Table of Contents

Introduction

Welcome to *A Practical Introduction to JavaScript.* Thanks for reading. In order to help you get the most out of this book, it is useful to consider exactly what this book does and what it does not do.

This book aims to teach fundamental concepts of programming using JavaScript. It assumes no background with programming languages. The goal of the book is to help novice developers learn about assembling basic programs using common approaches. It uses modern approaches to writing code, but it does not assume a familiarity with the evolution of software development that has led us to this place.

This book is not a book written for experienced developers who wish to add another language to their repertoire. It does not push the boundaries of thinking about algorithms or demonstrate the flashiest solutions to problems. If you are looking for a quick read about the latest changes in JavaScript, then this probably isn't the book for you.

This book recognizes that programming, in general, and JavaScript, specifically, are both things that happen everywhere in our digital world: as more and more devices contain computing hardware, that hardware is always running software. Software is always powered by some form of programming.

This book also privileges programming for the Web, since that is the origin of JavaScript and still the primary place where JavaScript is used. Much of the discussion and examples will lean on Web-based scenarios and use cases.

How to Use This Book

This book is written with the goal of helping novice developers learn to program. In order to accomplish that goal, there are several repeating patterns that are designed to help everyone get the most from the book. Here is how you can use those patterns to get the most out of this book experience.

Read

Being a book, this involves a lot of reading. Each section contains several sub-sections. Some of those pages also contain media, code examples, and more. It's important to read through each section in detail, but it's often best to begin by reviewing the structure and skimming the content of a section as you first encounter it. Note the names of the sub-

sections in the sidebar navigation, and click through all of the pages to get an idea of what kind of material is on each page. You might feel more like looking at the pictures, or looking for new things in the code examples, than reading the text, and that's OK. Reviewing questions in the Quiz for each section of the book is also a great way to start reading the section because it will help you identify and focus on the most important details.

Exercise

Most of the sub-sections throughout this book contain interactive code exercises. These are only available in the web version of the book, but you are strongly encouraged to visit the book online and work through these exercises. These are a great way to firm up your understanding of different concepts, and you can always click the "solution" button to see how the code should go.

You are also encouraged to copy and paste the code from the code samples into a JavaScript console to see how they work. The code examples have all been written in a way to make it as easy as possible to copy and paste them into a JavaScript console and run them there. This is a great way to experiment and test concepts.

(**Note:** Due to JavaScript sandbox restrictions, you cannot normally access the full context of a JavaScript environment in the book exercises, which is another reason why using your own JS console is a great idea for practice.)

Check Yourself

Each section of the book contains a Quiz. These quizzes are designed to allow you to check your understanding and to underscore the most important details covered in each section. You can attempt each quiz as often as you would like, and you can get an explanation for each answer. As mentioned above, reviewing the quizzes before reading each section is a great way to focus on the main details of each section.

Practice Some More

Each section contains an Extra Practice page, which presents you with code examples and asks you questions about each example. You are encouraged to attempt to figure out the answers interpreting the code in your head, but each example has also been written so that it will run if copy/pasted into a JavaScript console. You are encouraged to dig into these practice problems to strengthen your understanding of core concepts and see more examples of how the code looks.

What is Programming?

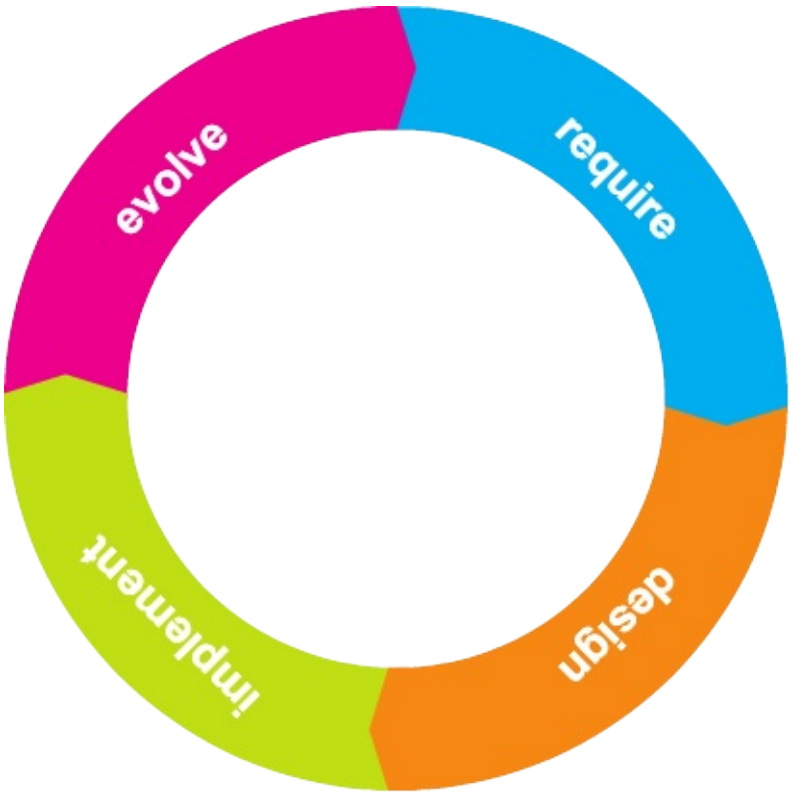
Programming is an umbrella term that we use to mean all of the activities related to creating software that runs on computer hardware. That's a very general description of what programming is, but it's tough to be much more specific and also accurate. The truth is that programming is different depending on what you're trying to do. In some cases, "programming" means writing code that is extremely difficult, uses obscure markings, and is almost incomprehensible to all but the most expert humans. In other cases, "programming" involves creating straightforward sets of instructions that use almost-human language to describe tasks that make sense even to people with no special training. Sometimes "programming" involves writing small scripts to connect discrete tools or applications in order to accomplish a set of goals. Other times "programming" means writing an entire application in a self-contained environment with well-defined boundaries.

Mostly, programming falls somewhere between these extremes.

There are several activities that tend to be lumped together as "programming":

1. We **identify a problem** that we can solve with some form of computer software. These problems may be huge or tiny, but it's always necessary to start with some goal in mind.
2. We **identify a solution** we can "implement" in code. When talking about software, implementation is the process of building the code and other media assets required to make a solution work.
3. We **write code**, create media assets, and populate datasets in order to make the software work. Each of these activities uses a different set of skills and requires precise planning.
4. We **make sure the software works** and fix the problems we find. Testing software requires a whole unique set of skills and approaches different from creating the software in the first place.
5. We **deploy** our software somewhere. In software terms, "deployment" means packaging and delivering code, media assets, and data for usage by an end user. (An "end user" is a person who wants to use the software we've created.)

This is, of course, a gross simplification of what it means to "program". In fact, each of these things could be broken out into much longer lists, detailing all the different efforts, techniques, knowledge, and learning that goes into every step of the process. We have the notion of a Software Development Lifecycle that contextualizes these steps into an ongoing process of development, revision, improvement, deployment, and further development.



In the diagram above, **require** refers to the process of deciding *what* problem will be solved. The goals of the software, the **required functionalities**, are defined. The **design** phase is the process of deciding *how* the software will be built: What technologies will be used to achieve the required functionalities, and how will everything be put together so the user can actually use the software?

Once those decisions are made, the **implement** phase is when the code is written, media assets are created, and data is populated. Eventually everything is ready and verified for deployment, which puts the software in front of the user. Once implementation is complete, the process begins to **evolve** the software in order to improve its usefulness: better features might be added, the interface may be improved, and other changes may become necessary throughout the lifespan of the software.

We see this process at play in many systems we use on a daily basis. You can probably note updates to software you use in your work or education that has changed in significant ways. The changes may have brought more features, or they may have brought more bugs, but the one thing that is consistent is that software is always changing.

The process of creating software to run on computing devices is often called **programming**, but this is a shorthand. The actual process of creating software involves many, many tasks. One of the core tasks, of course, is **writing code**. For most developers, programming refers to that specific activity of writing code.

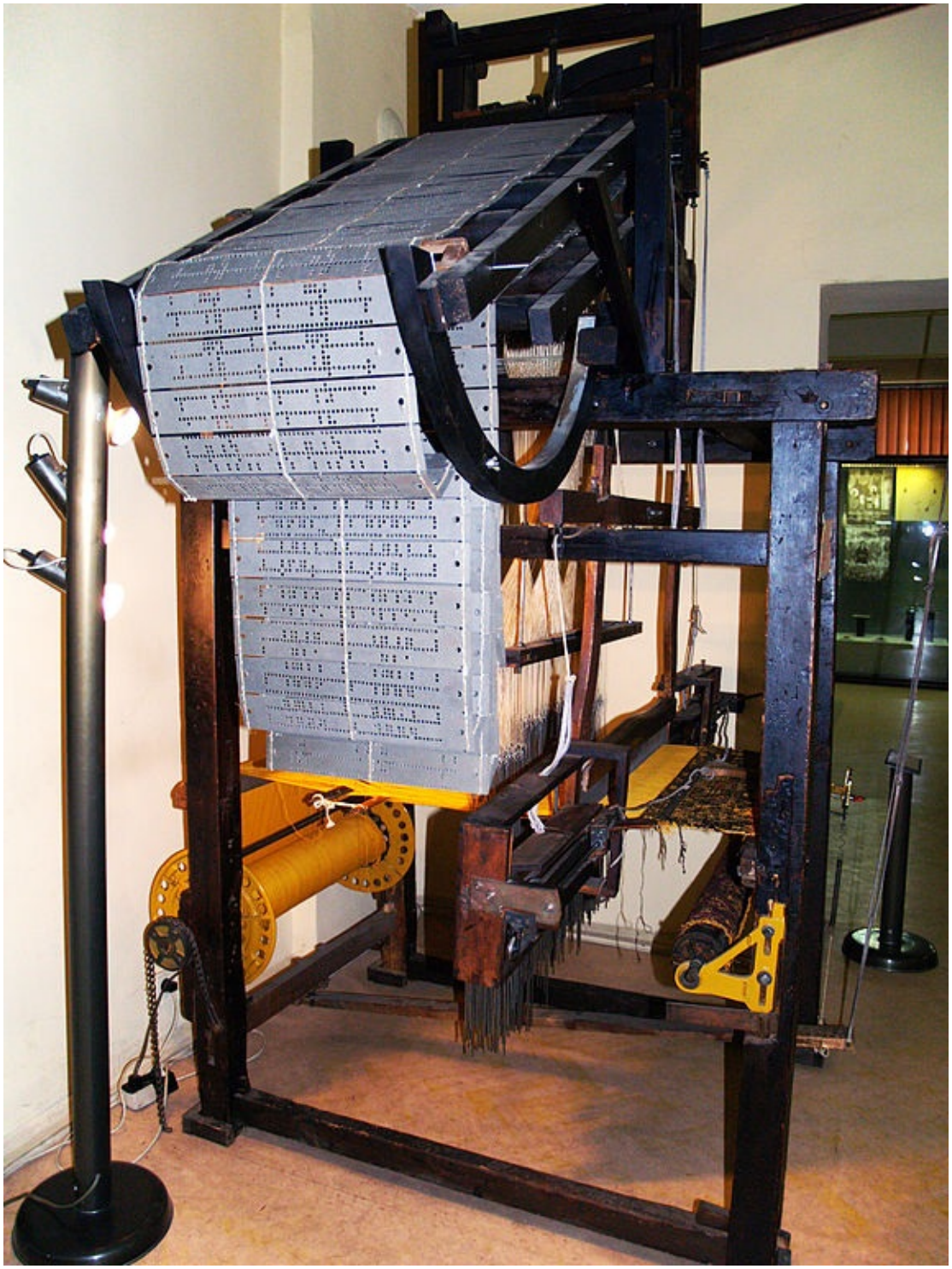
As with every other aspect of the Software Development Lifecycle, writing code is an activity that combines many smaller activities. It is the process of writing the instructions for a computer to follow. But it is also the process of describing the relationships between parts of the software, for identifying metadata required to make use of media assets, and for describing data models required to make use of datasets.

Programming, coding, hacking—whatever you call it, the instructions we write for computers form the muscle of any software application. This is the core of what it means to create content and functionality for computing devices.

History of Programming

Throughout the history of technology, we have created machines that repeat some process. Mechanical innovations have led to the creation of incredibly complex machines that could repeat defined processes to produce amazing output.

Most of us are familiar with the mechanical systems generally at play in an engine, or with the level of mechanical complexity we see in farm equipment (especially antique machines). But it's often difficult for people to realize just how long we have been working towards machines that could reproduce arbitrary instructions. One early example of a machine that could reproduce sets of instructions is the Jacquard Loom, invented by Joseph Marie Jacquard in 1804.



The Jacquard Loom could read a series of punch cards and produce complex woven patterns based on the instructions. This was a mechanized form of weaving detailed patterns, and it was an essentially "programmable" machine. Granted, it could only be programmed to produce different patterns, but it used a combination of hardware (the loom) and software (the punch cards) to do the job.

Software Development

Later in the 19th Century, Charles Babbage came up with the idea of the Difference Engine, which was a calculation device specifically designed to solve polynomial equations. Babbage continued his work by designing an Analytical Engine, which would be able to interpret more general instructions to perform mathematical calculations. Although he was never able to build an operational version of the machine, he worked with Ada Lovelace to describe different algorithms and software methods of how the machine could theoretically work. Lovelace is credited with describing the first software algorithm thanks to her work with Babbage.

In the early 20th Century, calculation machines became more and more common. By the 1940s, electronic machines known as computers were being developed, and these machines could interpret arbitrary instructions to perform different calculations.

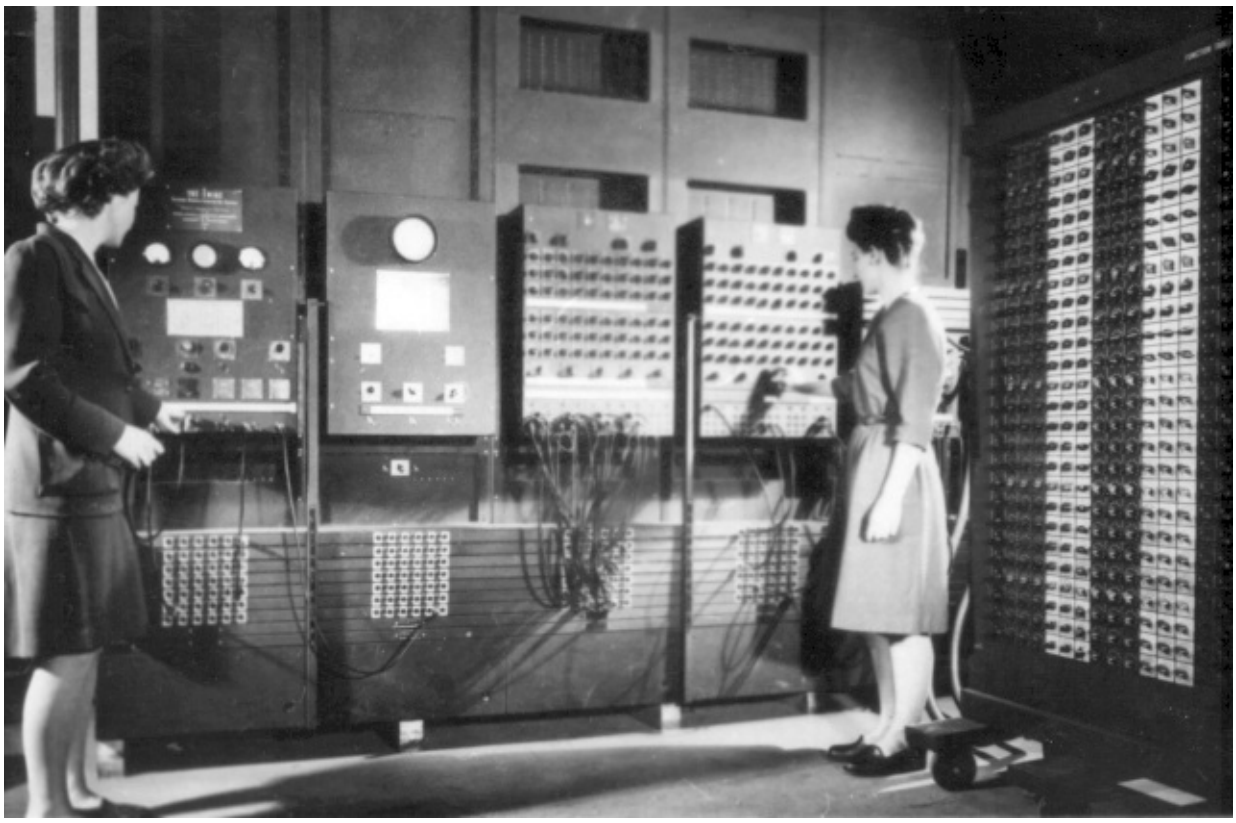


Photo: Betty Jean Jennings and Fran Bilas operating ENIAC

Computers like ENIAC, pictured above, were used to perform complex calculations at speeds that had never before been achievable. These machines allowed scientists and engineers to do more than was previously possible.

ENIAC was "programmed" by hard wiring specific circuits in the machine. Other computers of the time also used punch cards or magnetic tape to store their instructions. At this point, the people writing software were using very primitive interfaces (wires or holes in cards) to express their instructions.

Bigger Aspirations

As soon as computational power began to be created, we started thinking about how we would use these new devices. Vannevar Bush, head of the Office of Scientific Research and Development for the US Government through the 1940s, had come up with the idea of the Memex in the 1930s. In 1945 he published his influential essay "As We May Think", which describes his thoughts about the future of computing, information, and media.

> "The Encyclopædia Britannica could be reduced to the volume of a matchbox. A library of a million volumes could be compressed into one end of a desk."
> **— Vannevar Bush, "As We May Think" (1945)**

In the essay, Bush describes a desk that can access all of the information in the world. He posits the many new interfaces we will need to best make use of the data we have collected. He understands that storage will become smaller and smaller. He sees the collapse of all media into the digital form. And he even imagines small cameras we can use to record every moment of our lives in order to be able to archive and preserve those moments we wish to remember, or in order to retrace the history of our movement through the world.

Bush even imagines a method of linking documents and describing relationships between discrete pieces of information. He wants people to be able to define paths through information that can be shared with others in order to bring together unique realizations or understanding.

That last part sounds a lot like our World Wide Web, and that is not by accident. Creators of technologies like Hypertext and the Web were inspired by Bush's writing and worked to bring their own versions of his ideas to fruition.

Throughout the late 20th Century, computing hardware continued to develop at incredibly fast rates. Processing power grew exponentially, hardware size shrank considerably, and computers began to proliferate outside of dedicated server farms and research facilities.

By the end of the millennium, computers had permeated every type of business and almost every corner of the globe. Most businesses were connected to international computer networks, and tools like the World Wide Web were quickly established thanks to the rise of home computing.

The Modern Era

As computers have become more and more ubiquitous, so has software. We now live in a world where in many places it's impossible to obtain healthcare, register for college, or pay our taxes without a computer. Of course, all of these functions that computers perform are

thanks to the software that computers can run.

We have created platforms for all different kinds of software. From games and virtual reality experiences that require monumental computing power to smartphones and web-based systems that operate on minimal processing performance, software is everywhere.

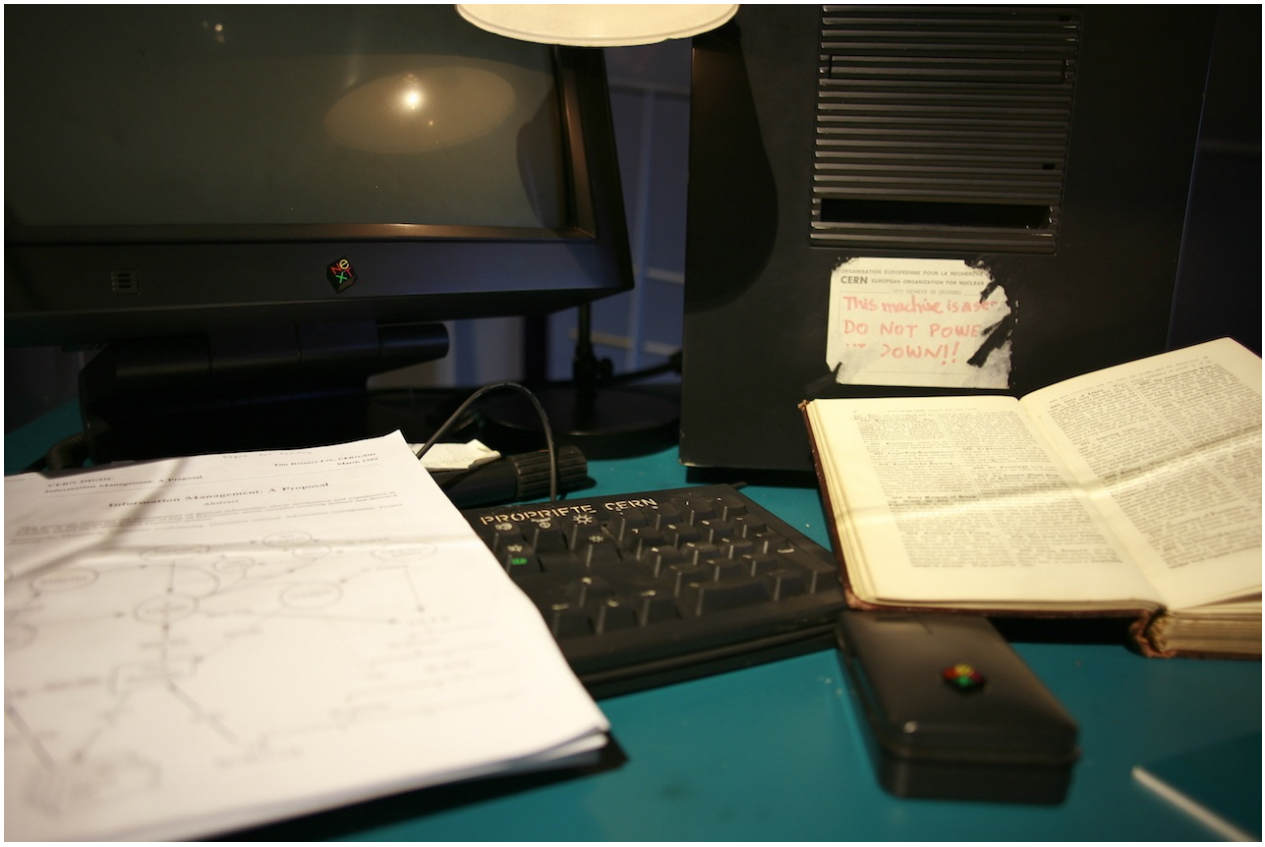


Photo: Tim Berners-Lee's Computer at CERN.

Tim Berners-Lee was a researcher at CERN when he came up with the idea of the World Wide Web. He first built an internal information management system for his colleagues, but quickly decided to make the project open source and to encourage others to use it. The notion of the World Wide Web and HyperText Transport Protocol (HTTP) caught on, and websites proliferated all over the world.

The web quickly became successful, which meant that software became intimately involved in spreading information and conducting business for the first time. The web is a platform that can deliver all sorts of content, but it also provides a bi-directional communication like no other ubiquitous system we've ever seen. The web brought the global network into our homes and changed our lives significantly.

The impact of software on our day-to-day lives was made even more profound with the advent of smartphones in the mid-2000s. Computing power had advanced sufficiently to give everyone a pocket-size device that rivaled the supercomputers of a decade prior. For the first time powerful software could move around with us and leverage the freedom and expanse of the entire world to bring us new and exciting capabilities.

Programming Languages

Programming languages have developed alongside computing hardware in order to facilitate different approaches to writing software. What is a programming language, exactly? A programming language is a "notation for writing programs, which are specifications of a computation or algorithm" (Wikipedia). It is the *way* we tell the computing hardware what we want it to do.

Sometimes programming languages were constrained by the capabilities of their target hardware. Other times, these languages were tailored to specific types of calculations or algorithms in order to more easily accomplish their users' goals. This book deals primarily with JavaScript, but the following list will help you gain an understanding of some of the many types of programming languages. Please note that this list is in no way exhaustive.

Assembly

The very first programming languages were Assembly Languages. These are "low level" languages, meaning that the developer writes instructions that are directly executed by the computing hardware. This often means that developers typed cryptic sequences of characters or otherwise indicated binary information that often corresponded directly with how the hardware worked.

- Assembly Language

Compiled Languages

Writing Assembly code is so arduous that developers almost instantly wished for a more expressive way to write their software. This quickly led to the development of "compiled" programming languages. The developer would write code in a "higher level" language that corresponded less to the machine instructions and more to the way humans think and speak. Those instructions were then interpreted by a compiler, which would produce a "binary" file that could be executed by the computing hardware. These languages remain popular, and new compiled languages are regularly created, because they allow for a good balance of developer experience and machine efficiency.

- Fortran
- COBOL
- C++

Scripted Languages

Although compiled languages still thrive in the modern software ecosystem, there has been a distinct rise in the past few decades of "scripted" programming languages. These languages are "interpreted" or "compiled" at runtime, skipping the step of compilation and the creation of a binary file. This is handy for developers who want to see their code execute as quickly as possible during development, and thanks to the strength of modern computing hardware these languages are able to perform well for their tasks. Many websites are built using scripted languages (Facebook, Instagram, etc.), and many applications use embedded scripted language interpreters to allow users to enhance their work (such as Flash, or World of Warcraft). Scripted languages are some of the most popular in the world right now.

- Python
- PHP
- JavaScript
- Ruby

Typed Languages

Data management is such a significant part of programming that languages are often grouped based on whether or not they require developers to "declare" the datatype of the variables they use. This means that if you declare a variable `foo` is an `integer` then it could be equal to `1`, `2`, or `3`, but it cannot be equal to `"bar"`. Languages that require developers to say that a variable is going to be a specific type of data are called "typed" and are valued for their ability to help prevent the errors that are caught by "type enforcement."

- C++
- Java
- Go
- Swift

Dynamic Languages

Dynamic languages allow developers to alter the type of a variable in mid-usage, and they do not require the developer to declare the datatype. This means that if a variable `foo` is defined, then `foo` could be equal to `1`, `2`, or `"bar"` at any point in the execution of the code. This is often seen as beneficial because it allows developers to be more free-form with how they handle data and does not require the same rigid foresight as a typed language.

- Python

- PHP
- JavaScript
- Ruby

Functional Languages, Object Oriented Languages, etc.

There are many ways to group programming languages, and the more we learn about programming languages then the easier it is to appreciate the differences between languages. Some differences, such as "functional programming" or "object oriented programming" (OOP) are difficult to grasp unless you know something about how languages work. For the sake of this historical introduction I'm going to skip over all of these very interesting and super useful languages and what makes them unique. Nonetheless, you may be curious to look into them more deeply.

- Scala (functional)
- Clojure (functional)
- Haskell (strongly typed)
- R (statistical programming)
- Lisp (lots of parentheses)

Core Concepts in Programming

After the advent of computers, programming languages grew and proliferated. Programming languages have gone through an evolution of type and purpose, which has often coincided with the capabilities of hardware and the needs of the people using the computing devices. However, there is also an evolutionary quality to the history of programming languages. As concepts are developed they are worked through, enhanced, revised, discarded, or kept. Since there are many goals and purposes for writing software, there are many ways of doing it.

Often, when we talk about "writing code" or "programming" we mean the act of writing instructions for the computer to follow. These instructions define different concepts and actions that the computer hardware can use to make different things happen: a pixel lights up on the screen, or a value gets written to disk. Writing the correct instructions to create our desired outcome is the challenge of coding (please note: I use the words "coding" and "programming" interchangeably since they are generally used that way in the field).

Over time, some concepts have permeated the world of programming so they exist in almost all languages. These are core concepts that make it possible to achieve different logical expressions. Using these core concepts allows us to create sets of instructions for computing hardware and to define *algorithms* that perform specific tasks. We will review some of these core concepts below, and throughout this book we will dig more deeply into how many of these work within the JavaScript programming language.

Algorithms and Instructions

An **algorithm** is a "self-contained sequence of actions to be performed" (Wikipedia). Algorithms can be designed to process information, calculate a specific value, modify a digital file, or perform a sequence of actions using computer-controlled machines. This might include ingesting and annotating data received from a weather sensor, applying a filter to a photo so that it looks like it was taken by an old-time camera, calculating interest on a bank account, or controlling a mechanical arm on a factory assembly line.

