Follow @derekknox

Chapters

+

Light
-

Chapter 4 Interactive Code

S
o far we've broken down some barriers around coding and computers in general. We've learned about coding structure, style, and behavior too. Additionally, we've introduced programming relative to visual design's elements, principles, and constructs. Though there is always more to learn, your confidence should be magnitudes higher than when we started. Additionally, a mental model should be forming. Let's strengthen it.

To do so, we will visit one more set of concepts before getting into JavaScript itself. Virtually all of these concepts are language agnostic. You are welcome.

In this section, we will visit the most important programming concepts that will help shape your future programming efforts. We will talk about the three main and distinct timeframes when code exists. These are *authoring time*, *compile time*, and *execution time*. This knowledge will help you understand the transition from authoring code to interactive creations people can use. Following this, we will visit the *frame rate* and *event loop* concepts in detail so we have a fine-tuned grasp of what actually happens during *execution time*.

Next we will explore the differences between *sync* (happens now) and *async* (happens later) behavior. This exploration will naturally tie into visiting *interfacing* and the *client-server model*. As a result, we will learn how code talks to other code while also learning the specific benefits of having multiple computers communicate and do dedicated and unique work.

Second to last, we will look at the *anatomy of HTML (`.html`), CSS (`.css`), and JavaScript (`.js`)*. This will allow us to reference real code against previously covered concepts. Lastly, we will cover the *work. right. better.* mantra to help you improve your code authoring over time. All coders live by this, myself included.

Upon completion of this chapter, your mental model will be primed for programming in JavaScript. Let's get going.

#

Authoring, Compiling, and Executing

As mentioned previously there are three distinct timeframes when code exists. In order, these are:

1. Authoring time
2. Compile time
3. Execution time

These distinct timeframes are useful for understanding how the static code we author in a text file enables a dynamic and interactive creation to exist in real-time. We will also touch on why this transformation even needs to occur.

Authoring time is pretty self-explanatory right? It is the time when we are authoring code. That's it. We have a text editor program of some kind open and we are typing or pasting into it. Simple as that. Some programs even exist—like Webflow and Unity for example—that enable a more visual programming approach. This simply means that the program has some amount of UI interaction that enables you as the author to *generate* code in a non-text based way. Admittedly, this is often more fun than text-based programming... but there is a catch.

Inevitably, these visual tools are limiting as they do not allow you to pull off *exactly* what you want. This is why we're concentrating on authoring custom code in a text editor. We get *ultimate control* as to how something works because *we get to write the code*.

After we have authored the code, we need to "run" or "execute" it. Executing it helps us confirm that it actually does what we want in real-time. Before we can do this however, a step needs to be taken that transforms our static authored code into code that can be executed. Welcome to *compile time*.

Compile time is how we get from our authored *static* code to *dynamic* code that manifests as an interactive application or game. We won't go into the details of this step as it is complex. All you need to know is that this step exists so *we* can use words *we* understand instead of coding in binary, the only thing computers fundamentally understand. A prime example is how we use English words (and English abbreviations) within the programming language JavaScript.

In essence, smart and nerdy authors write *compiler programs* that compile and convert a high-level programming language like JavaScript (easier for us humans to use) to lower-level languages (easier for super-nerdy humans to use) to binary (easier for super-super-nerdy humans and computers to use).

Author Compile Execute

The output of this compilation process results in code that the computer *executes* in real-time. This process often manifests as an interactive user interface in an application or an interactive environment in a game.

With Unity, the process is:

1. *Author* code in a text editor or with Unity's built-in UI that generates code
2. Click the "Build" button to *compile* the authored code into an executable package
3. *Execute* this package file's code on the target operating system in real-time
    - Remember file extensions?
    - Remember decoding?

With JavaScript in a web browser, the process is:

1. *Author* code in a text editor or a tool that helps generate code
2. View a web page that hosts your authored JavaScript code, the browser's JavaScript engine *compiles* it on the fly
3. This compiled code is then automatically *executed* in real-time

To recap, we write in a high-level language using English words and abbreviations. This is author time. Then a compiler program converts the code we humans can understand into code the computer understands. This is compile time. Finally, if the conversion process has no errors, the result is an application or game that is interactive. This is execution time.

How is execution time actually interactive though? One or more users could take any number of actions within our application or game at any given moment. We can't possibly handle all these potential scenarios in our code can we? Thankfully the answer is yes we can. We just need to understand the *frame rate* and the *event loop*.

#

Frame Rate

We know computers are dumb and fast. Some computers are faster than others however. How do we ensure the code we author will execute well on any given computer? To do so, we need to author our code with a target *frame rate* in mind.

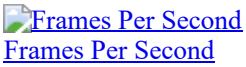
A frame rate is simply the amount of frames that are shown in a second. A frame is a single rendered image. Frames shown back to back over time create the illusion of motion. Common target frame rates are:

- 30fps (video)
- 60fps (apps/games)
- 90fps (AR/VR)

As you likely guessed, a high frame rate results in the illusion of smooth motion. A slow frame rate results in an illusion of less smooth motion. The human eye typically needs 24fps or more for a smooth perception of motion to be recognized. Within an interactive medium like applications or games, 60fps is a common target.

It is worth noting that if your application or game does not change often or lacks animation, you will likely be able to get away with 10fps, 5fps, or lower. The more motion and animation you have however, the more likely you'll want to target 60fps. Generally, the richer and more fluid the experience, the greater the target frame rate.

An application or game that targets 60fps must render a new image every 16.6 milliseconds. Damn, that's fast.


Frames Per Second
Frames Per Second

Milliseconds might be intimidating at first, but you'll get used to them quickly. In fact, you likely already have an intuitive understanding of milliseconds. Do you ever feel an animation within an application or game is too slow or too fast? The milliseconds chosen as the animation time is the culprit.

You usually don't have to worry about the 16.6ms window however. I present it here simply for you to keep it in the back of your mind.

