the item at index 3 of the `seasons` array to be `'Autumn'` instead of what is already there.

6.5 Arrays with let and const

You may recall that you can declare variables with both the `let` and `const` keywords. Variables declared with `let` can be reassigned.

Variables declared with the `const` keyword cannot be reassigned. However, elements in an array declared with `const` remain mutable. Meaning that we can change the contents of a `const` array, but cannot reassign a new array or a different value.

The instructions below will illustrate this concept more clearly. Pay close attention to the similarities and differences between the `condiments` array and the `utensils` array as you complete the steps.

```
let condiments = ["Ketchup", "Mustard", "Soy Sauce", "Sriracha"];
const utensils = ["Fork", "Knife", "Chopsticks", "Spork"];

condiments[0] = "Mayo";
console.log(condiments);
// Output: [ "Mayo", "Mustard", "Soy Sauce", "Sriracha" ]

condiments = ["Mayo"];
console.log(condiments);
// Output: [ "Mayo" ]

utensils[3] = "Spoon";
console.log(utensils);
// Output: [ "Fork", "Knife", "Chopsticks", "Spoon" ]

utensils = ["Spoon"];
// TypeError: Assignment to constant variable.
```

6.6 The .length property

One of an array's built-in properties is `length` and it returns the number of items in the array. We access the `.length` property just like we do with strings. Check the example below:

```
const newYearsResolutions = ["Keep a journal", "Take a falconry class"];
console.log(newYearsResolutions.length); // Output: 2
```

In the example above, we log `newYearsResolutions.length` to the console using the following steps:

- We use *dot notation*, chaining a period with the property name to the array, to access the `length` property of the `newYearsResolutions` array.
- Then we log the `length` of `newYearsResolution` to the console.
- Since `newYearsResolution` has two elements, so `2` would be logged to the console.

When we want to know how many elements are in an array, we can access the `.length` property.

6.7 The .push() Method

Let us learn about some built-in JavaScript methods that make working with arrays easier. These methods are specifically called on arrays to make common tasks, like adding and removing elements, more

straightforward.

One method, `.push()` allows us to add items to the end of an array. Here is an example of how this is used:

```
const itemTracker = ["item 0", "item 1", "item 2"];

itemTracker.push("item 3", "item 4");

console.log(itemTracker);
// Output: ["item 0", "item 1", "item 2", "item 3", "item 4"];
```

So, how does `.push()` work?

- We access the `push` method by using dot notation, connecting `push` to `itemTracker` with a period.
- Then we call it like a function. That is because `.push()` is a function and one that JavaScript allows us to use right on an array.
- `.push()` can take a single argument or multiple arguments separated by commas. In this case, we're adding two elements: `'item 3'` and `'item 4'` to `itemTracker`.
- Notice that `.push()` changes, or *mutates*, `itemTracker`. You might also see `.push()` referred to as a *destructive* array method since it changes the initial array.

If you are looking for a method that will mutate an array by adding elements to it, then `.push()` is the method for you.

6.8 The `.pop()` Method

Another array method, `.pop()`, removes the last item of an array.

```
const newItemTracker = ["item 0", "item 1", "item 2"];

const removed = newItemTracker.pop();

console.log(newItemTracker); // Output: [ "item 0", "item 1" ]
console.log(removed); // Output: item 2
```

- In the example above, calling `.pop()` on the `newItemTracker` array removed `item 2` from the end.
- `.pop()` does not take any arguments, it simply removes the last element of `newItemTracker`.
- `.pop()` returns the value of the last element. In the example, we store the returned value in a variable `removed` to be used for later.
- `.pop()` is a method that mutates the initial array.

When you need to mutate an array by removing the last element, use `.pop()`.

6.9 More Array Methods

There are many more array methods than just `.push()` and `.pop()`. You can read about all of the array methods that exist on the Mozilla Developer Network (MDN) array documentation.

`.pop()` and `.push()` mutate the array on which they are called. However, there are times that we do not want to mutate the original array and we can use non-mutating array methods.

Some arrays methods that are available to JavaScript developers include: `.join()`, `.slice()`, `.splice()`, `.shift()`, `.unshift()`, and `.concat()` amongst many others. Using these built-in methods make it easier to do some common tasks when working with arrays.

Below, we will explore `.shift()`, `.unshift()`, and `.slice()`.

6.9.1 The `.shift()` Method

The `.shift()` method removes the first element from an array and returns that removed element. This method changes the length of the array. The original array is modified.

```
const array1 = [1, 2, 3];
const firstElement = array1.shift();
console.log(array1); // Output: Array [2, 3]

console.log(firstElement); // Output: 1
```

6.9.2 The `.unshift()` Method

The `.unshift()` method adds one or more elements to the beginning of an array and returns the new length of the array. The original array is modified.

```
const array1 = [1, 2, 3];
console.log(array1.unshift(4, 5)); // Output: 5

console.log(array1); // Output: Array [4, 5, 1, 2, 3]
```

6.9.3 The `.slice()` Method

The `.slice()` method returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included) where start and end represent the index of items in that array. The original array will not be modified.

```
const animals = ["ant", "bison", "camel", "duck", "elephant"];
console.log(animals.slice(2)); // Output: Array ["camel", "duck", "elephant"]

console.log(animals.slice(2, 4)); // Output: Array ["camel", "duck"]

console.log(animals.slice(1, 5));
// Output: Array ["bison", "camel", "duck", "elephant"]
```

6.10 Arrays and Functions

Throughout the lesson we went over arrays being mutable, or changeable. What happens if we try to change an array inside a function? Does the array keep the change after the function call or is it scoped to inside the function?

Take a look at the following example where we call `.push()` on an array inside a function. Recall, the `.push()` method mutates, or changes, an array:

```
const flowers = ["peony", "daffodil", "marigold"];

function addFlower(arr) {
  arr.push("lily");
}

addFlower(flowers);
console.log(flowers); // Output: ["peony", "daffodil", "marigold", "lily"]
```

Let's go over what happened in the example:

- The `flowers` array that has 3 elements.
- The function `addFlower()` has a parameter of `arr` uses `.push()` to add a `'lily'` element into `arr`.
- We call `addFlower()` with an argument of `flowers` which will execute the code inside `addFlower`.
- We check the value of `flowers` and it now includes the `'lily'` element! The array was mutated!

So when you pass an array into a function, if the array is mutated inside the function, that change will be maintained outside the function as well. You might also see this concept explained as *pass-by-reference* since what we are actually passing the function is a reference to where the variable memory is stored and changing the memory.

6.11 Nested Arrays

Earlier we mentioned that arrays can store other arrays. When an array contains another array it is known as a *nested array*. Examine the example below:

```
const nestedArr = [[1], [2, 3]];
```

To access the nested arrays we can use bracket notation with the index value, just like we did to access any other element:

```
const nestedArr = [[1], [2, 3]];

console.log(nestedArr[1]); // Output: [2, 3]
```

Notice that `nestedArr[1]` will grab the element in index 1 which is the array `[2, 3]`. Then, if we wanted to access the elements within the nested array we can *chain*, or add on, more bracket notation with index values.

```
const nestedArr = [[1], [2, 3]];

console.log(nestedArr[1]); // Output: [2, 3]
console.log(nestedArr[1][0]); // Output: 2
```

In the second `console.log()` statement, we have two bracket notations chained to `nestedArr`. We know that `nestedArr[1]` is the array `[2, 3]`. Then to grab the first element from that array, we use `nestedArr[1][0]` and we get the value of `2`.

6.12 Review Arrays

In this chapter, we learned these concepts regarding arrays:

- Arrays are lists that store data in JavaScript.
- Arrays are created with brackets `[]`.
- Each item inside of an array is at a numbered position, or index, starting at `0`.
- We can access one item in an array using its index, with syntax like: `myArray[0]`.
- We can also change an item in an array using its index, with syntax like `myArray[0] = 'new string';`
- Arrays have a `length` property, which allows you to see how many items are in an array.
- Arrays have their own methods, including `.push()` and `.pop()`, which add and remove items from an array, respectively.
- Arrays have many methods that perform different tasks, such as `.slice()` and `.shift()`, you can find documentation at the Mozilla Developer Network website.
- Some built-in methods are mutating, meaning the method will change the array, while others are not mutating. You can always check the documentation.
- Variables that contain arrays can be declared with `let` or `const`. Even when declared with `const`, arrays are still mutable. However, a variable declared with `const` cannot be reassigned.
- Arrays mutated inside of a function will keep that change even outside the function.
- Arrays can be nested inside other arrays.
- To access elements in nested arrays chain indices using bracket notation.

Learning how to work with and manipulate arrays will help you work with chunks of data.

7 Loops

7.1 Loops

A *loop* is a programming tool that repeats a set of instructions until a specified condition, called a *stopping condition* is reached. As a programmer, you will find that you rely on loops all the time! You'll hear the generic term *iterate* when referring to loops; *iterate* simply means “to repeat”.

When we need to reuse a task in our code, we often bundle that action in a function. Similarly, when we see that a process has to repeat multiple times in a row, we write a loop. Loops allow us to create efficient code that automates processes to make scalable, manageable programs.

Loops iterate or repeat an action until a specific condition is met. When the condition is met, the loop stops and the computer moves on to the next part of the program.

7.2 The For Loop

Instead of writing out the same code over and over, loops allow us to tell computers to repeat a given block of code on its own. One way to give computers these instructions is with a `for` loop.

The typical `for` loop includes an *iterator variable* that usually appears in all three expressions. The iterator variable is initialized, checked against the stopping condition, and assigned a new value on each loop iteration. Iterator variables can have any name, but it is best practice to use a descriptive variable name.

A `for` loop contains three expressions separated by `;` inside the parentheses:

1. an *initialization* starts the loop and can also be used to declare the iterator variable.
2. a *stopping condition* is the condition that the iterator variable is evaluated against— if the condition evaluates to `true` the code block will run, and if it evaluates to `false` the code will stop.
3. an *iteration statement* is used to update the iterator variable on each loop.

The `for` loop syntax looks like this:

```
for (let counter = 0; counter < 4; counter++) {  
  console.log(counter);  
}
```

In this example, the output would be the following:

```
0  
1  
2  
3
```

Let us break down the example:

- The initialization is `let counter = 0`, so the loop will start counting at `0`.
- The stopping condition is `counter < 4`, meaning the loop will run as long as the iterator variable, `counter`, is less than 4.

- The iteration statement is `counter++`. This means after each loop, the value of `counter` will increase by 1. For the first iteration `counter` will equal 0, for the second iteration `counter` will equal 1, and so on.
- The code block is inside of the curly braces, `console.log(counter)`, will execute until the condition evaluates to `false`. The condition will be `false` when counter is greater than or equal to 4 — the point that the condition becomes false is sometimes called the *stop condition*.

This `for` loop makes it possible to write 0, 1, 2, and 3 programmatically.

7.3 Looping in Reverse

What if we want the `for` loop to log 3, 2, 1, and then 0? With simple modifications to the expressions, we can make our loop run backward.

To run a backward `for` loop, we must:

- Set the iterator variable to the highest desired value in the initialization expression.
- Set the stopping condition for when the iterator variable is less than the desired amount.
- The iterator should decrease in intervals after each iteration.

```
// The loop below loops from 3 to 0.
for (let counter = 3; counter >= 0; counter--){
  console.log(counter);
}
```

7.4 Looping through Arrays

`for` loops are very handy for iterating over data structures. For example, we can use a `for` loop to perform the same operation on each element on an array. Arrays hold lists of data, like customer names or product information. Imagine we owned a store and wanted to increase the price of every product in our catalog. That could be a lot of repeating code, but by using a `for` loop to iterate through the array we could accomplish this task easily.

To loop through each element in an array, a `for` loop should use the array's `.length` property in its condition. Check out the example below to see how `for` loops iterate on arrays:

```
const animals = ["Grizzly Bear", "Sloth", "Sea Lion"];
for (let i = 0; i < animals.length; i++){
  console.log(animals[i]);
}
```

This example would give you the following output:

```
Grizzly Bear
Sloth
Sea Lion
```

In the loop above, we have named our iterator variable `i`. This is a variable naming convention you will see in a lot of loops. When we use `i` to iterate through arrays we can think of it as being short-hand for the word *index*. Notice how our stopping condition checks that `i` is less than `animals.length`. Remember

that arrays are zero-indexed, the index of the last element of an array is equivalent to the length of that array minus 1. If we tried to access an element at the index of `animals.length` we will have gone too far. With `for` loops, it's easier for us to work with elements in arrays.

7.5 Nested Loops

When we have a loop running inside another loop, we call that a *nested loop*. One use for a nested `for` loop is to compare the elements in two arrays. For each round of the outer `for` loop, the inner `for` loop will run completely.

Let us look at an example of a nested `for` loop:

```
const myArray = [6, 19, 20];
const yourArray = [19, 81, 2];
for (let i = 0; i < myArray.length; i++) {
  for (let j = 0; j < yourArray.length; j++) {
    if (myArray[i] === yourArray[j]) {
      console.log("Both loops have the number: " + yourArray[j])
    }
  }
};
```

Let us think about what is happening in the nested loop in the example above. For each element in the outer loop array, `myArray`, the inner loop will run in its entirety comparing the current element from the outer array, `myArray[i]`, to each element in the inner array, `yourArray[j]`. When it finds a match, it prints a string to the console.

7.6 The While Loop

To start, let us convert a `for` loop into a `while` loop:

```
// A for loop that prints 1, 2, and 3
for (let counterOne = 1; counterOne < 4; counterOne++){
  console.log(counterOne);
}

// A while loop that prints 1, 2, and 3
let counterTwo = 1;
while (counterTwo < 4) {
  console.log(counterTwo);
  counterTwo++;
}
```

Let us break down what was happening with the `while` loop syntax:

- The `counterTwo` variable is declared before the loop. We can access it inside our `while` loop since it is in the global scope.
- We start our loop with the keyword `while` followed by our stopping condition, or *test condition*. This will be evaluated before each round of the loop. While the condition evaluates to `true`, the block will continue to run. Once it evaluates to `false` the loop will stop.
- Next, we have our loop's code block which prints `counterTwo` to the console and increments `counterTwo`.

What would happen if we did not increment `counterTwo` inside our block? If we did not include this, `counterTwo` would always have its initial value, `1`. That would mean the testing condition `counterTwo < 4` would always evaluate to `true` and our loop would never stop running. This is called an *infinite loop* and it is something we always want to **avoid**. Infinite loops can take up all of your computer's processing power potentially freezing your computer.

So you may be wondering when to use a `while` loop. The syntax of a `for` loop is ideal when we know how many times the loop should run, but we do not always know this in advance. Think of eating like a `while` loop: when you start taking bites, you do not know the exact number you will need to become full. Rather you will eat while you are hungry. In situations when we want a loop to execute an undetermined number of times, `while` loops are the best choice.

7.7 Do...While Statements

In some cases, you want a piece of code to run at least once and then loop based on a specific condition after its initial run. This is where the `do...while` statement comes in. A `do...while` statement says to do a task once and then keep doing it until a specified condition is no longer met. The syntax for a `do...while` statement looks like this:

```
let countString = "";
let i = 0;

do {
  countString = countString + i;
  i++;
} while (i < 5);

console.log(countString);
```

In this example, the code block makes changes to the `countString` variable by appending the string form of the `i` variable to it. First, the code block after the `do` keyword is executed once. Then the condition is evaluated. If the condition evaluates to `true`, the block will execute again. The looping stops when the condition evaluates to `false`.