Algorithms are designed to accomplish a specific action, and a software application will often make use of several algorithms in order to deliver its full set of features. If you imagine an app like Facebook, you might recognize ways in which different algorithms are used: An algorithm governs how user authentication is handled. An algorithm modifies files that a user uploads with the app. An algorithm determines which content to show you in your news feed.

Algorithms are connected with **instructions**. Much of any software application's code is concerned with connecting discrete actions (often powered by specific *algorithms*) using sets of instructions. These instructions are what determines that, for example, when a user comes to the site they are asked to authenticate their identity. The instructions would lead the user to a page or screen where they are asked for the information they need, and then that information would be handed off to the code that comprises the *authentication algorithm* in order to verify the user's identity. That algorithm would return a result that would then allow the app to know the user is authenticated, and other instructions would then tell the app how to behave.

It is useful to understand that as we write code we are doing different things at different times. Sometimes we are concerned with creating complex logic that performs specific tasks in an efficient manner. This practice is, often, the creation of algorithms and tends to lead to the creation of software components that can be used in many different situations.

At other times, we are writing instructions that are designed to move the user through the application, present different features to the user, and move data between all of the components of the application. This kind of coding is similar to writing algorithms, but it has different constraints and criteria for success.

In order to create either algorithms or instructions, software developers make use of a common set of concepts that exist across most programming languages. These concepts, which are built into the programming languages themselves, allow for data to be manipulated, logic to be described and enforced, and code to be interpreted by the computing hardware.

Syntax

In all programming languages, there is a definition of **syntax**. The syntax of a language determines the specific way lines of code must be written. Many languages require a semicolon (`;`) to end each line. Many languages use curly braces (`{` and `}`) to denote the contents of a function or loop.

Each language has a different syntax, but they all enforce their syntax, often ruthlessly. In most programming languages, a "syntax error" will break the entire application. This is often a source of frustration for new developers, but over time we get better at parsing our own code and seeing that we have missed a comma (or included an extra comma). It often seems frustrating from a human perspective to have a misplaced comma or semicolon break everything in our software (after all, in real life we can usually read past bad punctuation with little impact on our understanding). But our computing hardware has no notion of nuance or contextual interpretation (unless we've programmed it that way). It's impossible for our

programming language interpreter to "figure out" what we meant, and, in the end, we don't want our interpreter to "guess" about too much. We want reliable execution of our code that is the same every time. That requires us to be as attentive to detail as the machine.

In addition to line endings and markers denoting sections of code, **reserved words** are an important concept in a programming language's syntax. Words such as `for` or `while` are often reserved words that note specific commands within the programming language. These words are usually off-limits for making variable or function names because they are reserved for use by the programming language itself. Languages typically try to minimize the number of words they reserve in order to allow developers as much freedom of expression as possible.

Using an example from JavaScript, the reserved word `var` is used to scope a variable (don't worry too much about what that means, we will cover it later). But you could not name a variable `var` because it would conflict with the reserved word. To define a variable named `foo` that equals the number `6`, we would write:

```
var foo = 6;
```

The line above specifies the creation of that variable, `foo`.

Variables and Data Types

Within the code of a software application, different segments of code are named. This includes the names of files, functions, and **variables**. Variables are the labels we give to data values within the software. We use variables to reference data values at different points in the code.

For example, within an application there may be a variable called `user` that provides a reference to information within the software about the current user. The information contained in `user` will change depending on which person logs into the system at any given time, but within the software code that person's information will always be referenced using the variable `user`.

Each variable references a stored value that is of a specific **data type**. Data types allow the programming language to handle different kinds of values in useful ways. This is mostly designed to ease the task of writing code and increase the reliability of the resulting software. There are many data types, but below are a few of the most common.

Text

The **string** data type references what we typically think of as a *word* or *text*. Strings are normally denoted with quotation marks, such as:

```
var foo = "some text";
```

Strings can contain a huge variety of characters (including numbers), but if they are defined as a string then they are given a set of features that are useful for working with text. In various languages, this includes functions to break strings apart (such as separating all the words following spaces), to combine strings together, or to change/replace various characters within the string.

Numbers

There are two major data types used for working with numbers: **Integers** are whole numbers such as `1`, `42`, or `1337`. These are often used as counters or index numbers. This is what it looks like in JavaScript to define an integer:

```
let myInteger = 4;
```

Floating point numbers are numbers with decimal points. These are often required when doing calculations that might result in decimal remainders, when dealing with measurements, or in other situations where the value of a number may need to be more precise than a whole number. A decimal variable definition in JavaScript would look like this:

```
let myDecimal = 3.14;
```

It's very useful to note that, as of recent versions of the ECMAScript standard, JavaScript has only one type for Numbers.

Boolean

Within software development, we often talk about the "binary" nature of computer hardware. All computer hardware is, ultimately, comprised of very tiny switches. Billions of switches are etched onto a chip, and all of our programming logic is reduced to an "on" or "off" signal for each switch. Turning these switches on and off allows different actions to be executed by the hardware.

In Mathematics, Boolean Algebra is the practice of using logical deduction to determine *true* and *false* values within a system. We often do this kind of work in our code, and in order to support these calculations, we have the **Boolean** data type.

For example, in a software application like Facebook, it's crucial to know whether the user is logged in or not. This might involve making several checks, and with each check a boolean value might be recorded. In JavaScript, that could look something like this:

```
user.loggedIn = true;
```

The two values for a boolean data type within JavaScript are `true` and `false`. We use boolean variables to control logical flow throughout applications in most programming languages.

Data Structures

In addition to variables, software developers make use of **data structures**. Data structures are ways of containing data that make it more convenient for the developer to work with that data. Data structures often make it more efficient for the computing hardware to handle the data being processed, so using the correct data structure for the task is a key factor in writing performant applications.

Within JavaScript there are two data structures that are similar to structures in many other programming languages: objects and arrays.

Arrays

Arrays are sometimes known as "lists" or "indexed arrays" in other programming languages. An Array contains an *indexed* list of items. The items can be of any data type, and they can be referenced using their numerical index. For example, here is an Array assignment in JavaScript:

```
let myArray = ['hello', 'world', 42];
```

The example Array above has three items and they can be accessed using their numerical index. The value of `myArray[2]` is `42`. The value of `myArray[0]` is `'hello'`.

Arrays are very useful for storing lists of values.

Objects

Objects are foundational in JavaScript, and they are very common in other languages. In other programming languages they are sometimes known as "hashes" or "dictionaries" due to the way they work.

An object is defined with specific *attributes*. These attributes are accessed by name. The JavaScript Object Notation (JSON) is used across many platforms and languages because of its clean, simple syntax. Here is what a JavaScript Object definition looks like:

```
let myObject = {
    name: 'Grace Hopper',
    rank: 'Rear Admiral',
    favoriteTech: ['COBOL', 'compilers']
};
```

In order to access the values within the Object, we would reference the attributes by name: `myObject.name` equals `'Grace Hopper'`. `myObject.rank` equals `'Rear Admiral'`. It's interesting to note that the value of `myObject.favoriteTech` is an array. Object attributes may contain Arrays. Likewise, Arrays may contain items that are Objects.

Controlling Logical Flow

In order to define the instructions and algorithms of an application, it's important to control the way code executes--the "flow" of the program. We organize code into different sections so that we can make good use of the different pieces of logic we have created. As a user interacts with our software, it's important to move through the code in ways to support our desired outcomes.

For example, if the user wishes to login, we must move the user to the login page/screen and then execute the login process for the user: take in their login credentials, process that, and provide a response based on our analysis of the information they have supplied.

In order to make this happen we primarily use three different kinds of structures: functions, conditionals, and loops.

Functions

Most programming languages allow you to define **functions** (sometimes called "subroutines"). These are contained lines of code that are typically dedicated to just one purpose. Functions are an abstraction that is typically used to encapsulate some specific logic that can be called from multiple places within the software code.

For example, let's assume we are writing an application in JavaScript that requires us to take mixed-case strings and make them entirely uppercase. We can define a function like this:

```
function makeUppercase(text){
    return text.toUppercase();
}
```

The function defined above is called `makeUppercase` and it expects to receive a value that it labels `text`. The function then returns the results of the `text.toUppercase()` function, which is a built-in function for strings in JavaScript (and part of the value of having data types).

To use this function we might write:

```
let mixedText = "Hello, world!";
let uppercaseText = makeUppercase(mixedText);
console.log(uppercaseText); // Would print "HELLO, WORLD!" in the console.
```

This is a mundane example, but functions are used extensively in most types of programming languages and software projects.

Conditionals

As we read above, Boolean data types are useful for managing logical flow. We use conditionals to evaluate Boolean and other variables and then execute code accordingly. In most languages conditionals are structured as `IF`, `THEN`, `ELSE` statements. These statements cause the programming language interpreter to evaluate our statements and then determine which statement is `true`. The statement that is `true` is the one that gets executed.

Here is an example (again, in JavaScript) from an imaginary web application:

```
if (!user.logged_in){
    // User is not yet logged in
    // Redirect to login page
    app.location.go('/login');
} else if (user.role === 'staff') {
    // User is logged in and their role is staff
    // Redirect to admin homepage
    app.location.go('/admin');
} else {
    // User is logged in, but is not staff
    // Redirect to user homepage
    app.location.go('/home')
}
```

Conditional statements are able to be strung together. In the example above, we can see that the conditional check would progress from top to bottom, and the first condition that results in a `true` value would be executed. The first check determines whether the user is logged in, and if they are **not** logged in then it will redirect them to a login page. If the user **is**

logged in, then it would check the user's role. If the role is `'staff'`, then the user is redirected to the admin dashboard. However, if the user is not a staff member, then they are redirected to their user homepage.

This is just an imaginary example, but it's close to what we actually do in web apps to manage where users should go when they arrive on the site. Much of our efforts as software developers are put towards setting up the ability to write these conditionals so that we can determine what our app should do next. Conditionals are used to determine almost every decision our applications make.

Loops

We have reviewed the data structures Object and Array, which are used to store and relate complex information. When working with these structures, or just when executing functions in general, it's often useful to be able to **loop** through tasks. For example, if I have a large Array containing information about every book about a topic, then I might want to loop through those books and display information about each one for the user.

In most languages we have two kinds of loops: **for** and **while**. The `for` loop is used to cycle through all of the data stored in a data structure (usually an Array). They are often used to output the results of some data.

Example `for` loop:

```
for (i=0; i<=myArray.length; i++){
    // Do something with data
    console.log(myArray[i]);
}
```

The `for` loop above goes through the Array `myArray` and outputs the value of each item to the console. The syntax for this loop varies depending on language, and many languages have adopted an easier `foreach` loop syntax that essentially performs the same function.

The `while` loop executes as long as some condition is true. An example would look like:

```
var counter = 0;
while(counter < 100){
    // this code will execute until the value of counter is 100
    counter = counter + 1;
}
```

In the example above, the `while` loop continues until the counter reaches a certain number. In other cases, a `while` loop may be watching for another value to change within the system, such as when a process has finished and data can be updated. In general, `while`

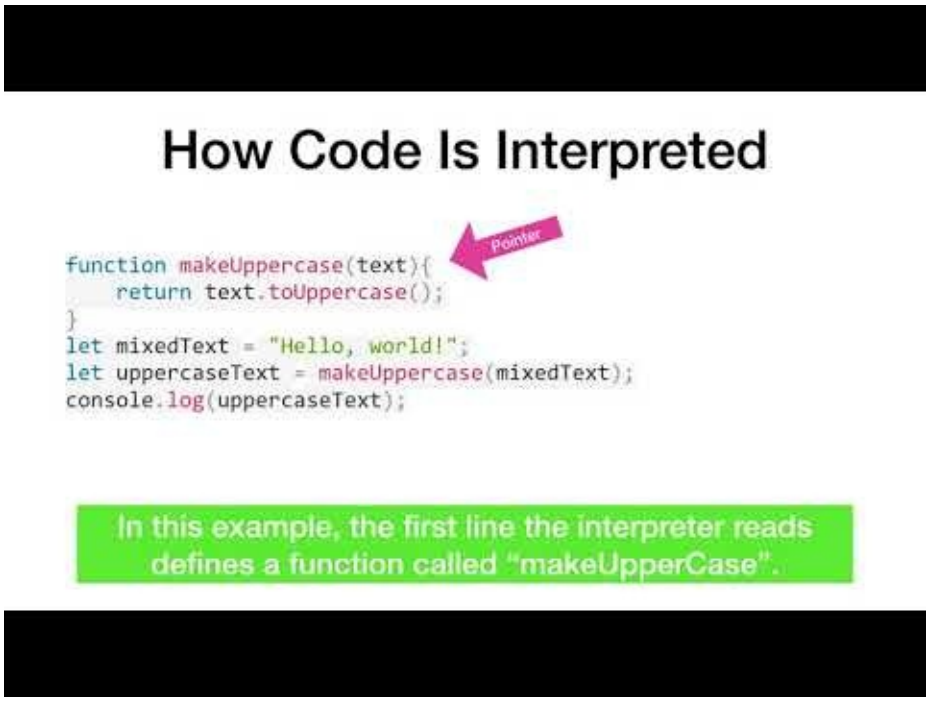
loops are used less often and are seen as being a little more risky because it's possible to create situations where the `while` loop might run forever (which breaks our applications).

How Code is Interpreted

All code is interpreted by some kind of system. That system reads through the code following specific rules and interpreting each line. Some common coding structures define code that does not run until it needs to be invoked. Other lines of code are executed at the moment they are read.

A common concept in a code interpreter is a "pointer". A pointer is the indication of where in the code the interpreter is reading at any given moment. The pointer indicates what line and character the interpreter is reading, and when the interpreter encounters an error it can report the position of the pointer to help us hone in on where the error might have happened.

Visualizing the pointer moving through code can be a good way to better understand what's happening when you "run" a program. Here is a video illustration that demonstrates how the pointer moves through some example code.



Video link

Conclusion

This is just a brief, general overview of some core concepts that tend to exist across programming languages. The rest of this book will go much deeper into the ways these concepts are expressed in JavaScript.

The Code Landscape

Computing hardware has become ubiquitous. That means that software has also become a part of nearly every moment of our lives. We live in a world that is mediated by computerized devices, whether those are automatic doors, the keys to our vehicles, or the app that tells us what time the bus will arrive. There is virtually no aspect of modern life that has not been affected by the rise of digital technology and software development.

Different programming languages have evolved alongside different computer hardware platforms and different user needs. There is a huge ecosystem of platforms and supporting technologies that facilitate all of the great functions we appreciate in our software. It can be daunting to decide what programming languages to learn about, or even to begin understanding the general landscape of how languages are used in the real world.

In order to provide a partial description of this broad landscape of software and coding, a list of some of the industries where software is critical and the languages used for those purposes.

Web Development

The web has changed the world. The variety of websites is impossible to summarize, and it's probably not necessary to underscore just how important websites are in the day-to-day business of the 21st Century. There is a wide variety of software running online, and many different programming languages are used actively. Here is a partial list of the most popular programming languages used online. (Please note that HTML and CSS are not programming languages, so they are not included here.)

- JavaScript
- Python
- PHP
- Ruby
- Java
- C#
- Go
- Scala
- Clojure

Videogames

Games are a huge part of the software ecosystem, often existing before much more serious software on new platforms. Videogames have become a part of our global cultural experience, and they rely on the development of computing hardware to continue pushing the bar for graphic and interactive innovations. Many languages are used in the videogames industry, but some of them are:

- C++
- C#
- Java
- Lua
- Python
- JavaScript

Smartphones

Since before the advent of smartphones we have been working towards pocket-sized computing devices that can help us navigate the world more easily and enjoyably. The current reigning champs of smartphone app development are Apple's iOS, Google's Android, and Microsoft's Windows Mobile. These platforms all support multiple approaches to software development, but they provide privileged support for certain programming languages and development platforms.

- Swift
- Java
- Objective-C
- C#

Media Production

Within the world of media production there are many sub-disciplines: film, video, television, animation, special effects, etc. Each of these crafts have become more and more digitized, and they are each reliant on software tools for modern production. Studios and producers use many different languages and platforms, and they are constantly making new tools to push their art.

- C++
- Python

Enterprise Business

Big, big businesses—like banks, insurance companies, large retailers, etc.—have used computers for years to manage backroom processes for manufacturing, fulfillment, communications, and more. They invest large amounts of money in big systems that become integrated tightly into the business functions and often take ages to change or evolve. This means that a wide variety of languages will be used by these large organizations as their new initiatives will leverage the latest, best tech while legacy systems might require antique knowledge.

- Python
- PHP
- Ruby
- COBOL
- Fortran

Scientific Research

Scientists and researchers collect and analyze data using software tools. They rely on computerized hardware to do the collection of data, which provides additional opportunities for software to play a role in their work. These users tend to adopt technologies for longer terms, but they are also constantly looking for new solutions to problems. There is a huge range of technologies used across all of the scientific disciplines, but these are a few of the languages commonly used to run scientific equipment and to report scientific results.

- Python
- R
- Java
- C++

Hardware

There is a huge rise in the creation of additional hardware that uses computerized interfaces. The "Internet of Things" is a popular subject, and we have seen network connected versions of everything from thermostats to refrigerators. In automobiles we see dashboards that are powered by network-connected computers. Children's toys possess intense amounts of sensors and software-powered behaviors. Autopilot software for drones has become so reliable that we can now have flying robots film our outdoor adventures. All of this innovation and development makes use of a broad range of languages and technologies, but these are some common languages used in modern hardware:

- C++

- Java
- Python
- JavaScript

Quiz: What is Programming?

Please enjoy this self-check quiz to help you identify key concepts, points, and techniques discussed in this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

What is JavaScript?

JavaScript is a **programming language**. It is the only programming language that is interpreted and executed directly in the web browser. JavaScript is supported in all web browsers. It is part of the three primary technologies that make World Wide Web content available: HTML, CSS, and JavaScript (JS).

JavaScript is defined by a standards organization called Ecma International, and that organization stewards the definition of the language and the efforts to keep it updated. In Ecma International standards, JavaScript is called ECMAScript, but in day-to-day usage the term JavaScript remains more popular. For our purposes in this book, the terms JavaScript and ECMAScript are synonymous.

JavaScript is a dynamic and scripted language. The dynamic aspect of the language means that variable datatypes are *inferred* by the JavaScript interpreter. The scripted aspect of the language refers to the fact that *no binary files are created and stored* (or "compiled" from the source code) in order to run JavaScript programs. There is never an "executable" created for a JavaScript application*.

JavaScript is also the official programming language supported in web browsers according to W3C specifications for how the World Wide Web should operate. There are no other programming languages that can be interpreted, by default, by all of the different web browsers. JavaScript is used across the web to bring websites to life and provide complex, engaging interactions to users.

Even as JavaScript remains important in web browsers, it has become ubiquitous everywhere. Through projects like node.js, JavaScript has been able to run anywhere. JavaScript powers website servers, tiny robots, flying drones, and everything in-between. JavaScript developers have proliferated and brought their tools to every corner of software development.

This ubiquity has made JavaScript a valuable tool for software development in general. The JavaScript community continues to grow by leaps and bounds, and JavaScript shows no signs of waning popularity.

* In modern JavaScript all sorts of capabilities have been developed beyond the conventional usage of the language. Although JavaScript will always be considered a scripting language, it's possible to create applications (such as Github's Atom editor) that run *like* regular, compiled applications.

The History of JavaScript

JavaScript was created in 1995 by Brendan Eich, who was working at the time for Netscape on Netscape Navigator 2 (Wikipedia). Netscape was founded, in part, by Marc Andreessen who had previously helped create the Mosaic web browser at the University of Illinois National Center for Super Computing Applications (NCSA). The graphical approach taken by Mosaic and, subsequently, by Netscape Navigator was popular and the browser was gaining popularity alongside the web in general.

Andreessen believed the web would need some kind of glue language, and Eich was given the task to create a scripting language very quickly to add to the Netscape project to facilitate the "glue" functionality. Eich and the team believed that they should capitalize on the growing popularity of Java as a web programming language, so they adopted some syntactical similarities of that language. More importantly, they called the language JavaScript to create the false assumption that Java and JavaScript had some more significant shared heritage. In fact, there is very little like Java in JavaScript, and the languages are not at all related other than this historical anecdote.

JavaScript was generally successful in the browser, but in the early days of the web there were many competing technologies. Microsoft came up with JScript and, eventually, Dynamic HTML, both of which sought to do things like JavaScript could do. The competing landscape prevented client-side JavaScript from becoming a major factor in website design for quite awhile.

Standardization

In 1996, Netscape submitted JavaScript to Ecma International in order to create a standard for the language. The result was, after a turbulent debate, ECMAScript, which persists today. Once JavaScript was more standardized and implemented in compatible ways across all browsers, the language began to be more important to the creation of modern websites and web-based applications.

In 2005, Jesse James Garret (Wikipedia) published a white paper in which he coined the term "AJAX" (which stands for Asynchronous JavaScript and XML). AJAX described the way in which companies like Google were building applications that felt more like desktop apps on the web. Tools like Gmail and Google Maps were able to update information without requiring a browser refresh, making them more responsive to the user and efficient to use.

New design paradigms such as AJAX helped push JavaScript as a foundational component of modern websites, and the language has continued to be evolved and enhanced via the ECMAScript standard and the input of many major players in the tech world.

JavaScript on the Web

On the modern web, JavaScript is a foundational element of software development. JavaScript runs in the web browser, making it a critical client-side scripting language. But implementations like Node.js allow JavaScript to run on a server allowing for server-side applications to be written in JavaScript, too.

Websites might use backends written in Python or Ruby or Java, but every website uses JavaScript. It's the most ubiquitous language online, and it is used to do virtually everything.

Application Frameworks

Building JavaScript applications like Gmail or Facebook requires a cohesive framework that will allow developers to coordinate different data, views, and functions. To accomplish this, many different JavaScript frameworks have been created.

- React.js
- Angular
- Vue.js
- Ember.js
- Backbone

To help get an idea of the differences between frameworks, check out the TodoMVC website, which archives versions of the same Task List application written in different technologies.

Server-Side JavaScript

With the advent of Node.js, server-side JavaScript has become more and more popular. These frameworks help you build server-side components of websites (and some of them crossover to helping with the client-side implementation, too).

- Meteor.js
- Express.js
- Socket.io

JavaScript is also popular for creating some of the most hyped non-relational databases:

- MongoDB
- CouchDB

Game and Animation

JavaScript has replaced a lot of proprietary technologies that were used for doing game and animation programming. In the move to eliminate plugins (which are painful for both developers and users) and bring the web to a more standardized place, JavaScript has filled this niche. Here are some interesting JavaScript game and animation frameworks that are popular online today.

- D3.js
- SnapSVG
- Pixi.js
- Phaser

General Purpose

There's too much to list all of the different popular JavaScript modules and libraries used in web development today. To browse more of the popular tools available to modern web devs, take a look at JavaScripting.com. That site indexes hundreds of JavaScript modules and libraries, all sorted and searchable for your browsing convenience. The biggest problem with the JavaScript ecosystem today is that there are so many players.

JavaScript in the Wild

JavaScript has burst past the boundaries of web browsers and even web servers. It lives on operating systems and in devices all over the world. It is used for all sorts of tasks beyond just creating and serving web pages. As web technology becomes embedded throughout our world, and screens are increasingly powered by open web standards, JavaScript will always play a role in the software we create.

Non-Browser-Based JavaScript Engines

In order to make JavaScript run everywhere, it's crucial to have a JavaScript engine that can run on your hardware. Several projects have been created to do just that: Allow JavaScript programs to execute like regular applications. These are some of the most popular non-browser-based JavaScript engines in use today.

- Node.js
- Rhino
- MynaJS

Scripting in Apps

Many applications allow users to create scripts using JavaScript to automate features or tasks. The following applications all allow for this sort of scripting:

- Adobe Creative Suite
- Chrome Web Browser Extensions
- RPG Maker
- Google Apps

Application Development

JavaScript is increasingly used in application development for outside the browser. Apache's Cordova project is a JavaScript-based framework for allowing developers to create native mobile applications using web technologies. The Unity game engine allows JavaScript to be used as a language for creating platform native games, too.

Hardware Hacking

The popularity of JavaScript-based platforms like Node.js have even led to JavaScript being used as the primary programming language for self-contained hardware projects. Devices like the Tessel run Node.js as a platform and allow JavaScript developers to create software that can control a wide array of devices and sensors.

Quiz: What is JavaScript?

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Basic Syntax

JavaScript uses a specific syntax to write code. This allows the JavaScript interpreters in web browsers to properly parse and execute that code. Like with any programming language, JavaScript interpreters can be thrown off with even the slightest mistake in syntax. A misplaced comma (`,`) or missing curly brace (`}`) can make your entire program throw an error and fail to execute.

These sorts of issues are often the most frustrating for new developers to deal with because they are difficult to find when they happen, and they represent a failure at the "easy" part of writing code (getting the punctuation correct). It often feels like the more difficult parts of writing code **should** be the large concepts and design techniques we struggle to grasp when we first learn them. But, in fact, much of the experience that comes with coding is about being able to scan through a set of lines and quickly pick out misplaced brackets, curly braces, parentheses, and commas.

There are some ways to help grasp syntax more easily. It's always good to use a dedicated editor that includes a "linter". A linter is a tool that helps us see issues in the code we've written, especially from a stylistic and syntactical perspective. Linters are like grammar checkers for programming languages.

Throughout this section, we will review the basics of how JavaScript gets into the web browser via HTML tags and how the fundamental building blocks of JavaScript work. We will review comments, operators, and variable declarations.

General Syntax Rules

There are a few things to know about JavaScript that apply everywhere. Understanding how these aspects of the language work can help you avoid mistakes in the future.

Line Endings

JavaScript uses the semicolon (`;`) to denote a line ending. Semicolons should be placed after every command in JavaScript like this:

```
let x = 12;
```

Line endings help JavaScript interpreters parse code and make it more obvious for other developers to understand when commands begin and end. JavaScript does not recognize indentation (although you should still indent your code properly), so different developers or groups will have different ways of spacing and breaking lines. Line endings help us understand the code regardless of the stylistic choices of the developers.

Brackets

In JavaScript, brackets (`[ ]`) are used to denote an Array and also to reference an index within an array. Here's an example:

```
let list = ['chocolate', 'ice cream', 'cookies'];
let myFave = list[2];
```

In the example above the brackets are used to denote the Array (which is stored in a variable called `list`). The brackets are also used to reference a specific *index* of the Array called `list` . (It's OK if this is a bit unclear; we will cover Arrays in the next section.)

Braces

Braces (aka "curly braces") look like this: `{ }` . They denote an Object in JavaScript, and they are also used to denote code blocks. Here is an example of the two ways braces are used:

```
let foo = {
   'name': 'Leia Organa',
   'rank': 'General'
};
```

In the example above, `foo` is an Object with two attributes: `name` and `rank` . The braces denote the contents of the Object.

In the next example, braces are used to set off the logic for a conditional statement:

```

if (someValue) {
    console.log('This value exists!');
} else {
    console.log('This value does not exist!');
}
```

In this example, the braces are used to denote the logic that happens for each case in the conditional. (Again, it's OK if it's fuzzy how these things work; we will cover them in an upcoming section.)

Case Sensitivity

Throughout JavaScript, everything is case sensitive. The object `window` is not the same as the object `Window`. We must use consistent case in our naming in order to have our code work. In future sections we will talk about stylistic conventions used for naming things in our programs, and the reason those stylistic conventions exist is to allow us to more predictably name things and not have so many issues with case or formatting.

In the following example, we would see an error result because the case is inconsistent:

```
let myValue = 42;
console.log('The meaning of life is ' + myvalue);
```

The code above would result in an "Uncaught reference error" because `myvalue` has not been declared as a variable. However, if we altered the code like this it would work:

```
let myValue = 42;
console.log('The meaning of life is ' + myValue);
```

The result of this code would be our intended message printed out to the JavaScript console.

Take heart!

Learning a programming language for the first time can be incredibly difficult. It's a lot like learning a foreign language for the first time. Just like we become more aware of how grammar works by studying a foreign language, we will become more aware of how "coding" works by learning to program. Programming is a combination of problem solving and then *encoding* the solution into something that a machine can interpret.

We must allow ourselves the space to learn how to express our logic through code, and we cannot make the mistake of believing that any of this stuff *should* be easy. This stuff is difficult. It is, in many ways, the

culmination of thousands of years of human thought and intellectual development. Don't feel bad if it takes more than a week to learn how to do it.

Adding JavaScript to an HTML File

JavaScript is related to an HTML file using the `<script>` tag. There are a few different ways to load JavaScript in an HTML file. The two primary choices are between inline JavaScript and external JavaScript. For the most part, we will use external JavaScript files because that allows for a nicer organization of our project and a better "separation of concerns". When we want to edit JS we shouldn't have to deal with all the HTML code in our page. The JS files are probably plenty long enough.