In the future, if your application's animation and motion seems to have hiccups or it looks janky, then you likely have a problem or two. Typically this means your code, or a third-party's code, is:

1. doing too much work each frame
2. doing work inefficiently

We'll revisit these two problems later, but just be aware of them.

So a frame rate is vital for perceived motion, that's great. But how do we actually allow interactivity to be reflected in this motion? *Event loop* time.

#

Event Loop

An interactive application or game is simply the manifestation of a target sixty rendered images, shown in sequence, per second. The only reason this frame rate is important is so the user's experience is perceptually fluid and responsive. If fluidity and responsiveness were not a perceptual goal, our application or game could leverage 10fps, 5fps, or an even lower target frame rate. More often than not however we desire the interactive content we create to be extremely fluid and responsive. 60fps? Yes please.

A non-interactive animation hitting 60fps undoubtedly looks smooth, but it is just that, *non-interactive*. How do we account for this desired interactivity?

Thankfully, the program responsible for *executing* our compiled code helps us out. This program is called the *engine*. Additionally, the engine gets help from its parent program, the *runtime*. Together, the runtime and engine provide an interactive system that executes code in real-time. If we were to code the relationship in an HTML-like code, it would look like this:

```
<runtime>
    <engine></engine>
</runtime>
```

For us to take advantage of the runtime and engine, we just need to get an idea of how they work together. This allows us to author code that reacts interactively as the engine executes it in real-time.

The JavaScript runtime in a web browser has four core parts and the engine has two:

1. engine
    - *heap*
    - *stack*
2. *runtime APIs*
3. *event queue*
4. *event loop*

We can update our HTML-like example from before:

```
<runtime>
    <engine>
        <heap></heap>
        <stack></stack>
    </engine>
    <apis></apis>
    <queue></queue>
    <loop></loop>
</runtime>
```

So what do each of these runtime parts do anyway? Here is a succint breakdown where the use of *work* implies some amount of code reading or executing:

1. engine - does work
    - *heap* - shared memory for work

    - *stack* - organizes the engine's work
2. *runtime APIs* - does special work the engine cannot
3. *event queue* - organizes the results of special work as packages of engine work
4. *event loop* - gives the engine queued packages of special work

During compile time, just before execution time, the engine quickly does three things with our authored code:

1. reads
2. reorganizes
3. optimizes

These steps enable the engine to run fast and efficiently during execution time. Execution time is when the runtime and engine work together to achieve two goals:

1. clear the stack
2. clear the event queue

Once these two goals are met, the engine can relax. It relaxes until new work is added to its stack. How does it get new work though? Do you remember the input triggers we covered in the *Behavior* section? Bingo. As a reminder, these input triggers, often called *events*, are:

1. User interaction (tap, click, hover, gesture, voice, etc.)
2. Environment (layout resizing, operating system, device sensors, etc.)
3. Time (delays, schedules, etc.)

When an event occurs, the runtime APIs manage the special work required and then update the event queue when finished. Since the event loop has been cycling while the engine was relaxing, it now notices the updated event queue. Consequently, it takes one item from the queue and gives it to the engine. The engine then puts it on the stack. You guessed it, new work for the engine. This process cycles.

Though all of the runtime's parts are vital, the event loop most ensures our interactive creations come to life. Thank you event loop.

#

Sync and Async

Though we didn't explicitly state it, the nature of how the event loop works gives rise to the notion of *sync* and *async* work. Synchronous and asynchronous are the full terms, but we will use the shortcuts sync and async instead.

Sync work is referred to as *blocking* where async work is *non-blocking*. Using your new knowledge of the event loop, how would you categorize the engine's stack work? How about the runtime APIs work? Take a moment before continuing.

Answer time:

- Sync
    - blocking (stack work)
- Async
    - non-blocking (runtime APIs work)

The event loop cannot cycle when work is on the stack. The loop is blocked. The event queue does not prevent the event loop from cycling. The loop is not blocked.

Let's look at code examples of both work types to see this in action. We haven't specifically covered JavaScript outside the brief introduction in the *Elements and Elements* section, but I am confident you'll get the gist. At the very least use this time to test your assumptions.

Sync work:

```
function makeBackgroundBlack() {
    document.body.style.backgroundColor = '#000000';
}

makeBackgroundBlack();
```

There are likely some details you don't understand, but that is to be expected. Let's look at the same code with some added explanations. Take note that the words following the `//` are plain English, not code. These plain English *comments* are useful for us humans when we read and share our code for others to read (including our future self). Comments are for humans not computers. The engine essentially ignores them.

```
// 1. We declare a function statement (stack work) for the engine
// 2. We name the function whatever we want, `makeBackgroundBlack` in this case
// 3. The engine doesn't do the work yet (code statement between the `{` and `}`)
function makeBackgroundBlack() {

    // We set the document's background color to black (remember the Coding Color section?)
    document.body.style.backgroundColor = '#000000';
}

// 1. We tell the engine to do work, the code sequence `()` is the trigger
// 2. Since we named the function, the engine knows the exact work to put on its stack
makeBackgroundBlack();
```

Without comments, there are naturally a lot less lines for us humans to read. As far as the engine is concerned however, these are the same program. As you gain more experience you will understand when and when not to use comments. When in doubt use them to help other coders, including your future self, understand your original intention.

It's worth mentioning that the descriptive function names used in this book are intentional. From the engine's perspective, a function named `x` would work as well as `makeBackgroundBlack`, but the descriptive alternative is much more useful to us humans.

Admittedly, the code above does not do a whole lot. The work inside of the `makeBackgroundBlack` function will happen really fast too. The event loop will technically be blocked for less than 1ms, but from a user's perspective the work will happen instantly and then the event loop will become unblocked. If the function instead counted to a million before changing the background color, then the user would percieve the program as slower (and rightly so).

