Follow @derekknox

Chapters

+

Light
-

Chapter 5
80/20
JavaScript

H
ave you ever heard of the Pareto principle? It states that roughly 80% of the output of systems results from roughly 20% of the input. This is not a universal rule, but it appears often enough to be a stand-alone principle. Put another way, a small amount of input in a system often produces a disproportionately large output.

The aim of this chapter is to introduce JavaScript with this 80/20 notion in mind. Though both numbers are rough, the takeaway is that you can be productive in JavaScript by knowing and using a small and specific subset of the language. This approach is in contrast to knowing all its keywords, their behavior, associated syntax, and the various subsystems of JavaScript. I recommend learning them later if you are interested, but you don't need them all to be productive.

The subset we'll focus on will help you author code that works. As you saw in *Work. Right. Better.*, this focus does not mean the code is not right. The mantra simply provides a path for improving code over time.

Prior to introducing this subset and detailing its parts we'll briefly cover JavaScript in terms of the *environment* in which it is executed. We will then cover a *mindset* that explains useful techniques for reading and authoring JavaScript. After these two sections we'll be ready to dive into the the code subset itself.

#

Environment

As mentioned in the *Interactive Code - Event Loop* section, there is a runtime that JavaScript is executed within. This runtime is also known as the runtime *environment*. The browser has been the runtime environment of focus thus far. This fact will remain. You should know however that JavaScript can be executed in different environments.

In the *Client and Server* section, we learned that the browser is on a client. Can JavaScript also be executed on a server? You bet. Some programs even embed an environment enabling coders to author plug-ins (new software) that extend the original program. JavaScript is a very flexible programming language and for this reason it is useful in many environments.

Though it may seem odd at first, each environment expects a particular language *version*. This idea makes a lot of sense in programming unlike English for example. A version consists of a specific set of keywords, syntax, and functionality. This allows a language to evolve where changes to it won't break existing programs. A language evolves and can improve when third-party and custom code is so useful that it should be built-in.

We will continue to focus on JavaScript in the browser, but just know it has more than one home and that its home expects a particular language version. The code in this book is version six (ES6). It's worth noting that the subset in this

book makes the code look like version five (ES5). This is intentional. The majority of ES6 additions fall in the *right* and *better* categories. Put another way, the additions are for advanced coders not beginners. This will likely be the case for all future versions in fact.

#

Mindset

The five principles below will influence your way of thinking about JavaScript throughout the rest of this book and beyond. They will simultaneously explain useful techniques for reading and authoring code. For brevity, I will use the term coding to encapsulate reading and authoring moving forward. The five principles are:

1. Thinking in three zoom levels
2. Functions are your friend
3. Prototype to learn
4. Don't repeat yourself
5. Refactor early and often

The first, *thinking in three zoom levels* will help you determine what to focus on at a given time. *Functions are your friend* will highlight why functions are so paramount. *Prototype to learn*, *don't repeat yourself*, and *refactor early and often* are programming principles found in *The Pragmatic Programmer* by Andrew Hunt and David Thomas. They encourage you to experiment, author reusable code, and aggressively refine it. Combined, the five principles will influence your actions while coding.

#

1. Thinking in Three Zoom Levels

There are three zoom levels you should consider when coding:

1. Scope tree
2. Statement pattern
3. Value resolution

These zoom levels are useful at authoring time (within code files) and execution time (within the program). This three zoom level approach is a technique for navigating and understanding code more quickly.

It is worth noting that any text editor is usable for coding. However, dedicated code editors are designed specifically to make our coding efforts less painful and more efficient. Ask online or local programmers that you trust to determine which code editor is best for you. SublimeText, Visual Studio Code, and WebStorm (among many others) are all great editors currently.

#

Zoom Level 1 - Scope Tree

Whenever you are starting to read or author code, place yourself at zoom level 1. Take note that your editor, authoring environment, and debugger (we'll cover this guy later) may each be of help at this step.

When reading, the goal is to *scan for functions*. You are looking at both their names and their nested structure. Not their implementation details. JavaScript programs are simply a tree of function executions and thus a tree of scopes. So identifying the names and nesting alone helps you understand the structural shape of a section of code or the program as a whole more quickly. The function's implementation details are to be ignored completely.


Zoom Level 1 - Scope Tree

When authoring, the goal is the same. If starting with no code you'll lack program shape, so this is only important when adding new code to existing code. When code already exists, this effort determines the target location for the code you plan to add. It can be difficult to select the location correctly the first time so don't stress. This fact is why we'll explore the *refactor early and often* principle.