Note that the `while` and `do...while` loop are different. Unlike the `while` loop, `do...while` will run at least once whether or not the condition evaluates to `true`.

```
const firstMessage = "I will print!";
const secondMessage = "I will not print!";

// A do while with a stopping condition that evaluates to false
do {
  console.log(firstMessage)
} while (true === false);
// Output: I will print!

// A while loop with a stopping condition that evaluates to false
while (true === false){
  console.log(secondMessage)
};
// No Output
```

7.8 The break Keyword

Imagine we are looking to adopt a dog. We plan to go to the shelter every day for a year and then give up. But what if we meet our dream dog on day 65? We do not want to keep going to the shelter for the next 300 days just because our original plan was to go for a whole year. In our code, when we want to stop a loop from continuing to execute even though the original stopping condition we wrote for our loop has not been met, we can use the keyword `break`.

The `break` keyword allows programs to “break” out of the loop from within the loop’s block. Let us check out the syntax of a break keyword:

```
for (let i = 0; i < 99; i++) {  
  if (i > 2 ) {  
    break;  
  }  
  console.log("Banana.");  
}  
  
console.log("Orange you glad I broke out the loop!");
```

This is the output for the above code:

```
Banana.  
Banana.  
Banana.  
Orange you glad I broke out the loop!
```

`break` statements can be especially helpful when we are looping through large data structures! With breaks, we can add test conditions besides the stopping condition, and exit the loop when they are met.

7.9 Review

In this chapter, we learned how to write cleaner code with loops.

- Loops perform repetitive actions so we do not have to code that process manually every time.
- How to write `for` loops with an iterator variable that increments or decrements.
- How to use a `for` loop to iterate through an array.
- A nested `for` loop is a loop inside another loop.
- `while` loops allow for different types of stopping conditions.
- Stopping conditions are crucial for avoiding infinite loops.
- `do...while` loops run code at least once— only checking the stopping condition after the first execution.
- The `break` keyword allows programs to leave a loop during the execution of its block.

8 Higher-Order Functions

8.1 Introduction

We are often unaware of the number of assumptions we make when we communicate with other people in our native languages. If we told you to “count to three,” we would expect you to say or think the numbers one, two and three. We assumed you would know to start with “one” and end with “three”. With programming, we are faced with needing to be more explicit with our directions to the computer. Here’s how we might tell the computer to “count to three”:

```
for (let i = 1; i<=3; i++) {  
  console.log(i)  
}
```

When we speak to other humans, we share a vocabulary that gives us quick ways to communicate complicated concepts. When we say “bake”, it calls to mind a familiar subroutine—preheating an oven, putting something into an oven for a set amount of time, and finally removing it. This allows us to *abstract* away a lot of the details and communicate key concepts more concisely. Instead of listing all those details, we can say, “We baked a cake,” and still impart all that meaning to you.

In programming, we can accomplish “abstraction” by writing functions. In addition to allowing us to reuse our code, functions help to make clear, readable programs. If you encountered `countToThree()` in a program, you might be able to quickly guess what the function did without having to stop and read the function’s body.

We are also going to learn about another way to add a level of abstraction to our programming: *higher-order functions*. *Higher-order functions* are functions that accept other functions as arguments and/or return functions as output. This enables us to build abstractions on other abstractions, just like “We hosted a birthday party” is an abstraction that may build on the abstraction “We made a cake.”

In summary, using more abstraction in our code allows us to write more modular code which is easier to read and debug.

8.2 Functions as Data

JavaScript functions behave like any other data type in the language; we can assign functions to variables, and we can reassign them to new variables.

Below, we have an annoyingly long function name that hurts the readability of any code in which it is used. Let us pretend this function does important work and needs to be called repeatedly!

```
const announceThatIAmDoingImportantWork = () => {  
  console.log("I am doing very important work!");  
};
```

What if we wanted to rename this function without sacrificing the source code? We can re-assign the function to a variable with a suitably short name:

```
const busy = announceThatIAmDoingImportantWork;  
  
busy(); // This function call barely takes any space!
```

`busy` is a variable that holds a reference to our original function. If we could look up the address in memory of `busy` and the address in memory of `announceThatIAmDoingImportantWork` they would point to the same place. Our new `busy()` function can be invoked with parentheses as if that was the name we originally gave our function.

Notice how we assign `announceThatIAmDoingImportantWork` without parentheses as the value to the `busy` variable. We want to assign the value of the function itself, not the value it returns when invoked. In JavaScript, functions are first class objects. This means that, like other objects you have encountered, JavaScript functions can have properties and methods. Since functions are a type of object, they have properties such as `.length` and `.name` and methods such as `.toString()`. You can see more about the methods and properties of functions in the documentation.

Functions are special because we can invoke them, but we can still treat them like any other type of data.

8.3 Functions as Parameters

Since functions can behave like any other type of data in JavaScript, it might not surprise you to learn that we can also pass functions (into other functions) as parameters. A *higher-order function* is a function that either accepts functions as parameters, returns a function, or both! We call the functions that get passed in as parameters and invoked *callback functions* because they get called during the execution of the higher-order function.

When we pass a function in as an argument to another function, we do not invoke it. Invoking the function would evaluate to the return value of that function call. With callbacks, we pass in the function itself by typing the function name *without* the parentheses (that would evaluate to the result of calling the function):

```
const timeFuncRuntime = funcParameter => {
  let t1 = Date.now();
  funcParameter();
  let t2 = Date.now();
  return t2 - t1;
}

const addOneToOne = () => 1 + 1;

timeFuncRuntime(addOneToOne);
```

We wrote a higher-order function, `timeFuncRuntime()`. It takes in a function as an argument, saves a starting time, invokes the callback function, records the time after the function was called, and returns the time the function took to run by subtracting the starting time from the ending time.

This higher-order function could be used with any callback function which makes it a potentially powerful piece of code.

We then invoked `timeFuncRuntime()` first with the `addOneToOne()` function - note how we passed in `addOneToOne` and did not invoke it.

Let us look at another example:

```
timeFuncRuntime(() => {  
  for (let i = 10; i>0; i--){  
    console.log(i);  
  }  
});
```

In this example, we invoked `timeFuncRuntime()` with an anonymous function that counts backwards from 10. Anonymous functions can be arguments too.

8.4 Review

By thinking about functions as data and learning about higher-order functions, you have taken important steps in being able to write clean, modular code and take advantage of JavaScript's flexibility. Let us review what we learned in this chapter:

- Abstraction allows us to write complicated code in a way that's easy to reuse, debug, and understand for human readers.
- We can work with functions the same way we would any other type of data including reassigning them to new variables.
- JavaScript functions are first-class objects, so they have properties and methods like any object.
- Functions can be passed into other functions as parameters.
- A higher-order function is a function that either accepts functions as parameters, returns a function, or both

9 Iterators

9.1 Introduction to Iterators

Imagine you had a grocery list and you wanted to know what each item on the list was. You would have to scan through each row and check for the item. This common task is similar to what we have to do when we want to iterate over, or loop through, an array. One tool at our disposal is the `for` loop. However, we also have access to built-in array methods which make looping easier.

The built-in JavaScript array methods that help us iterate are called *iteration methods*, at times referred to as *iterators*. Iterators are methods called on arrays to manipulate elements and return values.

In this chapter, you will learn the syntax for these methods, their return values, how to use the documentation to understand them, and how to choose the right iterator method for a given task.

9.2 The `.forEach()` Method

The first iteration method we learn is `.forEach()`. Aptly named, `.forEach()` will execute the same code for each element of an array.

```
const groceries = ['brown sugar', 'salt',  
                  'cranberries', 'walnuts'];  
  
groceries.forEach(function(groceryItem){  
  console.log(' - ' + groceryItem);  
});
```

IDENTIFIER

ARRAY

KEY

- Iterator
- Callback function

The code above will log a nicely formatted list of the groceries to the console. Let us explore the syntax of invoking `.forEach()`.

- `groceries.forEach()` calls the `forEach` method on the `groceries` array.
- `.forEach()` takes an argument of callback function. Remember, a callback function is a function passed as an argument into another function.
- `.forEach()` loops through the array and executes the callback function for each element. During each execution, the current element is passed as an argument to the callback function.
- The return value for `.forEach()` will always be `undefined`.

Another way to pass a callback for `.forEach()` is to use arrow function syntax.

```
groceries.forEach(groceryItem => console.log(groceryItem));
```

We can also define a function beforehand to be used as the callback function.

```
function printGrocery(element){
  console.log(element);
}

groceries.forEach(printGrocery);
```

The above example uses a function declaration but you can also use a function expression or arrow function as well.

All three code snippets do the same thing. In each array iteration method, we can use any of the three examples to supply a callback function as an argument to the iterator. It is good to be aware of the different ways to pass in callback functions as arguments in iterators because developers have different stylistic preferences. Nonetheless, due to the strong adoption of ES6, we will be using arrow function syntax in the later exercises.

```
const fruits = ["mango", "papaya", "pineapple", "apple"];

// Iterate over fruits below
fruits.forEach(item => console.log("I want to eat a " + item));
// Output:
// I want to eat a mango
// I want to eat a papaya
// I want to eat a pineapple
// I want to eat a apple
```

9.3 The .map() Method

The second iterator to cover is `.map()`. When `.map()` is called on an array, it takes an argument of a callback function and returns a new array! Take a look at an example of calling `.map()`:

```
const numbers = [1, 2, 3, 4, 5];

const bigNumbers = numbers.map(number => {
  return number * 10;
});
```

`.map()` works in a similar manner to `.forEach()`—the major difference is that `.map()` returns a new array.

In the example above:

- `numbers` is an array of numbers.
- `bigNumbers` will store the return value of calling `.map()` on `numbers`.
- `numbers.map` will iterate through each element in the `numbers` array and pass the element into the callback function.
- `return number * 10` is the code we wish to execute upon each element in the array. This will save each value from the `numbers` array, multiplied by `10`, to a new array.

If we take a look at `numbers` and `bigNumbers`:


```
console.log(numbers); // Output: [1, 2, 3, 4, 5]
console.log(bigNumbers); // Output: [10, 20, 30, 40, 50]
```

Notice that the elements in `numbers` were not altered and `bigNumbers` is a new array.

9.4 The `.filter()` Method

Another useful iterator method is `.filter()`. Like `.map()`, `.filter()` returns a new array. However, `.filter()` returns an array of elements after filtering out certain elements from the original array. The callback function for the `.filter()` method should return `true` or `false` depending on the element that is passed to it. The elements that cause the callback function to return `true` are added to the new array. Take a look at the following example:

```
const words = ["chair", "music", "pillow", "brick", "pen", "door"];

const shortWords = words.filter(word => {
  return word.length < 6;
});
```

- `words` is an array that contains string elements.
- `const shortWords =` declares a new variable that will store the returned array from invoking `.filter()`.
- The callback function is an arrow function has a single parameter, `word`. Each element in the `words` array will be passed to this function as an argument.
- `word.length < 6;` is the condition in the callback function. Any `word` from the `words` array that has fewer than 6 characters will be added to the `shortWords` array.

Let's also check the values of `words` and `shortWords`:

```
console.log(words);
// Output: ["chair", "music", "pillow", "brick", "pen", "door"]

console.log(shortWords);
// Output: ["chair", "music", "brick", "pen", "door"]
```

Observe how `words` was not mutated, i.e. changed, and `shortWords` is a new array.

9.5 The `.findIndex()` Method

We sometimes want to find the location of an element in an array. That is where the `.findIndex()` method comes in. Calling `.findIndex()` on an array will return the index of the first element that evaluates to `true` in the callback function.

```
const jumbledNums = [123, 25, 78, 5, 9];

const lessThanTen = jumbledNums.findIndex(num => {
  return num < 10;
});
```

- `jumbledNums` is an array that contains elements that are numbers.
- `const lessThanTen =` declares a new variable that stores the returned index number from invoking `.findIndex()`.

- The callback function is an arrow function has a single parameter, `num`. Each element in the `jumbledNums` array will be passed to this function as an argument.
- `num < 10;` is the condition that elements are checked against. `.findIndex()` will return the index of the first element which evaluates to `true` for that condition.

Let us take a look at what `lessThanTen` evaluates to:

```
console.log(lessThanTen); // Output: 3
```

If we check what element has index of 3:

```
console.log(jumbledNums[3]); // Output: 5
```

Great, the element in index `3` is the number `5`. This makes sense since `5` is the first element that is less than 10.

If there is not a single element in the array that satisfies the condition in the callback, then `.findIndex()` will return `-1`.

```
const greaterThan1000 = jumbledNums.findIndex(num => {
  return num > 1000;
});

console.log(greaterThan1000); // Output: -1
```

9.6 The `.reduce()` Method

Another widely used iteration method is `.reduce()`. The `.reduce()` method returns a single value after iterating through the elements of an array, thereby reducing the array. Take a look at the example below:

```
const numbers = [1, 2, 4, 10];

const summedNums = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue
})

console.log(summedNums) // Output: 17
```

Here are the values of `accumulator` and `currentValue` as we iterate through the numbers array:

Iteration	accumulator	currentValue	return value
First	1	2	3
Second	3	4	7
Third	7	10	17

Now let us go over the use of `.reduce()` from the example above:

- `numbers` is an array that contains numbers.
- `summedNums` is a variable that stores the returned value of invoking `.reduce()` on numbers.
- `numbers.reduce()` calls the `.reduce()` method on the `numbers` array and takes in a callback function as argument.

- The callback function has two parameters, `accumulator` and `currentValue`. The value of `accumulator` starts off as the value of the first element in the array and the `currentValue` starts as the second element. To see the value of `accumulator` and `currentValue` change, review the chart above.
- As `.reduce()` iterates through the array, the return value of the callback function becomes the `accumulator` value for the next iteration, `currentValue` takes on the value of the current element in the looping process.

The `.reduce()` method can also take an optional second parameter to set an initial value for `accumulator` (remember, the first argument is the callback function). For instance:

```
const numbers = [1, 2, 4, 10];

const summedNums = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue
}, 100) // <- Second argument for .reduce()

console.log(summedNums); // Output: 117
```

Here is an updated chart that accounts for the second argument of `100`:

Iteration	accumulator	currentValue	return value
First	100	1	101
Second	101	2	103
Third	103	4	107
Fourth	107	10	117

9.7 Iterator Documentation

There are many additional built-in array methods, a complete list of which is on the MDN's Array iteration methods page. The documentation for each method contains several sections:

1. A short definition.
2. A block with the correct syntax for using the method.
3. A list of parameters the method accepts or requires.
4. The return value of the function.
5. An extended description.
6. Examples of the method's use.
7. Other additional information.

9.8 More Iterators

Let us briefly learn two other popular iterators. Below we will explore `some()` and `every()`.