The takeaway here is that you should try to do small and efficient work in your functions. As you author code over time, you will develop an intuitive understanding of what that really means. For now, put this idea in the back of your mind. Remember, the faster the stack and event queue are cleared, the greater the chance you will hit your target frame rate. 60fps? Yes please.

Let's now look at an async example. We will make it similar to the sync example for comparison. In fact, we will make the program identical except for a single statement and its comments.

Async work:

```
function makeBackgroundBlack() {
    document.body.style.backgroundColor = '#000000';
}

// 1. We tell the engine to do work, the code sequence `(makeBackgroundBlack, 1000)` is the trigger
// 2. `setTimeout` is a named built-in function that the runtime APIs provide, thank you runtime
// 3. The `setTimeout` function expects two argument values
setTimeout(makeBackgroundBlack, 1000);
```

Before reading on, try to guess what `setTimeout` does. How do you think it uses the argument values?

This async program is identical to the sync program except for one code statement. If we look up how the `setTimeout` function uses its arguments to do work, we can learn what work it does. I want to stress again that professional coders—just like beginners—use references to learn and remind themselves what certain functions do. Even the best cannot remember everything.

I will just tell you what arguments, in order, `setTimeout` expects:

1. a function
2. a time in milliseconds

The work `setTimout` actually does, in English, is:

1. create a timer
2. run the timer for the time in milliseconds provided
3. wait for the timer to complete
4. upon completion, update the event queue with the function provided

What is cool about many functions, `setTimeout` included, is that they can be designed with an amount of flexibility built-in. As you may have guessed, as long as we give `setTimeout` valid arguments, it will always do the work we want (via the function provided) after a delay (via the time provided). Pretty damn cool. You can design your own functions with flexibility as well.

As a quick example, let's design a new function named `changeBackgroundColor`.

```
function changeBackgroundColor(newColor) {
    document.body.style.backgroundColor = newColor;
}
```

Here are a few examples of how we could use it:

```
changeBackgroundColor('#FF0000');
```

or

```
changeBackgroundColor('00FF00');
```

or

```
changeBackgroundColor('0000FF');
```

You get the idea.

We explored some JavaScript a little bit in this section, but the main takeaway is the difference between sync and async. If you understand how to keep your stack work small and fast, you'll be running 60fps with no problem.

As you learn more about runtime APIs you will discover that the runtime will do a lot of really cool non-blocking work for you. If you haven't already noticed, the runtime APIs provide a simple *interface* for you as a code author to work with. Let's dig deeper into what that means.

#

Interfacing

It wasn't important until now to tell you what the aforementioned *API* of runtime APIs stands for. Now is the time. Welcome to *Application Programming Interface*. Yea, that is a mouthful. Coders instead just say API (each letter pronounced individually).

An API is just an interface. The doorknob of a door is an interface. The steering wheel of a vehicle is an interface. The keys of a virtual keyboard are too. Interfaces provide a way to interact. The doorknob and steering wheel are physical interfaces designed for humans. A virtual keyboard is a digital interface also designed for humans. An API is a programming interface designed for code and humans. An API allows code to interact with other code (where humans can read and understand it), this is the takeaway.

As a code author, you get ease-of-use while the underlying complexity is the code designer's responsibility. Think about it. As a driver of a vehicle, you do not need to know all the details of how the engine works to use it. The designers take that responsibility. Similarly with code, you just use the API provided to gain benefit. This is a pretty solid trade.

In the last section we saw one API in action, the built-in `setTimeout` function. We wrote our code to interact with that API by giving it two argument values. Behind the scenes, the runtime wired up and provided the functionality we expected. Both you and I don't really care how the work is done as long as it gets done. This is the trade-off an API provides, a simpler way to do work. APIs rock!

Earlier in the *Constructs and Components* section we covered built-in, third-party, and custom components. Naturally, they each provide various APIs. As previously mentioned, we can create useful code for others to use too. Another author using our code APIs doesn't have to care about how the work actually gets done. They get the same trade-off as us, a simpler way to do work.

As you gain coding and programming experience, you'll memorize certain built-in and third-party APIs. This translates to improving your authoring speed. Don't worry about being slow when you start out, you'll get better in time.

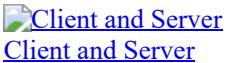
This might sound weird at first, but some APIs exist on a different computer than the one your program runs on. Whoa. Have you ever wondered how a particular app or game can provide new content without actually updating? Welcome to the *client-server model*.

#

Client and Server

Clients and servers are computers that communicate over a network. The internet is one such network that facilitates this communication. Some of these computers are clients and some are servers. Sometimes a computer is both a client and a server. The distinction is simple. A client asks for data and the server provides it. The network between them is the communication channel.

The details of how the internet facilitates this communication is elegant and impressive, but not a focus of this book. I highly recommend *Introduction to Networking* by Charles Severance if you want to dig deeper, but it is not required reading to move forward. Just know that the internet enables a client computer and a server computer to talk.


Client and Server

Let's look at two example programs that utilize the internet to help solidify your understanding. One program will be a web browser and the other an internet-connected mobile game. Let's just say that each example is running on a smartphone. Both the browser and the game are on the client, the smartphone. Each program leverages distinct APIs to request information from distinct and remote computers, the servers. Pretty simple relationship really.

You already know that a web browser may be used to access a particular website or web app. While the site or web app is running, its code can make *additional requests* to a server in an effort to get additional data. For example, when a sport statistics app is starting up—or another input trigger occurs—the app could use an API to get the most up-to-date scores and stats. An updated version of the app is not required. Instead, an API interaction provides updated data. This is super useful.

An internet-connected game works exactly the same way. When it starts up—or another input trigger occurs—it could make an API request to get current world-wide player rankings for the game. Additionally, if the designers and code authors designed the game in such a way, they could make different API requests for new worlds, levels, characters, etc. No update required. Creativity and experience are the only limiting factors. Pretty damn cool.