Grasping the function names and their nested structure (program shape) is vital. We will cover a function's three fundamental use cases in the *Functions* section that soon follows. For now, their name and nested structure are the focus

points.

Once a particular function is of enough interest, you enter zoom level 2.

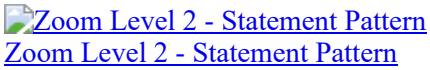
#

Zoom Level 2 - Statement Pattern

Now you can pay attention to a function's implementation details. We do this line by line.

Zoom level 2 is all about prepping yourself to get answers in zoom level 3. Answers in this context are synonymous with values. Once you have values, they can be understood and operated on. Values ensure a function can actually do work.

We will cover the specific statement patterns to look out for later in the *Statements* section. Until then, try to intuit meaning from the code snippets and look for general patterns that are repeated.

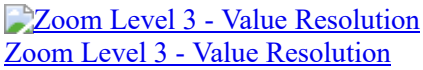

Zoom Level 2 - Statement Pattern

#

Zoom Level 3 - Value Resolution

At zoom level 3 we have determined the statement pattern for a given code statement. Once determined, this informs the steps we take in resolving the values of it. Again, functions can't do work unless values exist. Since a program is a dynamic and living thing, these values can be different at different times. We take the three zoom level approach to help us determine the exact values at a given time.

The takeaway is that the three zoom levels help you navigate and understand code more quickly using a repeatable approach.


Zoom Level 3 - Value Resolution

#

2. Functions are your Friend

We will continue to reinforce the importance of functions throughout this book. They enable interactive code and the manifestation of the amazing games, tools, and software we love, to exist. It is in your best interest to make functions your friend.

It is a bummer for creatives like us that the term `function` is used. The term originates from math, but functions aren't often mathematical in programming. They can be, but they're more about *doing work*. If I had it my way the keyword would instead be `work`, but `function` it is.

Each function has one of three fundamental use cases in JavaScript. Remember that a function encloses its own scope regardless of use case. Each use case provides a means to a desired end. These use cases are:

Function as:

- Organizational unit
- Instantiatable unit
- Reusable work unit

The takeaway is that functions are paramount to a program. Without them, a program is of little use. We'll cover these three use cases in great detail in the *Functions* section below.

#

3. Prototype to Learn

Code is virtual not physical. We must use this reality to our advantage. Authoring code and throwing it away is cheap and easy. We can undo and redo in an instant. This quality enables us to iterate quickly toward a solution or desired result

with minimal consequence.

Cheap and easy is not to be confused with being subpar however. Prototyping is extremely valuable as you can explore a solution space quickly to learn. Learning through quick iteration eventually manifests as a clearer path to follow. Following this path either leads to a solution or surfaces a new idea to prototype against.

The takeaway is that prototyping is how we sketch and experiment when coding. Taking shortcuts and ignoring best practices is OK here. Only after we have a working prototype are we concerned with not taking shortcuts and following best practices. Prototypes require minimal effort but result in great value. Completing a prototype ensures we have something that works. Right and better are for later.

#

4. Don't Repeat Yourself

Programming enables us to break the restrictions of the physical world. Every single day we do work to get results. Often times we've done this work thousands of times before. A few things I do everyday that fit this description include eating, drinking water, and reading. In each example I use energy to accomplish work in an effort to get a result. The work takes some amount of time before I get the result. The result is not instantaneous though I wish it was. Regardless of how good or fast I get at the tasks that comprise the work, the result will never be instantaneous.

We lack this restriction in the virtual world. We can get results instantly. In software we can *encode the work*. Once encoded we can simply execute a particular function to do work and to get a desired result. The same effort that was initially required no longer exists. It is encoded. Electricity is so fast the result of the work seems instantaneous.

The takeaway is that we strive to create functions that are reusable. We briefly covered this idea earlier in the *Programming and Visual Design - Elements and Elements* section. Additionally, we provided a concrete example in the *Sync and Async* section with the `changeBackgroundColor(newColor)` function. We will further explore this idea and more concrete examples in the *Functions* section.

#

5. Refactor Early and Often

As previously mentioned, it can be difficult to select the correct location when first adding code. Additionally, it can be difficult enough to get code to work through prototyping. It can even be difficult to name identifiers well (believe it or not this is one of the more difficult aspects of programming). Thankfully the code is not set in stone. It is easy to move, change, and rename. This is refactoring. We are working in the virtual not physical world after all. Use this to your advantage. Our editors help us accomplish this faster while minimizing mistakes.