9.8.1 The `.some()` Method

The `some()` method tests whether at least one element in the array passes the test implemented by the provided function. It returns a Boolean value.

```
const array = [1, 2, 3, 4, 5];

// checks whether an element is even
const even = (element) => element % 2 === 0;

console.log(array.some(even)); // Output: true
```

9.8.2 The .every() Method

The `every()` method tests whether all elements in the array pass the test implemented by the provided function. It returns a Boolean value.

```
const isBelowThreshold = (currentValue) => currentValue < 40;

const array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(isBelowThreshold)); // Output: true
```

9.9 Review

We have learned a number of useful methods in this chapter as well as how to use the JavaScript documentation from the Mozilla Developer Network to discover and understand additional methods.

- `.forEach()` is used to execute the same code on every element in an array but does not change the array and returns undefined.
- `.map()` executes the same code on every element in an array and returns a new array with the updated elements.
- `.filter()` checks every element in an array to see if it meets certain criteria and returns a new array with the elements that return truthy for the criteria.
- `.findIndex()` returns the index of the first element of an array which satisfies a condition in the callback function. It returns -1 if none of the elements in the array satisfies the condition.
- `.reduce()` iterates through an array and takes the values of the elements and returns a single value.
- All iterator methods takes a callback function that can be pre-defined, or a function expression, or an arrow function.
- You can visit the Mozilla Developer Network to learn more about iterator methods (and all other parts of JavaScript!).

10 Objects

10.1 Introduction to Objects

It is time to learn more about the basic structure that permeates nearly every aspect of JavaScript programming: objects.

You are probably already more comfortable with objects than you think, because JavaScript loves objects. Many components of the language are actually objects under the hood, and even the parts that are not—like strings or numbers—can still act like objects in some instances.

There are only seven fundamental data types in JavaScript, and six of those are the primitive data types: string, number, boolean, null, undefined, and symbol. With the seventh type, objects, we open our code to more complex possibilities. We can use JavaScript objects to model real-world things, like a basketball, or we can use objects to build the data structures that make the web possible.

At their core, JavaScript objects are containers storing related data and functionality, but that deceptively simple task is extremely powerful in practice. You have been using the power of objects all along, but now it is time to understand the mechanics of objects and start making your own.

10.2 Creating Object Literals

Objects can be assigned to variables just like any JavaScript type. We use curly braces, `{}`, to designate an *object literal*:

```
let spaceship = {}; // spaceship is an empty object
```

We fill an object with unordered data. This data is organized into *key-value pairs*. A key is like a variable name that points to a location in memory that holds a value. A key's value can be of any data type in the language including functions or other objects.

We make a key-value pair by writing the key's name, or *identifier*, followed by a colon and then the value. We separate each key-value pair in an object literal with a comma (`,`). Keys are strings, but when we have a key that does not have any special characters in it, JavaScript allows us to omit the quotation marks:

```
let spaceship = {  
    'Fuel Type': 'diesel',  
    color: 'silver'  
};
```

└── PROPERTIES

● OBJECT ● KEY ● VALUE

```
// An object literal with two key-value pairs
let spaceship = {
  "Fuel Type": "diesel",
  color: "silver"
};
```

The `spaceship` object has two properties `Fuel Type` and `color`. `'Fuel Type'` has quotation marks because it contains a space character.

10.3 Accessing Properties

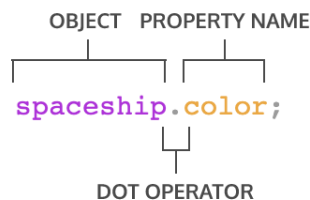
There are two ways we can access an object's property. Let us explore the first way— dot notation, `.`.

You have used dot notation to access the properties and methods of built-in objects and data instances:

```
"hello".length; // Returns 5
```

With property dot notation, we write the object's name, followed by the dot operator and then the property name (key):

```
let spaceship = {
  homePlanet: "Earth",
  color: "silver"
};
spaceship.homePlanet; // Returns "Earth",
spaceship.color; // Returns "silver",
```



If we try to access a property that does not exist on that object, `undefined` will be returned.

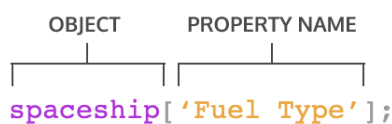
```
spaceship.favoriteIcecream; // Returns undefined
```

10.4 Bracket Notation

The second way to access a key's value is by using bracket notation, `[]`. You have used bracket notation when indexing an array:

```
["A", "B", "C"][0]; // Returns "A"
```

To use bracket notation to access an object's property, we pass in the property name (key) as a string.



We **must** use bracket notation when accessing keys that have numbers, spaces, or special characters in them. Without bracket notation in these situations, our code would throw an error.

```
let spaceship = {
  "Fuel Type": "Turbo Fuel",
  "Active Duty": true,
  homePlanet: "Earth",
  numCrew: 5
};
spaceship["Active Duty"]; // Returns true
spaceship["Fuel Type"]; // Returns "Turbo Fuel"
spaceship["numCrew"]; // Returns 5
spaceship["!!!!!!!!!!!!!!"]; // Returns undefined
```

With bracket notation you can also use a variable inside the brackets to select the keys of an object. This can be especially helpful when working with functions:

```
let returnAnyProp = (objectName, propName) => objectName[propName];

returnAnyProp(spaceship, "homePlanet"); // Returns "Earth"
```

If we tried to write our `returnAnyProp()` function with dot notation (`objectName.propName`) the computer would look for a key of `'propName'` on our object and not the value of the `propName` parameter.

10.5 Property Assignment

Once we have defined an object, we are not stuck with all the properties we wrote. Objects are mutable meaning we can update them after we create them. We can use either dot notation, `.`, or bracket notation, `[]`, and the assignment operator, `=` to add new key-value pairs to an object or change an existing property.

OBJECT	PROPERTY NAME	ASSIGNMENT OPERATOR	VALUE
<code>spaceship</code>	<code>['Fuel Type']</code>	<code>=</code>	<code>'vegetable oil'</code>
 <code>spaceship.color = 'gold';</code>			

One of two things can happen with property assignment:

- If the property already exists on the object, whatever value it held before will be replaced with the newly assigned value.
- If there was no property with that name, a new property will be added to the object.

It is important to know that although we cannot reassign an object declared with `const`, we can still mutate it, meaning we can add new properties and change the properties that are there.

```
const spaceship = {type: "shuttle"};
spaceship = {type: "alien"}; // TypeError: Assignment to constant variable.
spaceship.type = "alien"; // Changes the value of the type property
spaceship.speed = "Mach 5";
// Creates a new key of "speed" with a value of "Mach 5"
```

You can delete a property from an object with the `delete` operator.

```
const spaceship = {
  "Fuel Type": "Turbo Fuel",
  homePlanet: "Earth",
  mission: "Explore the universe"
};

delete spaceship.mission; // Removes the mission property
```

10.6 Methods

When the data stored on an object is a function we call that a *method*. A property is what an object has, while a method is what an object does. Object methods seem familiar because we have been using them all along. For example `console` is a global javascript object and `.log()` is a method on that object. `Math` is also a global javascript object and `.floor()` is a method on it.

We can include methods in our object literals by creating ordinary, comma-separated key-value pairs. The key serves as our method's name, while the value is an anonymous function expression.

```
const alienShip = {
  invade: function () {
    console.log("Hello! We have come to dominate your planet.")
  }
};
```

With the new method syntax introduced in ES6 we can omit the colon and the `function` keyword.

```
const alienShip = {
  invade () {
    console.log("Hello! We have come to dominate your planet.")
  }
};
```

Object methods are invoked by appending the object's name with the dot operator followed by the method name and parentheses:

```
alienShip.invade(); // Prints "Hello! We have come to dominate your planet."
```

10.7 Nested Objects

In application code, objects are often nested— an object might have another object as a property which in turn could have a property that is an array of even more objects.

In our `spaceship` object, we want a `crew` object. This will contain all the crew members who do important work on the craft. Each of those `crew` members are objects themselves. They have properties like `name`, and `degree`, and they each have unique methods based on their roles. We can also nest other objects in the `spaceship` such as a `telescope` or nest details about the spaceship's computers inside a parent `nanoelectronics` object.

```
const spaceship = {
  telescope: {
```



```

    yearBuilt: 2018,
    model: "91031-XLT",
    focalLength: 2032
  },
  crew: {
    captain: {
      name: "Sandra",
      degree: "Computer Engineering",
      encourageTeam() { console.log("We got this!") }
    }
  },
  engine: {
    model: "Nimbus2000"
  },
  nanoelectronics: {
    computer: {
      terabytes: 100,
      monitors: "HD"
    },
    "back-up": {
      battery: "Lithium",
      terabytes: 50
    }
  }
};

```

We can chain operators to access nested properties. We will have to pay attention to which operator makes sense to use in each layer. It can be helpful to pretend you are the computer and evaluate each expression from left to right so that each operation starts to feel a little more manageable.

```
spaceship.nanoelectronics["back-up"].battery; // Returns "Lithium"
```

In the preceding code:

- First the computer evaluates `spaceship.nanoelectronics`, which results in an object containing the `back-up` and `computer` objects.
- We accessed the `back-up` object by appending `["back-up"]`.
- The `back-up` object has a `battery` property, accessed with `.battery` which returned the value stored there: `"Lithium"`.

10.8 Pass By Reference

Objects are *passed by reference*. This means when we pass a variable assigned to an object into a function as an argument, the computer interprets the parameter name as pointing to the space in memory holding that object. As a result, functions which change object properties actually mutate the object permanently (even when the object is assigned to a `const` variable).

```

const spaceship = {
  homePlanet : "Earth",
  color : "silver"
};

```

```
let paintIt = obj => {
  obj.color = "glorious gold"
};

paintIt(spaceship);

spaceship.color // Returns "glorious gold"
```

Our function `paintIt()` permanently changed the color of our `spaceship` object. However, reassignment of the `spaceship` variable would not work in the same way:

```
let spaceship = {
  homePlanet : "Earth",
  color : "red"
};
let tryReassignment = obj => {
  obj = {
    identified : false,
    "transport type": "flying"
  }
  console.log(obj) // Prints {"identified": false, "transport type": "flying"}
};
tryReassignment(spaceship) // The attempt at reassignment does not work.
spaceship // Still returns {homePlanet : "Earth", color : "red"};

spaceship = {
  identified : false,
  "transport type": "flying"
}; // Regular reassignment still works.
```

Let us look at what happened in the code example:

- We declared this `spaceship` object with `let`. This allowed us to reassign it to a new object with `identified` and `'transport type'` properties with no problems.
- When we tried the same thing using a function designed to reassign the object passed into it, the reassignment didn't stick (even though calling `console.log()` on the object produced the expected result).
- When we passed `spaceship` into that function, `obj` became a reference to the memory location of the `spaceship` object, but *not* to the `spaceship` variable. This is because the `obj` parameter of the `tryReassignment()` function is a variable in its own right. The body of `tryReassignment()` has no knowledge of the `spaceship` variable at all!
- When we did the reassignment in the body of `tryReassignment()`, the `obj` variable came to refer to the memory location of the object `{'identified': false, 'transport type': 'flying'}`, while the `spaceship` variable was completely unchanged from its earlier value.

10.9 Looping Through Objects

Loops are programming tools that repeat a block of code until a condition is met. We learned how to iterate through arrays using their numerical indexing, but the key-value pairs in objects are not ordered. JavaScript has given us alternative solution for iterating through objects with the `for...in` syntax.

`for...in` will execute a given block of code for each property in an object.

```
let spaceship = {
  crew: {
    captain: {
      name: "Lily",
      degree: "Computer Engineering",
      cheerTeam() { console.log("You got this!") }
    },
    "chief officer": {
      name: "Dan",
      degree: "Aerospace Engineering",
      agree() { console.log("I agree, captain!") }
    },
    medic: {
      name: "Clementine",
      degree: "Physics",
      announce() { console.log("Jets on!") } },
    translator: {
      name: "Shauna",
      degree: 'Conservation Science',
      powerFuel() { console.log("The tank is full!") }
    }
  }
};
// for...in
for (let crewMember in spaceship.crew) {
  console.log(`${crewMember}: ${spaceship.crew[crewMember].name}`)
};
```

Our `for...in` will iterate through each element of the `spaceship.crew` object. In each iteration, the variable `crewMember` is set to one of `spaceship.crew`'s keys, enabling us to log a list of crew members' role and name.

10.10 Review

Let us review what we learned in this chapter:

- Objects store collections of *key-value* pairs.
- Each key-value pair is a property—when a property is a function it is known as a method.
- An object literal is composed of comma-separated key-value pairs surrounded by curly braces.
- You can access, add or edit a property within an object by using dot notation or bracket notation.
- We can add methods to our object literals using key-value syntax with anonymous function expressions as values or by using the new ES6 method syntax.
- We can navigate complex, nested objects by chaining operators.
- Objects are mutable—we can change their properties even when they are declared with `const`.

- Objects are passed by reference— when we make changes to an object passed into a function, those changes are permanent.
- We can iterate through objects using the `For...in` syntax.

11 Advanced Objects

11.1 Advanced Objects Introduction

Remember, objects in JavaScript are containers that store data and functionality. In this lesson, we will build upon the fundamentals of creating objects and explore some advanced concepts. In this chapter we will learn to:

- use the `this` keyword.
- convey privacy in JavaScript methods.
- define getters and setters in objects.
- create factory functions.
- use destructuring techniques.

11.2 The `this` Keyword

Objects are collections of related data and functionality. We store that functionality in methods on our objects:

```
const goat = {
  dietType: "herbivore",
  makeSound() {
    console.log("baaa");
  }
};
```

In our `goat` object we have a `.makeSound()` method. We can invoke the `.makeSound()` method on `goat`.

```
goat.makeSound(); // Prints baaa
```

We have a `goat` object that can print `baaa` to the console. What if we wanted to add a new method to our `goat` object called `.diet()` that prints the goat's `dietType`?

```
const goat = {
  dietType: "herbivore",
  makeSound() {
    console.log("baaa");
  },
  diet() {
    console.log(dietType);
  }
};
goat.diet();
// Output will be "ReferenceError: dietType is not defined"
```

That is strange, why is `dietType` not defined even though it is a property of `goat`? That is because inside the scope of the `.diet()` method, we do not automatically have access to other properties of the `goat` object. Here is where the `this` keyword comes to the rescue. If we change the `.diet()` method to use the `this`, the `.diet()` works.

```
const goat = {
  dietType: "herbivore",
  makeSound() {
    console.log("baaa");
  },
  diet() {
    console.log(this.dietType);
  }
};

goat.diet(); // Output: herbivore
```

The `this` keyword references the *calling object* which provides access to the calling object's properties. In the example above, the calling object is `goat` and by using `this` we are accessing the `goat` object itself, and then the `dietType` property of `goat` by using property dot notation.

11.3 Arrow Functions and this

We saw in the previous exercise that for a method, the calling object is the object the method belongs to. If we use the `this` keyword in a method then the value of `this` is the calling object. However, it becomes more complicated when we start using arrow functions for methods. Take a look at the example below:

```
const goat = {
  dietType: "herbivore",
  makeSound() {
    console.log("baaa");
  },
  diet: () => {
    console.log(this.dietType);
  }
};

goat.diet(); // Prints undefined
```

In the comment, you can see that `goat.diet()` would log `undefined`. What happened? Notice that in the `.diet()` is defined using an arrow function.