In each case, inline or external, we use a `<script>` tag. Read more about the `<script>` tag here.

Inline JavaScript

Inline JavaScript is written into the HTML of a document and contained in a `<script>` tag. There are cases where this is necessary in a production website/app, and it can be a nice way to experiment with specific features or solutions.

Here is an example of inline JavaScript:

```
<html>
<head>
  <title>Inline JS Demo</title>
</head>
<body>
  <h1 id="page-title">Replace This Text</h1>
  <script>
    let heading = document.querySelector("#page-title");
    heading.innerHTML = "Hello World!";
  </script>
</body>
</html>
```

The `<script>` tag can be placed in either the `<head>` or `<body>` of the HTML file, and it is executed whenever the browser encounters this tag while rendering the web page. There can be multiple `<script>` tags (and multiple inline scripts) in a single HTML file.

External JavaScript

External JavaScript is written in a separate file and then linked into the HTML file using a `<script>` tag with the `src` attribute set. We can imagine that we have a website with two files: `index.html` and `main.js`.

The `index.html` file looks like this:

```
<html>
<head>
  <title>External JS Example</title>
  <script src="main.js" defer></script>
</head>
<body>
  <h1 id="page-title">Replace This Text</h1>
</body>
</html>
```

The `main.js` file looks like this:

```
  let heading = document.querySelector("#page-title");
  heading.innerHTML = "Hello World!";
```

Notice that the code in the HTML file is the same, except the `<script>` tag has been moved to the `<head>` of the document and we've added two attributes to it. First, we added a `src` attribute that indicates the file we want to load. This link can be either absolute or relative to the HTML file (in this example, it is a relative link). We have also added a `defer` attribute to the `<script>` tag. This tells the browser that it's OK to defer the execution of this script until it has finished rendering the HTML. We typically add either the `async` or `defer` attributes to `<script>` tags when loading external JavaScript files. The `async` attribute tells the browser it's OK to continue rendering the HTML and that it's OK to execute the script whenever the file has finished loading.

In the `main.js` file we have the same code from the previous example. The only difference is that we now have a file that is fully dedicated to JavaScript, affording us a better experience editing the JS functionality on the page.

In general, it's preferable to use external JavaScript when building websites and applications. When building with web technology, each component (HTML, CSS, and JS) tends to become large and complex. It takes a lot of HTML to structure a site and define content. It takes many lines of CSS to define visual styles for a site. And it can take a lot of JavaScript to build all of the functionality wanted for a project. When dealing with large, complex projects, it's helpful to focus on one component at a time. That is easier when each

component is contained in dedicated files. (**Note:** For the moment I'm discounting and avoiding discussing technologies like JSX and how they might be changing this paradigm in the future.)

Similarly to CSS, JavaScript is also something that can be applied to multiple pages within a site. The features we define with JavaScript can be written in a way that they can operate on any page of our site that contains compatible HTML. Being able to link multiple HTML files to the same JavaScript file is a crucial part of making efficient use of the code we write.

For presentation purposes throughout this book, we will occasionally use examples that feature inline JavaScript, but for the most part in the working world we prefer to use external JavaScript files.

Blocking Out the Bad Old Days

Once upon a time, there was a great debate about how to load scripts in HTML. By default, when the browser encounters a `script` tag with a `src` attribute it stops rendering the page and waits for that file to download. This is a concept known as "blocking": All processing of the HTML is "blocked" until the linked JavaScript file is downloaded and executed. Unfortunately, this delays the processing of the page and increases the likelihood of user frustration.

To avoid blocking in the past, it was common for developers to put their script tags at the bottom of their HTML file, right before the closing `body` tag. That was an OK solution, but it put a whole set of important lines at the bottom of the HTML file, which made it a little more difficult to notice as a developer. It was also undesirable because there are situations when we want JavaScript to block the page load (for example, when delivering A/B content tests to end users) and in these cases we would end up with JavaScript in both the `head` of the document and the `body` of the document.

The HTML5 specification solved this problem by introducing the `async` and `defer` attributes for the `script` tag. This allows us to keep all of our JavaScript `script` tags in one place (in the `head` of our HTML documents) and still avoid blocking the browser's rendering of the page.

Comments

All programming languages allow developers to make "comments" on their code. Comments are non-functional lines that are meant to explain the logic of the code to our future selves or other developers. We use comments to make it clear why something is happening, how some component in our code should be used, or other details we wish to note for people reading our source code in the future.

Comments are essential for properly documenting our code and explaining how it works. We usually work in teams, so what seems obvious to one person will almost never be obvious to another. It's crucial to learn to comment well and to use comments strategically to enhance the readability of our code.

One-line Comments

One way of writing JavaScript comments is to precede any line with two slashes (`//`). This indicates that whatever follows is a comment and should not be executed by the interpreter. Here is an example:

```
// Declare a variable called "foo" and set it equal to 12.
let foo = 12;
```

We can use this format to comment *parts* of a line, too:

```
// Set up API details
let apiDOMAIN = "api.example.com"; // Domain of API endpoint.
let apiPort = 8080; // Port for API endpoint.
let apiVersion = "1.3"; // Specify API version.

// Combine API details into base path for API.
let apiBasePath = `http://${apiURL}:${apiPort}/${apiVersion}/`;
```

As we see above, we can combine one-line and partial-line comments however we'd like. The key is to maintain readability and to use comments to help other developers understand the code more fully.

Multi-line Comments

Multi-line comments have opening and closing symbols. They begin with `/*` and they end with `*/`. This style is often used when a comment needs to stretch across different lines. Sometimes this is necessary because of how much information needs to be conveyed, but it's often used to simply enhance the presentation of the code and provide more clear documentation.

Rewriting our example from above, we might see something like this:

```
/* API Details

   These details should be configured per deployment. Additional
   information can be found in the runbook.

   apiDomain   =  Domain of the API endpoint.
   apiPort     =  Port for the API endpoint.
   apiVersion  =  Version of the API to use.
   apiBasePath =  The path for the API used throughout this app.
*/

let apiDOMAIN = "api.example.com";
let apiPort = 8080;
let apiVersion = "1.3";

let apiBasePath = `http://${apiURL}:${apiPort}/${apiVersion}/`;
```

The multi-line comments are able to contain a much larger block of text, and we can use any simple text formatting we want to make the content of the comments more readable. Many developers will use some ASCII art to make their most important comments stand out.

Commenting Out Code

Both styles of comment can be useful when trying to debug code. It's often necessary to remove a line of code when trying to track down problems with your program. However, we usually don't want to just delete the line and risk losing our work. In these cases, commenting out lines can be very helpful.

Here is what it looks like to comment out a line using the one-line comment format:

```
// let titleHeading = document.querySelector("#title");
titleHeading.innerHTML = "Hello World!";
```

The first line of that code has been commented out. We could accomplish the same thing with multi-line comments (even though it's just one line):

```
/* let titleHeading = document.querySelector("#title"); */
titleHeading.innerHTML = "Hello World!";
```

But we could also use the multi-line comment to comment out both lines:

```
/* let titleHeading = document.querySelector("#title");
titleHeading.innerHTML = "Hello World!"; */
```

TODO: Make the Most of Comments

In any program, comments are probably used for a few specific reasons. One is to document what the code is doing in plain language that can be more quickly understood by collaborators. Another is to sketch out ideas in "pseudocode" so we can better understand what tasks we need the code to accomplish. We often use different commenting conventions to allow for comments to be understood by machines, usually in order to generate "automatic documentation". Some of the popular packages that can make comments into usable documentation include KSS and Autodoc and a lot more.

One of the most common ways to use comments is to track tasks yet to be completed with the phrase "TODO". This is done so that small tasks can be tracked in place. A developer can run a "find in files" search from their IDE or text editor to see all of the TODO notes across their code. By writing TODOs directly into the code, it's easy to see where functionality may be missing or temporary. When the next wave of improvements is made to the code, hopefully some of the TODOs will be completed.

When making notes like this in our code, it's critical to update the status of TODO notes. One of the worst things in a developer's day is to come across a TODO that is actually already implemented. In order to figure that out, we must review the code in detail, and it can be exceptionally confusing. We must be sure to note when our TODOs become TODONE, and we should remove those comments and replace them with more useful comments that help another developer understand the code more quickly.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Mark the comment with one-line (`//`) syntax.

```
Comment me out or I'll cause an error!
```

Exercise

Mark the comment with multi-line (`/*`) syntax.

```
Comment both of these lines out
or we'll cause an error!
```

Exercise

Comment out the broken line below using either form of commenting.

```
let foo { bar };
```

Operators

Arithmetic Operators are used to modify numbers in JavaScript. Operators include addition, subtraction, multiplication, and division. These are used throughout code to calculate different values and modify data flowing through the system.

Standard Operators

The standard operators are addition, subtraction, multiplication, and division. These operators always take in two numeric values (operands) as their input and return a single numeric value. They operate as we expect:

```
let a = 2 + 2; // Sets the variable "a" equal to 4.
let b = 2 - 2; // Sets the variable "b" equal to 0.
let c = 2 * 2; // Sets the variable "c" equal to 4.
let d = 2 / 2; // Sets the variable "d" equal to 1.
```

We could also use operators to calculate the value between variables:

```
let e = c * a; // Sets the variable "e" equal to 16.
```

Advanced Operators

In addition to the standard operators, there are advanced operators that we often use to do calculations in JavaScript. These operators also accept two numeric values (operands) and return a single numeric value as a result, but their behavior is a little more complex and less obvious than the previous operators.

Remainder

The `%` operator returns the remainder after the two operands are divided. Here are some examples:

```
let a = 4 % 2; // Sets the variable "a" equal to 0;
let b = 5 % 2; // Sets the variable "b" equal to 1;
```

In the example above, we can see that `a` is equal to `0` because `4 / 2 = 2` and there is no remainder. But `b` is equal to `1` because `5 / 2` equals `2` with a remainder of `1`.

This may seem like an odd way to express division. After all, most of the time we are happy to know that `5 / 2 = 2.5`. But knowing whether there is a remainder after division, and what the value of that remainder is, can be useful. For example, in order to determine if we're dealing with an even or odd number, we can check the remainder of any number after dividing by `2`:

```
let c = 1236 % 2; // Sets the variable "c" equal to 0;
```

If we were, for example, trying to color rows of a table differently to increase readability, we could check for "even/odd" and then add a class accordingly. If `c = 0` then we would know that row is "even".

Exponent

Sometimes we need to find the value of some number raised to some exponent. This is used to calculate all sorts of equations in Geometry, Physics, or numerous other situations. In order to do that we use the `**` notation:

```
let a = 2**4; // Sets the value of "a" to 16.
```

In this example, we are not multiplying `2 * 4` (which would be `8`). We are raising the number `2` to the exponent `4` (we usually describe this as "two to the fourth power"). This is equivalent to `2 * 2 * 2 * 2`, which equals `16`.

Increment

We often use different variables to count things in our programs. This happens all the time: counting times through a loop, items in a list, votes, etc. Because it's such a common thing to do, there is a shorthand for how we write it. We could write:

```
let counter = 0; // Initialize "counter" to 0.
counter = counter + 1; // Sets the value of "counter" to 1.
```

The code above works, but it can be shorthanded like this:

```
let counter = 0; // Initialize "counter" to 0.
counter++; // Sets the value of "counter" to 1.
```

Whenever `counter++` is executed in the code, it will increase the value of `counter` by `1`.

Decrement

As a companion operator to the increment operator, decrement uses the `--` symbol to denote that a value should be decremented by `1`. Here is an example:

```
let counter = 100; // Initialize "counter" to 100.
counter--; // Sets the value of "counter" to 99.
```

Each time `counter--` is run it would decrease the value of `counter` by `1`.

Grouping Operations

Just like with normal arithmetic, we can group operations using parentheses. The normal order of operations is maintained in JavaScript, and parentheses affect computation just as they would in regular math. For example:

```
let x = 12;
let y = 3.14;

let z = x + y * 3 - (x - y/2);
```

The value of `z` would be `10.990000000000002`. Following the order of operations, JavaScript would first figure out `y/2` (which is `1.57`). Then it would calculate `x - 1.57` to get `10.43`. Next it would figure out `y * 3`, which is `9.42`. Finally, it would work out `x + 9.42 - 10.43`. For some reason, thanks to the inaccuracy of Decimal data types in JS (again, we will cover this more later), the final result is `10.990000000000002`. Normally we would run another command to round that down to `10.99`.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Set `x` equal to three times seven.

```
let x =
```

Exercise

Set `x` equal to twelve minus eight.

```
let x =
```

Exercise

Set `x` equal to seven to the twelfth power.

```
let x =
```

Exercise

Add parentheses so the value of `x` equals `42`.

```
let x = 6 + 2 * 3 * 4 - 2 * 3;
```

Variables

In any programming language, data and discrete blocks of code can be labeled for reference throughout the program. We call these references "variables". Variables can have different values at different points in the code execution. For example, a variable named `counter` might be initialized to zero, but its value would likely be changed throughout the execution of the code as different logic is processed. It might be counting all sorts of objects or actions in the program.

When we create a variable, we call that "declaring" the variable. At the same time we declare a variable, we could also initialize the value of that variable to something. If we do not initialize the value of the variable to something, then JavaScript assigns the value `undefined` to that variable.

There are three ways to declare a variable in JavaScript: `var`, `let`, and `const`.

`var`

The generic way to declare a variable in JavaScript is to use the `var` command. This creates a variable that can be accessed in a variety of different ways. It is close to being a "global" variable, although that is something unique in JavaScript (and something we cover in a future section of this book).

Here is an example of declaring a variable using `var`:

```
var foo = "some value";
```

The Old Way: `var`

Due to the way that scope worked in JavaScript prior to ECMAScript 6, `var` was the old way of making local variables. Unfortunately, the `var` command does not actually make a variable local in the way most developers expect, and it has been replaced with the `let` command. The `var` command still works as it always has, but modern JavaScript should avoid using it as much as possible.

`let`

Most of the time in JavaScript, we want to control the "scope" of our variables. Scope is a term that we will dig into much more deeply later in this book, but for now it's worthwhile to understand that the `let` command allows us to create a "local" variable, which can only be used within its own code block. Here is an example:

```
let bar = "another value";
```

`const`

Most programming languages have a notion of "constant" variables, which are variables whose value cannot be changed during the execution of the program. Once we declare and initialize a variable with the `const` command we cannot change the value of that variable. This is useful for setting up parameters or preferences for our code. Here is an example:

```
const userID = 42;
```

Declaring and Initializing Variables

When we write code, we want to make it as clear as possible for ourselves (or our collaborators) to understand when we return to it. Sometimes this is helped by declaring variables at the top of a file even though we might not initialize them until later. It's not uncommon to see a list of variable declarations like this at the top of a file:

```
var foo,
    bar,
    baz;

const apiDOMAIN = "api.example.com";
const apiPort = 8080;
const apiVersion = "1.3";

const apiBasePath = `http://${apiURL}:${apiPort}/${apiVersion}/`;
```

In the example above, we have declared three variables for use later: `foo`, `bar`, and `baz`. At the time they are declared, they are not initialized to any value. A variable that is declared without a value automatically has the value `undefined` in JavaScript.

The other variables we declare are used to assemble the `apiBasePath` value. These values would likely be derived from inspecting some other aspect of the environment, and that information would be stored in constants so they cannot be altered during runtime. This insures that these values will not accidentally get overwritten with bad information.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Define a variable `x` equal to 10.

```
let x =
```

Exercise

Define a variable `x`, but do not initialize it.

```

```

Coding Style

It's important to write clean code that we can return to at a later date, or which our collaborators can review, and make sense of. The more consistent we are with how we write our code, the easier it is to return to it later and make improvements or fixes. In all programming languages there is a notion of a standard style and approach that grows out of common usage patterns and design choices in the language itself. JavaScript, of course, is no different.

And just as each newspaper has a slightly customized style guide that governs its newsroom writers, each development group will have a somewhat unique take on things. With a language like JavaScript, which has grown and changed a lot in recent years, it can be especially confusing to read multiple style guides that come from different eras. It's crucial to consult updated material for the latest, best practices. A resource like the JavaScript Standard Style is useful, and their site contains a living guide to the latest in generally accepted JavaScript style standards.

The guidelines presented below are not exhaustive. And for every feature of JavaScript there are preferred ways to write code using those features. So we will constantly be learning more best practices and evolving our approaches a we learn more and as the language evolves. But being aware of the stylistic patterns below will help you understand the code you read more quickly and write more readable code for others.

Linters Are Awesome!

A development in code editors over the past 10 years has been the rapid growth of "linters"—helper utilities that dynamically alert us to stylistic flaws or syntactical errors in our code. These can be thought of as a "grammar check" for code, and they work in similar ways. We can install a linter into our favorite editor (most editors support them) and they will usually underline, highlight, or mark in some way each line that causes either a stylistic warning or a syntactical error.

Linters are super helpful, especially as we are learning new languages. They make it easier to see the small syntax errors that so often cause new developers a lot of frustration, and they help us develop good habits writing code that meets stylistic guidelines. Look up linters for whatever

editor you prefer. If you want to try an editor an linter combo, download the [Atom](https://atom.io) editor and install the [JavaScript Standard Style Linter](https://atom.io/packages/linter-js-standard) to give it a try.

Variable and Function Naming

Variables and functions should be named with "camelCase". The variable or function name should always start with a lowercase letter, and then each word in the name should be capitalized. Here are some examples:

```
let my_var = true; // bad
let myVar = true; // good

let Foo = 42; // bad
let foo = 42; // good

function somelongname(){}; // bad
function someLongName(){}; // good

function another_long_name(){}; // bad
function anotherLongName(){}; // good
```

Class Naming

Classes should be named in "CapitalCamelCase", which capitalizes the first letter of the name. This indicates they are class objects and helps us understand better what we're working with. Examples:

```
class myClass {}; // bad
class MyClass {}; // good

class a_very_long_class_name {}; // bad
class AVeryLongClassName {}; // good
```

Spacing

Consistent spacing is very useful for creating more readable code. We should put spaces after commas, around operators, and after keywords. Here are some examples:

```
// Spacing after commas.
let myArray = [1,2,3]; // bad
let myArray = [1, 2, 3]; // good

myFunction(true,12,'Jane'); // bad
myFunction(true, 12, 'Jane'); // good

// Spacing around operators (and equals signs).
let x = 12/2; // bad
let x = 12 / 2; // good
```

Indentation

Indentation is extremely important for writing readable code. It's nearly impossible to decipher code that is poorly indented, and bad indentation is generally seen as a huge red flag for code quality. Use 2 spaces to indent JavaScript for each code block.

```
function capitalizetext(text){ // bad
return text.toUpperCase();
}
function capitalizeText(text){ // good
  return text.toUpperCase();
}

  if (someValue) {               // bad
console.log('This is true.');
}else{
console.log('This is not true.');
}

if (someValue) {                 // good
  console.log('This is true.');
} else {
  console.log('This is not true.');
}
```

Code Blocks

Code blocks are defined using the curly braces (`{ }`), and they are important in JavaScript. They are used by loops, conditionals, functions, classes, and other objects in the system to define sets of logical instructions. In general, we should place the opening curly brace (`{`) on the first line opening the codeblock, and the closing curly brace (`}`) should align with that first line. Here is an example:

```
// bad
if (x < 12)
{
  if (x % 2 === 0)
  {
    console.log('X is even and it is less than twelve.');
  }
  else
  {
    console.log('X is odd, and it is less than twelve.');
  }
}
else
{
  if (x % 2 === 0)
  {
    console.log('X is even and it is greater than twelve.');
  }
  else
  {
    console.log('X is odd, and it is greater than twelve.');
  }
}

// good
if (x < 12) {
  if (x % 2 === 0) {
    console.log('X is even and it is less than twelve.');
  } else {
    console.log('X is odd, and it is less than twelve.');
  }
} else {
  if (x % 2 === 0){
    console.log('X is even and it is greater than twelve.');
  } else {
    console.log('X is odd, and it is greater than twelve.');
  }
}
```

And Much More

Understanding these fundamental aspects of JavaScript style will help us get started reading code and writing better coe, but we will be learning a lot more about how to write JavaScript, and as we learn new techniques we will have more use for additional style guidelines. As with almost every other form of writing, consistency is the key here, so the most important

thing is for us to find a style guide we like and then stick with it. It will be much easier to accomplish that goal if we install a linter into our preferred code editor, which would be a good idea.

Quiz: Basic Syntax

Try this self-check quiz to test your knowledge!

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Extra Practice

Remember: Learning to program takes practice! It helps to see concepts over and over, and it's always good to try things more than once. We learn much more the second time we do something. Use the exercises and additional self-checks below to practice.

```
// Refer to this code to answer the questions.
let x = 12;
let y = 3;

let z = x % y;

let foo = 5;
x = 16;
let bar = x % foo;

/* let z = 42; */

let baz = z + bar;
```

Visit Content Online

The content on this page has been removed from your PDF or ebook format. You may view the content by visiting this book online.

Data Types and Structures

JavaScript is a dynamic language, which means that it determines the Data Type of a variable based on the value assigned to that variable. For example, we can assign different values to the same variable throughout our script and that variable will take on the features of different data types:

```
let myData = "some text"; // myData is a string
myData = false; // myData is a boolean
myData = 5446; // myData is a number
myData = ['text', 42, {'name': 'Jane Doe'}]; // myData is an object
```

In the example above, we have made four different value assignments to `myData`. Each time, we have changed the Data Type for the variable. If you repeat this experiment in your web browser's JavaScript console, then you will see that you can run the `typeof()` command on this variable to find out it's value at each step:

```
> 16:00:00.000 var myData = "some text"; // myData is a string
               typeof(myData);
<· 16:00:00.000 "string"
> 16:00:00.000 myData = false; // myData is a boolean
               typeof(myData);
<· 16:00:00.000 "boolean"
> 08:18:30.253 myData = 5446; // myData is a number
               typeof(myData);
<· 08:18:30.253 "number"
> 08:18:30.253 myData = ['text', 42, {'name': 'Jane Doe'}]; // myData is an object
               typeof(myData);
<· 08:18:30.253 "object"
>
```

Anytime in your code we can check the type of a value using the `typeof()` command. This allows us to make better decisions, prevent errors, and provide more functionality to our users.

Data Structures

The primary Data Structure used in JavaScript is the Object. This is a structure that underlies virtually every aspect of JavaScript, and it's one that we use often in our regular programming work. Objects allow hierarchical relationships to be created between attributes and methods, which lets us set up conceptual frameworks to contain any idea we wish to work with in our code.

These are big ideas, so we will unpack them each on their own. Remember that if you've never learned programming before these are probably brand new ideas, and it will take some time and practice to fully wrap our heads around them. That's OK.

Duck Typing

JavaScript's dynamic typing is a concept that is sometimes hard to grasp. It is probably most accurately called dynamic (or "late") binding, which is not a particularly friendly term. Computer scientists and developers have discussed JavaScript's typing system at great length, largely due to the fact that the method provides both a great deal of flexibility and a not-inconsequential amount of variability. Sometimes, in JavaScript you have to try the code to know for sure what you're going to get, and that unpredictability can sometimes lead to breakage in software.

A common, not entirely accurate, but much more friendly way to think about JavaScript's typing system is with the term "Duck Typing". This term refers to the fact that JavaScript does not actually *care* what the type of a value is, only that it has all of the features needed by your code at the moment.

For example, if we are trying to perform a `split()` on some text (to break apart words or letters), the JavaScript interpreter only looks for that `split()` method to exist on our variable. If the variable is, indeed, currently set to a type that supports `split()` then the code executes properly. If not, then an error is raised.

Another way to express this is like so: JavaScript doesn't care whether it's dealing with a Duck. If it quacks like a duck, walks like a duck, flies like a duck, and swims like a duck, then it's a duck.

Numbers

Most programming languages make a division between an integer and a decimal. JavaScript does not make any distinction between the two. JavaScript has only one numeric Data Type, called Number. If a variable is given a numeric value, then it will be of the type "number":

```
let x = 12;
if (typeof(x) === "number"){
    // This code will execute because 12 is a value of the type "number".
    console.log('x is a number!');
}
```

We could change the value of `x` to `3.14` and the conditional above would still evaluate to `true` in the same way because as far as JavaScript is concerned there is no difference between a whole number and a number with a decimal point.

It is possible to determine whether or not a Number is an integer using a tool provided by the `Number` object. consider the following code:

```
let x = 12;
if (Number.isSafeInteger(x)){
    // This code will execute because x is currently set to an integer value.
    console.log('x is an integer!');
}
x = 3.14; // Change x to a decimal number.
if (Number.isSafeInteger(x)){
    // This code will not execute because x is currently set to a decimal value.
}
```

The `Number.isSafeInteger()` function can be used to determine if a number is an integer or not. If it is an integer, the function will return `true`. If it is not an integer, then it will return `false`. In the cases where differentiation between types of numbers matters, this is a helpful tool.

Unexpected Math

JavaScript does not have a way of restricting "floating point" or "decimal" numbers to a specific number of places. We can use methods like `Number.toFixed()` to round off at a specific decimal point, but we can

always expect to see weird things like this:

```
let total = 0.10 + 0.20; // expected result is 0.3
(or maybe 0.30)
console.log(`Actual value of total is: ${total}`);
// prints 0.30000000000000004 to the console.
```

This all stems from the fact that numbers with decimal points in JavaScript are all treated as "floating point" numbers, which is not the precise "decimal" numbers we are used to dealing with in our finances or other aspects of our daily lives. We won't dive into what makes "floating point" numbers work, but we can recognize and anticipate that we might encounter trouble working with decimal numbers in JavaScript when dealing with decimals. If we are working on certain types of applications, we may need to explore some different methods for working around the unexpected results of floating point math.

The savvy developer will search around for the latest articles, cheatsheets, and examples for dealing with floating point issues in their code.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Define a variable `x` equal to an integer.

```
let x =
```

Exercise

Define a variable `x` equal to a decimal.

```
let x =
```

Strings

In most programming languages, the Data Type for what we would normally consider "text" is the String. Strings can contain any text (including numbers and symbols) that can be typed. Here are some examples:

```
let x = 'Hello World!';
let y = "Hello World!";
let z = "中文 español deutsch English हिन्दी العربية português বাংলা русский 日本語 ▫▫▫▫▫▫
한국어 தமிழ் עברית";
```

Each of the variables above has been assigned a value that is of the type "string". We can use either single quotes (`'`) or double quotes (`"`), but the quotes around our string must match.

Strings may contain any amount of text desired, and sometimes it may be necessary to break a String value across multiple lines. There are two ways to do this. The first way uses the `+` sign to *concatenate* each line:

```
let longText = "This text is " +
    "very long and contains " +
    "many different characters.";
```

The other way to define long String values is to use the backslash character at the end of the line to indicate that the string will continue on the next line:

```
let longText = "This text is \
    very long and contains \
    many different characters.";
```

Escaping Characters

Although we can choose to use either single or double quotes to indicate a String, there are some things to consider when choosing between the two. A String defined with single quotes may contain double quotes without breaking the String. And a String defined with double quotes may contain a single quote (or apostrophe) without breaking the String. Sometimes it makes sense to selectively choose one or the other based on what the content of the String will be. Take a look at this example code:

```
let a = "Today's Featured Video"; // The single quote can be used as an apostrophe bec
ause the String is defined with double quotes.

let b = 'The important critic said, "I love this app!"'; // The double quotes can be u
sed because the single quotes are used to define the String.
```

In the example above, we can see that using single or double quotes can be advantageous for certain types of content. However, it's not the nicest style to switch between single and double quotes like this, and there are some characters that remain problematic within Strings regardless of which style of quote you choose.

Remember that backslash (`\`) we used to indicate a multi-line string? What if we need to make a string that contains backslashes? What if we have text that requires a mixture of double quotes and apostrophes? What if we want to add a carriage return or a tab to the String? Those characters are visible when we read them (they make blank space for us), but how do we even type them?

In order to solve this problem, we can **escape** characters in a String. Escaping is done by prefixing the character with a backslash (`\`). Here is an example:

```
let a = 'Today\'s Featured Video'; // The apostrophe has been escaped using a backslas
h character.

let b = "The important critic said, \"I love this app!\""; // The double quotes have n
ow been escaped with a backslash.
```

In the example above we can see that using the backslash allows us to choose which quotes we prefer regardless of the content of our String.