The core benefit of refactoring is that it enables us to improve code readability. Remember the three zoom levels? Refactoring helps us here. Code complexity can also be reduced which makes it more understandable to ourself, other coders, and our future self.

Take note that refactoring requires that the correct work still gets done. In other words, the functional behavior remains where the implementation of that behavior may differ. This idea parallels the *Work. Right. Better.* mantra in that refactoring leads to right and better.

It is worth noting however that refactoring is a susceptible step in that it can lead to bugs. A bug is code that unintentionally prevents work or that does work incorrectly. We'll cover this more and explore concrete examples in the *Errors* and *Debugging* sections.

The takeaway is that code is a living thing. It can be molded into a more perfect shape. Code can be made more understandable during authoring time while becoming more efficient during execution time. By refactoring early and often we can author more readable, less complex, and more efficient code.

#

Subset

Just like natural languages have many words, rules, and exceptions, so too do programming languages. As we all know from experience, we only use a small fraction of English to communicate. JavaScript—and programming languages in general—are no different.

The question is then, why do extra words in a language even exist? Extra words are useful to those more experienced with a given language. They enable concise communication between those in-the-know. They are intended as a shortcut to shared meaning and understanding. The tradeoff of using these words is a risk of increased misunderstanding for those unfamiliar with them. The subset approach helps mitigates this risk.

The JavaScript subset—and the language itself—is organized in four groups:

1. Keywords
2. Expressions
3. Operators
4. Statements

These four groupings make a program useful by enabling it to do work during execution time. Think of the respective groupings as:

1. Named shortcuts to values
2. Values
3. Special characters for changing values
4. Useful patterns of special characters and keywords

Keywords are named shortcuts to values. They enable us to use a natural-language-like *key* for identifying something as meaningful to us.

Expressions are values. They unsurprisingly can be represented with keywords, but also with literals (like the number `360`) or to-be-*expressed* evaluations (which we'll cover later). Without values, we would not be able to translate something meaningful to us to a computer.

Operators are special characters for changing values. Operators enable values to be *operated on* and changed. You already know about the arithmetic operators `+, -, *, and /` from math class.

Statements are useful patterns of special characters and keywords. They enable us to group and reason about the various special characters, keywords, values, and operators in a particular portion of code using a small pattern. Put another way, we *state* something meaningful using a small pattern.

Combined, all four groups enable us to author code that a JavaScript engine understands. Expression by expression and statement by statement. When this happens in the browser, the browser works with other programs on the computer to ensure we see our animated and interactive creations on-screen. Let's dig into each of the four groups above one-by-one.

#

Keywords

So we now know keywords are named shortcuts to values. Each keyword is a *key* for accessing a value using a natural-language-like word. Keywords are not always complete words as you would expect in English. They are sometimes abbreviations (like `var` is for variable) or aggregations (like `makeBackgroundBlack` is for "make background black"). The former happens to be a *reserved keyword* where the latter happens to be a *non-reserved keyword*. These are the two types of keywords in JavaScript:

1. Reserved keywords
2. Non-reserved keywords

Put another way, reserved keywords are those that cannot be reassigned a value. They have a predefined value assigned by JavaScript. This value cannot be changed. In contrast, non-reserved keywords *can* be reassigned a value. They also have a predefined value by default, but it can be changed.

Non-reserved keywords are also known as *identifiers*. An identifier is simply any keyword that isn't already reserved. Non-reserved keywords are organized in three groups:

1. JavaScript identifiers
2. Environment identifiers
3. Custom identifiers

A JavaScript identifier is a keyword with a predetermined-by-JavaScript value. This value is useful to your code as it can help facilitate *language*-specific work. An environment identifier is a keyword with a predetermined-by-the-environment value. Its value is useful to your code also, but it helps facilitate *environment*-specific work. Custom identifiers have the

special JavaScript value `undefined` until the identifier is reassigned a value by you or through another coder's third-party code.

So JavaScript identifiers and environment identifiers each have predetermined non-`undefined` values. These values are set by the language and environment respectively. Custom identifiers have the predetermined value `undefined` until we reassign them a value.