Arrow functions inherently *bind*, or tie, an already defined `this` value to the function itself that is NOT the calling object. In the code snippet above, the value of `this` is the global object, or an object that exists in the *global scope*, which does not have a `dietType` property and therefore returns `undefined`.

To read more about either arrow functions or the global object check out the MDN documentation of the global object and arrow functions.

The key takeaway from the example above is to *avoid* using arrow functions when using `this` in a method!

11.4 Privacy

Accessing and updating properties is fundamental in working with objects. However, there are cases in which we don't want other code simply accessing and updating an object's properties. When discussing *privacy* in objects, we define it as the idea that only certain properties should be mutable or able to change

in value.

Certain languages have privacy built-in for objects, but JavaScript does not have this feature. Rather, JavaScript developers follow naming conventions that signal to other developers how to interact with a property. One common convention is to place an underscore `_` before the name of a property to mean that the property should not be altered. Here's an example of using `_` to prepend a property.

```
const bankAccount = {
  _amount: 1000
}
```

In the example above, the `_amount` is not intended to be directly manipulated. Even so, it is still possible to reassign `_amount`:

```
bankAccount._amount = 1000000;
```

Later, we will cover the use of methods called *getters* and *setters*. Both methods are used to respect the intention of properties prepended, or began, with `_`. Getters can return the value of internal properties and setters can safely reassign property values.

11.5 Getters

Getters are methods that get and return the internal properties of an object. But they can do more than just retrieve the value of a property. Let us take a look at a getter method:

```
const person = {
  _firstName: "John",
  _lastName: "Doe",
  get fullName() {
    if (this._firstName && this._lastName){
      return `${this._firstName} ${this._lastName}`;
    } else {
      return "Missing a first name or a last name.";
    }
  }
}

// To call the getter method:
person.fullName; // 'John Doe'
```

Notice that in the getter method above:

- We use the `get` keyword followed by a function.
- We use an `if...else` conditional to check if both `_firstName` and `_lastName` exist (by making sure they both return truthy values) and then return a different value depending on the result.
- We can access the calling object's internal properties using `this`. In `fullName`, we are accessing both `this._firstName` and `this._lastName`.
- In the last line we call `fullName` on `person`. In general, getter methods do not need to be called with a set of parentheses. Syntactically, it looks like we are accessing a property.

Now that we have gone over syntax, let's discuss some notable advantages of using getter methods:

- Getters can perform an action on the data when getting a property.
- Getters can return different values using conditionals.
- In a getter, we can access the properties of the calling object using `this`.
- The functionality of our code is easier for other developers to understand.

Another thing to keep in mind when using getter (and setter) methods is that properties cannot share the same name as the getter/setter function. If we do so, then calling the method will result in an infinite call stack error. One workaround is to add an underscore before the property name like we did in the example above.

11.6 Setters

Along with getter methods, we can also create *setter* methods which reassign values of existing properties within an object. Let us see an example of a setter method:

```
const person = {
  _age: 37,
  set age(newAge){
    if (typeof newAge === "number"){
      this._age = newAge;
    } else {
      console.log("You must assign a number to age");
    }
  }
};
```

Notice that in the example above:

- We can perform a check for what value is being assigned to `this._age`.
- When we use the setter method, only values that are numbers will reassign `this._age`.
- There are different outputs depending on what values are used to reassign `this._age`.

Then to use the setter method:

```
person.age = 40;
console.log(person._age); // Logs: 40
person.age = "40"; // Logs: You must assign a number to age
```

Setter methods like `age` do not need to be called with a set of parentheses. Syntactically, it looks like we are reassigning the value of a property.

Like getter methods, there are similar advantages to using setter methods that include checking input, performing actions on properties, and displaying a clear intention for how the object is supposed to be used. Nonetheless, even with a setter method, it is still possible to directly reassign properties. For example, in the example above, we can still set `_age` directly:

```
person._age = "forty-five"
console.log(person._age); // Prints forty-five
```


11.7 Factory Functions

So far we have been creating objects individually, but there are times where we want to create many instances of an object quickly. Here is where factory functions come in. A real world factory manufactures multiple copies of an item quickly and on a massive scale. A factory function is a function that returns an object and can be reused to make multiple object instances. Factory functions can also have parameters allowing us to customize the object that gets returned.

Let us say we wanted to create an object to represent monsters in JavaScript. There are many different types of monsters and we could go about making each monster individually but we can also use a factory function to make our lives easier. To achieve this diabolical plan of creating multiple monsters objects, we can use a factory function that has parameters:

```
const monsterFactory = (name, age, energySource, catchPhrase) => {
  return {
    name: name,
    age: age,
    energySource: energySource,
    scare() {
      console.log(catchPhrase);
    }
  }
};
```

In the `monsterFactory` function above, it has four parameters and returns an object that has the properties: `name`, `age`, `energySource`, and `scare()`. To make an object that represents a specific monster like a ghost, we can call `monsterFactory` with the necessary arguments and assign the return value to a variable:

```
const ghost = monsterFactory("Ghouly", 251, "ectoplasm", "BOO!");
ghost.scare(); // "BOO!"
```

Now we have a `ghost` object as a result of calling `monsterFactory()` with the needed arguments. With `monsterFactory` in place, we do not have to create an object literal every time we need a new monster. Instead, we can invoke the `monsterFactory` function with the necessary arguments to make a monster.

11.8 Property Value Shorthand

ES6 introduced some new shortcuts for assigning properties to variables known as *destructuring*.

In the previous chapter, we created a factory function that helped us create objects. We had to assign each property a key and value even though the key name was the same as the parameter name we assigned to it. To remind ourselves, here is a truncated version of the factory function:

```
const monsterFactory = (name, age) => {
  return {
    name: name,
    age: age
  }
};
```

Imagine if we had to include more properties, that process would quickly become tedious. But we can use a destructuring technique, called *property value shorthand*, to save ourselves some keystrokes. The example below works exactly like the example above:

```
const monsterFactory = (name, age) => {
  return {
    name,
    age
  }
};
```

Notice that we do not have to repeat ourselves for property assignments.

11.9 Destructured Assignment

We often want to extract key-value pairs from objects and save them as variables. Take for example the following object:

```
const vampire = {
  name: "Dracula",
  residence: "Transylvania",
  preferences: {
    day: "stay inside",
    night: "satisfy appetite"
  }
};
```

If we wanted to extract the `residence` property as a variable, we could use the following code:

```
const residence = vampire.residence;
console.log(residence); // Prints "Transylvania"
```

However, we can also take advantage of a destructuring technique called *destructured assignment* to save ourselves some keystrokes. In destructured assignment we create a variable with the name of an object's key that is wrapped in curly braces `{ }` and assign to it the object. Take a look at the example below:

```
const { residence } = vampire;
console.log(residence); // Prints "Transylvania"
```

Look back at the `vampire` object's properties in the first code example. Then, in the example above, we declare a new variable `residence` that extracts the value of the `residence` property of `vampire`. When we log the value of `residence` to the console, `'Transylvania'` is printed.

We can even use destructured assignment to grab nested properties of an object:

```
const { day } = vampire.preferences;
console.log(day); // Prints "stay inside"
```

Variables created by destructured assignment reference the object directly.

```
const robot = {
  functionality: {
    beep() {
      console.log("Beep Boop");
    },
  },
};
const { functionality } = robot;
functionality.beep(); //Output: Beep Boop
```

Since `functionality` is referencing `robot.functionality`, we can call the methods available to `robot.functionality` simply through `functionality`.

11.10 Built-in Object Methods

Previously, we have been creating instances of objects that have their own methods. But, we can also take advantage of built-in methods for Objects. For example, we have access to object instance methods like: `.hasOwnProperty()`, `.valueOf()`, and many others. Check out: MDN's object instance documentation.

There are also useful Object class methods such as `Object.assign()`, `Object.entries()`, and `Object.keys()` just to name a few. For a comprehensive list, browse: MDN's object instance documentation.

Below we will explore `Object.keys()`, `Object.entries()`, and `Object.assign()`.

11.10.1 The `Object.keys()` Method

The `Object.keys()` method returns an array of a given object's own enumerable property **names**, iterated in the same order that a normal loop would.

```
const object1 = {
  a: "somestring",
  b: 42,
  c: false
};

console.log(Object.keys(object1)); // Output: Array ["a", "b", "c"]
```

11.10.2 The `Object.entries()` Method

The `Object.entries()` method returns an array of a given object's own enumerable string-keyed property `[key, value]` pairs, in the same order as that provided by a `for...in` loop. (The only important difference is that a `for...in` loop enumerates properties in the prototype chain as well).

The order of the array returned by `Object.entries()` does not depend on how an object is defined. If there is a need for certain ordering, then the array should be sorted first.

Like `Object.entries(obj).sort((a, b) => b[0].localeCompare(a[0]));`.

```
const object1 = {
  a: "somestring",
  b: 42
};

for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`);
}

// Output:
// "a: somestring"
// "b: 42"
// order is not guaranteed
```

11.10.3 The Object.assign() Method

The `Object.assign()` method copies all enumerable own properties from one or more source objects to a target object. It returns the target object.

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target); // Output: Object { a: 1, b: 4, c: 5 }
console.log(returnedTarget); // Output: Object { a: 1, b: 4, c: 5 }
```

11.11 Review

- The object that a method belongs to is called the *calling object*.
- The `this` keyword refers the calling object and can be used to access properties of the calling object.
- Methods do not automatically have access to other internal properties of the calling object.
- The value of `this` depends on where the `this` is being accessed from.
- We cannot use arrow functions as methods if we want to access other internal properties.
- JavaScript objects do not have built-in privacy, rather there are conventions to follow to notify other developers about the intent of the code.
- The usage of an underscore before a property name means that the original developer did not intend for that property to be directly changed.
- Setters and getter methods allow for more detailed ways of accessing and assigning properties.
- Factory functions allow us to create object instances quickly and repeatedly.
- There are different ways to use object destructuring: one way is the property value shorthand and another is destructured assignment. As with any concept, it is a good skill to learn how to use the documentation with objects.

12 Classes

12.1 Introduction to Classes

JavaScript is an *object-oriented programming* (OOP) language we can use to model real-world items. In this chapter, you will learn how to make *classes*. Classes are a tool that developers use to quickly produce similar objects.

Take, for example, an object representing a dog named `halley`. This dog's `name` (a key) is `"Halley"` (a value) and has an `age` (another key) of `3` (another value). We create the `halley` object below:

```
let halley = {
  _name: "Halley",
  _behavior: 0,

  get name() {
    return this._name;
  },

  get behavior() {
    return this._behavior;
  },

  incrementBehavior() {
    this._behavior++;
  }
}
```

Now, imagine you own a dog daycare and want to create a catalog of all the dogs who belong to the daycare. Instead of using the syntax above for every dog that joins the daycare, we can create a `Dog` class that serves as a template for creating new `Dog` objects. For each new dog, you can provide a value for their name. Afterwards, we will introduce inheritance and static methods — two features that will make your code more efficient and meaningful.

```
class Dog {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }

  get name() {
    return this._name;
  }
  get behavior() {
    return this._behavior;
  }

  incrementBehavior() {
    this._behavior ++;
  }
}

const halley = new Dog("Halley");
console.log(halley.name); // Print name value to console
```

```
console.log(halley.behavior); // Print behavior value to console
```

12.2 Constructor

Previously, we saw the creation of a class called `Dog`, and used it to produce a `Dog` object.

Although you may see similarities between class and object syntax, there is one important method that sets them apart. It is called the *constructor* method. JavaScript calls the `constructor()` method every time it creates a *new instance* of a class.

```
class Dog {  
  constructor(name) {  
    this.name = name;  
    this.behavior = 0;  
  }  
}
```

- `Dog` is the name of our class. By convention, we capitalize and CamelCase class names.
- JavaScript will invoke the `constructor()` method every time we create a new instance of our `Dog` class.
- This `constructor()` method accepts one argument, `name`.
- Inside of the `constructor()` method, we use the `this` keyword. In the context of a class, `this` refers to an instance of that class. In the `Dog` class, we use `this` to set the value of the `Dog` instance's `name` property to the `name` argument.
- Under `this.name`, we create a property called `behavior`, which will keep track of the number of times a dog misbehaves. The `behavior` property is always initialized to zero.

12.3 Instance

Now, let us create class instances. An *instance* is an object that contains the property names and methods of a class, but with unique property values. Let us look at our `Dog` class example.

```
class Dog {  
  constructor(name) {  
    this.name = name;  
    this.behavior = 0;  
  }  
}  
  
const halley = new Dog("Halley"); // Create new Dog instance  
console.log(halley.name); // Log the name value saved to halley  
// Output: "Halley"
```

Below our `Dog` class, we use the `new` keyword to create an instance of our `Dog` class. Let us consider the line of code step-by-step.

- We create a new variable named `halley` that will store an instance of our `Dog` class.
- We use the `new` keyword to generate a new instance of the `Dog` class. The `new` keyword calls the `constructor()`, runs the code inside of it, and then returns the new instance.

- We pass the 'Halley' string to the `Dog` constructor, which sets the `name` property to 'Halley'.
- Finally, we log the value saved to the `name` key in our `halley` object, which logs 'Halley' to the console.

12.4 Methods

At this point, we have a `Dog` class that spins up objects with `name` and `behavior` properties. Below, we will add getters and a method to bring our class to life. Class method and getter syntax is the same as it is for objects **except you can not include commas between methods**.

```
class Dog {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }

  get name() {
    return this._name;
  }

  get behavior() {
    return this._behavior;
  }

  incrementBehavior() {
    this._behavior++;
  }
}
```

In the example above, we add getter methods for `name` and `behavior`. Notice, we also prepended our property names with underscores (`_name` and `_behavior`), which indicate these properties should not be accessed directly. Under the getters, we add a method named `.incrementBehavior()`. When you call `.incrementBehavior()` on a `Dog` instance, it adds `1` to the `_behavior` property. Between each of our methods, we did not include commas.

12.5 Method Calls

Finally, let us use our new methods to access and manipulate data from `Dog` instances.

```
class Dog {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }

  get name() {
    return this._name;
  }

  get behavior() {
    return this._behavior;
  }
}
```

```

    incrementBehavior() {
        this._behavior++;
    }
}
const halley = new Dog('Halley');

```

In the example above, we create the `Dog` class, then create an instance, and save it to a variable named `halley`. The syntax for calling methods and getters on an instance is the same as calling them on an object — append the instance with a period, then the property or method name. For methods, you must also include opening and closing parentheses.

Let us take a moment to create two `Dog` instances and call our `.incrementBehavior()` method on one of them.