Another set of important characters to escape are Unicode characters. If we wanted to insert a ❤ character in a String, then we would use the following code:

```
let heart = "Here is a heart: \u2764"; // The backslash escapes the unicode character.
```

If we did not escape the Unicode value then our String would be "Here is a heart: u2764" instead of "Here is a heart: ❤".

We can find a full table showing how to escape characters on the MDN String page. Here it is for convenience:

Code	Output
\0	the NULL character
\'	single quote
\"	double quote
\	backslash
\n	new line
\r	carriage return
\v	vertical tab
\t	tab
\b	backspace
\f	form feed
\uXXXX	unicode codepoint
\xXX	the Latin-1 character

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Fix the String definition below so it works.

```
let x = "Courage is not the absence of fear,
    but rather the judgement that something
    else is more important than fear.";
```

Exercise

Fix the String so the text shows up properly.

```
let x = "You will need a 3/8" socket for this job.";
```

Exercise

Fix the String so the text shows up properly.

```
let x = 'All's I know is, my gut says, "Maybe"';
```

Exercise

Fix the String so the text shows up properly.

```
let x = 'In Seattle we don't use an u2602.';
```

Other Primitive Data Types

JavaScript also gives us a few Data Types that are used in more specific ways: Boolean, Undefined, Null, and Symbols.

Boolean

The Boolean Data Type has a value that is either `true` or `false`. These types of variables are helpful when controlling the logical flow of a program. We often use different logic to determine a `true` or `false` so that we can later have an easy check to determine the next commands the program should execute.

Boolean values are generally checked within a conditional (an `if` statement) or to control the execution of a `while` loop. These will be used throughout our code.

Undefined

Undefined is the type for any value that has been declared but not initialized to a value. When we write something like this, the type of the variable `foo` is `undefined`:

```
let foo;
```

The following example indicates how this could be used in code:

```
let foo;
if (typeof(foo) === undefined) {
    // Code here would execute because foo is undefined.
}
```

This is the default type for any variable that has not been assigned a value.

Null

Null is a Data Type that is used to indicate there is nothing assigned to the variable. At first, it seems as if it's a duplicate of Undefined, but in fact it's different in an important way. We would probably never assign `undefined` as a value to a variable in our system. Remember, it is the *default* type of a non-initialized variable.

But imagine the case where we have some variable where there *could* be a value assigned, but at some point in our program it *does not* have a value assigned. This is where the Null data type becomes helpful. We can set the value of a variable to `null` to indicate the variable does not currently point to a value. This is used often in API design to indicate that where there *could* be information it is missing.

Here's an example of using `null` in code. Imagine we are writing a system trying to manage a `currentUser` object:

```
let currentUser;
if (typeof(foo) === undefined) {
    // This code executes because foo has not been assigned a value.
}
```

Later on in our code, we might want to set the `currentUser` back to `null` to indicate no user is logged in:

```
currentUser = null;
if (typeof(foo) === undefined) {
    // This code will not execute because currentUser is equal to null.
} else if (typeof(foo) === null) {
    // This code will execute.
    // Do something to login the user
}
```

Now, when we do the check against `undefined` it's `false` because `currentUser` has been defined as `null`.

It's common for developers to use `null` to represent missing data. Imagine an object that stores an address. We could picture that object with attributes for both "street address" and "street address line two". However, many people do not have an additional street address line to use in their addresses. In this case, the optional "street address line two" might be equal to `null` to represent that value is missing in a purposeful way.

Objects

Just about everything in JavaScript is an Object at its base level. An Object is a Data Type (if we create an object and run `typeof()` on it, we can see that it returns `"object"`), but it is also a Data Structure. The structure of an Object allows attributes to be related. Each attribute of an Object has its own Data Type and can be used just like any standalone variable of that Data Type. Consider the following code:

```
let foo = {}; // This is common syntax for "initializing" an empty Object.

foo.name = "Foo Object"; // In this case foo.name is a String.
foo.counter = 0; // In this case, foo.counter is a Number.
```

Looking at the code above, we can see that `foo` is initially created as an empty Object, meaning that our program would know that `foo` is an Object, but it does not yet have any custom attributes defined. In the next lines, we define two attributes of `foo`: `foo.name` and `foo.counter`. The `foo.name` attribute is a String, and the `foo.counter` attribute is a Number. These two values can be used in the same way as normal String or Number variables.

It's also useful to note the basic syntax of creating and using Objects. Curly braces (`{ }`) indicate an object, so we can use them as above to initialize an empty Object. This is a very common practice in writing JavaScript code. We can also see that assigning new attributes to an Object is just like creating new variables, except we name them using the "dot naming syntax" that is common to JavaScript. We could define as many attributes as we'd like on this Object (or any other), and we can add/change attributes at will.

Attributes can be set to any kind of Object, any Data Type, or even Functions (which are actually just Objects, too). Consider the following code:

```
let venue = {}; // initialize the empty object `venue`
venue.name = "The JS Palace";
venue.description = "The best place to work on your JavaScript code!";
venue.address = {}; // initialize another empty object called `venue.address`
venue.address.streetAddress = "123 Fake St.";
venue.address.city = "Seattle";
venue.address.state = "WA";
venue.address.zip = 98112;
```

In the code above, we can see that `venue` is an object containing information about a business location. The attributes defined on `venue` are pretty normal, but it's worth pausing to look at the `venue.address` attribute. That attribute is, itself, an Object. It is able to contain the address information in a self-contained, well-labeled block of data.

Objects allow us to create relationships and hierarchies between information, and they are very useful for doing so. In fact, the JavaScript Object Notation (JSON) format has become so popular that it is used across virtually every other programming language to create well-defined hierarchical data structures. We will explore more about JSON in a couple sections, but first we must discuss Arrays.

Referencing Properties of Objects

Throughout our code we will need to reference properties of Objects. This can be accomplished in two ways: "dot-notation" and "keys". These two methods can be used interchangeably, and each is valuable for specific purposes. It's important to keep in mind these two methods of accessing the values of an Object in order to get the most from this Data Structure.

Here is some example code to review:

```
let clubMembers = {
    grace: {
        name: "Grace Hopper",
        email: "grace@example.com"
    },
    guido: {
        name: "Guido van Rossum",
        email: "guido@example.com"
    },
    brendan: {
        name: "Brendan Eich",
        email: "brendan@example.com"
    }
}
```

In the code above, we have an Object defined that uses the usernames of club members for the primary index. This means that we could access these values in two ways:

```
console.log(clubMembers.grace.name); // Outputs "Grace Hopper" to the console.

console.log(clubMembers['grace'].name); // Outputs "Grace Hopper" to the console.
```

The two `console.log` statements above have the same results: They print the `name` property from the object contained in the `grace` property of the `clubMembers` object. The first method of referring to the value is "dot notation" and it is often used when we know exactly which property we need to access in an Object. The second method uses a reference to a "key", which can be useful when we want to use a variable to supply that name. Consider this example:

```
let username = "guido";

console.log(clubMembers[username].name); // Outputs "Guido van Rossum" to the console.

console.log(clubMembers['guido'].name); // Outputs "Guido van Rossum" to the console.

console.log(clubMembers.guido.name); // Outputs "Guido van Rossum" to the console.
```

In this example, you can see that the variable `username` has a value assigned to it. The value is the String, `"guido"`. This matches the "key" for the Object defined at `clubMembers.guido`. So we can access the `name` property of the Object stored at `clubMembers.guido` using either of the three syntax methods above.

The dot notation method is very clean when we know what property we need to access. But when we're using variables (such as when looping through Arrays or when correlating data from another variable or source), it's often necessary to use the "key" notation to use an expression to reference the value stored in a given property (or at a given "key").

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Define an Object called `foo`.

```
let
```

Exercise

Define an Object called `foo` with a `name` attribute equal to "Bob".

```
let
```

Exercise

Set `baz` equal to the attribute `bar` in the Object `foo` without using dot notation.

```
let key = 'bar',
let foo = {
    bar: 42
}

let baz =
```

Arrays

Objects work a lot like dictionaries: If you know what you want to look up, then it's very easy to get information. But there are many times when we don't need a full dictionary, we only need a list of items. Consider the case where we want to keep track of a list of items to purchase at the grocery? Or a list of people in a group? Or a list of tags that have been applied to content in our system?

In these cases, it's better to use a Data Structure like an Array. Arrays are *indexed* collections of information. This means that each item in the Array can be referenced by a number that indicates its position in the Array.

Here is some code that demonstrates creating and accessing information in an Array:

```
let sandwich = ['lettuce', 'mayo', 'turkey']; // A list called `sandwich` has been cre
ated with three items in it.
```

In the example above, a list called `sandwich` has been created with three items in it: 'lettuce', 'mayo', and 'turkey'. Each of these items is a String, but Arrays can contain any data types:

```
let sampleObject = {};
sampleObject.name = "Demo";
let crazyList = ['Zimbabwe', 42, sampleObject, false];
```

In this second example, `crazyList` contains items of differing Data Types. The first is a String, then a Number, then an Object, and then a Boolean. Arrays are flexible and useful for holding sets of things that need to go together.

Arrays also come with some features that make it easy to work with them. One of the most useful is the `length` attribute. We can determine how many items are in an array by using this attribute like so:

```
let flowers = ['rose', 'carnation', 'daffodil'];
console.log(flowers.length) // Will output `3` to the console.
```

Accessing the length attribute on an Array is especially useful when presenting information about data to users (e.g. How many tags on this content item?, etc.) or when you're looping through Arrays using other methods (which we will discuss in the next section).

Accessing Items in an Array

JavaScript Arrays are "zero indexed" which means that they begin counting their items with the number `0`. To access any item in an Array, we can reference the position of the item using the number that corresponds to that item's position. Here is an example:

```
let dogs = ['retriever', 'hound', 'mutt'];
console.log(dogs[0]); // Prints 'retriever' to the console.
console.log(dogs[1]); // Prints 'hound' to the console.
console.log(dogs[2]); // Prints 'mutt' to the console.
```

As we can see in this example, a list with three items will fill positions `0`, `1`, and `2` in the Array. To access any of these positions, we can use the square bracket notation to reference a specific index within the Array: `dogs[1]` to reference `'hound'`, for example.

It's also possible and common to loop over items in an Array, and there are some built-in tools that make it very convenient to do that in our code. We will dig deeper into that in the next section when we talk about loops more generally.

Adding Items to an Array

A list is much more useful when you can add new items to it. To accomplish this, Arrays have a `push()` command that can add an item to the end of the list:

```
let tags = ['js', 'programming', 'educational']; // An Array called `tags` is created
with three items.
tags.push('book'); // The item `book` has been added to the Array.
```

By using the `push()` command, we have added the String `book` to the Array called `tags`. That Array now looks like this in our code: `['js', 'programming', 'educational', 'book']`. We could continue adding new items as needed.

Removing Items from an Array

There are three ways to remove items from an Array: `shift`, `pop`, and `splice`. Each of these provides a very specific way to remove items from the Array.

To remove the item at the beginning of the Array, use the `shift()` command:

```
let fruits = ['apple', 'orange', 'banana'];
fruits.shift(); // Removes 'apple' from the list leaving ['orange', 'banana'].
// `fruits` is now equal to ['orange', 'banana']
```

To remove the item at the end of the Array, use the `pop()` command:

```
let fruits = ['apple', 'orange', 'banana'];
fruits.pop(); // Removes 'banana' from the list leaving ['apple', 'orange'].
// `fruits` is now equal to ['apple', 'orange']
```

The two previous commands are simple and straightforward. The next way to remove items from an Array is a little more complex.

To remove one or more items from within an Array, use the `splice()` command. The `splice()` command takes two parameters: The position to begin removing, and how many items to remove after that position. Of course, in order to get the most from this command we often need to first use another command, `indexOf()`. We can use `indexOf()` to find the position of an item in an Array. Here is an example of using both of these tools to remove several items from an Array:

```
let colors = ['blue', 'red', 'green', 'yellow', 'black', 'white'];
let position = colors.indexOf('yellow'); // The `position` variable will be set to `3`
 since that is the index for the item 'yellow' in this Array.
colors.splice(position, 2); // This command removes 'yellow' plus the one item followi
ng 'yellow' (which is 'black').
// `colors` is now equal to ['blue', 'red', 'green', 'white']
```

In this example we can see that the `colors` array consists of six items. We use `indexOf('yellow')` to determine the position of `'yellow'` in the Array. Then, we use that position value with the `splice(position, 2)` command to remove `'yellow'` and the item following it (which is `'black'`). The `2` in our `splice()` command indicates that two items should be removed from the Array.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Define an Array called `names` with a length of 4.

```
let names =
```

Exercise

Remove the first item from this Array.

```
let vehicles = ['truck', 'bus', 'ship', 'airplane', 'rocket'];
```

Exercise

Remove the last item from this Array.

```
let vehicles = ['truck', 'bus', 'ship', 'airplane', 'rocket'];
```

Exercise

Remove the the items 'bus' and 'ship' from this Array.

```
let vehicles = ['truck', 'bus', 'ship', 'airplane', 'rocket'];
```

JSON

Representing data in structured formats is crucial to many forms of programming, including JavaScript. In order to solve the problem of how to structure data in a way that could be portable online but still retain proper relationships, developers have gravitated towards JavaScript Object Notation (commonly called "JSON"). JSON was derived based on how Objects in JavaScript are defined, but it has become a standard data interchange format that is used in virtually every programming language.

Understanding JSON data structures is crucial for developers in any language. But if we're familiar with JavaScript Objects and Arrays, then the JSON data structure does not seem so odd. Here is an example taken from Wikipedia:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

JSON data structures are written as "anonymous" Objects. That means there is no `var objectName =` at the beginning of the data structure. When applications are using JSON, they typically understand that they should interpret the entire structure as a single value. For example, the JSON data above may have come from an API call to a server to fetch the profile information for the user `John Smith`.

We can see that JSON uses basically the same format as Object definitions in JavaScript. The curly braces (`{ }`) indicate Objects, and the square brackets (`[ ]`) indicate Arrays. Look, for example, at the `phoneNumber` Array: It contains two Objects. Each of the `phoneNumber` objects have two attributes: `type` and `number`. The user `John Smith` has two phone numbers; one of them is his home number and the other is his fax number. This is a common case where users have multiple phone numbers and wish to label each one accordingly. By using the structure of JSON we can easily understand how all this information is related, and we could use all of the information within an application.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section. The exercise below refers to the following JSON object, which is given the name `userData` in the JavaScript context.

```
let userData = {
  'name': {
    'firstName': 'Grace',
    'lastName': 'Hopper',
    'derbyName': 'Amazing Grace'
  },
  'email': 'grace@example.com',
  'phone': '555-555-5555',
  'dob': "December 9, 1906",
  'dod': "January 1, 1992",
  'militaryService': {
    'rank': 'Rear Admiral',
    'awards': [
      'Defense Distinguished Service Medal',
      'Legion of Merit',
      'Meritorious Service Medal',
      'American Campaign Medal',
      'World War II Victory Medal',
      'National Defense Service Medal',
      'Armed Forces Reserve Medal with two Hourglass Devices',
      'Naval Reserve Medal',
      'Presidential Medal of Freedom (posthumous)'
    ],
    'startYear': 1943,
    'endYear': 1986
  },
  'keywords': [
    'UNIVAC',
    'COBOL',
    'programming'
  ]
}
```

Exercise

Refer to the JSON object defined above to set the proper values for each of the variables in the exercise.

```
// For each of the tasks below, refer to the `userData` object like this:
let name = userData.name;

// TODO: Set the value of `email` to the user's email address
let email = ;

// TODO: Set the value of `derbyName` to the user's derby name.
let derbyName = ;

// TODO: Set the value of `milServiceStart` to the first year of the user's militar
y service.
let milServiceStart = ;

// TODO: Set the value of `rank` to the military rank of the user.
let rank = ;

// TODO: Set the value of `thisKeyword` to "COBOL".
let thisKeyword = ;

// TODO: Set the value of `thisAward` to "Meritorious Service Medal".
let thisAward = ;
```

Working with Text

Since so much of what we do as programmers revolves around expressing logic and describing concepts or objects in text, it's crucial to build up some skills for working with text in our code. By this we mean not only working with String Data Types, but also combining Data Types to create the text we need for any given purpose.

Concatenating Strings

Probably the most straightforward way of combining text and data is to use the concatenation operator (`+`) to cobble together strings. Here is what that looks like in code:

```
let name = "Jane";
let destination = "library";
let distance = 1.2;

let message = "Your trip: " + name + " is going to the " + destination + " which is "
+ distance + "miles away.";
// `message` now contains the text: "Your trip: Jane is going to the library which is
1.2 miles away.";
```

This is a fairly easy way to understand putting text and variables together. We can combine Numbers and Strings into another string, and then we could output that `message` to the user. Note that we use the plus signs to concatenate the values together into a big string, and we must be sure to include spacing in the quotes so that we don't slam the values and text together. (Try running these lines in a JavaScript console and see what it looks like if you remove those extra spaces.)

But this method can also be difficult to manage: What if we need to put variable into a *lot* of text? What if we need to put information from an Array into the text? What if we don't like writing so many plus signs and making sure to put spaces in our quotes correctly?

Luckily, we have more tools at our disposal.

Joining Arrays and Splitting Strings

A common task when writing code is combining items in a list into some textual representation. Sometimes we can solve this problem using the `join()` command that comes with every Array. The `join()` command allows us to specify a character that will be

used between items in the list, and it outputs a string containing the generated text. Here's how it works:

```
let fruits = ['apple', 'banana', 'orange', 'lime'];
let fruitText = fruits.join(', '); // Joins all Array items and inserts a ', ' between
 them.
// fruitText now contains the String: 'apple, banana, orange, lime'
```

There are many situations where using `join()` like this is a good approach. It can be even more useful when combined with the String `split()` command. We can use `split()` to break a String apart like so:

```
let ogText = "Sometimes I worry about being a success in a mediocre world." // Quote i
s courtesy Lily Tomlin!
let textArray = ogText.split(' ');
// textArray equals ["Sometimes", "I", "worry", "about", "being", "a", "success", "in"
, "a", "mediocre", "world."]

let shoppingText = "eggs,butter,milk,bread";
let shoppingArray = shoppingText.split(',');
// shoppingArray now equals ["eggs", "butter", "milk", "bread"]
```

In the example above, we can see that we have two different strings which we can split apart on characters of our choosing. We could also use the `join()` command to join them again into a string. If we use the same character to join, then we would end up with the original string:

```
let ogText = "Sometimes I worry about being a success in a mediocre world." // Quote i
s courtesy Lily Tomlin!
let textArray = ogText.split(' ');

let newText = textArray.join(' ');
if (ogText === newText){
    // This code will execute because `ogText` and `newText` are identical.
    console.log('We have re-made the original text!');
}
```

Being able to break apart strings and re-combine them is helpful in many situations. For example, we often do this sort of thing when analyzing file paths or URLs. It can be helpful for allowing users to enter data in different interfaces, too, where providing a comma-separated list is more convenient than typing each item of the Array individually.

BEWARE Although these tools for breaking apart Strings and making them Arrays are useful, be wary of using them to replace proper Data Structures in your code or HTML interfaces.

Template Literals

Finally, we have a great tool in JavaScript that allows us to insert data values into text in a very convenient way. Template Literals are defined with the backtick character (```). We can work more easily Template Literals and avoid many of the issues we previously had with String concatenation. For example, consider our concatenation example updated to use Template Literals:

```
let name = "Jane";
let destination = "library";
let distance = 1.2;

let message = `Your trip: ${name} is going to the ${destination} which is ${distance}
miles away.`;
// `message` now contains the text: "Your trip: Jane is going to the library which is
1.2 miles away.";
```

We can see that the end result is the same, but getting there is much easier on the eyes. Template Literals allow us to drop variables into Strings by referencing them with expression tags (`${expression}`). This allows us to write code that is much more simple. Template Literals can also span multiple lines, so we can do things like this:

```
let salutation = "Dearest";
let name = "John";
let numberOfDays = 15;
let numberOfHours = 7;

let message = `${salutation} ${name},
    It's been ${numberOfHours} hours and ${numberOfDays} days
    since you took your love away.

    Please return it soon`;

console.log(message);
```

This code would output:

```
Dearest John,
It's been 7 hours and 15 days
since you took your love away.

Please return it soon.
```

Template Literals make it much easier to work with large chunks of text. They can span multiple lines, and their formatting stays generally intact. But they can also be made a little more dynamic by adding JavaScript expressions in addition to just inserting values from variables. Here is an example:

```
let itemName = "Widget";
let price = 4;
let tax = 0.05;
let quantity = 3;

let receipt = `RECEIPT OF PURCHASE
    ${itemName} x ${quantity} @ ${price}

    Price: ${quantity * price}
    Tax: ${price * tax}
    -----------------------------------
    Total: ${price + (price * tax)}`;

console.log(receipt);
```

The code above would output the following text to the JavaScript console:

```
RECEIPT OF PURCHASE
Widget x 3 @ 4
Price: 12
Tax: 0.2
-----------------------------------
Total: 4.2
```

As we can see, it's possible to do operations within the expression tags. This makes Template Literals even more flexible and useful.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Split the String `shoppingList` into an array on the commas.

```
let textList = "bread,milk,butter,eggs";
let shoppingList =
```

Exercise

Join the Array using a slash (`/`) as the joining character.

```
let pathArray = ['Projects','web','resources','js','main.js'];
let filePath =
```

Exercise

Replace the expression tags in this template literal to insert the corresponding data.

```
let name = "Grace Hopper";
let source = "Ships Ahoy Magazine";
let date = "July 1986";

let message = `"It's easier to ask forgiveness than it is to get permission."
    Name, Source
    Date`;
```

Quiz: Data Types and Structures

Please enjoy this self-check quiz to review some of the concepts from this section.

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Extra Practice

Remember: Learning to program takes practice! It helps to see concepts over and over, and it's always good to try things more than once. We learn much more the second time we do something. Use the exercises and additional self-checks below to practice.

1. Data Types and Variable Assignment

```
let score = 42;

let quote = "Move fast and break stuff.";

let fruits = ['apple', 'orange', 'pear'];

let foo,
    bar,
    baz;

let stopLoop = false;

let userRegistered = true;
```

2. JSON Interpretation

```
let weatherData = {
    "city": {
        "id": 5809844,
        "name": "Seattle",
        "coord": {
            "lon": -122.3321,
            "lat": 47.6062
        },
        "country": "US",
    },
    "list": [
        {
            "dt": 1500148800,
            "temp": {
                "day": 74.05,
                "min": 55.96,
                "max": 75.09,
                "night": 55.96,
                "eve": 69.6,
```

```
                "morn": 68.9
            },
            "pressure": 1024.13,
            "humidity": 51,
            "weather": [
                {
                    "id": 800,
                    "main": "Clear",
                    "description": "sky is clear",
                    "icon": "01d"
                }
            ],
            "speed": 5.17,
            "deg": 222,
            "clouds": 0
        },
        {
            "dt": 1500235200,
            "temp": {
                "day": 67.3,
                "min": 55.78,
                "max": 68.94,
                "night": 56.08,
                "eve": 65.71,
                "morn": 55.78
            },
            "pressure": 1026.81,
            "humidity": 57,
            "weather": [
                {
                    "id": 800,
                    "main": "Clear",
                    "description": "sky is clear",
                    "icon": "01d"
                }
            ],
            "speed": 3.38,
            "deg": 353,
            "clouds": 0
        },
        {
            "dt": 1500321600,
            "temp": {
                "day": 69.85,
                "min": 52.63,
                "max": 70.54,
                "night": 52.63,
                "eve": 66.49,
                "morn": 58.32
            },
            "pressure": 1019.85,
            "humidity": 51,
            "weather": [
```

```
                {
                    "id": 801,
                    "main": "Clouds",
                    "description": "few clouds",
                    "icon": "02d"
                }
            ],
            "speed": 6.71,
            "deg": 357,
            "clouds": 12
        }
    ]
}
```


The data above represents a three-day weather forecast for Seattle, WA. Use this data structure to answer the questions below.

Visit Content Online

The content on this page has been removed from your PDF or ebook format. You may view the content by visiting this book online.

Controlling Logical Flow

In all programming languages, and in every program beyond the most basic set of instructions, it's crucial to control the flow of logic to achieve the desired result. Most programs are built to handle some task, and most of those tasks require the program logic to make decisions about what should be done next. The process of determining what instructions the program should execute next is one of the tasks we work hardest on as we write our code.

A Big Example

Imagine a situation where we have a website that requires users to be logged in and have a verified email address in order to use the features the site provides. This is a common case, and requires the application powering the website to determine a few things:

- Is the user currently logged in?

This seems like an easy question, and on one level it is: The user's login state could be represented with a Boolean (`true` or `false`). But in order to determine that, we've already decided at the very beginning of our application that we need to check if the user is logged in.

Let's imagine the user is not currently logged in. What next? We might move the user to a login page so they can login. Now we would encounter even more branching logic:

- Does the user have an account?
    - If so, did the user provide the required credentials to login?
        - If yes, then did the user verify their email?
            - If they *did* verify their email, then we can finally move them into the app.
            - If they did *not yet* verify their email, then we need to move into the email verification logic.
    - If the credentials are bad, redirect the user to an account recovery screen.

But all of that is assuming the user has an account. What if that's not the case? If the user has not yet created an account, then we know we must go through the account registration *and* the email verification process before allowing the user to access the website.

A Smaller Example

All of the decisions about big picture things like "user registration and verification" can be overwhelming to think about for novice developers. There are a lot of parts to the puzzle, and depending on the business logic (the rules and policies governing business operations) they can be more or less complicated. However, we encounter the need for logical flow control in many smaller ways, too.

Imagine that we're writing a page that fetches the posts from a blog. The data might be retrieved from a database or API, but once it has been fetched it's up to our application code to process the information into something that can be read by a user. In order to do this, we must properly insert the data into an HTML template for processing by the web browser.

We typically use a loop to process sets of data. Within the loop we might even need to do some checking in order to determine different aspects of the presentation, but in general we would be controlling our logical flow sort of like this:

- Got data. Begin processing.
- Process output of an item.
- Is there another item?
    - If so, process the next item.
    - If not, stop looping and move on to next section of code.

This kind of processing is very common and used in large and small ways. Within a loop to present content items there might be smaller loops to, perhaps, display each tag associated with the content, or to list each author on the content.

Controlling logical flow through different parts of our programs is an essential part of development. In this section we will explore the concepts of conditionals and loops to control logical flow.

Conditionals

Conditionals are sets of code instructions that execute if a given assertion is `true`. These are commonly expressed as "if/then" statements, and they are essential to controlling logical flow.

Basic Conditional Syntax

We often call conditionals "if statements" because in most programming languages the keyword "if" is used. In JavaScript, we write a conditional statement like so:

```
if (true) {
    // code here executes when the condition is true
    console.log('It's true!');
}
```

This is a simple statement that checks if an assertion is true. In this case, since the assertion is simply `true` it will always execute the code. But we can do more with conditionals. Here is a more complex example:

```
let foo = 12;
let bar = 42;

if (foo > bar ) {
    // In this example, this `foo` is less than `bar`, so this code will not execute.
    console.log('Foo is bigger.');
} else {
    // Since `bar` is greater than `foo`, this code will execute.
    console.log('Bar is bigger.');
}
```

The `else` clause is used to provide instructions for what to do if the initial assertion is not `true`. For very complex conditionals, it's possible to provide multiple assertions that can be checked:

```
let foo = 12;
let bar = 12;

if (foo > bar ) {
    // If `foo` is greater than `bar`, then this code executes.
    console.log('Foo is bigger.');
} else if (foo == bar) {
    // If `foo` and `bar` are equal, then this code executes.
    console.log('Foo and Bar are equal.');
} else {
    // If `bar` is greater than `foo`, this code will execute.
    console.log('Bar is bigger.');
}
```

In the example above, the `else if` clause provides a second assertion that is checked. If the first assertion is `true`, then the second assertion will never be checked. But if the first assertion is `false`, then the second assertion will be checked. If the second assertion is `true`, then that code executes. If the second assertion is also `false`, then the `else` clause will be executed.