A server, if you didn't guess already, is also responsible for providing a client web browser the *initial* HTML, CSS, and JavaScript for a website or web app. If too many people try to access the same data at the same time however, the server can crash. It is worth noting that each client is actually getting a copy of the data to reconstruct in its browser. Typically a crash occurs because the server program can't properly provide the various clients the copies fast enough. I mention this simply because many people don't grasp the fact that the client reconstructs what the server provides.

Having talked about HTML, CSS, and JavaScript at solely a high level thus far, now is a great time to venture deeper. We'll do so next by exploring the anatomy of the three file types `.html`, `.css`, and `.js`.

#

Anatomy of HTML, CSS, and JavaScript

This section will be the first where we really start digging into the existing languages of HTML, CSS, and JavaScript. We have covered quite a bit in an effort to get to this point. Specifically, we learned about the powerful concepts of binary and states in addition to encoding and decoding in the *Breaking Barriers* chapter. We learned in *Structure, Style, and Behavior* how each concept plays a valuable and distinct role in an interactive creation. In *Programming and Visual Design* we mapped familiar visual design concepts to programming concepts to help establish a knowledge bridge. More recently, we explored programming-specific concepts to help shape your understanding of how code lives and communicates in real-time.

We know code is a system for converting meaning between forms. HTML, CSS, and JavaScript are just specific forms. HTML is a markup language used to define structure. CSS is a style language used to define style. JavaScript is a programming language used to define behavior. The web browser knows how to decode HTML into divisions of content and render them. It also knows how to decode CSS property-value pairs to adorn the HTML elements with style and functionality. Lastly, we know a browser's runtime and engine understand how to decode and execute JavaScript which enables an interactive creation to exist.

We will now start to dig into HTML, CSS, and JavaScript since we have a greater understanding of how the various concepts we have covered play-off each other. Your mental model should be primed.

Let's squash a myth real quick first. Designers and other non-coders new to coding think they need to learn an *entire* new language including all its words, syntax, and idiosyncrasies.

- Word - you know what a word is
- Syntax - language-specific rules of meaningful word and character patterns
- Idiosyncrasies - exceptions to rules

In the context of the web, that would mean *three* sets need to be learned. There is a better way to spend time and energy learning however. What really needs to be learned is:

1. Distinction between structure, style, and behavior
2. Common syntax subset of each structure, style, and behavior language
3. Common words subset of each language (80/20 rule)

For 2D and the web, that means we need to know the difference between HTML structure, CSS styling, and JavaScript behavior. We have this distinction covered already. Bonus. Let's dig into number two and three. Learning these two subsets for each language may not be easy, but it can be simple. In fact HTML and CSS have a simpler learning curve compared to JavaScript. As such, I will briefly cover the former and let other resources guide you following this chapter. For JavaScript however, the next chapter *80/20 JavaScript* will be your go-to guide. Regardless, there is no substitue for practice.

#

HTML

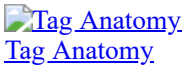
In keeping with the structure, style, and behavior order, let's look at the anatomy of each starting with HTML. We will title our file `index.html` to align with a best practice naming convention. Technically you can name it anything your OS allows, but make it easier on yourself and follow this best practice.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Coding for Designers</title>
    </head>
    <body>Content goes here.</body>
</html>
```

This is the most structurally simple, complete, and valid HTML a browser uses. Let's walk through each tag using *comments* to clarify each tag's purpose. We saw earlier that JavaScript uses the `//` character sequence for comments. HTML uses the `<!---->` sequence where the actual comment rests in the middle of the dashes. Just take note that each language may have a uniqe approach to comments, but the purpose is the same. Yes, it would be ideal if all languages used the same character sequence for denoting comments. It's not a perfect world. Sad face.

```
<!-- Tell the browser the document type - we only care about html -->
<!DOCTYPE html>

<!-- Tell the browser where we've defined our html - additional data exists outside it -->
<html>

    <!-- Define the page metadata - useful data for the browser, other programs, and search engines -->
    <head>

        <!-- Define the character encoding - inform the browser how to reliably decode this document -->
        <meta charset="utf-8">

        <!-- Define the page title - useful for the browser, other programs, and search engines -->
        <title>Coding for Designers</title>

    <!-- Define the page metadata end -->
    </head>

    <!-- Define the page content - what gets rendered for viewing and user interaction -->
    <body>Content goes here.</body>

