Introduction to Javascript

Qitian Liao

University of California, Berkeley

Contents

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 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:

3

```
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' .

4

- 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.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects
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.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

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.
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var

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.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var

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:

```js
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: `

```js
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.

```js
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 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 `case`s. 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 `case`s are true, then the code in the `default` statement will run.

18

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 `case`s 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:
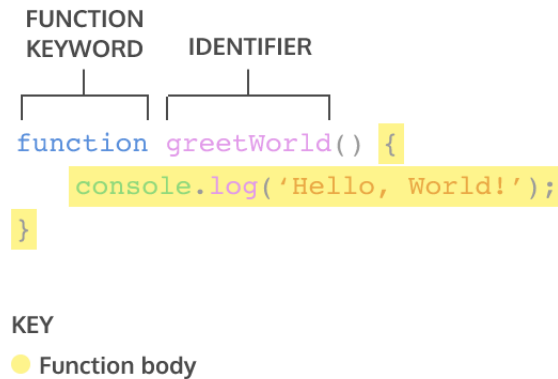
A function declaration consists of:

- 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:

```javascript
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.

```
① function getGreeting() {
③     console.log("Hello, World!");
   }

② getGreeting();

④ // Code after function call
```
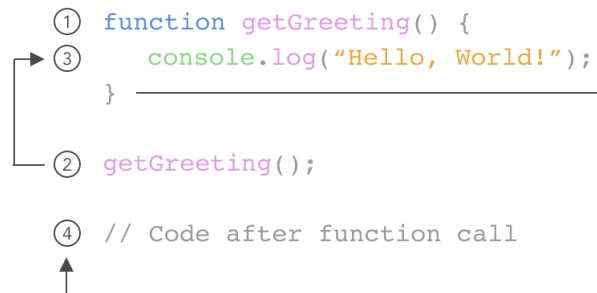
We can call the same function as many times as needed.

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.

22

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.

ARGUMENTS AS VALUES

```
calculateArea(10, 6);
```

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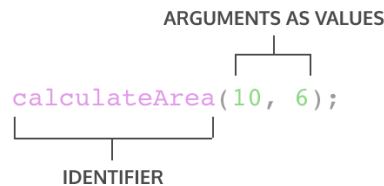
In the function call above, the number 10 is passed as the width and 6 is passed as height . Notice that the order in which arguments are passed and assigned follows the order that the parameters are declared.

```
const rectWidth = 10;
const rectHeight = 6;

calculateArea(rectWidth, rectHeight);
```

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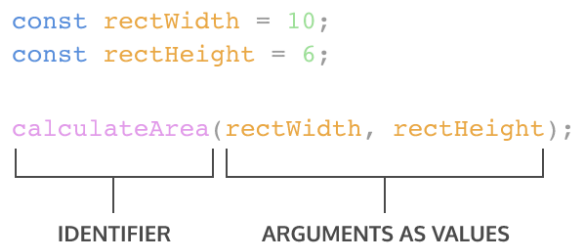
The variables rectWidth and rectHeight are initialized with the values for the height and width of a rectangle before being used in the function call.

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.

23

- 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.
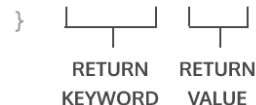
4.6   Return

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;
}       └───┬──┘ └──┬──┘

        RETURN    RETURN
        KEYWORD   VALUE
```

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:

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*:

```
            FUNCTION
            KEYWORD      IDENTIFIER

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

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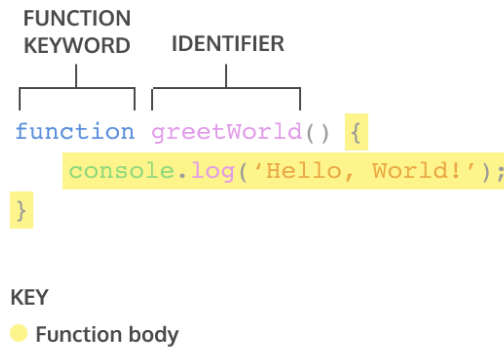
- A parameter is a named variable inside a function's block which will be assigned the value of the argument passed in when the function is invoked:

```
                        PARAMETERS

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

- To *call* a function in your code:

```
              IDENTIFIER

greetWorld();
```
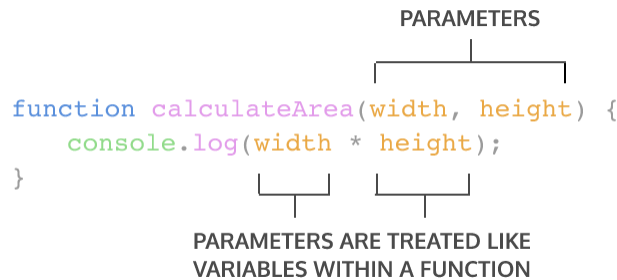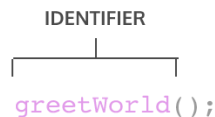
- ES6 introduces new ways of handling arbitrary parameters through *default parameters* which allow us to assign a default value to a parameter in case no argument is passed into the function.

- To return a value from a function, we use a *return statement.*

- To define a function using *function* expressions:

28

```
                                    FUNCTION
                IDENTIFIER          KEYWORD        PARAMETERS

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

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

```
                IDENTIFIER              PARAMETERS   ARROW

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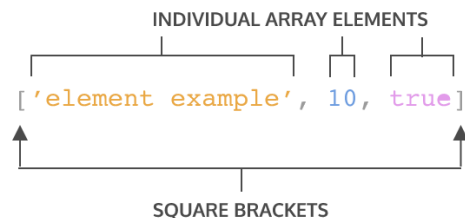
Let us take a closer look at the syntax in the array example:

- 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:
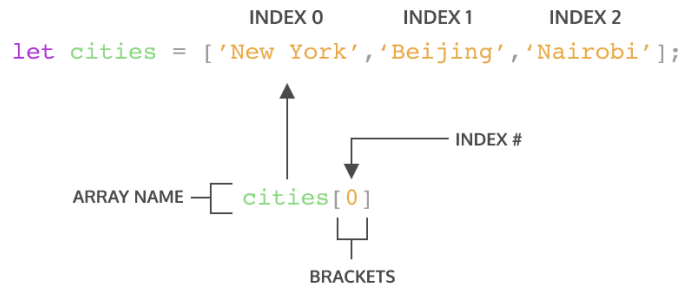
In the code snippet above:

- 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.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

`.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.

41

- 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 **i**ndex. 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.

45

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.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function
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
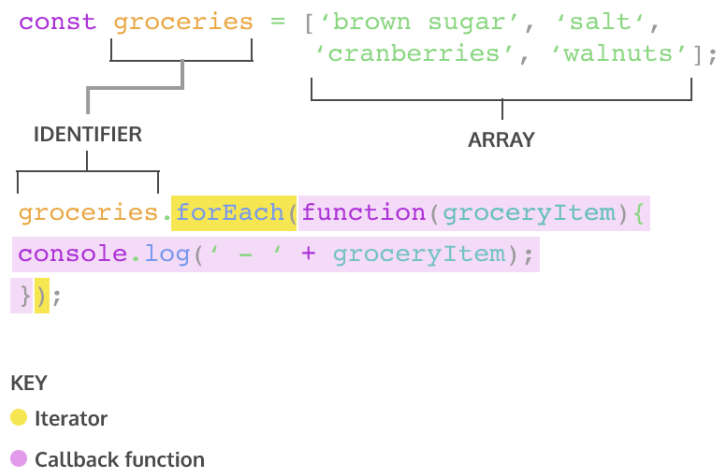
9 Iterators

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'];
```
          IDENTIFIER   ARRAY

```
groceries.forEach(function(groceryItem){
  console.log(' - ' + groceryItem);
});
```

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.

50

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;
});
```

51

- 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:

```javascript
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	17

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

11 Classes

12   Browser Compatibility and Transpilation

13   Modules

14   Promises

15   Async-Await

16   Requests