Of course, it is possible to nest conditionals to create even more complex conditionals:

```
let foo = 12;
let bar = 42;
let baz = 7;

if (foo > bar ) {
    console.log('Foo is bigger than bar.');
    if (foo > baz) {
        console.log('Foo is bigger than baz, so it is the biggest!);
    } else {
        console.log('Baz is bigger than foo, so it is the biggest!);
    }
} else if (foo == bar) {
    // If `foo` and `bar` are equal, then this code executes.
    console.log('Foo and Bar are equal.');
    if (foo == baz) {
        console.log('Foo, bar, and baz are all equal.');
    } else if (foo > baz) {
        console.log('Foo and bar are larger than baz.');
    } else {
        console.log('Foo and bar are smaller than baz.');
    }
} else {
    // If `bar` is greater than `foo`, this code will execute.
    console.log('Bar is bigger than foo.');
    if (bar > baz) {
        console.log('Bar is bigger than baz, so it is the biggest!);
    } else {
        console.log('Baz is bigger than bar, so it is the biggest!);
    }
}
```

As you can see in the example above, we can nest conditionals in all kinds of ways to achieve whatever logical control we need.

Comparison Operators

In order to make assertions that conditional statements can check, we must use a few more operators. We are often making comparisons between values, so the comparison operators are important. We can see comparison operators at work in the examples above, but there are more than those. Here is a chart of commonly used comparison operators:

operator	description	example
`==`	**Equal**: Returns `true` if the operands on either side of the operator are equal.	`x == y`
`!=`	**Not Equal**: Returns `true` if the operands on either side of the operator are *not* equal.	`x != y`
`===`	**Strict Equal**: Returns `true` if the operands are equal and of the same type.	`x === 2`
`!==`	**Strict Not Equal**: Returns `true` if the operands are of the same type but not equal, or are of different type.	`x !== 'text'`
`>`	**Greater Than**: Returns `true` if the left operand is greater than the right operand.	`x > 42`
`>=`	**Greater Than or Equal To**: Returns `true` if the left operand is greater than or equal to the right operand.	`foo >= 32`
`<`	**Less Than**: Returns `true` if the left operand is less than the right operand.	`y < 1337`
`<=`	**Less Than or Equal To**: Returns `true` if the left operand is less than or equal to the right operand.	`bar <= 42`

Using these comparison operators we can compare the values of any two variables.

Logical Operators

Sometimes when we want to check especially complex conditionals, it's useful to utilize logical operators to make compound assertions. There are three main logical operators:

operator	description	example
`&&`	**AND**: Returns `true` if BOTH operands equal `true`.	`x == 12 && y == 42`
||	**OR**: Returns `true` if EITHER operand equals `true`.	`x > 3 || y < 31`
`!`	**NOT**: Returns the opposite of whatever the operand evaluates to be. That is `false` if the operand equals `true`, or `true` if the operand equals `false`.	`!foo`

These three logical operators round out our ability to check conditions between operands. `AND` and `OR` are especially useful for making compound assertions. We can use them in a conditional like this:

```
let x = 12;
let y = 42;

if (x==12 && y==42){
    // This code would execute because both `x==12` and `y==42` are `true`.
    console.log('This is true!');
}

if (x==144 && y==42){
    // This code would not execute because one side of the assertion is `false`.
    // The `AND` operator requires both sides to evaluate to `true`.
    console.log('This is true!');
}

if (x==36 || y==42) {
    // This code would execute because although `x==36` is `false`, the other side of
the assertion is true.
    // This happens because of the `OR` operator.
    console.log('One of these is true!');
}
```

Although it can be tricky if we string together several `AND` or `OR` clauses in an assertion, these example keep things simple. It's often best when writing code to keep things simple and break out complex logic into multiple conditionals.

It can sometimes be tougher to understand the `NOT` operator because it equals `true` when the value is `false`. This can be a bit of a head trip. Here is an example using `NOT`:

```
let loggedIn = false;

if (!loggedIn) {
    // Since `loggedIn` is `false`, this code will execute.
    console.log('This user is not logged in!');
}
```

Truthiness and Falsiness

Just like with Data Types, JavaScript is more than happy to attempt to arrive at a `true` or `false` value based on whatever we give it. Sometimes this can be very convenient. In JavaScript we have a notion of "truthy" and "falsy".

The following values all evaluate to `true` in a conditional:

```
if (true)
if (12)
if ("some text")
if ({})
if ([])
if (3.14)
```

Likewise, the following values all evaluate to `false` in a conditional:

```
if (false)
if (null)
if (undefined)
if (0)
if (NaN)
if ('')
if ("")
```

Using the principles of truthy and falsy, we could write a conditional like this:

```
let x;

if(x) {
    // This code would execute if `x` has any value assigned other than `null` or `und
efined`.
    // Since `x` is currently equal to `undefined`, this code will NOT execute.
} else {
    // This code will execute.
    console.log('X exists, but it has no value.');
}
```

Of course, when possible it's good to check explicitly against an expected value. If we expect `x` to be `undefined` then we should check for that (`if (x===undefined)`). But sometimes that is not possible. Given the dynamic typing of JavaScript and the complex nature of programs, we might find a situation where we are not sure if a value might be `undefined` or `null`, but we want to catch that condition regardless. Or, we might consider `0`, and `false` to be equivalent for some purpose in our code. In this case, it is helpful to be able to perform truthy and falsy checks.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Write a conditional to check if `x` and `y` are equal. If so, set `status` to "success". If not, set `status` to "failure".

```
let x = 12;
let y = 42;
let status;
```

Exercise

Write a conditional to check if `x` OR `y` are true. If so, set `status` to "success".

```
let x = true;
let y = false;
let status;
```

Exercise

Write a conditional to check if `x` AND `y` are true. If so, set `status` to "success". If not, set `status` to "failure".

```
let x = true;
let y = false;
let status;
```

For Loops

The most common kind of loop in any programming language is probably the `for` loop. This is a structure that allows us to define a clear number of times to loop through some set of instructions. The `for` loop can be used anytime we need to repeat a set of instructions for a specific number of iterations.

Here is an example of a `for` loop in JavaScript:

```
for(let i=0; i<10; i++) {
    console.log(`This is iteration number ${i}.`);
}
```

The `for` loop structure is not too complex, but it does require a specific syntax to make it happen. First, it identifies as a `for` loop with the keyword `for`. Then we list three statements in the top line of the loop:

1. **Initializer:** The first statement initializes a counter variable to a value. (This could be any variable name, but the letter `i` for "iteration" is often used.) This is the value where the counter will begin.
2. **Exit Condition:** The second statement provides a conditional assertion that is checked on each iteration of the loop. Each time the loop executes, it checks this assertion to see if it's `true`. If so, the loop continues. If not, the loop stops.
3. **Final Expression:** third statement increments the value of the counter by one. This causes the value of the counter to increase, so eventually the second statement will return `false`.

Here is another example of how a `for` loop can function:

```
let simpsons = ['Homer', 'Marge', 'Bart', 'Lisa', 'Maggie'];

for (let i=0; i<simpsons.length; i++){
    console.log(simpsons[i]);
}
```

Notice that in this example we are actually using the `i` variable (which is the counter for the `for` loop) as the index for which name we want to output. This code would output each of the names to the JavaScript console in the web browser.

Looping Arrays in the Old Days

It used to be necessary to use the standard form of the `for` loop through Arrays. To do this, we would use the `Array.length` property to set the exit condition for the loop. This method still works, and it's used in some examples on this page. But pay attention to the more modern way of looping arrays over the next couple of sections of the book.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Define a `for` loop that will double the variable `x` for 12 loops.

```
let x = 2;
```

While Loops

The other common kind of loop in programming languages is the `while` loop. These loops are defined with an opening assertion, and they execute as long as the condition remains true. Here is an example:

```
let i = 0;
while (i < 12) {
    console.log(`This is loop iteration number ${i}`);
    i++;
}
```

In this case, the `while` loop will execute 12 times, and it will output a message to the web browser's JavaScript console on each loop.

The `while` loop provides a way to keep executing some set of instructions until a condition changes. This can be powerful, but it can also be dangerous. It's very easy to write an infinite while loop:

```
while (true) {
    console.log('This is an infinite loop and will lock up your browser tab if you try
 to run it. (Don\'t do that!)');
}
```

It might seem insane to have such an easy way to write an infinite loop that could lock up a browser tab, but if used properly, it could work. Imagine we have a system that has a `checkValue()` function that would give us back a value we are watching for:

```
while (true) {
    status = checkValue();
    if (status.success == true) {
        break;
    }
}
```

In this example, we set up an infinite loop, but once a status is received and checked, it can stop execution using the `break` command. The `break` command can be used to stop any loop from executing and return to the execution of code outside the loop. (Please note: This is still probably not the best structure for a program, but it works by way of example.)

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Define a `while` loop that will add 1 to a number until it equals 42.

```
let x = 0;
```

Looping Arrays

The most common use for a loop in JavaScript is to loop over values in an Array. Arrays are meant for storing lists of items, and we often have a need to display those list items in different ways. Arrays are a great way to store and access this information, especially in modern JavaScript.

As we read in the `for` loop section, we can loop through Arrays in a very "manual" way by using a standard `for` loop. However, Arrays (and other iterable objects in JavaScript) can also be accessed using the `for ... of` syntax.

Let's look at the example we saw on the `for` loop page:

```
let simpsons = ['Homer', 'Marge', 'Bart', 'Lisa', 'Maggie'];

for (let i=0; i<simpsons.length; i++){
    console.log(simpsons[i]);
}
```

The above code shows the old way of accessing items in an Array. Here is the new (and preferred) approach:

```
let simpsons = ['Homer', 'Marge', 'Bart', 'Lisa', 'Maggie'];

for (let familyMember of simpsons){
    console.log(familyMember);
}
```

As we can see, using the `for ... of` syntax allows us to eliminate the need for a counter. Every item in the Array comes through and is accessible within the loop using the named variable we specify at the beginning of the loop (in this case, that variable is named `familyMember`).

The syntax to declare this kind of loop is more simple than the standard `for` loop syntax. We only have to define a name for each item in the Array to take on as it moves through the loop. Everything else is handled for us by JavaScript, which knows how many items are in the Array and can figure out how many times to execute the loop.

This pattern exists in many other programming languages, so getting used to the way this works is helpful even outside of JavaScript. Many templating languages, for example, have a `foreach` or similar kind of loop that can be executed.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Write a `for ... of` loop to add all the numbers in the `transactions` Array.

```
let transactions = [23, 44, 7, 89];
let total = 0;
```

Quiz: Controlling Logical Flow

Try this self-check quiz to test your knowledge!

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Extra Practice

Remember: Learning to program takes practice! It helps to see concepts over and over, and it's always good to try things more than once. We learn much more the second time we do something. Use the exercises and additional self-checks below to practice.

1. Looping and Conditionals

```
let maxNum = 20;

for (let i=1; i<=maxNum; i++) {
    if (i % 2 === 0) {
        console.log(`${i} is even.`;
    } else {
        console.log(`${i} is odd.`);
    }
}
```

Visit Content Online

The content on this page has been removed from your PDF or ebook format. You may view the content by visiting this book online.

Organizing Code

So far we have looked at code that represents basically step-by-step instructions. Loops provide a small way to re-use the same instructions, but for the most part we've focused in linear instructions where each line executes once. This is the base of programming, but it is not the most efficient way to write programs.

All programming languages allow for us to create groupings of code that can be executed on-demand. This means that we can write instructions that can be called to execute at various times in our program. By organizing our code into re-usable blocks, we can write more efficient code that is more easily maintained. Most languages call the fundamental form of this code grouping "functions" or "subroutines".

In JavaScript, we often use "functions" to accomplish this goal. The JavaScript function is a powerful way to label and contain a block of instructions that can be accessed from throughout the program. Functions operate as small programs within the larger program.

Impact on Code Organization

Functions allow us to organize code in more logical ways. The main body of our program becomes a set of higher-level instructions about what happens at runtime, and then each function will define more detailed instructions for each of those steps. We can group logic into well-named functions in order to provide a more clear presentation of our code for ourselves and other developers to read.

It's not unusual to have a function for each step of a process such as `login()` or `logout()`. Those two functions could be accessed from several places, but we never need to duplicate the lines of code within those functions. This means that if I were a developer who wanted to change something in the login process, I could look for the `login()` function and focus my attention there. There is no need to read through the entire program in order to find the lines that make logins work.

This also allows us to group logic in such a way that we can better account for missing data or logic that should not be run unless certain situations arise. It would be impossible to have a direct execution of code that both logs users in and logs users out. We need to execute those instructions at very specific times. Without functions we would be unable to manage even the most common of cases in our web applications.

Impact on Maintenance and Enhancements

As mentioned before, functions help us eliminate duplication within our code and make it more clear what blocks of code actually do. This has a huge impact on both long-term maintenance and building enhancements. If the API provider for ecommerce transactions changes their API, then we only need to change the functions that handle the transactions, and we can easily identify where that code lives based on names and grouping. Without proper organization, code becomes like "spaghetti": We can't tell where one feature starts or ends, and we can't change one part without affecting all the stuff jumbled around it. We seek to make our code more like a well-organized shelving unit: Parts that go together live together, and we can easily access one part of our code without jostling around everything else.

By using functions to help us organize code, we make all of our development tasks easier and we increase the reliability of our application for our end users.

Writing DRY Code

A popular axiom among developers is to "Keep it DRY," or to "Write DRY code." In these sayings, developers are referring to the acronym D-R-Y, which stands for "Don't Repeat Yourself". The label stems from the book *The Pragmatic Programmer* (1999) by Andy Hunt and Dave Thomas, in which they write, "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." This directive gets at the problem of creating confusing, redundant code or data systems that become ambiguous and where logic is tangled together.

Ideally, we could change any one part of our programs without having to change every other part. This is the core of the DRY principle. This is also related to the "separation of concerns" that we discuss occasionally. Each feature of a program should be as self-contained as possible, and we should strive to build clean points of integration to allow our different pieces to work together effectively.

Functions

As mentioned previously, functions are subroutines that can be executed whenever they are needed in our code. They are self-contained blocks of code, so they are convenient for creating "tools" or "components" that do just one thing. If we make a function that performs the "break apart a sentence into individual words and put them in an array" task, then anytime we need text capitalized in our program we can execute that function. When we execute a function's code, we use the term "call": We "call" a function, which often "returns" a result. This is how we verbalize using functions in our programs. Let's take a look at how to use them.

Function Syntax

Functions are defined using the `function` command. This command takes a *name*, a set of *arguments*, and then defines a *code block* that will be executed when the function is called. A standard function definition looks like this:

```
function name(parameter){
    // this code executes
}
```

That's the basic form of a function. We call the top line of the function the "declaration" or "signature". In the example above, we have just labelled the parts. Here is an example that actually does something:

```
function getWords(text){
    return text.split(' ');
}
```

The function above is called `getWords()`. When we write about functions, we tend to add the parentheses to indicate that the thing we're referring to is a callable function. `getWords()` takes a string of text and returns an Array of all the words in that text, separated using the space character (`' '`). The `name` of the function is `getWords`. To execute the function we would write the `name` with parentheses. Since this function also requires an `argument` (the variable specified between the parentheses after the `name` of the function), we could specify some text when we call it. The code between the curly braces is executed whenever the function is called. In this case, it is simple code, but this code could contain as many instructions as needed, and it could call other functions.

Here is what it might look like to use this function in a small program:

```
function getWords(text){
    return text.split(' ');
}

let words = getWords("I've always been more interested in the Future than the Past.");

if (Array.isArray(words)){
    console.log(`words is an Array with length ${words.length}.`);
    for (word of words){
        console.log(word);
    }
}
```

Function Parameters

Functions can accept parameters that are specified in the function declaration. These are named values that become variables available within the code block of the function (between the curly braces). We can write functions to generically reference the names of their parameters, and then when we call the functions in code we can send any values into those parameter "slots".

In the example above, the function `getWords()` accepted one parameter: the `text` value. This value is given the name `text` within the function, so we are able to write the code referencing `text` and fulfill our requirements. Here is another example:

```
function calculateArea(length, width, measure) {
    let area = length * width;
    return `${area} sq ${measure}`;
}

let boxArea = calculateArea(4,12,'feet');
console.log(boxArea);
// "48 sq feet"
```

In this example, we have a function that calculates the area of a rectangle. This formula for doing the area calculation is: `length * width`. We have written the function to accept `length`, then `width`, which we expect to be numbers. The `calculateArea()` function also accepts a `measure` parameter, which allows for a nicer response to be crafted. Calling `calculateArea()` gives us a nicely formatted statement about the area results.

It can be difficult to know exactly what Data Types a function wants, especially if the parameters are poorly named. To remedy this problem, it's customary to make some comment about what the function expects and what it returns:

```
function calculateArea(length, width, measure) {
    // Calculates area of rectangle.
    // params:
    //    length - integer
    //    width - integer
    //    measure - string
    // returns:
    //    string

    let area = length * width;
    return `${area} sq ${measure}`;
}

let boxArea = calculateArea(4,12,'feet');
console.log(boxArea);
// "48 sq feet"
```

As we can see, this is effective at conveying the Data Types of parameters, but it takes up a lot of space in the code and is somewhat redundant. Another way we can convey this information, and gain a couple other benefits is to use Default Parameters.

Default Parameters

Default parameters are parameters that are written with a default value. This allows us to better convey what Data Type each parameter should be. It's also handy to write functions that can execute with a default set of parameters most of the time, but whose parameters can be modified to fit special cases.

Here is an example of an updated `getWords()` function that uses default parameters:

```
function getWords(text="Demo text here.", splitter=" "){
    return text.split(splitter);
}

let words = getWords();

if (Array.isArray(words)){
    console.log(`words is an Array with length ${words.length}.`);
    for (word of words){
        console.log(word);
    }
}
```

Notice that the function declaration has been updated to provide default values for each parameter. We can now just call `getWords()` and supply no text. In that case, it would process the default text string ("Demo text here."). The function is also more versatile because it uses a `splitter` param that can be changed. In most cases, we probably want to split sentences apart using spaces to get the words. For those cases, it's no more difficult to use the function.

But in cases where we want to split text apart using a different character, we can now use our same `getWords()` function:

```
let words = getWords("bread,milk,lunch meat", ",");

if (Array.isArray(words)){
    console.log(`words is an Array with length ${words.length}.`);
    for (word of words){
        console.log(word);
    }
}
```

In this second example, we are sending in a list of comma-separated words and using the comma to perform the split. As we can see if we run the code in our JavaScript console, the modified `getWords()` function handles this special case with no trouble!

Function Arguments

All functions also have access to an `arguments` object. Arguments are more advanced features of JavaScript that we are not going to cover in-depth here, but they can be useful for adding extra parameters to your function calls or in situations where you can't know all of the things you will need to process (where you need to accept an arbitrary list of items, for example). Arguments can be confusing because they are not explicitly named in the function declaration. Here is an example:

```
function makeTextList(){
    args = Array.from(arguments);
    return args.join(',');
}

let shoppingList = makeTextList('bread','milk','lunch meat');
console.log(shoppingList);
// "bread,milk,lunch meat"
```

When using arguments in a function, keep in mind that the `arguments` object is not a standard Array, but we can use the `Array.from()` function to transform the `arguments` object into a standard Array. There are many more great ways to use arguments, but they are a bit beyond where we are right now.

Returning Data From Functions

Once we've completed some calculation or processing in a function, we often want to send some data back to the main logic of our program. To do this, we use the `return` statement.

In each example above we've seen how we can return data to whatever command called the function. Once that data has been returned, it can then be used by subsequent instructions. This is a very common pattern for using functions in code. Here is an example:

```
function calculateTax(amount = 9.42, tax = 0.065){
    return amount * (1 + tax);
}

let subtotal = 12.57;
let total = calculateTax(subtotal);
```

In this example, we have a simple function that calculates tax. We can use this function whenever we need it, and it helps us generate the value for the `total` variable. The `total` variable might be used in many places: in a template to render it for the user, stored in a database, emailed to an email receipt, etc.

In the case above, the `return` statement is returning the results of the calculation directly. It would be possible to store those results and return the variable where they were stored like this:

```
function calculateTax(amount = 9.42, tax = 0.065){
    let taxedAmount = amount * (1 + tax);
    return taxedAmount;
}

let subtotal = 12.57;
let total = calculateTax(subtotal);
```

We could even be building up a much more complex Object or Array of values that could be returned and stored in the `total` variable. Sometimes this is needed when we have to track additional metadata (such as what tax rate was applied, or what time the transaction was processed).

Using functions to process information and return data is a powerful tool for building complex programs.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Define a function that would uppercase a String. You can use the `String.toUpperCase()` to capitalize a String. An example is shown in the starter code. Package that example into a function that will take a parameter called `text` and return the uppercase version of it.

```
// Example of using `String.toUppercase()`
let example = "abcd";
console.log("Uppercased result: " + example.toUpperCase());

// Function goes here
var myText = capitalizeText("http");
```

Exercise

Define a function that utilizes default parameters to produce the default message "Hello, world!"

```
// Function goes here
var greeting = sayHello();
```

Scope and Scoping

Some commands we've used throughout this book without really explaining why are the `var` and `let` commands, which are used to declare a variable. We know they indicate that a new variable has been created, but we have not yet discussed how these commands are different, or what those differences mean for our code. In this section we will cover these differences and how to use them in a program.

Scope

In programming languages there is generally a notion of "scope". The scope is a term used to describe the current context of the code: all of the variables, functions, objects, etc. that are known to the programming language interpreter at the time of execution. Take the following snippet of JavaScript as an example:

```
var foo = "Hello, world!";
var bar = 42;

function baz(){
    console.log("The question of whether a computer can think is no more interesting t
han the question of whether a submarine can swim.");
}

var quote = baz();
console.log(`Foo: ${foo}`);
console.log(`Bar: ${bar}`);
console.log(`Quote: ${quote}`);
```

In the example above, the scope of this script includes all of the globally available objects available in JavaScript (objects like `Number` and `Object` and `for` and `if`, to name a few). The scope also includes three code objects that we have defined: `foo`, `bar`, and `baz()`. Although `foo` is a String, `bar` is a Number, and `baz()` is a Function, each of them is "known" to the JavaScript interpreter as it executes these lines. When we set `quote` equal to `baz()`, the interpreter understands which function we are calling because everything is accessible within the same scope.

There are, however, situations where variables or functions may not be known to all scopes. Sometimes we want to make variables exclusive to certain parts of our code. Each time we declare a variable, we must make a choice about whether we want that variable to be "global" or "local".

Global Variables: `var`

In ECMAScript 6 (the latest standard for JavaScript), there are two types of variables: Global and local. Global variables in JavaScript are declared with the `var` command:

```
function echoNum(){
    myNum = 42
    console.log(`myNum: ${myNum}`);
}

echoNum();
var foo = myNum;
console.log(`foo: ${foo}`);
```

In the code above, `myNum` is declared as a global variable within the `echoNum()` function. This means that we can reference `myNum` from outside the function where it is declared (and assign its value to `foo`).

The `var` command used to be the standard way of declaring a variable in a JavaScript program. There is a distinction between variables that are truly global (declared with no `var`, `let`, or `const` command) and variables declared with `var`, but we won't be getting into that at this point. In most cases, `var` effectively creates a variable that is usable by all parts of your program. **NOTE:** At this point in JavaScript history, it's preferred to use local variables declared with `let` (see below) over global variables or variables declared with `var`.

Variables declared with the `var` command should be treated as global variables (they aren't *exactly* global variables, but for now that's the safest way to treat them), which means their usage should be limited. In general we try to declare as few global variables as possible. This is for a few reasons, but primary among them are these three:

1. Global variables take up a name that cannot be used anywhere else in our code. Having global variables with generic names like `counter` or `item` would be difficult to accommodate, since we often want to use those words in different contexts and not have our data or operations get mixed up.
2. The larger the scope becomes for any given piece of code, the longer it takes to interpret. By controlling how many code objects we load into our scope we can make our code execute more quickly.
3. The more objects we have floating around in our global scope, the more chance there is for error. Operations can get mixed up. Data can get mangled. Generally "bad craziness" might ensue.

Local Variables: `let`

In most of our coding tasks, we attempt to keep our variables and functions local to small pieces of code. The `let` command allows us to declare a variable that only exists in the code block where it is declared and any code blocks contained within (children of the original code block). Here's an example:

```
function echoNum(){
    let myNum = 42;
    console.log(`myNum can be referenced inside the function: ${myNum}`);
    if (myNum === 42){
        console.log(`myNum can be referenced in this conditional: ${myNum}`);
        let foo = "Even more local.";
        console.log(`foo has been declared and scoped to exist only within this condit
ional: ${foo}`);
    }
    console.log(`foo cannot be referenced outside of the conditional where it is decla
red: ${foo}`);
}
console.log(`myNum CANNOT be referenced outside the function: ${myNum}`);
```

In this example, we see that `myNum` is declared inside of `echoNum()`, but it is declared with the `let` command. This means that `myNum` only exists inside of `echoNum()` and any code blocks contained within `echoNum()`. The conditional inside of `echoNum()` is another code block (indicated by the curly braces), and it can reference `myNum`. Within the conditional codeblock the variable `foo` is declared. This variable only exists within the conditional statement. It cannot be referenced outside of the conditional at all.

This example shows several ways in which `let` restricts the availability of local variables. Local variables are available inside their own scope (code block) and any "child" scopes (code blocks).

It's useful to be able to make local variables with `let` because we often need to make quick variables that perform similar tasks within different code blocks. For example, we may wish to use variables to control a loop (e.g. `counter`, `index`, `item`) or to help with a conditional (e.g. `isSaved`, `isDirty`) and not have those values persist outside of their specific code block. We could use these variable names in multiple contexts, so keeping them local to each one is crucial.

Hoisting in JavaScript

A somewhat unique quality of JavaScript is the notion of "hoisting". A variable declared with `var` or function may be used before it is declared, as long as it is declared somewhere later in the code. This can lead to situations where a variable may be referenced or a function may be invoked on a line preceding the actual declaration of that variable or function. In general, it's best to avoid reliance on this quality because it can lead to writing confusing code that is difficult for other developers to read and understand.

This is not an actual feature of JavaScript; it is a byproduct of the way that JavaScript is interpreted and executed. When a JavaScript program is interpreted, the code is reviewed and all objects that are defined are kept in memory. This allows the interpreter to review the code, find the important objects, and then understand references to those objects, even if the references in the code are made on lines preceding the object declarations. Although it's a reliable effect of interpretation, it's still not recommended to make a habit of using functions or variables before you have declared them. In fact, the standard practice is to declare all variables at the beginning of a code block or file, and to define functions in some location within our code that makes sense.

Quiz: Organizing Code

Try this self-check quiz to test your knowledge!

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Extra Practice

Remember: Learning to program takes practice! It helps to see concepts over and over, and it's always good to try things more than once. We learn much more the second time we do something. Use the exercises and additional self-checks below to practice.

1. Multiplying Two Numbers

Review the code, then answer the questions below.

```
function multiplyNumbers(firstNumber=2, secondNumber=4){
    let product = firstNumber * secondNumber;
    return product;
}

let foo = multiplyNumbers(3,12);

let bar = multiplyNumbers(15,4);

let baz = multiplyNumbers("two","three");
```

2. Calculating Area of a Box

```
function calculateArea(length, width, units="sq ft"){
    let area = length * width;
    return `${area} ${units}`;
}

let boxLength = 5;
let boxHeight = 12;

let boxArea = calculateArea(boxLength, boxHeight);
```

3. Scoping Variables

```
let myText,
    processedText;

const stopWords = { // stopWords is an Object defining words that should not be capita
lized.
    'of': true,
    'the': true,
    'a': true,
    'an': true,
    'and': true,
    'to': true
}


function capitalizeText(text){
    // This function expects a String of text. It breaks apart the String into an
    // Array of words, then it capitalizes the first letter of each word. Once it has
    // capitalized all words, it returns a new String with capitalized text.

    let wordArray = text.split(' '); // Split on space characters.
    let newWordArray = []; // Initialize an Array to store results.

    for (word of wordArray){
        if (!stopWords[word]){ // Check to make sure word is not in stopwords list.
            let newFirstLetter = word[0].toUpperCase(); // Change first letter in `wor
d` to uppercase.
            let slicedWord = word.slice(1); // Get the rest of the word after the firs
t letter.
            let newWord = newFirstLetter + slicedWord; // Put the word back together.
            newWordArray.push(newWord); // Add the newWord to the newWordArray of capi
talized words.
        } else {
            newWordArray.push(word); // If we hit a stopWord, just put that word back
in the list without altering.
        }
    }

    var newText = newWordArray.join(' ');
    return newText; // Return the string.
}

myText = "association of code writers";
processedText = capitalizeText(myText);
console.log(processedText);
```

Visit Content Online

The content on this page has been removed from your PDF or ebook format. You may view the content by visiting this book online.

Object Oriented JavaScript

Prior to ECMAScript 6, JavaScript supported "object oriented"-like functionality, but it was difficult and clunky to emulate a real object oriented approach to writing code. For that reason, JavaScript mostly used functions and events to chain together features, which worked well enough but caused some consternation for developers used to operating in an object oriented context.