<!-- Define the html end -->
</html>
```

There are many types of element tags that can be added in both the `<head>` and `<body>` element tags. As mentioned earlier, we will not go over them here as you, just like other coders, can use other resources to look up what tags are available. We will reveal the most common ones for the `<body>` however to fulfill the *common subset* bullet above. Instead of comments I will use English inline with the tags to show how the elements wrap content. Content is visible to the user where the tags that wrap the content are not. The tags simply tell the browser your structure, the building block parent-child relationships. The spacing *between elements* is not important as the browser understands the structure regardless. Spacing *between content* is what matters.


Tag Anatomy
Tag Anatomy

Spacing, tabs, and new lines do typically exist between elements however. This is solely to improve human readability. Imagine if the below HTML elements were all on a single line. The browser wouldn't care as the defined structure would remain the same (thanks to the opening and closing HTML tags), but we would have a much tougher time reading and easily seeing the parent-child relationships when authoring.

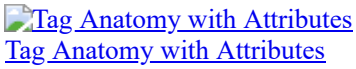
```
<body>
    <div>
        <h1>This is the Primary Title of the Page</h1>
        <p>This is a paragraph. It is followed by an image and a button.</p>
        <img>
        <button>Toggle Image Opacity</button>
    </div>
    <div>
        <h2>This is a Secondary Title</h2>
        <p>This is another paragraph. It has a <a>hyperlink</a></p>
        <p>A <span>span</span> helps style certain text in a paragraph.</p>
        <button>Or even in a <span>button</span></button>
    </div>
</body>
```

The first thing you will notice about six of the eight tags is that they are abbreviations, they are shortcuts. This admittedly sucks for beginners, but it is great for when you know the small subset by heart. You will learn it quickly, but it still is lame from a beginner perspective. Here's a breakdown to help:


HTML Tags Subset

A `<div>` tag simply denotes a division of content where the aesthetic and layout (position and dimension) of each division is dependent on style. This will likely become your most used element tag. Combining it with the virtually infinite variations of CSS styling enables you to create the same visual effect of virtually every other tag. If not for the specific *attributes* and *semantics* of certain tags that browsers, operating systems, and search engines use, we could almost get away with solely using `<div>`s with custom styles. I mention this to illustrate that a lot can be achieved with little.

Moving on, the `<h1>`, `<h2>`, `<p>`, `<span>`, and `<button>` tags are explained and covered within the inline examples above. The `<img>` and `<a>` tags are not however. This is because they are each lacking *attributes*. We'll use the shortcut *attrs* moving forward. All elements can leverage attrs, but these latter two elements *require* them to work as designed.


Tag Anatomy with Attributes

Think of attrs as APIs. Attrs enable the coder to achieve functionality based on a shared understanding of designed use. They are also where we start to transition toward CSS via the `class` attr. First we will update the `<img>` and `<a>` tags with each of their most useful attrs.

So `<img>` becomes `<img src='assets/img/cover-coding-for-designers.jpg'>` where the `src` attr is set with `=` to the value `assets/img/cover-coding-for-designers.jpg` between quotes. Quotes define the start and end of the attr value. The browser knows how the image tag is designed to work so its runtime APIs automatically download the image provided by the `src` attr, which is the path to the asset. Upon completion of the download, the tag embodies the downloaded image. This same functionality can be achieved using JavaScript only, but we'd have to write more code. Remember, APIs provide a tradeoff and a simpler way to do work.

Additionally, `<a>hyperlink</a>` becomes `<a href='https://www.CodingForDesignersBook.com'>hyperlink</a>` where the `href` attr is set with `=` to the value `https://www.CodingForDesignersBook.com` between quotes. The browser knows that the anchor tag, when hit, should change the web page to the value provided. This same functionality can be achieved using JavaScript only, but we'd have to write more code. Again, the goal is a simpler way to do work.

The takeaway is that the browser provides useful runtime API hooks via element attrs. Now that we've covered the basic implementation of HTML structure, let's dig into style. Welcome to the `class` attr.

#

CSS

The `class` attr gives you the creative power to style content. We will start by giving each `<div>` the same style. So `<div>` becomes `<div class='dark-background'>` where the `class` attr is set with `=` to the value `dark-background` between quotes. The `dark-background` name could be virtually anything we want, but there are rules for valid names. Make it easier on yourself and stick with:

- Lowercase English alphabet characters
- Use `-` instead of spaces

Now that we have set a `dark-background` value for the `class` attr, how do we actually define the style for the browser to render it? Remember when I mentioned that other tags can be added to the `<head>`? Welcome to the `<link>` tag. It allows us to link another file to our web page. Perfect. Our updated `<head>` is below and it now has a `<link>` tag. The attrs of it tell the browser to download and use a CSS file where our defined `dark-background` class definition resides.

```
<head>
    <meta charset="utf-8">

    <!-- `rel` defines the href relationship type - `href` defines the hyperlink reference -->
    <link rel="stylesheet" href="assets/css/style.css">

    <title>Coding for Designers</title>
</head>
```

The folder structure on the server computer that is providing these files to client computers is below. Take note that a lacking file extension means the item is a folder. The *relative* paths of the `<link>`'s `href` attr value and the `<img>`'s `src` attr value rely on the structure below. These relative paths—relative to the `index.html` that is—tell the browser exactly where to find the file to use.

```
index.html
assets
    css
        style.css
    img
        cover-coding-for-designers.jpg
```

Now is the time to look at the anatomy of a `.css` file which, for our purpose, consists of one class definition. Typically the file would have more than one, but currently we only need one. Take note that CSS uses *comments* via `/**/` where the comment itself resides in the middle. I too wish HTML, CSS, and JavaScript shared the same comment syntax, but I digress. Our `style.css` file is simply:

```
.dark-background {

    /* This is a CSS comment - the below property-value pair sets an element's background color to black */
    background-color: #000000;

}
```

Each class definition is denoted by a preceding `.` so our `dark-background` class becomes `.dark-background` within our `.css` file. Then, similar to the JavaScript scope concept we visited earlier, the property-value pairs are defined between the `{` and `}` characters. The `{` and `}` define the scope—the space and enclosure—where one or more property-value pairs are associated with a particular class.


Class Anatomy
Class Anatomy

Each property-value pair consists of a property name that uses the same rules for class naming suggested above, followed by a `:` and then an actual value. For ease of learning, I wish the `:` was `=` instead, but I again digress. With a class and its valid property-value pair(s) defined, any element that has a `class` attr value by the same name will get the styles applied. Pretty damn cool and very reusable.

If it isn't already obvious, each browser sets *default styles* as a baseline. Without default styles we would see nothing until we provided our own. So browser's provide defaults so we at least can render something before starting to customize. Think of default styles as a `browser-specific-style.css` that the browser provides each page behind the scenes. Defaults apply style based on element type instead of by the `class` attr however. More often than not, you'll want to set your own styles to override these defaults. This is exactly why we created `style.css`.

Since browsers typically default the color of text to black, our applied `dark-background` class makes our text blend with the dark background. We want to see the text too so we can fix this by updating our class definition to use another common property named `color`. Again, we won't go over all the valid properties that are possible as you—like other coders and designers—can reference resources. The takeaway is the relationship of property-value pairs within class definitions. Here is one approach to update our `style.css` file:

```
.dark-background {
    background-color: #000000;
    color: #ffffff;
}
```

Another approach could be:

```
.dark-background {
    background-color: #000000;
}

.light-text {
    color: #ffffff;
}
```

Our `<div>`s would need to be updated for the second solution. Take note that spaces are used between class names to enable more than one class style to be applied to a single element. This is a very powerful aspect of CSS, one which you will grasp more intuitively in time. Updated HTML using the second solution is below:

```
<div class='dark-background light-text'>
```

With either solution, both our `<div>`s will have dark backgrounds and the text within them will be light. You might wonder how the text of the child elements can be light without explicitly setting styles on them. This is where the *cascading* part of cascading style sheets comes in. Think of cascading as style inheritance. Long story short, a majority of property-value styles can be inherited regardless of nested element depth. Cascading is extremely powerful and you will learn its nuances in time.

As a creation evolves, feel free to change the names of classes if it makes sense to. A class name may make less sense as new property-value pairs are added or removed. Over time you will leverage established naming practices and develop your own to gain an intuitive feel of when to update a name. Additionally, you'll develop a sense of how best to group certain property-value pairs for reuse by different elements. Just know that it is extremely common to change names as a creation evolves. We are working in the virtual not physical world after all.

#

JavaScript

Thus far we've covered the anatomy of HTML and CSS, so now it is time for JavaScript. In a way we already looked at the anatomy of JavaScript in the *Programming and Visual Design - Elements and Elements* section. We'll dive even deeper in the next chapter *80/20 JavaScript*. For now we will look at the high level anatomy of a JavaScript program. We will focus on shape, space, and form or more precisely function, scope, and object.

As you already know, a function is like a shape because it encloses scope just as a shape encloses space. An enclosure helps prevent clashing of what exists between different scopes or spaces. No clashing please. Objects, like forms, are a higher level enclosure. More often than not, they each embody one or more functions or shapes respectively. JavaScript, like the majority of programming languages, has specific *types* of objects. JavaScript is fairly unique however in that a function is *also* an object type. This takes time to get used to.

Long story short, at execution time, a JavaScript program is a nested tree of function executions. As a byproduct of functions each enclosing a scope, a JavaScript program can also be thought of as a nested tree of scopes. What is interesting is that the tree changes at runtime. This occurs because the stack of function executions grows and shrinks over time.

For example, our familiar snippet:

```
function makeBackgroundBlack() {
    document.body.style.backgroundColor = '#000000';
}

makeBackgroundBlack();
```

can be sequenced visually to represent what occurs at runtime step-by-step:


Stack Changes
Stack Changes

Think of *script1* as a function execution that's created automatically. It's a runtime starting point. As you'll soon learn, *many* script files can be loaded in an HTML page. The runtime thus creates starting point executions automatically for each script. Without them, the engine would never get to your code or third-party code.

In our visual example above the scope tree grew from zero to two and back to zero again over time.

1. Empty, no scope
2. *script1*, default global scope
3. `makeBackgroundBlack`, local scope nested in its parent scope
4. *script1*, default global scope
5. Empty, no scope

When the JavaScript engine evaluates an expression that has an identifier reference (shortcut name for a value), it works like this in an effort to get the bound value:

1. Look in the local (this function's) scope for the identifier
2. If not found, look in that scope's parent scope
3. Repeat until the identifier is found or the root parent scope is hit and not found

When the identifier reference is found in one of the scopes, the engine uses the value that is bound to it and continues to do work. This is what we want. If the root scope is hit and the value does not exist, then we have a problem. We will go into detail later regarding this scenario, so just be aware of it.

Now that we know the anatomy of a JavaScript program is simply a nested tree of scopes, let's add a simple example based on this section's HTML and CSS examples. As you might imagine, HTML has a tag that allows us to add JavaScript just as easily as we added CSS. Welcome to the `<script>` tag. You can add a `<script>` tag to the `<head>` or the `<body>`, but the latter is best practice for non-blocking reasons. Specifically, `<script>` tags should reside just before the closing body, the `</body>`. Here is an updated excerpt from our `index.html` file.

```
    <!-- The previous HTML code from our index.html file is excluded for brevity -->
    <script src='assets/js/main.js'></script>
</body>
```

Naturally, the server folder structure needs the path to our `main.js` so our updated structure becomes:

```
index.html
assets
    css
        style.css
    img
        cover-coding-for-designers.jpg
    js
        main.js
```

The browser understands the script tag and its `src` attr and then—you guessed it—the browser automatically downloads it. When completely downloaded, the runtime and engine take over with compilation and execution of its contents. Before we look at the contents of our `main.js` file, let's update our `<img>` and `<button>` HTML to more easily use each with JavaScript.

The updated `<img>` becomes:

```
<img id='image-to-toggle' src='assets/img/cover-coding-for-designers.jpg'></img>
```

The updated `<button>` becomes:

```
<button onclick='toggleImageOpacity()'>Toggle Image Opacity</button>
```

The `id` attr is how we identify an element in our HTML as unique to our document. An id means we intend there to be only one element with a specific name. A unique id enables us to gain a reference to the element's corresponding object for use in JavaScript. With this JavaScript reference we can use its API to do all sorts of cool things.

The result of clicking the button above during execution time results in the function named `toggleImageOpacity` to be called via `()`. This results in the function's body being executed by the engine. Behind the scenes, the runtime APIs transform the hardware input (mouse, trackpad, stylus, touch, etc.) to event queue work. The event loop then picks up the resulting package and it gets placed on the stack. This all happens in milliseconds or even microseconds when the event loop is not blocked. Lastly, the engine actually executes the code in the function body. Now is a great time to look at the contents residing in our `main.js` file.