```

let nikko = new Dog("Nikko"); // Create dog named Nikko
nikko.incrementBehavior(); // Add 1 to nikko instance's behavior
let bradford = new Dog("Bradford"); // Create dog name Bradford
console.log(nikko.behavior); // Logs 1 to the console
console.log(bradford.behavior); // Logs 0 to the console

```

In the example above, we create two new `Dog` instances, `nikko` and `bradford`. Because we increment the behavior of our `nikko` instance, but not `bradford`, accessing `nikko.behavior` returns `1` and accessing `bradford.behavior` returns `0`.

12.6 Inheritance

Imagine our doggy daycare is so successful that we decide to expand the business and open a kitty daycare. Before the daycare opens, we need to create a `Cat` class so we can quickly generate `Cat` instances. We know that the properties in our `Cat` class (`name`, `behavior`) are similar to the properties in our `Dog` class, though, there will be some differences.

Let us say that our `Cat` class looks like this:

```

class Cat {
    constructor(name, usesLitter) {
        this._name = name;
        this._usesLitter = usesLitter;
        this._behavior = 0;
    }

    get name() {
        return this._name;
    }

    get usesLitter() {
        return this._usesLitter;
    }

    get behavior() {
        return this._behavior;
    }
}

```



```

    incrementBehavior() {
        this._behavior++;
    }
}

```

In the example above, we create a `Cat` class. It shares a couple of properties (`_name` and `_behavior`) and a method (`.incrementBehavior()`) with the `Dog` class from earlier chapters. The `Cat` class also contains one additional property (`_usesLitter`), that holds a boolean value to indicate whether a cat can use their litter box.

When multiple classes share properties or methods, they become candidates for *inheritance* — a tool developers use to decrease the amount of code they need to write. With inheritance, you can create a *parent* class (also known as a superclass) with properties and methods that multiple *child* classes (also known as subclasses) share. The child classes inherit the properties and methods from their parent class.

Let us abstract the shared properties and methods from our `Cat` and `Dog` classes into a parent class called `Animal`.

```

class Animal {
    constructor(name) {
        this._name = name;
        this._behavior = 0;
    }

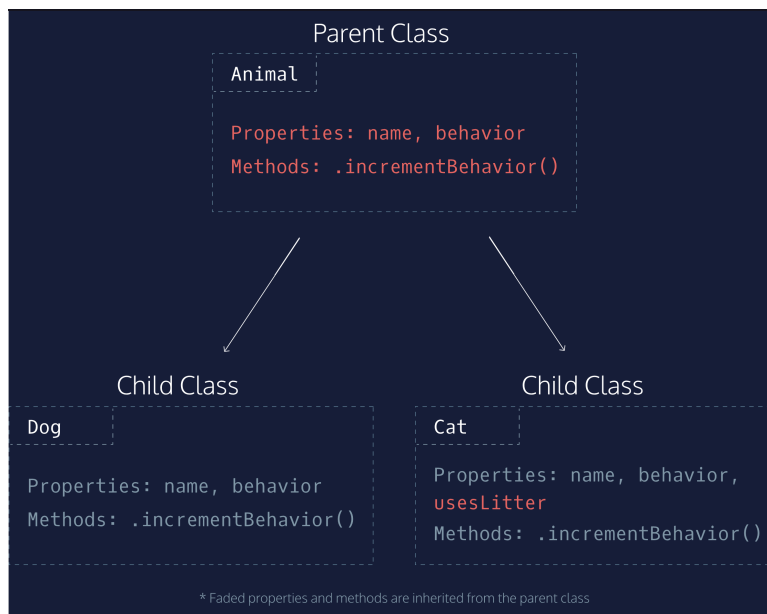
    get name() {
        return this._name;
    }

    get behavior() {
        return this._behavior;
    }

    incrementBehavior() {
        this._behavior++;
    }
}

```

In the example above, the `Animal` class contains the properties and methods that the `Cat` and `Dog` classes share (`name`, `behavior`, `.incrementBehavior()`). The diagram to the right shows the relationships we want to create between the `Animal`, `Cat`, and `Dog` classes.



Now that we have these shared properties and methods in the parent `Animal` class, we can extend them to the subclass, `Cat`.

```
class Cat extends Animal {
  constructor(name, usesLitter) {
    super(name);
    this._usesLitter = usesLitter;
  }
}
```

In the example above, we create a new class named `Cat` that extends the `Animal` class. Let us pay special attention to the new keywords: `extends` and `super`.

- The `extends` keyword makes the methods of the animal class available inside the cat class.
- The constructor, called when you create a new `Cat` object, accepts two arguments, `name` and `usesLitter`.
- The `super` keyword calls the constructor of the parent class. In this case, `super(name)` passes the name argument of the `Cat` class to the constructor of the `Animal` class. When the `Animal` constructor runs, it sets `this._name = name;` for new `Cat` instances.
- `_usesLitter` is a new property that is unique to the `Cat` class, so we set it in the `Cat` constructor.

Notice, we call `super` on the first line of our `constructor()`, then set the `usesLitter` property on the second line. In a `constructor()`, you must always call the `super` method before you can use the `this` keyword — if you do not, JavaScript will throw a reference error. To avoid reference errors, it is best practice to call `super` on the first line of subclass constructors.

Below, we create a new `Cat` instance and call its name with the same syntax as we did with the `Dog` class:

```
const bryceCat = new Cat("Bryce", false);
console.log(bryceCat._name); // output: Bryce
```

In the example above, we create a new instance the `Cat` class, named `bryceCat`. We pass it `'Bryce'` and `false` for our `name` and `usesLitter` arguments. When we call `console.log(bryceCat._name)` our program prints, `Bryce`.

In the example above, we abandoned best practices by calling our `_name` property directly. In the next exercise, we will address this by calling an inherited getter method for our `name` property.

Now that we know how to create an object that inherits properties from a parent class let us turn our attention to methods. When we call `extends` in a class declaration, all of the parent methods are available to the child class.

Here, we extend our `Animal` class to a `Cat` subclass.

```

class Animal {
  constructor(name) {
    this._name = name;
    this._behavior = 0;
  }

  get name() {
    return this._name;
  }

  get behavior() {
    return this._behavior;
  }

  incrementBehavior() {
    this._behavior++;
  }
}

class Cat extends Animal {
  constructor(name, usesLitter) {
    super(name);
    this._usesLitter = usesLitter;
  }
}

const bryceCat = new Cat("Bryce", false);

```

In the example above, our `Cat` class extends `Animal`. As a result, the `Cat` class has access to the `Animal` getters and the `.incrementBehavior()` method.

Also in the code above, we create a `Cat` instance named `bryceCat`. Because `bryceCat` has access to the `name` getter, the code below logs `'Bryce'` to the console.

```
console.log(bryceCat.name);
```

Since the `extends` keyword brings all of the parent's getters and methods into the child class, `bryceCat.name` accesses the `name` getter and returns the value saved to the `name` property.

Now consider a more involved example.

```

bryceCat.incrementBehavior(); // Call .incrementBehavior() on Cat instance
console.log(bryceCat.behavior); // Log value saved to behavior

```

`1` is logged to the console. Let us investigate why.

- The `Cat` class inherits the `_behavior` property, behavior getter, and the `.incrementBehavior()` method from the `Animal` class.
- When we created the `bryceCat` instance, the `Animal` constructor set the `_behavior` property to zero.
- The first line of code calls the inherited `.incrementBehavior()` method, which increases the `bryceCat` `_behavior` value from zero to one.

- The second line of code calls the `behavior` getter and logs the value saved to `_behavior` (1).

In addition to the inherited features, child classes can contain their own properties, getters, setters, and methods.

Below, we will add a `usesLitter` getter. The syntax for creating getters, setters, and methods is the same as it is in any other class.

```
class Cat extends Animal {
  constructor(name, usesLitter) {
    super(name);
    this._usesLitter = usesLitter;
  }

  get usesLitter() {
    return this._usesLitter;
  }
}
```

In the example above, we create a `usesLitter` getter in the `Cat` class that returns the value saved to `_usesLitter`. Compare the `Cat` class above to the one we created without inheritance, we decreased the number of lines required to create the `Cat` class by about half. It did require an extra class (`Animal`). However, the benefits (time saved, readability, efficiency) of inheritance grow as the number and size of your subclasses increase. One benefit is that when you need to change a method or property that multiple classes share, you can change the parent class, instead of each subclass.

Let us see how we would create an additional subclass, called `Dog`.

```
class Dog extends Animal {
  constructor(name) {
    super(name);
  }
}
```

This `Dog` class has access to the same properties, getters, setters, and methods as the `Dog` class we made without inheritance, and is a quarter the size. Now that we have abstracted animal daycare features, it's easy to see how you can extend `Animal` to support other classes, like `Rabbit`, `Bird` or `Snake`.

12.7 Static Methods

Sometimes you will want a class to have methods that are not available in individual instances, but that you can call directly from the class.

Take the `Date` class, for example — you can both create `Date` instances to represent whatever date you want, and call static methods, like `Date.now()` which returns the current date, directly from the class. The `.now()` method is static, so you can call it directly from the class, but not from an instance of the class.

Let us see how to use the `static` keyword to create a static method called `generateName` method in our `Animal` class:

```
class Animal {
  constructor(name) {
```

```

    this._name = name;
    this._behavior = 0;
  }

  static generateName() {
    const names = ["Angel", "Spike", "Buffy", "Willow", "Tara"];
    const randomNumber = Math.floor(Math.random()*5);
    return names[randomNumber];
  }
}

```

In the example above, we create a `static` method called `.generateName()` that returns a random name when it is called. Because of the `static` keyword, we can only access `.generateName()` by appending it to the `Animal` class.

We call the `.generateName()` method with the following syntax:

```
console.log(Animal.generateName()); // returns a name
```

You cannot access the `.generateName()` method from instances of the `Animal` class or instances of its subclasses.

```
const tyson = new Animal("Tyson");
tyson.generateName(); // TypeError
```

The example above will result in an error, because you cannot call static methods (`.generateName()`) on an instance (`tyson`).

12.8 Review: Classes

- *Classes* are templates for objects.
- Javascript calls a *constructor* method when we create a new instance of a class.
- *Inheritance* is when we create a parent class with properties and methods that we can extend to child classes.
- We use the `extends` keyword to create a subclass.
- The `super` keyword calls the `constructor()` of a parent class.
- Static methods are called on the class, but not on instances of the class.

13 Browser Compatibility and Transpilation

13.1 Introduction

Everyone is prompted to update your web browser every few months. A few reasons include addressing security vulnerabilities, adding features, and supporting new HTML, CSS, and JavaScript syntax. The reasons above imply there is a period before a software update is released when there are security vulnerabilities and unsupported language syntax. This chapter focuses on the latter. Specifically, how developers address the gap between the new JavaScript syntax that they use and the JavaScript syntax that web browsers recognize.

This has become a widespread concern for web developers since Ecma International, the organization responsible for standardizing JavaScript, released a new version of it in 2015, called ECMAScript2015, commonly referred to as ES6. Note, the 6 refers to the version of JavaScript and is not related to the year it was released (the previous version was ES5). Upon release, web developers quickly adopted the new ES6 syntax, as it improved readability and efficiency. However, ES6 was not supported by most web browsers, so developers ran into browser compatibility issues.

This chapter covers two important tools for addressing browser compatibility issues.

- [caniuse.com](#) — A website that provides data on web browser compatibility for HTML, CSS, and JavaScript features. You will learn how to use it to look up ES6 feature support.
- Babel — A Javascript library that you can use to convert new, unsupported JavaScript (ES6), into an older version (ES5) that is recognized by most modern browsers.

13.2 [caniuse.com](#)

Since Ecma's release of ECMAScript2015 (ES6), software companies have slowly added support for ES6 features and syntax. While most new browser versions support the majority of the ES6 library, there are still a couple sources of compatibility issues:

- Some users have not updated to the latest, ES6 supported web browser version.
- A few ES6 features, like modules, are still not supported by most web browsers.

Because companies add support for ES6 features gradually, it is important for you to know how to look up browser support on a feature-by-feature basis. The website [caniuse.com](#) is the best resource for finding browser compatibility information.

In [caniuse](#), you can enter an ES6 feature, like `let`, and see the percentage of browsers that recognize it. You can also see when each major web browser (Chrome, Safari, Edge, etc.) added support for the keyword.

13.3 Why ES6?

Before we learn how to set up a JavaScript project that converts ES6 to an older version, it is worth understanding a few of the reasons Ecma made such substantial updates. The version of JavaScript that preceded ES6 is called JavaScript ES5. Three reasons for the ES5 to ES6 updates are listed below:

- Readability and economy of code — The new syntax is often easier to understand (more readable) and requires fewer characters to create the same functionality (economy of code).

- Addresses sources of ES5 bugs — Some ES5 syntax led to common bugs. With ES6, Ecma introduced syntax that mitigates some of the most common pitfalls.
- A similarity to other programming languages — JavaScript ES6 is syntactically more similar to other object-oriented programming languages. This leads to less friction when experienced, non-JavaScript developers want to learn JavaScript.

Because ES6 addressed the above issues, Ecma knew that adoption by web developers would occur quickly, while web browser support lagged behind. To limit the impact of ES6 browser compatibility issues, Ecma made the new syntax backwards compatible, which means you can map JavaScript ES6 code to ES5.

13.4 Transpilation With Babel

We can manually convert ES6 code to ES5. Although manual conversion only take a few minutes, it is unsustainable as the size of the JavaScript file increases. Because ES6 is predictably backwards compatible, a collection of JavaScript programmers developed a JavaScript library called Babel that *transpiles* ES6 JavaScript to ES5. **Transpilation is the process of converting one programming language to another.** In the next chapters, we will learn how to use Babel to transpile the new, easy-to-write version of JavaScript (ES6) to the old, browser-compatible version of JavaScript (ES5).