Some of the most popular languages in the world are object oriented programming languages: C++, Java, Python, Objective C, C#, etc. Although there are other popular programming paradigms, object oriented programming (often abbreviated as OOP) has been a major force in software development for several decades. It is a set of core principles and design concepts that has proven to be a useful way of thinking about information and functionality.

In this section we will learn more about OOP concepts and how we can use those concepts in our code. Keep in mind that the fundamental concepts behind OOP are not exclusive to JavaScript, so it is useful to know these concepts even beyond the realm of our browser-based code.

Prototype: Under the Hood

In JavaScript, the way that Objects are implemented is with the "prototype mechanism". The `Object.prototype` attribute points to a template for each specific kind of Object in the system. This `prototype` is inherited by any Objects derived from a given template (for example: Number, Boolean, Date, etc.). The `prototype` provides the model for all the functionality these different Object types afford us.

We can create new Objects and then use them to create instances of those Objects, as described in object oriented programming theory, but the way this is implemented in JavaScript is via the `Object.prototype` mechanism. The `prototype` of any Object can be modified and changed, and the `prototype` continues to exist in spite of the new ECMAScript 6 syntax that makes it easier to write and use more conventional Classes in JavaScript. We do not dive deep in the `Object.prototype` in this book, but it is there to explore as we discover more sophisticated cases to solve.

What is Object Oriented Programming?

At this point it should be clear that when we are programming we are primarily concerned with two things:

1. Representing what we know about the world (data) in a way that can be understood and manipulated to meet our needs.
2. Representing actions we can perform on data in a way that can be understood and used effectively.

In order for this to work, we must make many conceptual leaps: We need to figure out how to describe facts about the world in text. We must be able to represent measurements and the complexity of relationships using symbolic expressions. We want to create logical connections between parts of our programs that are both functional for the computing environment and understandable by human beings.

Because of these and many other requirements of writing software, we have developed the notion of Object Oriented Programming (OOP) to effectively describe and execute both data and functionality. OOP allows us to define "Classes" of objects that possess both "properties" and "methods". Object Classes allow us to describe objects and concepts in great detail; we can think of them as the "nouns" of OOP. Properties can be thought of as "adjectives"; they describe the details of the Class. The methods are the "verbs" of OOP; they *do* things, executing functionality, manipulating data, or providing some kind of input/output.

An example of a Class in JavaScript might look like this:

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  calcArea() {
    return this.height * this.width;
  }
}
```

In the example above, a Class is created called `Rectangle`. This Class has two properties: `height` and `width`. Those properties *describe* the rectangle. It also has a method called `calcArea()` that can be invoked in order to return the area of the rectangle. This method allows the Class to *do* something. This Class models the concept of a rectangle in a fairly thorough way. We could use it to represent rectangles in our code as much as we needed.

Of course the basic structure of the Class is just the beginning of how OOP works. OOP is ruled by four fundamental concepts that help us understand how to use Classes most effectively.

Abstraction

Abstraction is the idea that we want to use our Class objects to take away complexity as much as possible. Abstraction is part of our daily lives. We know how to drive a car: turn it on, put it in gear, press the gas or the brake as needed. We do not need to understand precisely how each of these actions is performed by the electromechanical systems of the car. We only need to understand the interface for the car. Once we learn the interface, we can make effective use of the car without learning how to build a car from scratch.

The notion of defining and using an interface for code we have written ourselves is actually one we've already gotten used to. We use abstraction anytime we organize code into functions or subroutines: We wrap up more complex logic and instructions into a simple command we can issue wherever we need it. Imagine the example below:

```
function calculateTax(subtotal){
  // Perform whatever complex logic needs to happen to determine `localTaxRate`
  let localTaxRate = 0.065;
  return subtotal * (1 + localTaxRate);
}

let total = calculateTax(subtotal);
```

In this example we can see that a function called `calculateTax()` has been created to handle the calculation of the tax. There could be any number of instructions or lines of logic used to determine the applicable tax rate, but we don't have to think about all of that code when we use the function to figure out the total cost of a transaction. We have "abstracted" the logic used to calculate the tax rate, and the only thing we need to remember as developers is to call the function and supply the subtotal.

We often define Class objects that have very complex code inside them. They might perform communication with remote systems, calculations beyond our limited Math skills, or other functions that require precision and many steps. By abstracting the complexity of these features way to simple interfaces we are able to define conceptual tools in our code that are easy and intuitive to use.

Encapsulation

Encapsulation is the idea that not all parts of a Class object should be available for public manipulation. By "hiding" or making certain data or functions unavailable outside of the Class object, we can make sure that those components are not altered by some unforeseen outside force. We do not want to accidentally change a value or execute a function that might cause our entire system to throw an error. Where possible, it is better to allow as few publicly available properties and methods as possible because it presents a cleaner interface to the Class object.

Encapsulation is supported in JavaScript through the `prototype` mechanism, but it is not easily achieved with strong enforcement and low effort. Instead, most developers have adopted the convention of prefixing any "private" property or method of a Class object with an underscore (`_`). This at least conveys to developers which properties and methods are not intended for direct use. When we encounter properties or methods named with an underscore, it's best to avoid using them directly. Thre should be a "public" property or method that provides an equivalent result.

Inheritance

Class objects are often hierarchical. This matches the way we have organized and categorized objects in the real world. For example, we can organize living things into "Plants" and "Animals", the Kingdoms of biological taxonomy. Within each of these Kingdoms are many sub-groupings of life forms that traverse organizational hierarchies like Phylum, Family, Genus, and Species. It is not unusual for Class objects in a system to incorporate a similar hierarchy.

For example, imagine a general Class object for `Animal` :

```
class Animal {
  constructor(name){
    this.name = name;
  }
}
```

The base class for `Animal` has only one property: `name` . If we wish to make a class to represent a `Dog` then we could inherit from the `Animal` class and extend its functionality:

```
class Animal {
    constructor(name){
        this.name = name;
    }
}
class Dog extends Animal {
    speak(){
        console.log(`${this.name} says, 'Woof!'`);
    }
}
let d = new Dog('Fido');
d.speak() // Outputs `Fido says, 'Woof!'` to the JS console.
```

In this example, we can see that the class `Dog` has been created by *extending* the class `Animal`. The `Dog` class does not need to re-declare the `constructor()` method because it has *inherited* this method. But the `Dog` class can *add* to the definition of the class. In this case, the `Dog` class extends the `Animal` class by adding a `speak()` method.

Polymorphism

Although building on the foundations of other Class objects in a hierarchical way is a powerful tool for modeling complex relationships and/or objects, we require one more ability in order to truly make efficient use of the Class objects we have defined. Polymorphism is the ability to *change* the things that have been inherited from one class to another. Here is an example:

```
class Animal {
  constructor(name){
    this.name = name;
  }
  speak(){
    console.log(`${this.name} makes a sound.`);
  }
}
let a = new Animal('Corvo');
a.speak(); // Outputs `Corvo makes a sound.` to the JS console.

class Cat extends Animal {
  speak(){
    console.log(`${this.name} meows.`);
  }
}
let k = new Cat('Fluffy');
k.speak(); // Outputs `Fluffy meows.` to the JS console.
```

In the example code above, the `Cat` class extends the `Animal` class. This time, the `Animal` class provides a `speak()` method, but it is very generic ("makes a sound"). The `Cat` class extends `Animal` and overrides the `speak()` method to provide a more cat-like response. This is polymorphism at work: the same method used with these two related Class objects will produce a different result.

It's important to notice that in spite of the complexity of the hierarchical relationships and the "mutation" or "change" of the `speak()` method, the interface of the `Animal` and `Cat` classes remains consistent and understandable for the developer. As long as a developer knows that they can instantiate an instance of either class and then call the `speak()` method they can make use of these Class objects. Changing what the `Cat` says does not require the developer to learn a new process or method.

We will investigate more of how Class objects are created and used in JavaScript on the following pages.

Note: Animal examples on this page were drawn largely from the Mozilla Developers Network page, "Classes".

Creating and Using Classes

In JavaScript we use the `class` command to define a new Class object. This command is followed by a specific structure that looks like this:

```
class ClassName {
    constructor([parameters]){

    }
    methodName([parameters]){

    }
}
```

The ECMAScript 6 syntax makes defining a Class object straightforward: We use the `class` command, then provide a name. By convention, class names should be "CapitalizedCamelCase" (first letter capitalized, and capitalize the first letter of each word in the name with no spaces).

Once the class has been created, it needs a `constructor` function. This method is invoked when a new instance of the class is created, and it sets up the class for working with later. Here is an example:

```
class Car {
    constructor(make, model, year, vin, mileage){
        this.make = make;
        this.model = model;
        this.year = year;
        this.vin = vin;
        this.mileage = mileage;
    }
}
let myCar = new Car('AMC', 'Pacer', '1981', 123456, 10000);
console.log(myCar.make); // Outputs 'AMC' to the JS console.
console.log(myCar.model); // Outputs 'Pacer' to the JS console.
```

Often a `constructor()` function is designed to accept parameters that define the specific instance of the Class object. In the case of the example code, we have created a `Car` class that accepts a range of parameters to describe a car. Then we create an instance of the `Car` class called `myCar`. The `myCar` object contains all the properties and methods of the `Car` class, but it has been populated with specific data because the `constructor()` function has been called.

Of course, what good is a car if we can't drive it?

Methods

In order to *do* things with our Class objects, we must create methods. Methods are functions that live within the object structure of the Class. They can perform actions on our data or within our code to fulfill the user's needs. We can enhance the `Car` example with some methods:

```
class Car {
    constructor(make, model, year, vin, mileage){
        this.make = make;
        this.model = model;
        this.year = year;
        this.vin = vin;
        this.mileage = mileage;
    }
    drive(miles){
        this.mileage = this.mileage + miles;
        console.log(`Vroom! We drove ${miles} miles.`);
    }
}
let myCar = new Car('AMC', 'Pacer', '1981', 123456, 10000);
myCar.drive(250); // Outputs `Vroom! We drove 250 miles.` to the JS console.
console.log(myCar.mileage); // Outputs '10250' to the JS console.
```

In this example the `Car` class has gained a `drive()` method. The `drive()` method takes in a parameter called `miles`, which it adds to the total mileage when it's called. So now we have a `Car` object called `myCar` that can drive some number of miles. The `drive()` method also adds the miles driven to our total mileage. As a developer, we do not need to make a separate call to keep a tally of the number of miles we've driven or use any extra code. This is a good example of "abstraction" in our Class definition that keeps the interface to our Class object simple.

Dynamic Properties

With properties and methods we can do a lot with our Class objects. However, we sometimes need to calculate properties of an object based on the values of some other properties. And sometimes calculating those values needs to happen each time the property is accessed. Consider the example of our car above. What if we wanted a quick and easy way to get a description that told us the information about our Car object? We would want to

include mileage, but that is a property that will change each time we `drive()` the car, so we cannot just calculate this description once and be done (otherwise we could just do the calculation in the `constructor()` method).

Using our Class and data from above, we want to output something like "1981 AMC Pacer with 10000 miles" as a description. We *could* define a method that would `return` that string, but then we'd have to "call" the method, which would be a little out of sync for a "property" like `description`. Instead, what we really want is to define a "dynamic property" that we can reference normally. To do that, we will use the `get` command:

```
class Car {
    constructor(make, model, year, vin, mileage){
        this.make = make;
        this.model = model;
        this.year = year;
        this.vin = vin;
        this.mileage = mileage;
    }
    get description() {
        return `${this.year} ${this.make} ${this.model} with ${this.mileage} miles`;
    }
    drive(miles){
        this.mileage = this.mileage + miles;
        console.log(`Vroom! We drove ${miles} miles.`);
    }
}
let myCar = new Car('AMC', 'Pacer', '1981', 123456, 10000);
myCar.drive(250); // Outputs `Vroom! We drove 250 miles.` to the JS console.
console.log(myCar.description); // Outputs "1981 AMC Pacer with 10250 miles" to the JS
 console.
myCar.drive(250); // Outputs `Vroom! We drove 250 miles.` to the JS console.
console.log(myCar.description); // Outputs "1981 AMC Pacer with 10500 miles" to the JS
 console.
```

The `get` command allows for accessing the `description()` method as if it is a property. The method is called each time we access `Car.description` like a normal property. We now have a dynamic property that updates every time it is called. The more we `drive()`, the more that is reflected in the `description`.

Now that we can create well-featured Class objects, we can move on to exploring how leverage the concept of inheritance.

What is `this`?

The keyword `this` in JavaScript can be tricky to track in all contexts. It usually refers to the current code block (Function, Class, etc.). In the case of a Class object definition, the `this` keyword refers to the Class itself. That's why we refer to `this.property` inside methods when accessing Class properties. (See this article for more about `this`)

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Define a Class called `Book` with `title` and `author` properties that can be set by the developer when instantiating the Class.

```
// Write Book Class here.

// Do not change this line.
let book1 = new Book('Slaughterhouse Five', 'Kurt Vonnegut Jr.');
```

Exercise

Define a Class called `Book` with `title`, `author`, `numReviews`, and `totalScore` properties. Create a method called `review` that accepts a number representing a review score. The `review` method must increment the `numReviews` property and add the review score to the `totalScore` property. Also, add a dynamic property called `rating` that calculates the average of the reviews (`totalScore` divided by `numReviews`).

```
// Write Book Class here.

// Do not change this line.
let book1 = new Book('Slaughterhouse Five', 'Kurt Vonnegut Jr.');
book1.review(4);
book1.review(2);
console.log(book.rating);
```

Extending Classes

When working with Class objects in our code, it's often useful to inherit and extend their functionality. This allows us to add custom functionality without recreating the entire structure of the Class. We can often use complex Class structures provided by third-party modules and libraries by extending their Class objects and adding the features we need.

As mentioned before, Class objects lend themselves to a hierarchical relationship. They can build complex and unique behavior by pulling together a whole chain of inheritance. Here is an example of a simple Class inheritance:

```
class Animal {
    constructor(name){
        this.name;
    }
    speak(){
        console.log(`${this.name} made a noise.`);
    }
}

class Bird extends Animal {
    speak(){
        console.log(`${this.name} chirps.`);
    }
    fly(){
        console.log(`${this.name} takes to the air!`);
    }
}
```

In this example, we have a base class for `Animal` that has been defined. The `Animal` class takes a `name` as a parameter to initialize an instance of the class. It also provides a `speak()` method that is generic to any animal. The `Bird` class *extends* the `Animal` class (using the `extends` command) and overrides the `speak()` method to provide a more bird-like voice. The `Bird` class *also* adds a `fly()` method, since birds can fly.

`super()`

In some cases, we want to keep everything about a method the same, but then add a little extra functionality to it. Take the following example into consideration:

```
class Animal {
    constructor(name){
        this.name;
    }
    speak(){
        console.log(`${this.name} made a noise.`);
    }
    move(){
        console.log(`${this.name} begins to move.`);
    }
}

class Bird extends Animal {
    constructor(name, color){
        super(name);
        this.color = color;
        this.numLegs = 2;
    }
    speak(){
        console.log(`${this.name} chirps.`);
    }
    move(){
        super();
        console.log(`${this.name} takes flight!`);
    }
}
class Dog extends Animal {
    constructor(name){
        super(name);
        this.numLegs = 4;
    }
    speak(){
        console.log(`${this.name} barks.`);
    }
    move(){
        super();
        console.log(`${this.name} walks.`);
    }
}
let tweetie = new Bird('Tweetie Pie', 'yellow');
tweetie.speak(); // Outputs `Tweetie Pie chirps.` to the JS console.
tweetie.move(); // Outputs `Tweetie Pie begins to move.` and `Tweetie Pie takes flight
!` to the JS console.

let rolf = new Dog('Rolf');
rolf.speak(); // Outputs `Rolf barks.` to the JS console.
rolf.move(); // Outputs `Rolf begins to move.` and `Rolf walks.` to the JS console.
```

In this example we start, again, with a base class called `Animal`. The `Animal` class takes in a `name` parameter, but nothing else. Two more classes are created by extending the `Animal` class: `Bird` and `Dog`. Each of these classes uses `super()` to preserve some of

the functionality of specific methods. Each one uses `super()` in the `constructor()` method to preserve the assignment of the `name` property on the Class. They also use the `super()` command in the `move()` method, which preserves the first line of the JS console output.

Using the ability to extend and alter existing Classes, we can accomplish a lot without having to rewrite code or re-design our data and code structures from scratch. Many third-party modules use these concepts to provide powerful tools that, like the cars we drive every day, can be effectively used without being fully understood. This empowers us as developers to accomplish more and to stand on the shoulders of developers who have come before us.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Extend the `User` class to create an Admin user that uses the `super()` command to add a property called `is_staff` and a method called `checkPermissions()` that will check to verify that `is_staff` equals `true` and will return `true` or `false` accordingly.

```
// User Class
class User {
    constructor(name, email){
        this.name = name;
        this.email = email;
    }
}
// Write Admin Class here.

// Do not change these lines.
let admin1 = new Admin('Jane Doe', 'jane@example.com');
let verified = admin1.checkPermissions();
```

Quiz: Object Oriented JavaScript

Try this self-check quiz to test your knowledge!

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Extra Practice

Remember: Learning to program takes practice! It helps to see concepts over and over, and it's always good to try things more than once. We learn much more the second time we do something. Use the exercises and additional self-checks below to practice.

1. Creating Classes and Class Instances

Review the code, then answer the questions below.

```
class Road {
    constructor(directions=['east','west'],sidewalk=true){
        this.directions = directions;
        this.numLanes = directions.length;
        this.sidewalk = sidewalk;
    }
}

let cityRoad = new Road(['north','north','south','south'], true);

let countryRoad = new Road(['east','west'], false);
```

2. Extending Classes

Review the code, then answer the questions below.

```
class Vehicle {
    constructor(license='ES64EVA'){
        this.license = license;
    }
    move(direction){
        console.log(`Vehicle ${this.license} moves ${direction}`);
    }
    get description(){
        return `Vehicle ${this.license} is a shapeless carriage.`;
    }

}

class Car extends Vehicle {
    constructor(license, numAxles=2, numWheels=4){
        super(license);
        this.numAxles = numAxles;
        this.numWheels = numWheels;
    }
    move(direction){
        return `Vehicle ${this.license} rolls ${direction}`;
    }
    get description(){
        return `Vehicle ${this.license} is a car with ${this.numAxles} axles and ${thi
s.numWheels} wheels.`;
    }
}
class Airplane extends Vehicle {
    constructor(license, numWings=2, numPropellers=1){
        super(license);
        this.numWings = numWings;
        this.numPropellers = numPropellers;
    }
    move(direction){
        return `Vehicle ${this.license} flies ${direction}`;
    }
    get description(){
        return `Vehicle ${this.license} is an airplane with ${this.numWings} wings and
${this.numPropellers} propellers.`;
    }
}

let myCar = new Car('HOTWLZ1');

let myPlane = new Airplane('FLYIN1', 4);
```

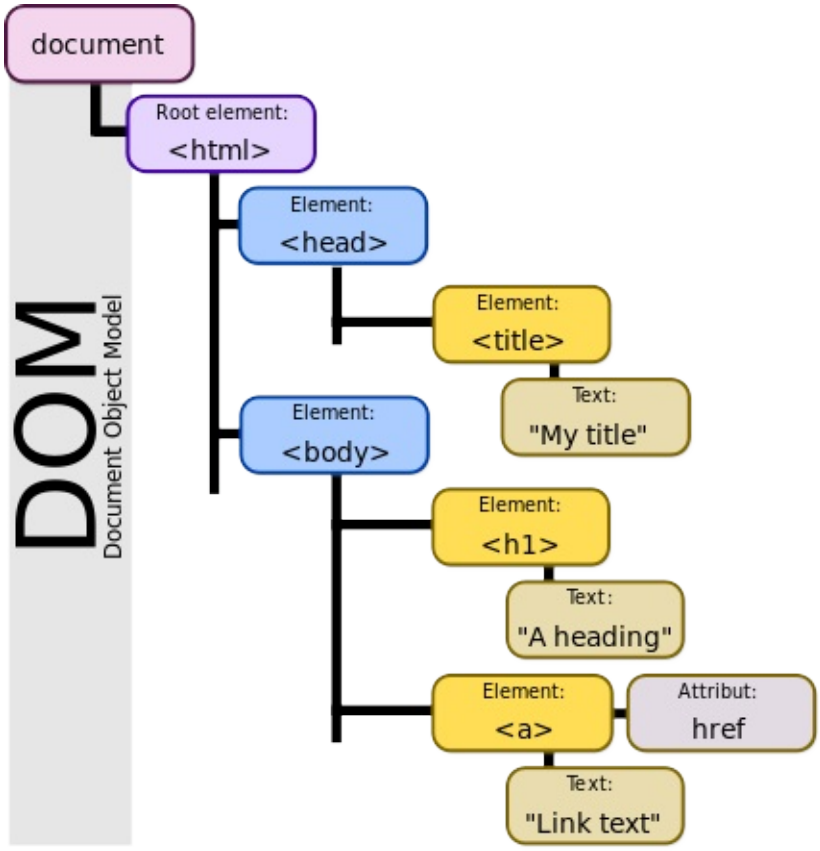
Visit Content Online

The content on this page has been removed from your PDF or ebook format. You may view the content by visiting this book online.

The Document Object Model

The Document Object Model (DOM) is the conceptual model used to represent an HTML document in the browser. JavaScript has built-in functionality to inspect, read, and modify every element in an HTML document using DOM-specific tools such as `document.querySelector()` and `element.addChild()`. There are many of these tools that we will use to interact with the DOM as we write our JavaScript programs.

The DOM defines the way that browsers understand the structure of HTML documents. It is a tree-like hierarchy that starts with a `document` node, then has one `html` node, and then branches out to `head` and `body` nodes. Here is an illustration of what the DOM generally "looks" like in diagram form:



The Document Object Model (DOM) by [Birger Eriksson](https://en.wikipedia.org/wiki/File:DOM-model.svg)

In JavaScript, we can use different commands to traverse the DOM's tree hierarchy in order to access any information contained within the HTML document. This makes it possible to read and interpret information, to grab sections of a document and alter them, to add elements to the document, or to remove them. It is not uncommon for a JavaScript program

to spend considerable time reading data from a document's elements and their attributes, writing new elements into the document, or modifying the class name or textual content of elements.

DOM Hierarchy

As we can see in the image above, the DOM is a hierarchical structure that involves branching. It lists each HTML element (defined by an HTML tag) and relates them as either a parent, child, and/or sibling. We often talk about "traversing" or "walking" the DOM, which refers to moving along the branches of this tree. Often we need to gather up all of the children of a particular HTML element, such as when we want to access all the items in a particular list. Or we want to know the parent element of a button or link so we can provide some responsiveness for the user.

More often, we need to add elements to the document, which requires understanding if we are appending, prepending, inserting as a child, or adding as a sibling, etc. It's critical to understand the DOM hierarchy of a given document clearly in order to effectively use JavaScript to modify the document.

Of course, DOM, hierarchy is just one way we have of selecting elements. We often rely on `id` or `class` attributes to select DOM elements based on CSS select queries. This provides us with easy ways to get at the pieces of the document we need, and it allows us to create groupings of elements outside of the normal DOM hierarchy relationships. All of the techniques for traversing and selecting DOM elements are critical for success in writing interactive web pages.

Minding Style

It's easier to understand DOM hierarchy if we have a solid habit of properly indenting our HTML code. Although web browsers don't care about indentation, it is a handy visual guide for us developers who need to easily understand the organization of our DOM elements. Some developers believe that it doesn't matter how they indent their code since HTML Inspector tools available in every web browser do a great job of cleaning up and revealing the DOM hierarchy. That is true, and those tools are incredibly valuable, but it's always a good idea to keep your code properly indented and reasonably organized. Doing so will not only make your code easier to read during development, but it will help you be more aware of how the DOM of your document will be organized.

Putting It All Together

Using the DOM standard tools, JavaScript programs can also alter the CSS attributes of a given element. This means that JavaScript can be used in conjunction with HTML and CSS to create very dynamic visual representations of HTML content. Elements can be made to move, take on different styles, perform various functions, and much more.

As an example, we can consider the following snippets of JavaScript, CSS, and HTML.

html

```
<button class="btn save">Save Item</button>
```

css

```
.save {
    background: blue;
    color: white;
    content: 'Save Item';
}
.saved {
    background: blue;
    color: yellow;
    content: 'Saved!';
}
```

js

```
let saveButtons = document.querySelectorAll('.btn .save');
saveButtons.forEach(function(button){
    button.addEventListener('click', function(event){
        event.target.setAttribute('class', 'btn saved');
        event.target.setAttribute('disabled', true);
    });
});
```

We will dive deeper into how each part of this code works over the course of the next few sections of this book, but for now it's useful to see that we have all three parts of our web technology trifecta at play: HTML, CSS, and JavaScript. The HTML specifies the content of the document, and it describes that content using HTML tags to define specific elements, their attributes, and other characteristics such as `href`, `class`, or `id`. The CSS

configures the visual presentation of specific HTML elements, using the CSS Selector syntax to match elements, and then describing visual attributes of those elements such as `background` or `color`.

JavaScript is the interactive glue that binds together each of these components and allows instructions to be executed in response to a user's interaction with the document. In the case above, we can see a `<button>` element defined in HTML. It has the `class` attribute, which contains `"btn save"`. We can imagine that this is a button on a content website, perhaps, where users may wish to save the content they find. (Think of the "Watch Later" button on YouTube, or the "favorite" button on so many sites.) This button is initially styled according to the classes defined on the HTML element using the `class` attribute. However, this can be changed via JavaScript, which is able to select all of those buttons out of the document and add an "event listener" (which we will discuss in the next section) to respond to the user's click on the button. When the user clicks the button, the event listener executes a set of instructions that alter the classes applied to the button and disable the button from being clicked again.

We will explore more about how each of these steps work in the JavaScript code, but for now it's useful to admire how the DOM provides the mechanism for each of these discrete technologies to operate in conjunction to give us a working web page. This allows a separation of concerns (content structuring, visual presentation, interactive instructions/logic) that makes it more convenient to build these deceptively complex media objects.

Selecting Elements

In order to work with the content of an HTML file, it's important to be able to retrieve specific elements and work with them in JavaScript. Most of the commands we can use to work with elements of the DOM are methods of the `document` object. The `document` object contains all sorts of properties that can tell us about the HTML, and it provides methods that can be used to perform actions on the HTML. The set of methods we will focus on for this section are those concerned with selecting elements out of the DOM.

Query Selectors

In modern ECMAScript, the `document` object provides two key methods to find and retrieve elements from the DOM: `document.querySelector()` and `document.querySelectorAll()`. These two methods allow us to select DOM elements using CSS selectors. Here is an example of some HTML and JS that works together:

html

```
<section id="profile">
    <p class="image-wrapper"><img src="user.jpg" alt="User Profile Image"></p>
    <p class="bio">Some information about the user.</p>
    <ul id="profile-actions">
        <li><a href="#edit">Edit</a></li>
        <li><a href="#settings">Settings</a></li>
    </ul>
</section>
```

js

```
// Select the first matching element
let profileSection = document.querySelector('#profile');
let profileBio = document.querySelector('p.bio');
profileBio.innerHTML = 'Updated text about the user.' // Updates content of user bio.

// Select multiple elements
let profileActions = document.querySelectorAll('ul#profile-actions li');
for (action of profileActions){
    console.log(action.innerHTML); // Outputs link tag for each action item.
});
```

As we can see in this example, we have HTML for a user profile page. There is a section for the profile, and that section contains several other elements. In the JavaScript, we can see how the `document.querySelector()` method can be used to select one element at a time (the *first* matching element). This is handy in many cases, but we can also use `document.querySelectorAll()` to select *all* of the elements matching our CSS selector. We can see that the `profileActions` variable is populated with a `document.querySelectorAll()` command, so it is equal to the entire set of results matching the CSS selector query. If we loop through those results using a standard `for ... of` loop, we can see that each element has been retrieved.

These two commands can leverage the full power of CSS Selectors. This means that we can use very complex syntax to define specific, dynamic, and highly customized queries. We can use everything we know about selecting elements for visual styling with CSS in order to enhance our pages. This is powerful stuff.

Old Style Selectors

As with so many older features of JavaScript, old style DOM selectors are still functional and we still run across them in code. They can be used to select based on more fundamental properties of elements, which can sometimes be convenient. These commands are still provided as methods on the `document` object. They are:

- `document.getElementById()` – Fetches the one element with the given ID (note: do not supply a "`#`" when using this method).
- `document.getElementsByClassName()` – Fetches an Array-like set of elements that match the given Class name (note: do not supply a "`.`" when using this method).
- `document.getElementsByTagName()` – Fetches an Array-like set of elements that match the given HTML Tag name.

These can be used like we would expect, given the descriptions above. Here is an example:

html

```
<section id="profile">
    <p class="image-wrapper"><img src="user.jpg" alt="User Profile Image"></p>
    <p class="bio">Some information about the user.</p>
    <ul id="profile-actions">
        <li><a href="#edit">Edit</a></li>
        <li><a href="#settings">Settings</a></li>
    </ul>
</section>
```

js

```
// Select the first matching element
let profileSection = document.getElementById('profile');
let profileBio = document.getElementsByClassName('bio');
// Note how the index must be used below due to the results of `getElementsByClassName
()`
profileBio[0].innerHTML = 'Updated text about the user.' // Updates content of user bi
o.