Let's explore what we just learned relative to a familiar code snippet:

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

makeBackgroundBlack();
```

Before reading on, test yourself by listing out what you think the keywords are. Then try to guess each keyword's reserved or non-reserved status (this will be challenging I know).

Answer time:

1. `function` - reserved
2. `makeBackgroundBlack` - custom identifier
3. `document` - environment identifier
4. `body` - environment identifier
5. `style` - environment identifier
6. `backgroundColor` - environment identifier

You may have thought `'#000000'` was a keyword, but it is a literal value. We'll cover the distinction between literal values and keywords in the next section.

The takeaway is that any portion of code resembling a natural-language-like word (abbreviations and aggregations included) is a keyword if it's not in quotation marks. Everything else is either a literal value, an operator, or a statement's special character(s).

#

Reserved Keywords

There is one reserved keyword in the snippet above. There are over forty in JavaScript. With our subset approach there are ten that we care about. Listed alphabetically they are:

1. `debugger`
2. `else`
3. `false` - (also a literal value)
4. `function`
5. `if`
6. `new`
7. `null` - (also a literal value)
8. `return`
9. `true` - (also a literal value)
10. `var`

These are the reserved keywords to get really familiar with. We'll explore them in more detail in the remaining sections of this chapter in addition to the *Deconstructing Designs* chapter. This way we'll explore them in context of when they are most useful.

Listing the words alphabetically works, but it is more useful to logically group them. Below are the groupings in addition to a concise description of how they help us code:

Custom Keyword Helpers

- `var` - helper for declaring a reusable value by a custom name
- `function` - helper for declaring a reusable function by a custom name

Instance Helper

- `new` - helper for creating unique instances

Code Flow Keywords

- `if` - helper for guiding the engine to read certain code
- `else` - helper for guiding the engine to read certain other code
- `return` - helper for a function to provide the result of its work

Literal Value Keywords

- `true` - helper for validating code flow
- `false` - helper for validating code flow
- `null` - helper for the special "absence of a value" value

Debugging Keyword Helper

- `debugger` - helper for debugging

The concise description accompanying each keyword may not make complete sense at the moment. This is OK. The takeaway is that these are the ten reserved keywords of JavaScript that you want to focus on. We'll better understand what they do for us as we explore more code.

#

Non-Reserved Keywords - JavaScript

In the snippet above, there is no example of a non-reserved JavaScript keyword. There are around seventy in JavaScript however. With our subset approach there are only six that we care about. They each specialize in working with common types of values.

1. `Date` - helper for working with dates
2. `Error` - helper for working with errors
3. `JSON` - helper for reading and writing data
4. `Math` - helper for doing math
5. `Number` - helper for working with numbers
6. `String` - helper for working with strings

The `Date` helps us work with dates and time. `Error`s you understand generally, but we'll explore them in the context of code in the *Errors* section at the end of this chapter. `JSON`, pronounced "Jason", is likely foreign. `JSON` is useful for reading and writing in a data format useful for communicating between clients and servers. `Math` provides a bunch of functions that help coders do complex work with numbers. It also allows us to do simple work with numbers like rounding. `Number` helps us do more generic work with numbers. Lastly, `String` helps us work with characters and natural language words that we don't want the engine to interpret as keywords, operators, or statements.

#

Non-Reserved Keywords - Environment

There are four non-reserved environment keywords in the snippet above. They are `document`, `body`, `style`, and `backgroundColor`. Each is parented by the former. Object parenting occurs when an object's identifier value is another object. We will cover the anatomy of an object in *Expressions - Complex Values - Object*, but here is a partial example:

```
window.document = {
    body: {
        style: {
            backgroundColor: ''
        }
    }
}
```

The `document` has a parent too. This parent is special and is known as the *host object*. The host object in a browser environment is the `window` object. The `window` object provides the runtime APIs we learned about in the *Interactive Code - Event Loop* section. As a result, the use of `document` and `window.document` are interchangeable. We'll see why this is the case in the *Functions - Scope Chain* section. As an aside, the `document` provides APIs for us to update our HTML during execution time. This is exactly what we want–and do–in our `makeBackgroundBlack` function.

Now is a great time to reinforce that professional coders don't remember all the runtime APIs (there are 700+). They reference resources *just like beginners*. We will do the same and simply focus on knowing about the special `window` object. Over time we'll memorize the APIs that we use most often.

I do however recommend exploring the list of all the web APIs sometime. The effort enables you to grasp the big picture of what is possible by default in the browser. You will be impressed and you'll undoubtedly find many that give you cool ideas.

#

Non-Reserved Keywords - Custom

In the snippet above, there is one non-reserved custom keyword. It is `makeBackgroundBlack`. There are naturally an infinite amount of custom keywords. Remember that the computer doesn't care what the keyword name is when it is custom. It just cares that it is unique to a scope while providing a value. We could have instead named our `makeBackgroundBlack` function `a` resulting in the snippet:

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

a();
```

The functionality is the same even though there are fewer characters. As a result there is less code–a smaller payload–to send from a client to a server and vice versa. Remember however that we are coding for humans first. Naming keywords meaningfully is the primary goal. We can decrease the payload size later through the aforementioned minification process among other steps. The takeaway is that the engine only cares that custom keywords are unique to a scope. The name itself is useful for us coders.