```
npm install babel-cli
npm install babel-preset-env
npm run build
```

13.5 npm init

Next you will learn how to setup a JavaScript project that transpiles code when you run `colorbox-lightgraynpm run build` from the root directory of a JavaScript project. The first step is to place your ES6 JavaScript file in a directory called **src**. From your root directory, the path to the ES6 file is `./src/main.js`. The initial JavaScript project file structure is:

```
project
|_ src
|___ main.js
```

Before we install Babel, we need to setup our project to use the node package manager (npm). Developers use *npm* to access and manage Node packages. Node packages are directories that contain JavaScript code written by other developers. You can use these packages to reduce duplication of work and avoid bugs.

Before we can add Babel to our project directory, we need to run `npm init`. The `npm init` command creates a **package.json** file in the root directory. A **package.json** file contains information about the current JavaScript project. Some of this information includes:

- Metadata — This includes a project title, description, authors, and more.
- A list of node packages required for the project — If another developer wants to run your project, npm looks inside **package.json** and downloads the packages in this list.
- Key-value pairs for command line scripts — You can use npm to run these shorthand scripts to perform some process. Later, we will add a script that runs Babel and transpiles ES6 to ES5.

If you have Node installed on your computer, you can create a **package.json** file by typing `npm init` into the terminal. The terminal prompts you to fill in fields for the project's metadata (name, description,

etc.) You are not required to answer the prompts, though we recommend at minimum, you add your own title and description. If you do not want to fill in a field, you can press enter. npm will leave fill these fields with default values or leave them empty when it creates the **package.json** file.

After you run `npm init` your directory structure will contain the following files and folders:

```
project
|_ src
|___ main.js
|_ package.json
```

13.6 Install Node Packages

We use the npm `install` command to install new Node packages locally. The `install` command creates a folder called **node_modules** and copies the package files to it. The `install` command also installs all of the dependencies for the given package.

To install Babel, we need to `npm install` two packages, `babel-cli` and `babel-preset-env`. However, npm installs over one hundred other packages that are dependencies for Babel to run properly.

We install Babel with the following two commands:

```
npm install babel-cli -D
npm install babel-preset-env -D
```

The `babel-cli` package includes command line Babel tools, and the `babel-preset-env` package has the code that maps any JavaScript feature, ES6 and above (ES6+), to ES5.

The `-D` flag instructs npm to add each package to a property called `devDependencies` in **package.json**. Once the project's dependencies are listed in `devDependencies`, other developers can run your project without installing each package separately. Instead, they can simply run `npm install` — it instructs npm to look inside **package.json** and download all of the packages listed in `devDependencies`.

Once you `npm install` packages, you can find the Babel packages and all their dependencies in the **node_modules** folder. The new directory structure contains the following:

```
project
|_ node_modules
|___ .bin
|___ ...
|_ src
|___ main.js
|_ package.json
```

The `...` in the file structure above is a placeholder for 100+ packages that npm installed.

13.7 .babelrc

Now that we have downloaded the Babel packages, we need to specify the version of the source JavaScript code. We can specify the initial JavaScript version inside of a file named **.babelrc**. In your root directory, you can run `touch .babelrc` to create this file.

Your project directory contains the following folders and files:

```
project
|_ node_modules
|_ .bin
|_ ...
|_ src
|_ main.js
|_ .babelrc
|_ package.json
```

Inside **.babelrc** we need to define the preset for our source JavaScript file. The preset specifies the version of our initial JavaScript file. Usually, we want to transpile JavaScript code from versions ES6 and later (ES6+) to ES5. From this point on, we will refer to our source code as ES6+, because Ecma introduces new syntax with each new version of JavaScript.

To specify that we are transpiling code from an ES6+ source, we have to add the following JavaScript object into **.babelrc**: "presets": ["env"]

```
"presets": ["env"]
```

When you run Babel, it looks in **.babelrc** to determine the version of the initial JavaScript file. In this case, ["env"] instructs Babel to transpile any code from versions ES6 and later.

13.8 Babel Source Lib

There is one last step before we can transpile our code. We need to specify a script in **package.json** that initiates the ES6+ to ES5 transpilation.

Inside of the **package.json** file, there is a property named "scripts" that holds an object for specifying command line shortcuts. It looks like this:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
}, ...
```

In the code above, the "scripts" property contains an object with one property called "test". Below the "test" property, we will add a script that runs Babel like this:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "babel src -d lib"
}
```

In the "scripts" object above, we add a property called "build". The property's value, "babel src -d lib", is a command line method that transpiles ES6+ code to ES5. Let us consider each argument in the method call:

- **babel** — The Babel command call responsible for transpiling code.
- **src** — Instructs Babel to transpile all JavaScript code inside the src directory.
- **-d** — Instructs Babel to write the transpiled code to a directory.

- `lib` — Babel writes the transpiled code to a directory called `lib`.

13.9 Build

We are ready to transpile our code. Previously, we wrote the following script in `package.json`:

```
"build": "babel src -d lib"
```

Now, we need to call `build` from the command line to transpile and write ES5 code to a directory called `lib`. From the command line, we type:

```
npm run build
```

The command above runs the `build` script in `package.json`. Babel writes the ES5 code to a file named `main.js` (it is always the same name as the original file), inside of a folder called `lib`. The resulting directory structure is:

```
project
|_ lib
|___ main.js
|_ node_modules
|___ .bin
|___ ...
|_ src
|___ main.js
|_ .babelrc
|_ package.json
```

Notice, the directory contains a new folder named `lib`, with one file, called `main.js`.

The `npm run build` command will transpile all JavaScript files inside of the `src` folder. This is helpful as you build larger JavaScript projects — regardless of the number of JavaScript files, you only need to run one command (`npm run build`) to transpile all of your code.

13.10 Review

In this chapter, we learned about browser compatibility and transpilation. Let us review the key concepts:

- ES5 — The old JavaScript version that is supported by all modern web browsers.
- ES6 — The new(er) JavaScript version that is *not* supported by all modern web browsers. The syntax is more readable, similar to other programming languages, and addresses the source of common bugs in ES5.
- caniuse.com — a website you can use to look up HTML, CSS, and JavaScript browser compatibility information.
- Babel — A JavaScript package that transpiles JavaScript ES6+ code to ES5.
- `npm init` — A terminal command that creates a `package.json` file.
- `package.json` — A file that contains information about a JavaScript project.
- `npm install` — A command that installs Node packages.

- `babel-cli` — A Node package that contains command line tools for Babel.
- `babel-preset-env` — A Node package that contains ES6+ to ES5 syntax mapping information.
- `.babelrc` — A file that specifies the version of the JavaScript source code.
- “`build`” script — A `package.json` script that you use to transpile ES6+ code to ES5.
- `npm run build` — A command that runs the `build` script and transpiles ES6+ code to ES5.

For future reference, here is a list of the steps needed to set up a project for transpilation:

1. Initialize your project using `npm init` and create a directory called `src`
2. Install babel dependencies by running

```
npm install babel-cli -D
npm install babel-preset-env -D
```

3. Create a `.babelrc` file inside your project and add the following code inside it: `"presets": ["env"]`

```
"presets": ["env"]
```

4. Add the following script to your `scripts` object in `package.json`:

```
"build": "babel src -d lib"
```

5. Run `npm run build` whenever you want to transpile your code from your `src` to `lib` directories.

14 Intermediate JavaScript Modules

14.1 Hello Modules

JavaScript *modules* are reusable pieces of code that can be exported from one program and imported for use in another program. Modules are particularly useful for a number of reasons. By separating code with similar logic into files called modules, we can:

- find, fix, and debug code more easily;
- reuse and recycle defined logic in different parts of our application;
- keep information private and protected from other modules;
- and, importantly, prevent pollution of the global namespace and potential naming collisions, by cautiously selecting variables and behavior we load into a program.

In this chapter, we will cover two ways to implement modules in JavaScript: Node.js's `module.exports` and `require()` syntax, as well as the ES6 `import/export` syntax.

14.2 `module.exports`

We can get started with modules by defining a module in one file and making the module available for use in another file with Node.js `module.exports` syntax. Every JavaScript file run in Node has a local `module` object with an `exports` property used to define what should be exported from the file.

Below is an example of how to define a module and how to export it using the statement `module.exports`. In `menu.js` we write:

```
let Menu = {};  
Menu.specialty = "Roasted Beet Burger with Mint Sauce";  
  
module.exports = Menu;
```

Let us consider what this code means.

1. `let Menu = {};` creates the object that represents the module `Menu`. The statement contains an uppercase variable named `Menu` which is set equal to an empty object.
2. `Menu.specialty` is defined as a property of the `Menu` module. We add data to the `Menu` object by setting properties on that object and giving the properties a value.
3. `"Roasted Beet Burger with Mint Sauce";` is the value stored in the `Menu.specialty` property.
4. `module.exports = Menu;` exports the `Menu` object as a module. `module` is a variable that represents the module, and `exports` exposes the module as an object.

The pattern we use to export modules is thus:

1. Create an object to represent the module.
2. Add properties or methods to the module object.
3. Export the module with `module.exports`.

14.3 require()

To make use of the exported module and the behavior we define within it, we import the module into another file. In Node.js, use the `require()` function to import modules.

For instance, say we want the module to control the menu's data and behavior, and we want a separate file to handle placing an order. We would create a separate file `order.js` and import the `Menu` module from `menu.js` to `order.js` using `require()`. `require()` takes a file path argument pointing to the original module file.

In `order.js` we would write:

```
const Menu = require("./menu.js");

function placeOrder() {
  console.log("My order is: " + Menu.specialty);
}

placeOrder();
```

We now have the entire behavior of `Menu` available in `order.js`. Here, we notice:

1. In `order.js` we import the module by creating a `const` variable called `Menu` and setting it equal to the value of the `require()` function. We can set the name of this variable to anything we like, such as `menuItems`.
2. `require()` is a JavaScript function that loads a module. Its argument is the file path of the module: `./menu.js`. With `require()`, the `.js` extension is optional and will be assumed if it is not included.
3. The `placeOrder()` function then uses the `Menu` module. By calling `Menu.specialty`, we access the property `specialty` defined in the `Menu` module.

Taking a closer look, the pattern to import a module is:

1. Import the module with `require()` and assign it to a local variable.
2. Use the module and its properties within a program.

14.4 module.exports II

We can also wrap any collection of data and functions in an object, and export the object using `module.exports`. In `menu.js`, we could write:

```
module.exports = {
  specialty: "Roasted Beet Burger with Mint Sauce",
  getSpecialty: function() {
    return this.specialty;
  }
};
```

In the above code, notice:

1. `module.exports` exposes the current module as an object.
2. `specialty` and `getSpecialty` are properties on the object.

Then in **order.js**, we write:

```
const Menu = require("./menu.js");  
console.log(Menu.getSpecialty());
```

Here we can still access the behavior in the **Menu** module.

14.5 export default

Node.js supports `require()`/`module.exports`, but as of ES6, JavaScript supports a new more readable and flexible syntax for exporting modules. These are usually broken down into one of two techniques, *default export* and *named exports*.

We will begin with the first syntax, default export. The default export syntax works similarly to the `module.exports` syntax, allowing us to export one module per file.

Let us look at an example in **menu.js**.

```
let Menu = {};  
export default Menu;
```

1. `export default` uses the JavaScript `export` statement to export JavaScript objects, functions, and primitive data types.
2. **Menu** refers to the name of the **Menu** object, the object that we are setting the properties on within our modules.

When using ES6 syntax, we use `export default` in place of `module.exports`. Node.js does not support `export default` by default, so `module.exports` is usually used for Node.js development and ES6 syntax is used for front-end development. As with most ES6 features, it is common to transpile code since ES6 is not supported by all browsers.

14.6 import

ES6 module syntax also introduces the `import` keyword for importing objects. In our **order.js** example, we import an object like this:

```
import Menu from "./menu";
```

1. The `import` keyword begins the statement.
2. The keyword **Menu** here specifies the name of the variable to store the default export in.
3. `from` specifies where to load the module from.
4. `'./menu'` is the name of the module to load. When dealing with local files, it specifically refers to the name of the file without the extension of the file.

We can then continue using the **Menu** module in the **order.js** file.

14.7 Named Exports

ES6 introduced a second common approach to export modules. In addition to `export default`, *named exports* allow us to export data through the use of variables.

Let us see how this works. In `menu.js` we would be sure to give each piece of data a distinct variable name:

```
let specialty = "";
function isVegetarian() {
};
function isLowSodium() {
};

export { specialty, isVegetarian };
```

1. Notice that, when we use named exports, we are not setting the properties on an object. Each export is stored in its own variable.
2. `specialty` is a string object, while `isVegetarian` and `isLowSodium` are objects in the form of functions. Recall that in JavaScript, every function is in fact a function object.
3. `export { specialty, isVegetarian };` exports objects by their variable names. Notice the keyword `export` is the prefix.
4. `specialty` and `isVegetarian` are exported, while `isLowSodium` is not exported, since it is not specified in the export syntax.

14.8 Named Imports

To import objects stored in a variable, we use the `import` keyword and include the variables in a set of `{}`. In the `order.js` file, for example, we would write:

```
import { specialty, isVegetarian } from "../menu";

console.log(specialty);
```

1. Here `specialty` and `isVegetarian` are imported.
2. If we did not want to import either of these variables, we could omit them from the `import` statement.
3. We can then use these objects as in within our code. For example, we would use `specialty` instead of `Menu.specialty`.

14.9 Export Named Exports

Named exports are also distinct in that they can be exported as soon as they are declared, by placing the keyword `export` in front of variable declarations.

In `menu.js`

```
export let specialty = "";
export function isVegetarian() {
};
```



```
function isLowSodium() {  
};
```

1. The `export` keyword allows us to export objects upon declaration, as shown in `export let specialty` and `export function isVegetarian() {}`.
2. We no longer need an `export` statement at the bottom of our file, since this behavior is handled above.

14.10 Import Named Imports

To import variables that are declared, we simply use the original syntax that describes the variable name. In other words, exporting upon declaration does not have an impact on how we import the variables.

```
import { specialty, isVegetarian } from "menu";
```

14.11 Export as

Named exports also conveniently offer a way to change the name of variables when we export or import them. We can do this with the `as` keyword. Let us see how this works. In our `menu.js` example:

```
let specialty = "";  
let isVegetarian = function() {  
};  
let isLowSodium = function() {  
};  
  
export { specialty as chefsSpecial, isVegetarian as isVeg, isLowSodium };
```

In the above example, take a look at the `export` statement at the bottom of the file.

1. The `as` keyword allows us to give a variable name an alias as demonstrated in `specialty as chefsSpecial` and `isVegetarian as isVeg`.
2. Since we did not give `isLowSodium` an alias, it will maintain its original name.

14.12 Import as

To import named export aliases with the `as` keyword, we add the aliased variable in our import statement.

```
import { chefsSpecial, isVeg } from "./menu";
```

In `orders.js`

1. We import `chefsSpecial` and `isVeg` from the `Menu` object.
2. Here, note that we have an option to alias an object that was not previously aliased when exported. For example, the `isLowSodium` object that we exported could be aliased with the `as` keyword when imported:

```
import {isLowSodium as saltFree} from "Menu";
```

Another way of using aliases is to import the entire module as an alias:

```
import * as Carte from "./menu";

Carte.chefsSpecial;
Carte.isVeg();
Carte.isLowSodium();
```

1. This allows us to import an entire module from **menu.js** as an alias **Carte**.
2. In this example, whatever name we exported would be available to us as properties of that module. For example, if we exported the aliases **chefsSpecial** and **isVeg**, these would be available to us. If we did not give an alias to **isLowSodium**, we would call it as defined on the **Carte** module.

14.13 Combining Export Statements

We can also use named exports and default exports together. In **menu.js**:

```
let specialty = "";
function isVegetarian() {
};
function isLowSodium() {
};
function isGlutenFree() {
};

export { specialty as chefsSpecial, isVegetarian as isVeg };
export default isGlutenFree;
```

Here we use the keyword **export** to export the named exports at the bottom of the file. Meanwhile, we export the **isGlutenFree** variable using the **export default** syntax. This would also work if we exported most of the variables as declared and exported others with the **export default** syntax.