// Note that the selector below would match all `li` elements in the entire page.
let profileActions = document.getElementsByTagName('li');
for (action of profileActions){
    console.log(action.innerHTML); // Outputs link tag for each action item.
});
```

The code in this example does the exact same thing as the code in the previous example. In cases where we control the HTML entirely ourselves, it is not too bad to write code using these legacy selectors. We can use very specific Class and ID names to identify the HTML elements we need to work with. It can be a bit clunky to work with the older style selectors, but we can get the job done.

The Perils of Rigid Naming

Part of the strength of CSS is that we can write selectors that are general enough to be very useful, but specific enough to perform the tasks we need. There has been a lot of thought put into how we can select elements within a DOM, and that thought has been expressed as the CSS Selector Syntax. For example, we can easily write selectors to select every other row in a table, every fifth item in a list, or diagonal elements in a grid. The CSS Selector language is very powerful, and without it, we become very rigid in how we name and structure elements.

When we rely entirely on explicit ID or Class names, our code becomes more fragile. There is more chance of conflict when developers change the names of IDs or Classes to suit the needs of either visual styling or interactivity. It becomes tempting to come up with complex naming schemes to differentiate between "functional" names and "design" names: classes that are found and used by JS versus classes found and used by CSS. For many years, developers struggled to have easier ways to select elements in the DOM.

From that struggle was born great tools like jQuery, which became the dominant JavaScript toolset for many years. Now that these features have become part of the standard JavaScript library, it's less necessary to always use a third-party module to make things more convenient. We can now leverage the full power of CSS Selectors from within JavaScript, and the world is a better place.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section. The exercises in this section all assume the following HTML. Please write JavaScript as if it is attached to this HTML snippet:

```
<ul id="search-results">
    <li class="result book">
        <p class="title">Dune <span class="format">Book</span></p>
        <p class="year">1965</p>
        <ul class="actions">
            <li><a href="#save">Save</a></li>
            <li><a href="#share">Share</a></li>
            <li><a href="#report">Report</a></li>
        </ul>
    </li>
    <li class="result movie">
        <p class="title">Dune <span class="format">Movie</span></p>
        <p class="year">1984</p>
        <ul class="actions">
            <li><a href="#save">Save</a></li>
            <li><a href="#share">Share</a></li>
            <li><a href="#report">Report</a></li>
        </ul>
    </li>
    <li class="result tv">
        <p class="title">Dune <span class="format">Television</span></p>
        <p class="year">2000</p>
        <ul class="actions">
            <li><a href="#save">Save</a></li>
            <li><a href="#share">Share</a></li>
            <li><a href="#report">Report</a></li>
        </ul>
    </li>
</ul>
```

Exercise

Select the `search-results` list and make it available in a variable called `resultsList`. Use `document.querySelector()`.

```
// Set `resultsList` equal to the DOM element for the search-results list.
```

Exercise

Use `document.querySelectorAll()` to select each of the result list items. Make this available as the variable `resultItems`.

```
// Set `resultItems` equal to the DOM elements for each item in the search results
list.
```

Creating and Removing DOM Elements

Now that we've explored how to select and retrieve elements from the DOM, we will move on to how to modify these elements. Modifying elements in the DOM, or adding/removing elements to/from the DOM, is crucial to creating a responsive experience for users. Users expect that if they interact with a page it will change: links will light up, buttons will click down, text will change, notifications will come and go, and their actions will have an effect.

Our pages should engage the user at all times in ways that are meaningful, predictable, and regular. If a user clicks "save" there should be suitable feedback to let the user know that the item has been saved. This helps the user understand what has happened and it matches what the user anticipated would happen. The next time the user clicks "save" the same interaction should unfold. This shows regularity in the interface.

Of course, making broad statements about how things *should* happen on a web page is one thing. Building those features is often another matter entirely. Let's take a look at how we can begin that task.

For the following examples, refer to this HTML markup for a contact list:

```
<ul id="contact-list">
    <li class="contact">
        <p class="name">Grace Hopper</p>
        <p class="phone">555-1234</p>
        <p class="email"><a href="mailto:grace@example.com">grace@example.com</a></p>
        <p class="address">123 Fake St.</p>
        <p class="city">Anytown</p>
        <p class="state">XZ</p>
    </li>
    <li class="contact">
        <p class="name">Bill Joy</p>
        <p class="phone">555-8754</p>
        <p class="email"><a href="mailto:bill@example.com">bill@example.com</a></p>
        <p class="address">42 Nowhere Ct.</p>
        <p class="city">Somewhere</p>
        <p class="state">ZY</p>
    </li>
    <li class="contact">
        <p class="name">Katherine Johnson</p>
        <p class="phone">555-3323</p>
        <p class="email"><a href="mailto:k8@example.com">k8@example.com</a></p>
        <p class="address">555 Secret Place</p>
        <p class="city">Nowhere</p>
        <p class="state">DV</p>
    </li>
</ul>
```

Adding Elements to the DOM

It's often necessary to add elements into the DOM. We can do this using a few different techniques. Often it's best to combine some techniques in order to maximize efficiency. Here is an example where we add a `contact` object to the list in the HTML example above.

```
function addContact(contactObj){
    // Set up a template literal to populate the data into an HTML structure.
    let contactContent =   `<p class="name">${contactObj.name}</p>
                            <p class="phone">${contactObj.phone}</p>
                            <p class="email"><a href="mailto:${contactObj.email}">${co
ntactObj.email}</a></p>
                            <p class="address">${contactObj.address}</p>
                            <p class="city">${contactObj.city}</p>
                            <p class="state">${contactObj.state}</p>`;

    // Make a new list item (`li`) to contain the contact content.
    let newContactLI = document.createElement('li');
    newContactLi.innerHTML = contactContent;

    // Select the `ul` containing all the contacts.
    let contactList = document.querySelector('#contact-list');

    // Append the new list item.
    contactList.appendChild(newContactLI);
}
```

In this example we can see a couple different approaches to creating new DOM elements and adding them to the DOM. First, we use a template literal that contains all the HTML markup populated with our data from `contactObj`, which is passed into this function as a parameter. After we have made the `contactContent` template literal, we then create the `newContactLI` object using the `document.createElement()` method. This method creates a new DOM element, which in this case is called `newContactLI`, but it does *not* add that element to the DOM (we do that later in the example).

Once we've created the `newContactLI` object, we set `newContactLI.innerHTML` equal to `contactContent`. This populates the `newContactLI` object with our template literal. The `innerHTML` attribute on any DOM element contains all of the HTML within that element. In this case, our new list item contains no HTML. It's the equivalent of: `<li></li>`. Once we set the element's `innerHTML` value to our `contactContent` template literal, the `innerHTML` is populated directly.

Now that we have our new list item all populated with content, we can add it to the contact list in the DOM. The new list item is not visible in our browser until we append it to some element in the DOM, so this step is absolutely necessary.

To append the new list item to the contact list, we must first select the unordered list containing the contacts (with the ID `#contact-list`). We do that and call it `contactList` in our code. Next, we use the `appendChild()` method of DOM elements to append the new contact list item to the contact list.

Although we often get by using just a few of the features of DOM elements, it's worthwhile to explore the full set of properties and methods available on DOM element objects. This includes properties like `scrollHeight` and `className`, which can be used when processing different types of interaction. There are also additional methods that can accommodate our computing needs such as `insertBefore()`, which inserts a new DOM element object *before* the selected object.

The trickiest part of creating new DOM elements is making decisions about the best way to build up the HTML involved. It's entirely possible to create elements and set all of their attributes using specific JavaScript commands. However, it's often much quicker and easier to maintain to populate large HTML structures using template literals and then drop that HTML into a container element. There's no single correct way to approach this task, so experiment with all the ways of adding elements into the DOM to get a feeling for what works.

Removing Elements from the DOM

Removing elements from the DOM is also possible using a variety of techniques. As with adding elements to the DOM, there is no single correct approach, and we will develop preferences for what works in different situations. A simple way of removing elements from the DOM is to select their parent element and then set the `innerHTML` value to an empty string (`''`). Here's an example of removing all of the list items from the `#contacts-list` element using `innerHTML`:

```
// Select the unordered list containing contacts.
let contactsList = document.querySelector('#contacts-list');

// Remove all list items (and their content) using the `innerHTML` property.
contactsList.innerHTML = '';
```

By setting the `innerHTML` of the `contactsList` object to `''`, we have removed all of the list items, too. This is easy and clean, but it is also a fairly blunt way of removing a DOM element. We often need a more precise or limited way of removing elements from the DOM. For that, we can use the `remove()` method of each DOM element object. Here is an example:

```
let firstContactItem = document.querySelector('.contact');
firstContactItem.remove()
```

This example selects the first contact item with the `.contact` class and removes it from the DOM. It uses the `remove()` method to accomplish this goal. Although this example is a bit glib, it illustrates how easy it is to remove an element from the DOM once we've selected it. We often have the case where we have a specific element object to work with, and in the next section we will explore more about how to respond to events that are triggered by individual DOM elements.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section. The exercises in this section all assume the following HTML. Please write JavaScript as if it is attached to this HTML snippet:

```
<ul id="search-results">
    <li class="result book">
        <p class="title">Dune <span class="format">Book</span></p>
        <p class="year">1965</p>
        <ul class="actions">
            <li><a href="#save">Save</a></li>
            <li><a href="#share">Share</a></li>
            <li><a href="#report">Report</a></li>
        </ul>
    </li>
    <li class="result movie">
        <p class="title">Dune <span class="format">Movie</span></p>
        <p class="year">1984</p>
        <ul class="actions">
            <li><a href="#save">Save</a></li>
            <li><a href="#share">Share</a></li>
            <li><a href="#report">Report</a></li>
        </ul>
    </li>
    <li class="result tv">
        <p class="title">Dune <span class="format">Television</span></p>
        <p class="year">2000</p>
        <ul class="actions">
            <li><a href="#save">Save</a></li>
            <li><a href="#share">Share</a></li>
            <li><a href="#report">Report</a></li>
        </ul>
    </li>
</ul>
```

Exercise

Select the `search-results` list and make it available in a variable called `resultsList`.

Use `document.querySelector()`. Create a new list item with the following info:

- Title: Children of Dune
- Format: Television
- Year: 2003

Once you've finished populating the Append the new list item as a new child at the end of the `resultsList`.

```
// Set `resultsList` equal to the DOM element for the search-results list.

// Create a new list item called `newLI`

// Set innerHTML of new list item to match HTML structure with new data specified a
bove.

// Append `newLI` to the `resultsList`
```

Modifying Elements

In addition to adding and removing elements from the DOM, it's important to be able to alter elements to reflect changes, updates, and other system status. These changes are not just changes in the text or information presented to the user, but also to the attributes and status of the elements in the DOM. It's necessary to consider all of the ways in which DOM elements may be modified by running JavaScript.

Modifying the Content of Elements

The content of elements is generally accessed through the `innerHTML` property. This property can be used to either retrieve the contents of an element (and all of its child elements) or to set those contents to something new. It's often useful to use `innerHTml` when setting the text value of an element that contains no child elements. Consider the following example:

html

```
<button id="123" class="favorite not-favorited">Click to add Favorite!</button>
```

js

```
let button = document.querySelector('#123');
button.innerHTML = "Added to Favorites!";
```

We can imagine that on a page with an "add favorite" button there is some JavaScript that will change the text of the button using the `innerHTML` property. This is a simple but common case: changing the text of a button or link to let the user know that an action has been performed. Sometimes it's necessary to change related elements, such as the counter displayed on another HTML element. Using `innerHTML` is a straightforward way to change the textual content of an element.

Modifying Element Attributes

Sometimes we need to change more than the textual content of an element. We might need to change the `src` attribute of an `<img>` tag, or the `value` attribute of a form input. We may want to change the `width` or `height` of an element, or any number of other HTML

attributes. In these cases, it's necessary to be able to "get" and "set" the values of element attributes. To do this, each element provides us with two methods: `getAttribute` and `setAttribute`.

Consider this example:

html

```
<img src="placeholder.jpg">
```

js

```
let image = document.querySelector('.bio img');

if (image.getAttribute('src') == 'placeholder.jpg'){
    image.setAttribute('src', 'new.jpg');
}
```

In this example we can see that the JavaScript code selects the image and then uses `getAttribute()` to retrieve the `src` attribute value. The `getAttribute()` method takes just one parameter: the name of the attribute. It checks that value and, if the values match, it replaces the `src` attribute value with a new filename using `setAttribute()`.

The `setAttribute()` method takes two parameters: the name of the attribute, and the new value. You can set as many attributes as you need, but it requires a separate instruction for each one.

Here is another example that demonstrates creating a new DOM element and adding attributes to it.

```
let myForm = document.querySelector('#myform');

let newInput = document.createElement('input');
newInput.setAttribute('name', 'username');
newInput.setAttribute('value', '');
newInput.setAttribute('placeholder', 'Enter Your Username');
newInput.setAttribute('type', 'text');

myForm.appendChild(newInput);
```

In this example, a new text input is created for a form. The `name`, `value`, `placeholder`, and `type` attributes are set using `setAttribute()`. The input is appended to the `form` element with the ID `#myform`.

It is also possible to remove an attribute altogether with the `removeAttribute()` method. Consider this HTML and JavaScript example:

html

```
<input name="username" type="text" value="myusername" disabled="true">
```

js

```
let usernameInput = document.querySelector('input[name="username"]');
usernameInput.removeAttribute('disabled');
```

In this example, the text input called `username` has been disabled. The JavaScript selects that text input and removes the `disabled` attribute, which would allow a user to edit this field. This is a common use case for providing safe access to data. Form fields use a lot of attributes to control behavior, so these methods are especially useful for working with form field DOM elements.

Between `setAttribute()` and `removeAttribute()` we have solid tools for modifying more than the textual content of a DOM element. These are powerful methods that can help in many cases.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section. The exercises in this section all assume the following HTML. Please write JavaScript as if it is attached to this HTML snippet:

```
<button class="btn save">Save Item</button>
```

Exercise

Given the HTML above, write JavaScript to change the text on the button to read "Saved!" and change the class attribute to be "btn saved".

```
// Select the button from the DOM using document.querySelector().
let saveButton =

// Change the innerHTML of the button

// Change the class attribute
```

Altering DOM Element Styles

Another task we perform often in our JavaScript programs is altering the visual styling of elements in the DOM. This allows us to create all sorts of interactive visual effects that can provide a compelling experience for the user. Although we can use the `setAttribute()` property on a DOM element to change the classes we have applied, there are other ways to modify styles either in groups or individually.

The `element.style` Attribute

Each DOM element has a `element.style` attribute, which is a JavaScript object. The `element.style` object has attributes that can be used to read or set style properties for the DOM element. Any time we have selected a DOM element we can read or set the values of all CSS properties by referencing the property name. Here is an example:

html

```
<button class="btn save">Save</button>
```

css

```
.save {
    background: blue;
    color: white;
    font-size: 1.2rem;
    font-family: sans-serif;
}
.btn {
    padding: 0.2rem;
    border: 1px solid yellow;
    background: black;
    color: white;
}
```

js

```
let saveButton = document.querySelector('button.save');
console.log(saveButton.style.background); // Prints: "blue"
console.log(saveButton.style.fontSize); // Prints: "1.2rem"
console.log(saveButton.style.fontFamily); // Prints: "sans-serif"
console.log(saveButton.style.border); // Prints: "1px solid yellow"
console.log(saveButton.style.padding); // Prints: "0.2rem"
```

Looking at the HTML, CSS, and JS above, we can see that the `<button>` element has two classes applied to it: `.btn` and `.save`. These two classes define several properties for the visual presentation of the button. In the JavaScript part of the example, we select the `saveButton` using a standard `document.querySelector()` call. Once we have selected the button element we can access the `element.style` attribute to read and print the CSS properties applied to this element.

We could also change the values dynamically using JavaScript:

```
saveButton.style.background = "red";
saveButton.style.color = "yellow";
```

These two lines of JavaScript would alter the background and text color of the button. We could set these style properties to any valid CSS value and they would override the existing style in the browser.

JS Mapping of CSS Properties

JavaScript objects have attributes. Those attributes cannot be named with dashes. However, some CSS properties (`font-face` , `background-color` , `border-size` , etc.) have dashes in their names. So how do we reference these properties as part of the `element.style` object?

There is a mapping between CSS Property names and the names they are given as attributes of the `element.style` object. For the most part, the dashes are removed and the name is written in "camelCase", which is common in JavaScript. So `font-face` becomes `element.style.fontFace` and `background-color` becomes `element.style.backgroundColor` . A full chart of property name conversions can be found here.

Setting Styles in Bulk: `element.style.cssText`

As we can see in the example above, the `element.style` attribute of a DOM element object is very useful for working with styles. But if we want to alter or read several styles all at once it can be a bit tedious to type it all out in separate lines of code. Fortunately, there is a `cssText` attribute on the `element.style` object that allows for writing styles in one big chunk of text. Consider this example:

```
let newItem = document.createElement('li');
newItem.style.cssText = "background:green; color: white; font-family: sans-serif; bord
er: 1px solid blue;";
```

Using the `cssText` attribute allows us to set many style properties with a more-or-less normal definition. It must all be compressed into a string, but we could use a template literal to allow for a more eye pleasing presentation in the code (and easier dynamic population of any values we've calculated for styles).

Reading Computed Styles

Sometimes DOM elements are affected by styles not directly applied to them. Many CSS properties are inherited from parent elements, and it's very common to use CSS inheritance to our advantage as web developers and interface designers. Keeping things like font families and sizes consistent is core to providing a robust, pleasing interaction for users. But sometimes we need to know the "computed styles" of a DOM element.

We can use the `window.getComputedStyle()` method to retrieve the computed styles that are applied to a specific DOM element. This method returns a `style` object very much like the `element.style` object we worked with previously. We can review the values of these properties and use them to make decisions about changes based on the exact styles the user sees applied to that element. Here is an example:

html

```
<body style="font-weight: bold;">
    <p style="color: green;">Page content.</p>
</body>
```

js

```
let paragraph = document.querySelector('p');
let computedStyle = window.getComputedStyle(paragraph);

console.log('Element style for font-weight: ' + paragraph.style.fontWeight); // prints
 empty string ("")
console.log('Computed style for font-weight: ' + computedStyle.fontWeight); // prints
"bold"

console.log('Element style for color: ' + paragraph.style.color); // prints "green"
console.log('Computed style for color: ' + computedStyle.color); // prints "rgb(0,255,
0)"
```

In this example we can see that the element style is restricted to styles explicitly applied to the element, and the properties use the same definition we specified in our stylesheets or HTML. However, the computed style returns the actual style being applied, even if it was inherited from a parent element. The computed style also converts synonyms and shortcuts (such as "green") to their computed value (in this case, `rgb(0,255,0)`).

Understanding which set of styles we want to work with, defined or computed, is important to getting the most out of our efforts. Regardless of which values we need, we can easily find them, work with them, and alter them using JavaScript.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section. The exercises in this section all assume the following HTML. Please write JavaScript as if it is attached to this HTML snippet:

```
<button class="btn save">Save Item</button>
```

Exercise

Given the HTML above, write JavaScript to change the styles to be: background: green, padding: 0.2rem, color: white, border: solid 1px yellow, border-radius: 4px.

```
// Select the button from the DOM using document.querySelector().
let saveButton = ;

// Alter styles using either Element.style or Element.style.cssText.
```

HTML Data Attributes

A feature of HTML that is often overlooked by non-developers is the ability to create "data" attributes on HTML elements that can store specific data values. These properties can be named (almost) whatever we want as long as they are preceded by the `data-` prefix. We could create properties such as `data-id`, `data-title`, `data-username`, etc. We can put as many of these attributes onto an HTML element as we would like.

Using HTML data attributes, we can retrieve a lot of information that is useful for making different features on our websites. Returning to our old example of a "Add to Favorites" button, it would not be uncommon to use a data attribute to carry through the ID of the content item being favorited. Here's an example:

html

```
<button data-contentID='1234' data-title='Fascinating Article'>Add to Favorites!</butt
on>
```

js

```
let faveButton = document.querySelector('button');
console.log(faveButton.dataset.contentID); // prints "1234" to the console
console.log(faveButton.dataset.title); // prints "Fascinating Article" to the console
```

In this example, we've used data attributes to store the `contentID` and the `title`. We can select this button however we wish in JavaScript, and then we can access the values of these attributes by accessing the `faveButton.dataset` object. We can also change values, or add values, to the `faveButton.dataset` object to facilitate whatever logic we need.

Every DOM element we select in JavaScript has an `element.dataset` object attached to it. We can make full use of this object to manage our data needs and make sure that our code can access that information when we need to. At this point, we have only been manually selecting elements in the DOM, but in the next section we will put everything together to respond to user interactions with the DOM using "event handlers". These HTML data attributes are even more handy when we are actually working with events initiated by our users.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.The exercises in this section all assume the following HTML. Please write JavaScript as if it is attached to this HTML snippet:

```
<button class="btn save" data-id="12234" data-saved="false">Save Item</button>
```

Exercise

Given the HTML above, write JavaScript to change the `data-saved` attribute to be `true` and read the `data-id` attribute to use in the `saveItem()` function.

```
// Select the button from the DOM using document.querySelector().
let saveButton =

// Alter the `data-saved` attribute to be 'true'.

// Use the provided `saveItem()` function to save the item.
// This function requires the item ID.

let saveSuccess = saveItem('Item ID goes here.');
```

Quiz: The Document Object Model

Try this self-check quiz to test your knowledge!

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Extra Practice

Remember: Learning to program takes practice! It helps to see concepts over and over, and it's always good to try things more than once. We learn much more the second time we do something. Use the exercises and additional self-checks below to practice.

1. Selecting and Adding DOM Elements

Review the HTML code, then answer the questions below.

```
<div id="content">
    <ul id="subnav">
        <li class="menu-item"><a href="#">Home</a></li>
        <li class="menu-item active"><a href="#">Archive</a></li>
        <li class="menu-item"><a href="#">About Us</a></li>
    </ul>
    <h1 class="page-title">Archive of Posts</h1>
    <p>Use this archive listing to find old posts.</p>
</div>
```

2. Modifying DOM Elements

Review the HTML code, then answer the questions below.

```
<div id="item-123" class="content-item-wrapper">
    <h3 class="item-title">Magnificent Search Result</h3>
    <p class="item-description">Some helpful description about this search result.</p>
    <ul id="item-actions">
        <li class="action save"><a href="#">Save</a></li>
        <li class="action share"><a href="#">Share</a></li>
    </ul>
</div>
```

Visit Content Online

The content on this page has been removed from your PDF or ebook format. You may view the content by visiting this book online.

Handling Events

In our code so far, we have worried primarily about doing things. We have created variables, objects, classes, and more. We have looped and conditionaled (?) and generally directed the logic, timing, and actions our code takes in a somewhat linear manner. This is all well and good, but it's missing a crucial element to truly be useful to us as web developers.

In the real-world, our JavaScript code lives in the browser, where a user is viewing a web page. The user is going to scroll, click, select, fill in forms, and do all sorts of other things on that web page. We want our code to be the thing that makes all these interactions possible, and in order to do that, we must be able to do one specific thing: React to a user's input.

We have to know when the user scrolled, clicked, filled in a field, submitted a form, and much more. We need to know when our page has finished downloading and is ready for the user to interact with it. We might very well need to know when certain portions of our code are done with their processing and have caused something to change in the context of our application.

In each of these cases, we can use built-in features of JavaScript to respond to **Events**. Events are signals that are emitted within the JavaScript context. There are many events that are triggered throughout a user's interaction with the web page: click events, scroll events, page load events, etc. These events always exist and they are always being emitted when the browser detects user interaction. But we must set up our JavaScript code to *listen* for these events and respond to them.

We can use the built-in events emitted by user interactions to effectively build interface elements, but we can also define our own events and use those to build even more complex systems that respond to changes in both user behavior and data. We could define an event, for example, that would update the interface with new data whenever a value in the system changes. Or we might define an event that would kick off a larger process, such as publishing a new content item.

Event-Driven Systems

There is a whole world of thought about event-driven system architecture, and many of those concepts are applicable to JavaScript. Events allow us to practice "loose coupling", which describes the practice of building systems that work together but which share few dependencies.

For example, a content management system might have one component that is concerned with content authoring and managing editorial approval. That system might be complex and dynamic, but at some point it would trigger a "publish article" event. The publishing system need not know anything about the business logic of reviewing and approving content or the product needs of the content outside of the publishing formats. The publishing system could be an independent process that picks up the data at the right time and packages it for publishing in whatever formats are configured. Once the publishing operation is completed, the publishing system could emit a "article published successfully" event and the content management system could listen for that event in order to trigger the next step in the process.

In this example, the content management system and the publish system could be said to be "loosely coupled"—meaning that they have a small dependency on one another, but generally do their job autonomously. The event signals are sent and received by each individual component of the system, and each component is able to listen for and respond to relevant events.

Event-driven systems are all over the place. They are popular in games, in asynchronous communications utilities, and other systems that manage complex processes where system changes might come from many different sources.

Adding Event Listeners

In order to use events in our code, it's important for us to create "event listeners". These are functions that watch for a particular event to be emitted in the JavaScript system and then execute when they detect that signal. It can be a little tricky to grasp exactly how events get triggered at first, but once we can understand and create listeners we can do amazing things with our events.

Creating Listeners

To create an event listener, we must first select a DOM element. Different DOM elements emit different events. For example, a `form` element will emit the `reset` and `submit` events. Once we select the DOM element, we can use the `Element.addEventListener()` method to create a new listener in our program. That means that if we wanted to respond to a form submission we could use the following HTML and JS code.

html

```
<form id="message-form" action="/some/action/" method="POST">
    <input type="text" name="message">
    <input type="submit" name="Send Message">
</form>
```

js

```
let messageForm = document.querySelector('#message-form');

function handleFormSubmission(event){
    // Code that does something when the form is submitted.
    console.log(`Form has been submitted.`);
}
messageForm.addEventListener('submit', handleFormSubmission);
```

In this example, we have selected the form with the ID `#message-form`. This is stored in the `messageForm` variable. We then use the `addEventListener()` method, which exists on every DOM element, to create a `submit` event listener. This means that anytime the user clicks the "Send Message" button to submit this form, the `submit` event will be emitted. That signal will be picked up by this listener, and the function named in the second parameter of the method call will be executed.

Here is another example that creates a `button` that changes color.

html

```
<button id="color-changer">Click me to change color.</button>
```

js

```
let button = document.querySelector('#color-changer');

function changeColor(event){
    if(event.target.style.backgroundColor === "cornflowerblue"){
        event.target.style.backgroundColor = "pink";
    } else {
        event.target.style.backgroundColor = "cornflowerblue";
    }
}

button.addEventListener('click', changeColor);
```

The result of the code above is this button:

Click me to change color.

In this example we use the same method to apply the event listener to the button as we did to apply the event listener to the form. We must first select the element to which we want to attach a listener. Then, we define the function that will be executed when the listener is triggered (when it detects the event signal). Finally, we attach the listener to the DOM element using the `addEventListener()` method.

This process is repeated often throughout a complex web app. Event listeners are, literally, the things that make our interfaces move. When properly applied, they can give us a compelling user experience impossible without the logic of JavaScript.

There are many more intricacies to successfully handling events, but we will get to those on the next pages.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Add a `click` event listener to the `button` element.

```
let myTest = false;
function myAction(event){
    myTest = true;
}
let button = document.querySelector('button');
// add event listener that executes `myAction()`
```

Exercise

Add a `submit` event listener to the `form` element.