```
function toggleImageOpacity() {

    // Reference work
    var imageToToggle = document.getElementById('image-to-toggle');

    // Core work
    imageToToggle.classList.toggle('halve-opacity');

}
```

We will update our `style.css` file with a `halve-opacity` class as well:

```
.halve-opacity {
    opacity: .5;
}
```

Do not worry if you cannot understand everything in the above function as we'll cover it in great detail in the next section. You should however be able to grasp a rough idea of what is happening each time the `toggleImageOpacity` function executes. The image's opacity is toggled between `.5` (50%) and `1` (100%) as a result of toggling the `halve-opacity` class on and off the `imageToToggle` element. Take note that the function is structured in two parts:

1. Reference work
2. Core work

A function is not always structured this way by a coder, but JavaScript compilation essentially enforces it. We don't go into the details of compilation as previously mentioned, but look into *JavaScript hoisting* if you want to dig deeper. Just save yourself some pain and declare your identifiers (`var`iables first and `function`s second) at the top of their parent function prior to referencing or executing them. Make the `var`iable and function names unique and meaningful while you're at it.

Admittedly, this program is tiny as it is comprised of only one small function. Take note however that programs of all sizes leverage this structure since they are just functions nested in functions. And consequently scopes nested in scopes.

Before advancing to the next chapter, let's dive really deep into the `toggleImageOpacity` function. The function *works*, but we can *improve* it.

#

Work. Right. Better.

In programming there is the axiom "make it right before you make it faster". This exact quote is found in *The Elements of Programming Style* by Brian W. Kernighan and P.J. Plauger and it is heavily respected in the programming community. Put another way, when authoring code you should make it:

1. Work
2. Right
3. Better

In this section we will explore the notion of *right* and *better* in reference to the working `toggleImageOpacity` function that we recently looked at. This effort will introduce reusable code improvement techniques and the thought process behind them. The three step process will become second nature in time.

First and foremost you need the code to *work*. No shit. Making it *right* is less obvious though. Typically this goal is to achieve the same functionality with less or more efficient code. Making it *better* is much more subjective.

Is the code better for beginner and junior coders? Is it better for advanced and senior coders? Is it better for the computer? Sometimes the solution is ideal for all, but this is not always the case. For example an advantageous language feature may be common sense to an advanced coder, but confusing to a beginner coder. The code is better for one group and less so for the other. Similarly, when program performance is a bottleneck, better may mean sacrificing human readability. This should be avoided, but it is sometimes necessary. This latter example translates to an advantage during compilation time or execution time, but a disadvantage to authoring time.

Below is the `toggleImageOpacity` function from the previous section with added comments. Each comment is numbered for reference as we will soon be investigating the code line-by-line. After we understand each line of code, we will entertain improvement ideas.

```
// 1. function declaration using `toggleImageOpacity` identifier
function toggleImageOpacity() {

    // 2. variable declaration and assignment using `imageToToggle` identifier
    var imageToToggle = document.getElementById('image-to-toggle');

    // 3. function call with 'halve-opacity' argument using `toggle` identifier
    imageToToggle.classList.toggle('halve-opacity');

// 4. end of `toggleImageOpacity` function
}
```

Now that the numbered comments provide a bit more context to each associated line of code, let's take a deeper look into each snippet. I do not expect you to understand everything we cover in the rest of this section however. Do not be discouraged if some words or ideas make no sense yet as this is expected. The intention of this section is to plant seeds in your mind regarding the vocabulary, concepts, and considerations of advanced coders. Extract what you can before we formally enter the *80/20 JavaScript* chapter.

In snippet one there are four parts comprising the line of code:

```
// 1. function declaration using `toggleImageOpacity` identifier
function toggleImageOpacity() {
```

1. `function` - keyword reserved by JavaScript denoting a function
2. `toggleImageOpacity` - custom identifier for referencing the function by name
3. `()` - *function signature input* defining how to call the function to do work
4. `{` - opening curly brace for declaring the beginning of the *function body*
    - `}` - a closing curly brace is expected after the *function body* for declaring its end
    - the `{` and `}` define the scope boundary of a function

The above four parts make up the anatomy of a function. First, the `function` keyword tells the JavaScript engine your intent to define a function. Second, a custom identifier is set to enable other code to properly reference and call the function by name. Third, the function's signature input defines what argument parameters the function expects to use when doing its work. No arguments are expected for `toggleImageOpacity` currently. Lastly, the `{` and `}` define the bounds of the function body. All functions return a value prior to closing the function body too, but this *function signature output* will be covered in the next chapter.

With respect to making snippet one *right* there is nothing we can do. A case could be made for making it *better* by renaming `toggleImageOpacity` to a single character alternative. This change is better for computers because there is less information to read which also translates to a smaller payload to send over a network. We won't make this change as we want to keep the code better for us humans. Research the automated *minification* process to learn about attaining the best of both worlds.

In snippet two there are seven parts comprising the line of code:

```
// 2. variable declaration and assignment using imageToToggle identifier
var imageToToggle = document.getElementById('image-to-toggle');
```

1. `var` - keyword reserved by JavaScript denoting a variable
2. `imageToToggle` - custom identifier for referencing a value by name
3. `=` - *assignment operator* that assigns the value on its right to the identifier on its left
4. `document.getElementById` - browser API function for accessing a specific HTML element's corresponding JavaScript object
5. `()` - special characters for function execution
6. `'image-to-toggle'` - argument value used in the function's work

7. `;` - character reserved by JavaScript denoting the explicit end of a code statement

The above seven parts make up the anatomy of a declaration and assignment statement. The parts work together to assign the resulting value from the `document.getElementById('image-to-toggle')` browser API call to the declared `imageToToggle` identifier. In subsequent code, the identifier can be used as a shortcut to reference the actual element object and then use its API. It is worth noting that every use of `imageToToggle` could be replaced with `document.getElementById('image-to-toggle')`. This approach would *work*, but it would be *less right* because we'd be doing the same work more times than needed.

Assigning an executed function's result to a variable identifier is an example of *caching*. Caching is a great approach for decreasing code volume and increasing execution time performance. The more expensive and time consuming the function call, the more valuable caching is. Preventing repetitive work is the win with caching.

With respect to making snippet two *more right*, there is one other thing we could do. We could cache the `imageToToggle` lookup and assignment *outside* of the `toggleImageOpacity` function. In doing so, we would use caching and prevent the repetitive work that occurs each time `toggleImageOpacity` executes. This approach would require the identifier to never be reassigned to continue to work.

Additionally, an even *more right* approach could be made if `imageToToggle`'s value would be useful to other code within our program. If this was the case then the variable wouldn't exist in the scope just outside the `toggleImageOpacity`'s scope, but could instead exist in a higher level scope via the singleton design pattern. We mentioned this approach in the *Programming and Visual Design - Principles and Patterns* section, but the details are outside the scope of this book. Again, we're just planting seeds here so do not concern yourself with the details. We'll leave the snippet as is for simplicity and because there isn't anything *better* we can do.

In snippet three there are four parts comprising the line of code:

```
// 3. function call with 'halve-opacity' argument using `toggle` identifier
imageToToggle.classList.toggle('halve-opacity');
```

1. `imageToToggle.classList.toggle` - browser API function for toggling the existence of a class on the element
2. `()` - special characters for function execution
3. `'halve-opacity'` - argument value used in the function's work
4. `;` - character reserved by JavaScript denoting the explicit end of a code statement

The above four parts make up the anatomy of one of the many expression statements. Specifically it is an execution statement. This occurs when a function has the `()` but is not preceded by the `function` keyword. We are not declaring it, but instead executing or *using it*. The parts work together to add the `halve-opacity` class to the `imageToToggle` element if it doesn't have it already. They also work together to remove the `halve-opacity` class from the `imageToToggle` element if it does. Toggling is another situation where the binary one-of-two-states concept again surfaces.

With respect to making snippet three *right* and *better*, there is not much we can do. We could take a caching approach as mentioned earlier with something like `var toggle = imageToToggle.classList.toggle;` and then subsequently call the function with `toggle()`. This particular example is not a good use case and it is an over-optimization. Again, caching is useful for time consuming and expensive work. You will learn in time when and when not to cache.

In snippet four there is one part comprising the line of code:

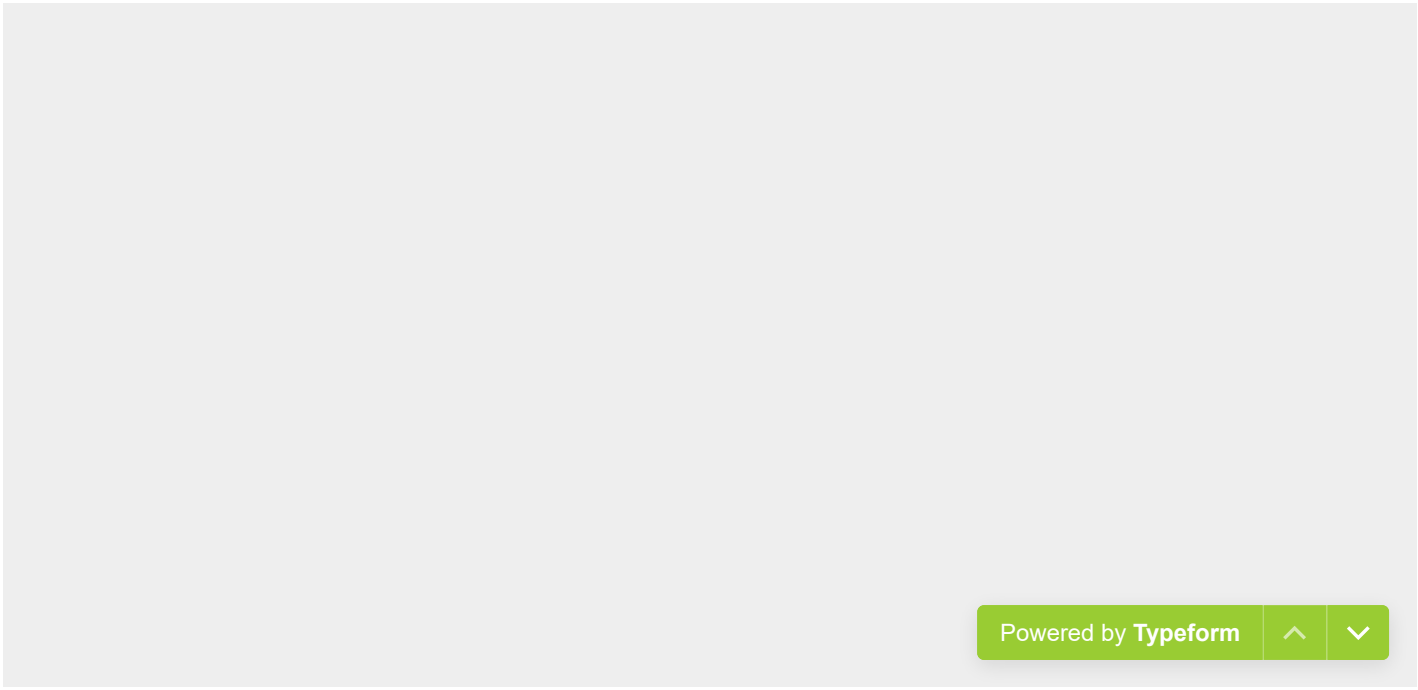
```
// 4. end of `toggleImageOpacity` function
}
```

1. `}` - closing curly brace for declaring the end of the *function body*
    - `{` - an opening curly brace is expected before the *function body* for declaring its beginning
    - the `{` and `}` define the scope boundary of a function

This closing curly brace simply defines the end of the `toggleImageOpacity` function. There are no *right* or *better* improvement possibilities.

In this section we introduced some of the vocabulary, concepts, and considerations of advanced coders. Overwhelming I know. Hopefully a few things made sense though. Regardless, seeds should be planted that will bear fruit later in your learn-to-code journey.

In the next chapter *80/20 JavaScript* we will bypass much of this advanced material in favor of a simplified approach. As you progress as a coder you can begin to make code *right* and *better*. Until then, the goal is to make code that works.



Chapter 3
Programming & Visual Design

Chapter 5
80/20 JavaScript