```
export let Menu = {};

export let specialty = "";
export let isVegetarian = function() {
};
export let isLowSodium = function() {
};
let isGlutenFree = function() {
};

export default isGlutenFree;
```

Here we use the **export** keyword to export the variables upon declaration, and again export the **isGlutenFree** variable using the **export default** syntax.

While it is better to avoid combining two methods of exporting, it is useful on occasion. For example, if you suspect developers may only be interested in importing a specific function and will not need to import the entire default export.

14.14 Combining Import Statements

We can import the collection of objects and functions with the same data. We can use an `import` keyword to import both types of variables as such:

```
import { specialty, isVegetarian, isLowSodium } from "./menu";  
import GlutenFree from "./menu";
```

14.15 Review

Let us review what we learned about JavaScript modules:

Modules in Node.js are reusable pieces of code that can be exported from one program and imported for use in another program.

- `module.exports` exports the module for use in another program.
- `require()` imports the module for use in the current program.

ES6 introduced a more flexible, easier syntax to export modules:

- default exports use `export default` to export JavaScript objects, functions, and primitive data types.
- named exports use the `export` keyword to export data in variables.
- named exports can be aliased with the `as` keyword.
- `import` is a keyword that imports any object, function, or data type.

15 JavaScript Promises

15.1 Introduction

In web development, asynchronous programming is notorious for being a challenging topic. An *asynchronous operation* is one that allows the computer to “move on” to other tasks while waiting for the asynchronous operation to complete. Asynchronous programming means that time-consuming operations do not have to bring everything else in our programs to a halt.

There are countless examples of asynchronicity in our everyday lives. Cleaning our house, for example, involves asynchronous operations such as a dishwasher washing our dishes or a washing machine washing our clothes. While we wait on the completion of those operations, we are free to do other chores.

Similarly, web development makes use of asynchronous operations. Operations like making a network request or querying a database can be time-consuming, but JavaScript allows us to execute other tasks while awaiting their completion.

This chapter will teach how modern JavaScript handles asynchronicity using the `Promise` object, introduced with ES6.

15.2 What is a Promise?

Promises are objects that represent the eventual outcome of an asynchronous operation. A `Promise` object can be in one of three states:

- **Pending:** The initial state—the operation has not completed yet.
- **Fulfilled:** The operation has completed successfully and the promise now has a *resolved value*. For example, a request’s promise might resolve with a JSON object as its value.
- **Rejected:** The operation has failed and the promise has a reason for the failure. This reason is usually an `Error` of some kind.

We refer to a promise as *settled* if it is no longer pending—it is either fulfilled or rejected. Let us think of a dishwasher as having the states of a promise:

- **Pending:** The dishwasher is running but has not completed the washing cycle.
- **Fulfilled:** The dishwasher has completed the washing cycle and is full of clean dishes.
- **Rejected:** The dishwasher encountered a problem (it did not receive soap) and returns unclean dishes.

If our dishwashing promise is fulfilled, we will be able to perform related tasks, such as unloading the clean dishes from the dishwasher. If it is rejected, we can take alternate steps, such as running it again with soap or washing the dishes by hand.

All promises eventually settle, enabling us to write logic for what to do if the promise fulfills or if it rejects.

15.3 Constructing a Promise Object

Let us construct a promise. To create a new `Promise` object, we use the `new` keyword and the `Promise` constructor method:

```
const executorFunction = (resolve, reject) => { };
const myFirstPromise = new Promise(executorFunction);
```

The `Promise` constructor method takes a function parameter called the *executor function* which runs automatically when the constructor is called. The executor function generally starts an asynchronous operation and dictates how the promise should be settled.

The executor function has two function parameters, usually referred to as the `resolve()` and `reject()` functions. The `resolve()` and `reject()` functions are not defined by the programmer. When the `Promise` constructor runs, JavaScript will pass **its own** `resolve()` and `reject()` functions into the executor function.

- `resolve` is a function with one argument. Under the hood, if invoked, `resolve()` will change the promise's status from `pending` to `fulfilled`, and the promise's resolved value will be set to the argument passed into `resolve()`.
- `reject` is a function that takes a reason or error as an argument. Under the hood, if invoked, `reject()` will change the promise's status from `pending` to `rejected`, and the promise's rejection reason will be set to the argument passed into `reject()`.

Let us look at an example executor function in a `Promise` constructor:

```
const executorFunction = (resolve, reject) => {
  if (someCondition) {
    resolve("I resolved!");
  } else {
    reject("I rejected!");
  }
}
const myFirstPromise = new Promise(executorFunction);
```

Let us break down what's happening above:

- We declare a variable `myFirstPromise`.
- `myFirstPromise` is constructed using `new Promise()` which is the `Promise` constructor method.
- `executorFunction()` is passed to the constructor and has two functions as parameters: `resolve` and `reject`.
- If `someCondition` evaluates to `true`, we invoke `resolve()` with the string `"I resolved!"`.
- If not, we invoke `reject()` with the string `"I rejected!"`.

In our example, `myFirstPromise` resolves or rejects based on a simple condition, but, in practice, promises settle based on the results of asynchronous operations. For example, a database request may fulfill with the data from a query or reject with an error thrown.

15.4 The Node `setTimeout()` Function

Knowing how to construct a promise is useful, but most of the time, knowing how to *consume*, or use, promises will be key. Rather than constructing promises, we will be handling `Promise` objects returned to us as the result of an asynchronous operation. These promises will start off pending but settle eventually.

Moving forward, we will be simulating this by providing you with functions that return promises which settle after some time. To accomplish this, we'll be using `setTimeout()`. `setTimeout()` is a Node API (a comparable API is provided by web browsers) that uses callback functions to schedule tasks to be performed after a delay. `setTimeout()` has two parameters: a callback function and a delay in milliseconds.

```
const delayedHello = () => {
  console.log("Hi! This is an asynchronous greeting!");
};

setTimeout(delayedHello, 2000);
```

Here, we invoke `setTimeout()` with the callback function `delayedHello()` and `2000`. In at least two seconds `delayedHello()` will be invoked. But why is it “at least” two seconds and not exactly two seconds?

This delay is performed asynchronously—the rest of our program will not stop executing during the delay. Asynchronous JavaScript uses something called the *event-loop*. After two seconds, `delayedHello()` is added to a line of code waiting to be run. Before it can run, any synchronous code from the program will run. Next, any code in front of it in the line will run. This means it might be more than two seconds before `delayedHello()` is actually executed.

Let us look at how we will be using `setTimeout()` to construct asynchronous promises:

```
const returnPromiseFunction = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {resolve("I resolved!")}, 1000);
  });
};

const prom = returnPromiseFunction();
```

In the example code, we invoked `returnPromiseFunction()` which returned a promise. We assigned that promise to the variable `prom`. Similar to the asynchronous promises you may encounter in production, `prom` will initially have a status of pending.

15.5 Consuming Promises

The initial state of an asynchronous promise is `pending`, but we have a guarantee that it will settle. How do we tell the computer what should happen then? Promise objects come with an aptly named `.then()` method. It allows us to say, “I have a promise, when it settles, **then** here’s what I want to happen...”

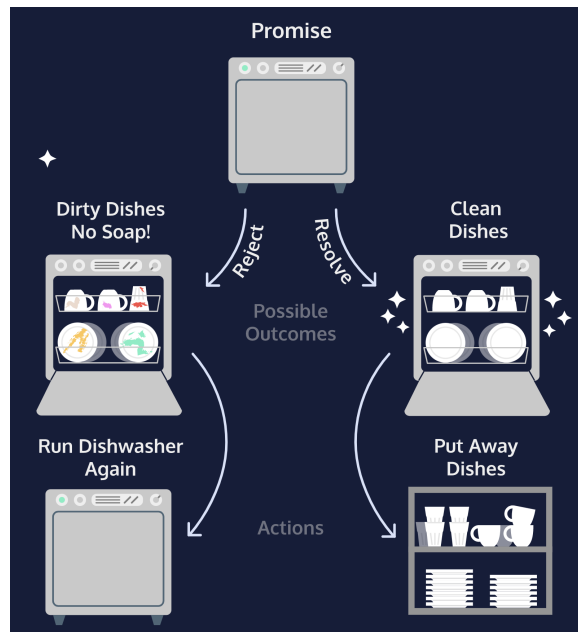
In the case of our dishwasher promise, the dishwasher will run **then**:

- If our promise rejects, this means we have dirty dishes, and we’ll add soap and run the dishwasher again.
- If our promise fulfills, this means we have clean dishes, and we’ll put the dishes away.

`.then()` is a higher-order function—it takes two callback functions as arguments. We refer to these callbacks as *handlers*. When the promise settles, the appropriate handler will be invoked with that settled value.

- The first handler, sometimes called `onFulfilled`, is a *success handler*, and it should contain the logic for the promise resolving.
- The second handler, sometimes called `onRejected`, is a *failure handler*, and it should contain the logic for the promise rejecting.

We can invoke `.then()` with one, both, or neither handler. This allows for flexibility, but it can also make for tricky debugging. If the appropriate handler is not provided, instead of throwing an error, `.then()` will just return a promise with the same settled value as the promise it was called on. One important feature of `.then()` is that it always returns a promise.



15.6 The onFulfilled and onRejected Functions

To handle a “successful” promise, or a promise that resolved, we invoke `.then()` on the promise, passing in a success handler callback function:

```
const prom = new Promise((resolve, reject) => {
  resolve("Yay!");
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};

prom.then(handleSuccess); // Prints: "Yay!"
```

Let us break down what is happening in the example code:

- `prom` is a promise which will resolve to `"Yay!"`.
- We define a function, `handleSuccess()`, which prints the argument passed to it.
- We invoke `prom`'s `.then()` function passing in our `handleSuccess()` function.
- Since `prom` resolves, `handleSuccess()` is invoked with `prom`'s resolved value, `"Yay"`, so `"Yay"` is logged to the console.

With typical promise consumption, we will not know whether a promise will resolve or reject, so we will need to provide the logic for either case. We can pass both an `onFulfilled` and `onRejected` callback to `.then()`.

```
let prom = new Promise((resolve, reject) => {
  let num = Math.random();
  if (num < .5 ){
    resolve("Yay!");
  } else {
    reject("Ohhh noooo!");
  }
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};

const handleFailure = (rejectionReason) => {
  console.log(rejectionReason);
};

prom.then(handleSuccess, handleFailure);
```

Let us break down what is happening in the example code:

- `prom` is a promise which will randomly either resolve with `"Yay!"` or reject with `"Ohhh noooo!"`.
- We pass two handler functions to `.then()`. The first will be invoked with `"Yay!"` if the promise resolves, and the second will be invoked with `"Ohhh noooo!"` if the promise rejects.

15.7 Using `catch()` with Promises

One way to write cleaner code is to follow a principle called *separation of concerns*. Separation of concerns means organizing code into distinct sections each handling a specific task. It enables us to quickly navigate our code and know where to look if something is not working.

Remember, `.then()` will return a promise with the same settled value as the promise it was called on if no appropriate handler was provided. This implementation allows us to separate our resolved logic from our rejected logic. Instead of passing both handlers into one `.then()`, we can chain a second `.then()` with a failure handler to a first `.then()` with a success handler and both cases will be handled.

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
```



```
.then(null, (rejectionReason) => {
  console.log(rejectionReason);
});
```

Since JavaScript does not mind whitespace, we follow a common convention of putting each part of this chain on a new line to make it easier to read. To create even more readable code, we can use a different promise function: `.catch()`.

The `.catch()` function takes only one argument, `onRejected`. In the case of a rejected promise, this failure handler will be invoked with the reason for rejection. Using `.catch()` accomplishes the same thing as using a `.then()` with only a failure handler.

Let us look at an example using `.catch()`:

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Let us break down what is happening in the example code:

- `prom` is a promise which randomly either resolves with `"Yay!"` or rejects with `"Ohhh noooo!"`.
- We pass a success handler to `.then()` and a failure handler to `.catch()`.
- If the promise resolves, `.then()`'s success handler will be invoked with `"Yay!"`.
- If the promise rejects, `.then()` will return a promise with the same rejection reason as the original promise and `.catch()`'s failure handler will be invoked with that rejection reason.

15.8 Chaining Multiple Promises

One common pattern we will see with asynchronous programming is multiple operations which depend on each other to execute or that must be executed in a certain order. We might make one request to a database and use the data returned to us to make another request and so on. Let us illustrate this with another cleaning example, washing clothes:

We take our dirty clothes and put them in the washing machine. If the clothes are cleaned, **then** we will want to put them in the dryer. After the dryer runs, if the clothes are dry, **then** we can fold them and put them away.

This process of chaining promises together is called *composition*. Promises are designed with composition in mind. Here is a simple promise chain in code:

```
firstPromiseFunction()
  .then((firstResolveVal) => {
    return secondPromiseFunction(firstResolveVal);
  })
  .then((secondResolveVal) => {
```

```
    console.log(secondResolveVal);
  });
```

Let us break down what is happening in the example:

- We invoke a function `firstPromiseFunction()` which returns a promise.
- We invoke `.then()` with an anonymous function as the success handler.
- Inside the success handler we **return** a new promise—the result of invoking a second function, `secondPromiseFunction()` with the first promise's resolved value.
- We invoke a second `.then()` to handle the logic for the second promise settling.
- Inside that `.then()`, we have a success handler which will log the second promise's resolved value to the console.

In order for our chain to work properly, we had to **return** the promise `secondPromiseFunction(firstResolveVal)`. This ensured that the return value of the first `.then()` was our second promise rather than the default return of a new promise with the same settled value as the initial.

Let us look at a more comprehensive example.

```
const {checkInventory, processPayment, shipOrder} = require("./library.js");

const order = {
  items: [{"sunglasses", 1}, {"bags", 2}],
  giftcardBalance: 79.82
};

checkInventory(order)
  .then((resolvedValueArray) => {
    return processPayment(resolvedValueArray);
  })
  .then((resolvedValueArray) => {
    return shipOrder(resolvedValueArray);
  })
  .then((successMessage) => {
    console.log(successMessage);
  })
  .catch((errorMessage) => {
    console.log(errorMessage);
  });
```

- `checkInventory()` expects an `order` argument and returns a promise. If there are enough items in stock to fill the order, the promise will resolve to an array. The first element in the resolved value array will be the same `order` and the second element will be the total cost of the order as a number.
- `processPayment()` expects an array argument with the `order` as the first element and the purchase total as the second. This function returns a promise. If there is a large enough balance on the giftcard associated with the order, it will resolve to an array. The first element in the resolved value array will be the same `order` and the second element will be a tracking number.
- `shipOrder()` expects an array argument with the `order` as the first element and a tracking number as the second. It returns a promise which resolves to a string confirming the order has shipped.

```
$ node app.js
All of the items are in stock. The total cost of the order is 35.97.
Payment processed with giftcard. Generating shipping label.
The order has been shipped. The tracking number is: 652449.
```

15.9 Avoiding Common Mistakes

Promise composition allows for much more readable code than the nested callback syntax that preceded it. However, it can still be easy to make mistakes. Here, we will go over two common mistakes with promise composition.

Mistake 1: Nesting promises instead of chaining them.

```
returnsFirstPromise()
  .then((firstResolveVal) => {
    return returnsSecondValue(firstResolveVal)
      .then((secondResolveVal) => {
        console.log(secondResolveVal);
      })
  })
```

Let us break down what is happening in the above code:

- We invoke `returnsFirstPromise()` which returns a promise.
- We invoke `.then()` with a success handler.
- Inside the success handler, we invoke `returnsSecondValue()` with `firstResolveVal` which will return a new promise.
- We invoke a second `.then()` to handle the logic for the second promise settling all **inside** the first `then()`.
- Inside that second `.then()`, we have a success handler which will log the second promise's resolved value to the console.

Instead of having a clean chain of promises, we have nested the logic for one inside the logic of the other. Imagine if we were handling five or ten promises.

Mistake 2: Forgetting to `return` a promise.

```
returnsFirstPromise()
  .then((firstResolveVal) => {
    returnsSecondValue(firstResolveVal)
  })
  .then((someVal) => {
    console.log(someVal);
  })
```

Let us break down what is happening in the example:

- We invoke `returnsFirstPromise()` which returns a promise.
- We invoke `.then()` with a success handler.
- Inside the success handler, we create our second promise, but we forget to `return` it.