```
let myTest = false;
function myAction(event){
    myTest = true;
}
let form = document.querySelector('form');
// add event listener that executes `myAction()`
```

Responding to Events

When an event listener detects an event signal in the system, it executes the specified function, which we call an *event handler*. This can be a named function or an anonymous function. The event listener passes in the `Event` object, which contains information about the event itself including the `target` of the event (which is usually the button or link that was clicked, the form that was submitted, etc.). We can use all of these components of an event to make decisions in our code about what lines to execute next.

Let's take a look at more examples of how we can use this information to respond to events, and how to prevent default browser actions from executing when an event is triggered.

Named vs. Anonymous Functions

There are two ways to define the functions that are executed when an event signal is detected by an event listener. The way used in the examples in the previous section uses a *named function* that is executed when the listener detects an event signal. Here is an example.

```
function myAction(event){
    console.log('My action has been triggered!');
}

let button = document.querySelector('button');
button.addEventListener('click', myAction);
```

In the example above, we define a function called `myAction()` that accepts a parameter called `event`. Then we select a `button` element from the DOM and add an event listener watching for the `'click'` event. This approach is clear and easy to read, and it allows for the same action to be attached to multiple events. In the case where you need to add a listener that would call the `myAction()` function to multiple DOM elements, this could be a more efficient and clear way to do so.

However, we often want to add a singular event listener to a single DOM element. In this case, many developers consider it more efficient and just as clear to create an event listener that calls an *anonymous function*. Here is an example of that approach.

```
let button = document.querySelector('button');
button.addEventListener('click', function(event){
    console.log('My action has been triggered!');
});
```

The result of this code is exactly the same as the code in the previous example: An event listener is added to a `button` element in the DOM that causes a function to be executed that, in this case, writes a log message to the console. We can understand why some developers prefer this approach because it uses fewer lines of code and it keeps all of the logic directly connected to the event listener contained within that code block. JavaScript makes a lot of use of anonymous functions in many situations, so this is not an unusual style choice.

Using Event Objects

Each event signal detected in the system triggers a call to the event handler that is defined in the event listener. The function that handles the event always receives an `Event` object as a parameter. This object can be used to access different information about the event. Here is an example of using information from the `Event` object within an event handler.

```
let button = document.querySelector('button');
button.addEventListener('click', function(event){
    console.log(`Action was triggered by ${event.target} at ${event.timestamp}.`);
    event.target.style.backgroundColor = 'goldenrod';
    let itemID = event.target.dataset.itemID;
});
```

There are several attributes, like `Event.timestamp` that can be used to respond to events signals, but the most common aspect of the `Event` object to use within an event handler is the `Event.target` object. `Event.target` is the DOM element object that triggered the event handler. In the example above, `event.target` is the `button` object.

Within the event handler, `event.target` can be used to access any information stored in the DOM element object. We can alter attributes of the object (such as the style alteration in the example code), and we can read in any attribute values (such as the `itemID`, which is read from the `event.target.dataset` object). This means we can use the same event to handle multiple elements on the page. Consider the following example of a system where favoriting content is possible.

html

```
<ul id="search-results">
    <li>Content Item Title 1 <a class="fave-link" href="#favorite" data-contentId="1">
Add to favorites</a></li>
    <li>Content Item Title 2 <a class="fave-link" href="#favorite" data-contentId="2">
Add to favorites</a></li>
    <li>Content Item Title 3 <a class="fave-link" href="#favorite" data-contentId="3">
Add to favorites</a></li>
    <li>Content Item Title 4 <a class="fave-link" href="#favorite" data-contentId="4">
Add to favorites</a></li>
    <li>Content Item Title 5 <a class="fave-link" href="#favorite" data-contentId="5">
Add to favorites</a></li>
</ul>
```

js

```
let faveLinks = document.querySelectorAll('.fave-link');
for (let link of faveLinks) {
    link.addEventListener('click', function(event){
        console.log(`Adding ${event.target.dataset.contentId} to user favorites.`);
    });
}
```

In the example above, we have some HTML that contains multiple content item listings. Each item has a "fave link" that allows a user to add the item to their "Favorites" list. Each of those links has a `data-contentId` attribute that stores the ID for the content item.

In our JavaScript, we have used a `document.querySelectorAll()` command to select all of the links, and then we loop through each link and add an event listener that will call an anonymous function. The anonymous function can then access the `event.target.dataset.contentId` property to determine which content item should be saved to the user's favorites list in the database. This allows us to use the same event listener to respond to event triggers in slightly different ways. Obviously we could not write a separate event listener for each content item in our database, so it's crucial to be able to pass in data using this method. The `Event.target` object allows us to make full use of the HTML `data-` attributes to pass data around in our web application.

Preventing Default Actions

Sometimes when we are responding to event triggers, we need to prevent the default action of the browser. This happens a lot when we are building more complex JavaScript applications that use elements like forms and links to provide user interface components. But it also comes up when we are using JavaScript to provide client-side validation of form

inputs or allow for users to do smaller changes to the interface. If we write enough JavaScript that uses event handling, we are eventually going to need to prevent some default behavior.

Preventing default behavior is as simple as including a single command in our event handler function: `Event.preventDefault()`. This command stops the browser from executing whatever the default action is for that event. Here is an example of using it to provide a form validation.

html

```
<form id="registration-form" action="register/" method="post">
    <p class="errors hidden">No errors to report.</p>
    <label>Username: <input type="text" value="" name="username" required></label>
    <label>Password: <input type="password" value="" name="pass1" required></label>
    <label>Confirm Password: <input type="password" value="" name="pass2" required></l
abel>
    <input type="submit" value="Register">
</form>
```

js

```
let myForm = document.querySelector('#registration-form');
let pass1 = document.querySelector('input[name="pass1"]');
let pass2 = document.querySelector('input[name="pass1"]');
let errorsParagraph = document.querySelector('p.errors');

myForm.addEventListener('submit', function(event){
    if (pass1.value != pass2.value){
        event.preventDefault();
        errorsParagraph.innerHTML = "Password values do not match.";
        errorsParagraph.setAttribute('class', 'errors visible');
    }
});
```

The example above shows the `event.preventDefault()` command used to stop a `form` from submitting when the password fields do not match. This kind of client-side form validation is helpful to a user because it provides an instantaneous feedback about whether or not they filled out the form properly. (**Note:** These sorts of validations are not a substitute for strong validation on the server or database side, too, but they provide a more enjoyable user experience.)

Our custom event handler will execute before the default actions take place in the browser, so we have a chance to determine whether or not we wish for those default actions to happen at all. If not, the `Event.preventDefault()` command stops those actions from

happening. This allows JavaScript developers the ability to truly alter the way the browser behaves in order to best serve their users.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Add a `submit` event listener to the `form` element using an anonymous function.

```
let myTest = false;
let form = document.querySelector('form');
// add event listener that executes `myAction()`
```

Exercise

Use the `event.target` object to change the `background-color` of the element by adding a `click` event to the `button` element.

```
let button = document.querySelector('button');
// add event listener that changes background-color of button
```

Exercise

Use a `preventDefault()` command to stop the `form` from submitting unless the user submits both `firstName` and `lastName` fields.

```
let myForm = document.querySelector('#registration-form');
let firstName = document.querySelector('input[name="first-name"]');
let lastName = document.querySelector('input[name="last-name"]');

// Add submit event listener to `myForm`.
// Verify that both `firstName` and `lastName` have been filled in.
```

Custom Events

Although we are often working with existing events that are triggered by elements in the DOM according to their specifications (things like "click", "submit", "mouseover", etc.), sometimes we want to create our own events based on actions in the system. We might have various processes happening in an asynchronous fashion, or we might want to alert people of new items available on a feed, or we could kick off a loosely coupled custom process based on our specific requirements. There are many reasons for creating a custom event, and they are not too difficult to manage.

General Purpose Custom Events

In order to create a custom event, we must first create a new `Event` object, and then we must create some listener to watch for our new event. When we wish to trigger the event we use the `document.dispatchEvent()` method, which also exists on all DOM elements.

```
let publishCompleted = new Event('publishCompleted');

let notification = document.querySelector('#notification');

notification.addEventListener('publishCompleted', function(event){
    event.target.innerHTML = "Publishing has completed.";
});

document.dispatchEvent(publishCompleted);
```

In the example above, we see that a new `Event` object is created for a `'publishCompleted'` event. This `Event` can now be dispatched whenever we need. The `notification` object is a DOM element that has an event listener watching for the `'publishCompleted'` signal. When it detects the `'publishCompleted'` event signal, it will trigger an anonymous function that changes the text of the `notification` element (since `notification` is the `event.target` in this case).

These kinds of custom events can help us build an event-driven system based upon loose coupling. But there are many events pre-defined in JavaScript that we can extend and use to fit our needs.

Using Built-in Events

There are many events defined in JavaScript that can be used to create custom event handlers. Here are a few:

- `AnimationEvent`
- `DragEvent`
- `KeyboardEvent`
- `MouseEvent`
- `UIEvent`
- `GamePadEvent`
- `ErrorEvent`
- `PointerEvent`
- `StorageEvent`
- and many more...

If we want to make an event listener that leverages more than the standard "click" style behavior, we can use these events to our advantage. We can intercept `MouseEvent` events and determine the `x, y` coordinates of the mouse pointer. We can listen for `KeyboardEvent` events to provide keyboard shortcuts for our users. Each of these `Event` types offers extra information and extra event signals that we can use in defining our own listeners. For example, the `KeyboardEvent` object makes a `keydown` event signal available. This signal is dispatched whenever the user presses a key on the keyboard. Here is an example of using `keydown` events to provide some interaction for the user.

```
document.addEventListener('keydown', function(event){
    console.log(`You pressed the ${event.key} key.`);
}
```

The code above will print a message into the console every time the user presses a key. One of the extra data points the `KeyboardEvent` event gives us is the `event.key` property, which tells us the key that was pressed. Other `KeyboardEvent` events that are dispatched include `keyup`, `keydown`, and `keypressed`. There are also many more properties that can help us build great keyboard interfaces.

For any family of events we wish to explore, we should read through the documentation so we understand exactly what events they signal and how those events are determined. It's useful to know, for example, the difference between a `keydown` and a `keyup` and the fact that most people want actions to happen on `keyup`, but not on `keydown` (because that's how most systems already work). There is no way to apply a general approach to all of these intricate details, so we must dive into each case on its own.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Add a `keyup` event listener to the `document` sets `keyCheck` to `true` when the `w`, `a`, `s`, `d` keys are pressed.

```
let keyCheck = false;

// create event listener

// determine which key was pressed

// if `w`, `a`, `s`, `d` keys are pressed, set `keyCheck = true`
```

Detecting Document Load Events

Early in this book we discussed the ways in which we attach JavaScript to a web page. We discussed the fact that we prefer to use external JavaScript files that are linked with `<script>` tags, and we mentioned that we should use either the `async` or `defer` attributes to prevent our JavaScript from "blocking" the load of other page assets.

Since the web is a dynamic place, and different resources for a page could be coming from all over the world, we can never know for sure how long it will take for a user to receive all of the page resources or how long it will take to parse everything on the user's unique hardware. There is no certainty of order of download for most page resources, and it's completely possible for unexpected thing to happen during the load process. It is possible, and sometimes likely, that your JavaScript file will finish downloading and begin processing before the HTML of your page has completed downloading.

As we've been manipulating the DOM and adding listeners to DOM elements, it may have occurred to us that in some situations we could receive errors if our JavaScript were running before the HTML of the page had actually finished downloading. In general, it is not safe to perform operations on the HTML of our web pages before the file has been completely downloaded and parsed into the DOM.

Luckily, there is a way of knowing when the DOM has finished loading the HTML content of the page.

The `DOMContentLoaded` Event

As we mentioned in the previous section, there are many events that are triggered by default in the process of viewing a web page. These events are being handled by default actions, but they are ignored by our code until we create a specific event listener. In order to prevent the problem of executing JavaScript that is trying to manipulate a partially-loaded DOM (because the HTML file has not completed downloading), we can set up an event listener for the `DOMContentLoaded` event.

The `DOMContentLoaded` event can be used just like any other event trigger (click, submit, etc.). We often attach an event listener to the `document` object that will execute our code when the `DOMContentLoaded` signal is sent. Let's imagine what that would look like in the context of an HTML-based game:

```
document.addEventListener('DOMContentLoaded', function(event){
    // Execute any functions that will manipulate the DOM
    setUpGameBoard();
    setUpScoreBoard();
    runGame();
});
```

We can imagine that this event listener is defined as the first block of code in the main JavaScript file for this web-based game. It is waiting for the `DOMContentLoaded` signal, and when that event is triggered it will execute a set of functions that appear to set up the game environment and kick off the execution of the game code. We can imagine that the `setUpGameBoard()` and `setUpScoreBoard()` functions do some significant DOM manipulation to create those structures. We can also assume that running the game will continue to alter those DOM elements as the game progresses.

Because so much DOM manipulation is happening, it's crucial for the full DOM to be loaded and available within the JavaScript context. If our code were trying to manipulate a partially-loaded DOM element, then we may run into errors due to some element not yet being loaded. By using an event listener with the `DOMContentLoaded` trigger we make sure that we never try to operate on a partially-loaded DOM.

Avoid the Load

The `load` and `DOMContentLoaded` events are often confused and used in appropriately. Most of the time, when we are concerned about whether a DOM element will have been loaded in order to be modified or selected, we want to use `DOMContentLoaded`. However, developers sometimes mistakenly use `load` instead.

Waiting until the `load` event is triggered could significantly impact your user because that event signal is not dispatched until after all of the resources for the page have loaded. This includes the images, audio, and video files that might be used in specific ways on the page. There could be a dramatic difference between time it takes to dispatch the `DOMContentLoaded` event signal and the `load` signal.

Exercises

Please try working these exercises to practice some of the skills we've learned in this section.

Exercise

Create an event listener that will watch for the `DOMContentLoaded` signal before it executes the `changeDOM()` function.

```
// create event listener on `document` object

// execute `changeDOM()` within event listener
```

Quiz: Handling Events

Try this self-check quiz to test your knowledge!

Visit Quiz Online

The quiz on this page has been removed from your PDF or ebook format. You may take the quiz by visiting this book online.

Extra Practice

Remember: Learning to program takes practice! It helps to see concepts over and over, and it's always good to try things more than once. We learn much more the second time we do something. Use the exercises and additional self-checks below to practice.

1. Save Buttons

Review the code, then answer the questions below.

```
let saveButtons = document.querySelectorAll('.save-button');

for (let button of saveButtons){
    button.addEventListener('click', function(event){
        let contentID = event.target.dataset.id;
        console.log(`Saving ${contentID}`);
    });
}
```

Visit Content Online

The content on this page has been removed from your PDF or ebook format. You may view the content by visiting this book online.

Conclusion

This is another one of those books where the end is really just the beginning.

We have covered the basics of what programming is, where it came from, and how we got to where we are now. We have done a whirlwind tour of the general landscape of software development and the kinds of programming languages used in various niches of the industry. We have considered common features of languages and which features are more defining than others.

We have jumped feet-first into the deep waters of JavaScript and ECMAScript 6, learning the latest methods and the most fundamental techniques. We have covered syntax, style, variables, data types, and other basics of the language. We have learned about conditionals and loops and functions. We extended our understanding into the realm of Object Oriented Programming and further enhanced the usefulness of all of these big concepts by practicing how to manipulate the Document Object Model and respond to user interaction.

Together, we have covered the basics that are required to do a remarkable amount of work with JavaScript in the web browser. We can use these skills to enhance web pages effectively and build complex interactive experiences. We can better understand code written by others and can begin to leverage their work to make ours better.

We have done a lot in a short amount of time. If you have come all this way with us, then you deserve congratulations.

But you also deserve encouragement, because we have only begun a journey towards proficiency with JavaScript. There are so many more things to learn, so many challenges to overcome, and so much work to do. We have not covered *everything*, and we have not learned it *all*. To name a few of the very important things that we have *not* covered, there are:

- Arrow Functions
- Importing and Exporting Modules
- Using AJAX design patterns to request data
- Working with localStorage and cookies
- Architecting complex web applications
- Optimizing JavaScript delivery and performance
- and so much more...

We must keep at it. This foundation will serve us well, and we are in a good position to build, grow, and develop into the creators we want to be. There are many resources out there that can push your knowledge further. Take full advantage of them to keep learning and you will accomplish all of your goals.

Resources for Learning More

As the author of this book, I recommend our Web Development Certificate program at Seattle University. We specialize in frontend development with a modern approach, and our program is designed to support mid-career change.

There are many resources online for learning more about JavaScript. Here are some links to useful sites. None of these sites are affiliated with this book.

References

- Mozilla Developers Network
- W3 Schools
- ECMAScript 6 Specification

Tutorials

- Learn ECMAScript 6
- ES6 Katas
- Introduction to ES6 from Code Mentor
- ECMAScript 6 Complete Tutorial

Other

- List of ES6 Learning Resources

Debugging Tips

Anytime we write code, we expect to encounter errors. In code, we call errors "bugs" and we call the process of tracking down and fixing errors "debugging". These bugs can come from syntax errors (when we misplace a comma, semicolon, brace, etc.) or from logical errors (when we make a mistake in calculating data or accessing a variable). Syntax errors are often the most frustrating for new developers because they consist mainly of things that are difficult for the untrained eye to spot. We have great tools for helping us find syntax errors, and once we get used to a language, those errors are not nearly as frustrating as they are to begin with. However, we still face the tricky task of figuring out our logical errors. Logical errors often require a more nuanced approach to debugging.

The process (and art) of debugging is enough to fill an entire book on its own, but we will review some methods that will often help find and fix bugs in our code.

Always Look at the Console

Whenever something isn't working properly in your JavaScript, always open up the developer tools in whatever browser you're using and check out the JavaScript console. The console will usually show what errors have been triggered by the page, and they will often give you a specific filename and line number where the error has been found. The console is an incredibly useful tool for developing JavaScript, so don't forget to consult the console (and your other developer tools) whenever something goes wrong.

Syntax Errors

Syntax errors consist of errors that are caused by some bad text in our code. Here are some common syntax errors:

```
let myString = 'Some text here; // Syntax error: no closing quote.
let myArray = [1, 2, 3,]; // Syntax error: trailing comma in Array definition.

if (x < 12) {
    console.log(x);

// Syntax error: conditional statement has no closing brace.
```

These errors are annoying, but they tend to be the easiest to fix.

Check the Console

Like the tip above says, we always begin by checking the console. In the case of syntax errors the console will usually lead you to the specific file and line number, often with a helpful link. Sometimes syntax errors actually exist on the line before the one that triggers the error, so be sure to look around the line noted in the console if you don't find the error right away. Here are some tips for where to look to resolve syntax errors:

- Look on the line (or lines) before the one indicated in the console. For many syntax errors they register on the line after the missing quote, semicolon, etc.
- When the console indicates the last line of the file, look closely at the curly braces and how we've closed all of the code blocks.
- Go through and fix up all of our indentation and spacing. We often find syntax errors when we clean up our style, and lining up our code blocks helps figure out where we've forgotten to close things.

Prevent the Error

If we use tools like linters in our editors, we can be alerted to syntax errors before we see them in the browser. Sometimes if we see a line number indicated in an error in the JavaScript console we can return to our editor and notice a warning or error that will help track down the error. Using tools in our editor to alert us to the simplest syntax errors is a great way to avoid needless frustration. Anticipate that everyone makes typos and mistakes, so it's no shame to rely on tools to help us discover those bugs.

Logical Errors

Logical errors are much more difficult to track down and fix. They involve a mistake in our logic that leads to an incorrect value or some erroneous logical flow. Because these errors do not trigger syntactical errors, and often do not trigger any error at all in the JavaScript interpreter, they can be very difficult to track down and fix.

The first step to fixing any bug of this sort is to **reproduce the error**. It is impossible to fix a bug that we cannot reproduce, and if we cannot reproduce a bug then we can never verify a fix. If the error happens every time, then we are lucky. The toughest bugs to fix are those that only show up in very specific (and possibly rare) circumstances.

When working to reproduce a bug, try to follow these steps:

- Note exactly what you did to cause the bug.
- Come up with a set of inputs or steps that lead to the bug and write them down.
- Work with a colleague to have them reproduce the bug on their own using your steps.
- If you cannot figure out how to reproduce a bug, be sure to capture as much information about your system and actions whenever you do manage to make the bug happen.

Remember: **If you cannot reproduce a bug then you cannot fix the bug.**

Once you can make a bug happen again, you can begin to track down a fix.

Console Logging

Throughout the code in this book we have used the `console.log()` command to output information to the JavaScript console. This is a valuable tool that can allow us to emit messages to the JavaScript console to track the movement of information and logic in our programs. We can use `console.log()` to help verify that things we can't otherwise "see" are happening. Here is an example:

```
let maxNum = 20;

for (let i = 1; i<=20; i++) {
    if (i % 2 === 0 ) {
        console.log(`${i} is even`);
        // additional program logic
    } else {
        console.log(`${i} is odd`);
        // additional program logic
    }
}
```

In the example above, the `console.log()` command is used to ping some information out to the console that we can use to verify our conditional is working properly. If we start seeing messages saying things like `"3 is even"` then we know that something is wrong.

Debugger

When `console.log()` is not enough, we can use the JavaScript Debugger, which exists in the developer tools for every major browser. When we use the debugger, we invoke a "breakpoint", which is like putting a big stop sign in our code. When the JavaScript interpreter hits the breakpoint, it stops and "freezes time" in the execution of the code. This can be very useful because we can then look at all the values of the code at that moment in the execution. This is extremely helpful in figuring out what went wrong.

Exactly how this works will vary just a little bit from browser to browser. Below I've linked to suitable tutorials covering the developer tools in all of the major browsers. But by way of an example, consider this:

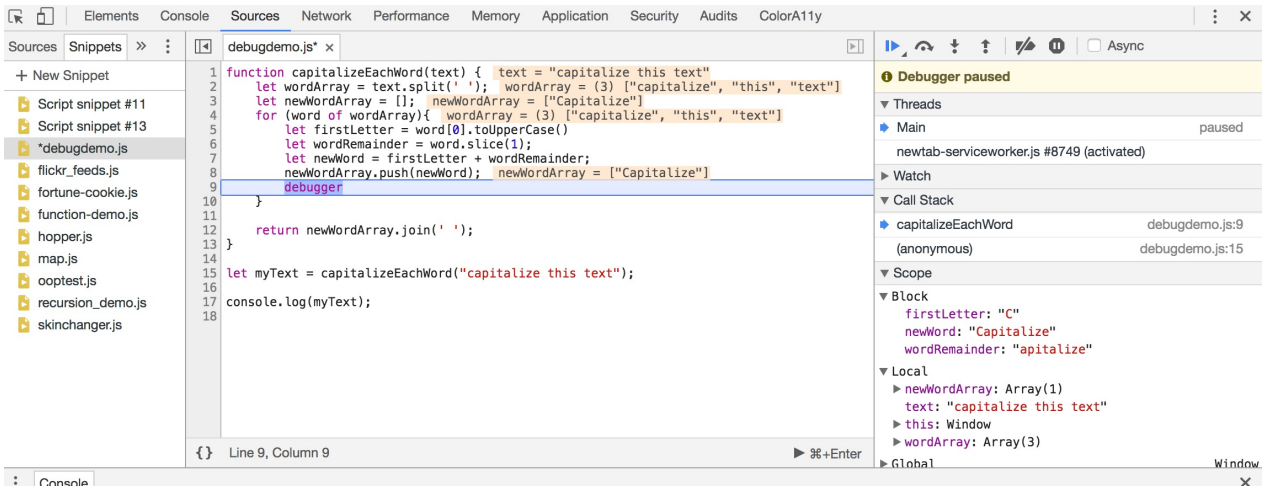
```
function capitalizeEachWord(text) {
    let wordArray = text.split(' ');
    let newWordArray = [];
    for (word of wordArray){
        let firstLetter = word[0].toUpperCase()
        let wordRemainder = word.slice(1);
        let newWord = firstLetter + wordRemainder;
        newWordArray.push(newWord);
        debugger;
    }

    return newWordArray.join(' ');
}

let myText = capitalizeEachWord("capitalize this text");

console.log(myText);
```

That code snippet has a `debugger` command in the `for` loop within the `capitalizeEachWord()` function. This will freeze the execution of the code like this:



Debugger Stopped at Breakpoint

In the image above we can see that the debugger has stopped at the line with the `debugger` command. This has caused the JavaScript debugger in Chrome Devtools to show us the current values of the variables in use around this line of code. We can take a closer look at these values in the "scope" panel:

Debugger Scope Panel

In the "scope" panel we can see the values of the variables available at the moment that the `debugger` line has been executed. We could alter these values using the JavaScript console and we can use the big "play" button at the top of the debugger window to continue executing code. Since our `debugger` command is in a `for` loop, it will pause the program each time it executes. In this way, we can watch the values change on each iteration of the `for` loop.

Learn Your Dev Tools

Although we can use the `debugger` command in pretty much any browser, there is always a danger that we could forget to remove that line from our code after we've figured out the problem. This is fairly common, and, as a result, developers tend not to use the `debugger` command to set a breakpoint.

> Every major browser provides developer tools of some kind, and these all allow you to set breakpoints by clicking on the line number in the code view within the developer tools. Setting breakpoints using the interface in our browser's developer tools is safer because we avoid making any changes directly to our code. It allows us to quickly set and remove breakpoints as we try to find and fix bugs. Consult the resources listed below for more information about how to get the most from our debugging tools.

Additional Resources

The amazing developers and community of people who make and use different tools to help with development and debugging also create a lot of educational resources. Check out these resources for more information about how to debug code.

Suitable for Beginners

Chrome

- Chrome Devtools Overview
- Get Started Debugging JavaScript with Chrome Developer Tools
- Pausing Your Code With Breakpoints

Firefox

- Firefox Debugger Overview
- Debugging JavaScript

MS Edge and IE

- F12 Devtools Guide (MS IE and Edge)

More Advanced Resources

- JavaScript Debugging Reference
- JavaScript Debugging Tips

anonymous function

A function that is defined as part of a method call or object definition such that it does not take on a name of it's own. Here is an example of an anonymous function used in an `addEventListener()` method call.

```
let button = document.querySelector('button');
button.addEventListener('click', function(event){
    console.log('Click event was triggered.');
});
```

compiled language

A description of a programming language that uses a "compiler" to create an executable file.

compiler

An application used to convert source code in a given programming language into an executable file that can be run by computing hardware. (See also: compiled.)

default actions

An action executed in the browser in response to a page event (e.g. click, submit, load, etc.).

dynamic language

A description of a programming language that infers or determines the datatypes of variables without requiring explicit declaration.

Ecma International

The governing body that stewards the ECMAScript standard, upon which JavaScript is based.

event handler

A function that is associated with an event listener and which is called when the event is triggered.

executable file

A file that can be run directly on the computing hardware without requiring any interpretation or compilation at runtime.

global

A global object is available at every scope and at every line of code. In JavaScript, the `Window` object is an example of a global object. Variables may be declared at a global or local scope level.

hoisting

A feature of JavaScript where variables and functions may be referenced before they are declared without error. (See this article on MDN for additional details.)

interpreter

An application used to convert source code in a given programming language into machine instructions that can be executed by the computing hardware.

iterable objects

JavaScript objects that implement the Iterator Protocol so they can be iterated over including Arrays, Maps, Objects, Symbols, and more.

linter

A tool used in a code editor to check syntax and style, usually in real-time. Most popular code editors support linting tools such as JSHint and ESLint. Refer to the documentation in your chosen editor for more information about installing and using linters for whatever

language you are writing.

local

Local objects are only known within the scope where they are defined. This is typically limited to a code block between two curly braces. Local objects are often used within functions, loops, and conditionals for specific tasks, and they may be named with generic names that are re-used in other local scope contexts.

method

A function related to a Class object. Class objects are made up of data properties and methods that perform some behavior or action. The methods are the "verbs" of the Class.

named function

A function that is declared with the `function functionName(){}` syntax. This function has a named reference that is available within its scope and context.

property

A piece of data related to a Class or object. Objects are made up of data properties, which contain information describing the object, and methods, which perform some behavior or action. The properties are the "adjectives" of the object.

scope

The context of the code, including all known object, function, and variable references, as it is executed by the interpreter.

scripted language

A description of a programming language that uses a runtime interpreter to convert source code into an executable file.

source code

The code a developer types in a given programming language using some form of text editor. This code is typically either compiled into an executable file, or it is interpreted at runtime by an interpreter.

typed language

A description of a programming language that requires developers to explicitly declare the datatype of each variable they use.