- We invoke a second `.then()`. It is supposed to handle the logic for the second promise, but since we did not return, this `.then()` is invoked on a promise with the same settled value as the original promise.

Since forgetting to return our promise will not throw an error, this can be a really tricky thing to debug.

15.10 Using `Promise.all()`

When done correctly, promise composition is a great way to handle situations where asynchronous operations depend on each other or execution order matters. What if we are dealing with multiple promises, but we do not care about the order? Let us think in terms of cleaning again.

For us to consider our house clean, we need our clothes to dry, our trash bins emptied, and the dishwasher to run. We need all of these tasks to complete but not in any particular order. Furthermore, since they are all getting done asynchronously, they should really all be happening at the same time.

To maximize efficiency we should use concurrency, multiple asynchronous operations happening together. With promises, we can do this with the function `Promise.all()`.

`Promise.all()` accepts an array of promises as its argument and returns a single promise. That single promise will settle in one of two ways:

- If every promise in the argument array resolves, the single promise returned from `Promise.all()` will resolve with an array containing the resolve value from each promise in the argument array.
- If any promise from the argument array rejects, the single promise returned from `Promise.all()` will immediately reject with the reason that promise rejected. This behavior is sometimes referred to as *failing fast*.

Let us look at a code example:

```
let myPromises = Promise.all([returnsPromOne, returnsPromTwo, returnsPromThree
]);

myPromises
  .then((arrayOfValues) => {
    console.log(arrayOfValues);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Let us break down what is happening:

- We declare `myPromises` assigned to invoking `Promise.all()`.
- We invoke `Promise.all()` with an array of three promises—the returned values from functions.
- We invoke `.then()` with a success handler which will print the array of resolved values if each promise resolves successfully.
- We invoke `.catch()` with a failure handler which will print the first rejection message if any promise rejects.

15.11 Review

Let us review about asynchronous JavaScript and promises.

- Promises are JavaScript objects that represent the eventual result of an asynchronous operation.
- Promises can be in one of three states: pending, resolved, or rejected.
- A promise is settled if it is either resolved or rejected.
- We construct a promise by using the `new` keyword and passing an executor function to the `Promise` constructor method.
- `setTimeout()` is a Node function which delays the execution of a callback function using the event-loop.
- We use `.then()` with a success handler callback containing the logic for what should happen if a promise resolves.
- We use `.catch()` with a failure handler callback containing the logic for what should happen if a promise rejects.
- Promise composition enables us to write complex, asynchronous code that is still readable. We do this by chaining multiple `.then()`'s and `.catch()`'s.
- To use promise composition correctly, we have to remember to `return` promises constructed within a `.then()`.
- We should chain multiple promises rather than nesting them.
- To take advantage of concurrency, we can use `Promise.all()`.

16 Async Await

16.1 Introduction

Often in web development, we need to handle asynchronous actions—actions we can wait on while moving on to other tasks. We make requests to networks, databases, or any number of similar operations. JavaScript is non-blocking: instead of stopping the execution of code while it waits, JavaScript uses an event-loop which allows it to efficiently execute other tasks while it awaits the completion of these asynchronous actions.

Originally, JavaScript used callback functions to handle asynchronous actions. The problem with callbacks is that they encourage complexly nested code which quickly becomes difficult to read, debug, and scale. With ES6, JavaScript integrated native promises which allow us to write significantly more readable code. JavaScript is continually improving, and ES8 provides a new syntax for handling our asynchronous action, `async...await`. The `async...await` syntax allows us to write asynchronous code that reads similarly to traditional synchronous, imperative programs.

The `async...await` syntax is syntactic sugar—it does not introduce new functionality into the language, but rather introduces a new syntax for using promises and generators. Both of these were already built in to the language. Despite this, `async...await` powerfully improves the readability and scalability of our code.

16.2 The `async` Keyword

The `async` keyword is used to write functions that handle asynchronous actions. We wrap our asynchronous logic inside a function prepended with the `async` keyword. Then, we invoke that function.

```
async function myFunc() {  
  // Function body here  
};  
  
myFunc();
```

We will be using `async` function declarations throughout this chapter, but we can also create `async` function expressions:

```
const myFunc = async () => {  
  // Function body here  
};  
  
myFunc();
```

`async` functions always return a promise. This means we can use traditional promise syntax, like `.then()` and `.catch` with our `async` functions. An `async` function will return in one of three ways:

- If there is nothing returned from the function, it will return a promise with a resolved value of `undefined`.
- If there is a non-promise value returned from the function, it will return a promise resolved to that value.
- If a promise is returned from the function, it will simply return that promise.

```

async function fivePromise() {
  return 5;
}

fivePromise()
  .then(resolvedValue => {
    console.log(resolvedValue);
  }) // Prints 5

```

In the example above, even though we return `5` inside the function body, what is actually returned when we invoke `fivePromise()` is a promise with a resolved value of `5`.

16.3 The await Operator

Previously, we covered the `async` keyword. By itself, it does not do much; `async` functions are almost always used with the additional keyword `await` inside the function body.

The `await` keyword can only be used inside an `async` function. `await` is an operator: it returns the resolved value of a promise. Since promises resolve in an indeterminate amount of time, `await` halts, or pauses, the execution of our `async` function until a given promise is resolved.

In most situations, we are dealing with promises that were returned from functions. Generally, these functions are through a library, and here we will be providing them. We can `await` the resolution of the promise it returns inside an `async` function. In the example below, `myPromise()` is a function that returns a promise which will resolve to the string `"I am resolved now!"`.

```

async function asyncFuncExample(){
  let resolvedValue = await myPromise();
  console.log(resolvedValue);
}

asyncFuncExample(); // Prints: I am resolved now!

```

Within our `async` function, `asyncFuncExample()`, we use `await` to halt our execution until `myPromise()` is resolved and assign its resolved value to the variable `resolvedValue`. Then we log `resolvedValue` to the console. We are able to handle the logic for a promise in a way that reads like synchronous code.

16.4 Writing async Functions

We have seen that the `await` keyword halts the execution of an `async` function until a promise is no longer pending. Do not forget the `await` keyword. It may seem obvious, but this can be a tricky mistake to catch because our function will still run—it just will not have the desired results. We are going to explore this using the following function, which returns a promise that resolves to `"Yay, I resolved!"` after a 1 second delay:

```

let myPromise = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Yay, I resolved!")
    }, 1000);
  });
}

```

Now we will write two async functions which invoke `myPromise()`:

```
async function noAwait() {
  let value = myPromise();
  console.log(value);
}

async function yesAwait() {
  let value = await myPromise();
  console.log(value);
}

noAwait(); // Prints: Promise { <pending> }
yesAwait(); // Prints: Yay, I resolved!
```

In the first `async` function, `noAwait()`, we left off the `await` keyword before `myPromise()`. In the second, `yesAwait()`, we included it. The `noAwait()` function logs `Promise { <pending> }` to the console. Without the `await` keyword, the function execution was not paused. The `console.log()` on the following line was executed before the promise had resolved.

Remember that the `await` operator returns the resolved value of a promise. When used properly in `yesAwait()`, the variable `value` was assigned the resolved value of the `myPromise()` promise, whereas in `noAwait()`, `value` was assigned the promise object itself.

16.5 Handling Dependent Promises

The true beauty of `async...await` is when we have a series of asynchronous actions which depend on one another. For example, we may make a network request based on a query to a database. In that case, we would need to wait to make the network request until we had the results from the database. With native promise syntax, we use a chain of `.then()` functions making sure to return correctly each one:

```
function nativePromiseVersion() {
  returnsFirstPromise()
  .then((firstValue) => {
    console.log(firstValue);
    return returnsSecondPromise(firstValue);
  })
  .then((secondValue) => {
    console.log(secondValue);
  });
}
```

Let us break down what is happening in the `nativePromiseVersion()` function:

- Within our function we use two functions which return promises: `returnsFirstPromise()` and `returnsSecondPromise()`.
- We invoke `returnsFirstPromise()` and ensure that the first promise resolved by using `.then()`.
- In the callback of our first `.then()`, we log the resolved value of the first promise, `firstValue`, and then return `returnsSecondPromise(firstValue)`.
- We use another `.then()` to print the second promise's resolved value to the console.

[leftmargin = *] Here is how we would write an async function to accomplish the same thing:

```
async function asyncAwaitVersion() {
  let firstValue = await returnsFirstPromise();
  console.log(firstValue);
  let secondValue = await returnsSecondPromise(firstValue);
  console.log(secondValue);
}
```

Let us break down what is happening in our `asyncAwaitVersion()` function:

- We mark our function as `async`.
- Inside our function, we create a variable `firstValue` assigned `await returnsFirstPromise()`. This means `firstValue` is assigned the resolved value of the awaited promise.
- Next, we log `firstValue` to the console.
- Then, we create a variable `secondValue` assigned to `await returnsSecondPromise(firstValue)`. Therefore, `secondValue` is assigned this promise's resolved value.
- Finally, we log `secondValue` to the console.

Though using the `async...await` syntax can save us some typing, the length reduction is not the main point. Given the two versions of the function, the `async...await` version more closely resembles synchronous code, which helps developers maintain and debug their code. The `async...await` syntax also makes it easy to store and refer to resolved values from promises further back in our chain which is a much more difficult task with native promise syntax.

16.6 Handling Errors

When `.catch()` is used with a long promise chain, there is no indication of where in the chain the error was thrown. This can make debugging challenging.

With `async...await`, we use `try...catch` statements for error handling. By using this syntax, not only are we able to handle errors in the same way we do with synchronous code, but we can also catch both synchronous and asynchronous errors. This makes for easier debugging.

```
async function usingTryCatch() {
  try {
    let resolveValue = await asyncFunction("thing that will fail");
    let secondValue = await secondAsyncFunction(resolveValue);
  } catch (err) {
    // Catches any errors in the try block
    console.log(err);
  }
}

usingTryCatch();
```

Remember, since `async` functions return promises we can still use native promise's `.catch()` with an `async` function

```
async function usingPromiseCatch() {
  let resolveValue = await asyncFunction("thing that will fail");
```

```

}

let rejectedPromise = usingPromiseCatch();
rejectedPromise.catch((rejectValue) => {
  console.log(rejectValue);
})

```

This is sometimes used in the global scope to catch final errors in complex code. Let us look at the following example.

```

const cookBeanSouffle = require("./library.js");

// Write your code below:
async function hostDinnerParty() {
  try {
    let resolvedValue = await cookBeanSouffle();
    console.log(`${resolvedValue} is served!`)
  } catch(error) {
    console.log(error);
    console.log("Ordering a pizza!");
  }
}

hostDinnerParty();

```

`cookBeanSouffle()` returns a promise that resolves or rejects randomly. When it resolves, the promise resolves with a value of “Bean Souffle” and, when it rejects, it rejects with a value of “Dinner is ruined!”.

```

$ node app.js
Fingers crossed... Putting the Bean Souffle in the oven
Bean Souffle is served!
$ node app.js
Fingers crossed... Putting the Bean Souffle in the oven
Dinner is ruined!
Ordering a pizza!

```

16.7 Handling Independent Promises

Remember that `await` halts the execution of our `async` function. This allows us to conveniently write synchronous-style code to handle dependent promises. But what if our `async` function contains multiple promises which are not dependent on the results of one another to execute?

```

async function waiting() {
  const firstValue = await firstAsyncThing();
  const secondValue = await secondAsyncThing();
  console.log(firstValue, secondValue);
}

async function concurrent() {
  const firstPromise = firstAsyncThing();
  const secondPromise = secondAsyncThing();
  console.log(await firstPromise, await secondPromise);
}

```

In the `waiting()` function, we pause our function until the first promise resolves, then we construct the second promise. Once that resolves, we print both resolved values to the console.

In our `concurrent()` function, both promises are constructed without using `await`. We then `await` each of their resolutions to print them to the console.

With our `concurrent()` function both promises' asynchronous operations can be run simultaneously. If possible, we want to get started on each asynchronous operation as soon as possible. Within our `async` functions we should still take advantage of *concurrency*, the ability to perform asynchronous actions at the same time.

Note: if we have multiple truly independent promises that we would like to execute fully in parallel, we must use individual `.then()` functions and avoid halting our execution with `await`.

16.8 Await Promise.all()

Another way to take advantage of concurrency when we have multiple promises which can be executed simultaneously is to `await` a `Promise.all()`.

We can pass an array of promises as the argument to `Promise.all()`, and it will return a single promise. This promise will resolve when all of the promises in the argument array have resolved. This promise's resolve value will be an array containing the resolved values of each promise from the argument array.

```
async function asyncPromAll() {
  const resultArray = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3(),
    asyncTask4()]);
  for (let i = 0; i < resultArray.length; i++){
    console.log(resultArray[i]);
  }
}
```

In our above example, we `await` the resolution of a `Promise.all()`. This `Promise.all()` was invoked with an argument array containing four promises (returned from required-in functions). Next, we loop through our `resultArray`, and log each item to the console. The first element in `resultArray` is the resolved value of the `asyncTask1()` promise, the second is the value of the `asyncTask2()` promise, and so on.

`Promise.all()` allows us to take advantage of asynchronicity— each of the four asynchronous tasks can process concurrently. `Promise.all()` also has the benefit of *failing fast*, meaning it will not wait for the rest of the asynchronous actions to complete once any one has rejected. As soon as the first promise in the array rejects, the promise returned from `Promise.all()` will reject with that reason. As it was when working with native promises, `Promise.all()` is a good choice if multiple asynchronous tasks are all required, but none must wait for any other before executing.

16.9 Review

Let us review what we have learned about the `async...await` syntax:

- `async...await` is syntactic sugar built on native JavaScript promises and generators.
- We declare an `async` function with the keyword `async`.

- Inside an `async` function we use the `await` operator to pause execution of our function until an asynchronous action completes and the awaited promise is no longer pending .
- `await` returns the resolved value of the awaited promise.
- We can write multiple `await` statements to produce code that reads like synchronous code.
- We use `try...catch` statements within our `async` functions for error handling.
- We should still take advantage of concurrency by writing `async` functions that allow asynchronous actions to happen in concurrently whenever possible.

17 Requests

17.1 Introduction to Requests

Have you ever wondered what happens after you click a “Submit” button on a web page? For instance, if you are submitting information, where does the information go? How is the information processed? The answer to the previous questions revolves around *HTTP* requests.

There are many types of HTTP requests. The four most commonly used types of HTTP requests are GET, POST, PUT, and DELETE. In this chapter, we will cover GET and POST requests. If you want to learn more about the different HTTP requests, we recommend the following documentation: Mozilla Developer Network: HTTP methods.

With a GET request, we are retrieving, or getting, information from some source (usually a website). For a POST request, we are posting information to a source that will process the information and send it back. In this lesson, we will explain how to make GET and POST requests by using JavaScript’s **XHR** object. We will also incorporate query strings into our requests.

We will use the Datamuse API for GET requests and the Rebrandly URL Shortener API for POST requests. To complete the exercise on POST, make sure you create a Rebrandly API Key by following the instructions in the article: Codecademy Articles: Rebrandly URL Shortener API.