In this last snippet you will notice that the `document`, `body`, `style`, and `backgroundColor` environment identifiers are left untouched. This is because they are not custom keywords. The runtime expects them to be associated with certain values. As such, they would not be shortened through minification for example.

It is worth noting that when naming custom keywords there are a set of rules. I will list them here for general familiarity, but there is a rule subset to instead focus on. Custom keywords can technically use the following characters:

- `a-z` (lowercase characters)
- `A-Z` (uppercase characters)
- `_` (underscore character)
- `$` (dollar sign character)
- `0-9` (number characters when not the first keyword's character)

There are three common case styles that are used as a result of the above rules:

1. upper camel (ex. `UpperCamelCase`)
2. lower camel (ex. `lowerCamelCase`)
3. underscore (ex. `underscore_case` aka `snake_case`)

I recommend using the `lowerCamelCase` just like we did with `makeBackgroundBlack`. We'll additionally use the `UpperCamelCase` style for certain functions, but we'll cover why in the *Functions* section later. The takeaway is that you can use all the rules above, but it is much simpler to stick to the `lowerCamelCase` and `UpperCamelCase` styles.

It is worth noting that these case styles exist for a reason. If blank spaces were allowed then our `makeBackgroundBlack` would be `make Background Black`. The engine would instead see three distinct custom identifier keywords as opposed to one. The `lowerCamelCase` and `UpperCamelCase` styles exist to mitigate this issue while maintaining readability.

Now that we've explored the various keyword types, now is the best time to explore the types of expressions—or values—a keyword can represent.

#

Expressions

Expressions are values. More precisely an expression is a piece of code that results in a value. The term expression is used to denote the fact that the engine may need to do some work to get the value—the expressed result. If this was not

the case then the official term could be value instead of expression.

In both examples below the value is the number `360`. The latter of the two requires work where the former does not:

- `360` (literal value expression)
- `300 + 60` (arithmetic expression)

Both examples are not really useful on their own however. An expression—the resulting value—becomes useful when used in the context of a code statement. A statement always consists of at least one expression. We will explore statements in greater detail in the *Statements* section but here are a few examples so you may begin to intuit how values are useful:

- `var maximumRotation = 360;`
- `if (currentRotation > maximumRotation) { currentRotation = maximumRotation; }`
- `function getMaximumRotation() { return maximumRotation; }`

The relationship of words to a sentence in natural languages is similar to the relationship of expressions to a statement in programming languages. A word is a basic unit of meaning just like an expression is. These units in aggregate provide greater meaning as a sentence in natural language or a statement in a programming language. In the *Statements* section—and in time—you'll begin to grasp what constitutes a valid statement.

#

Types & Forms

We've talked a lot about values, but we have not explicitly explored the built-in *types* of values in JavaScript. We only care about six of the seven types due to our subset approach. The types are organized in two groups:

1. Primitive Values
    - null (`null`)
    - undefined (`undefined`)
    - Boolean (`true` & `false`)
    - Number (`360`)
    - String (`"one or more characters wrapped in double quotes"` or `'single quotes'`)
2. Complex Values
    - Object (`{}` & `[]`)

The snippets within parenthesis above are all examples of the *literal form* of the respective value type. This form is most common and preferred. It is important to know that JavaScript has another way to create values however. This other way is called the *constructor form*. The constructor form leverages the `new` operator keyword followed by a JavaScript, environment, or custom identifier keyword. This keyword denotes the specific Object type. Here is the same list using `new` and the respective type's keyword (which is just a named shortcut to a function value):

1. Primitive Values
    - null (only literal form)
    - undefined (only literal form)
    - Boolean (`new Boolean(true)` & `new Boolean(false)`)
    - Number (`new Number(360)`)
    - String (`new String("one or more characters wrapped in double quotes")` or `new String('single quotes')`)
2. Complex Values
    - Object (`new Object()` & `new Array()`)

Unfortunately these `new` examples all result in a specific type of built-in Object when you really want the literal value.

The literal form is best for the above built-in types where the constructor form is best for all other types. Programmers like shortcuts, so this is also why the literal form is preferred above. Remember, working with the literal form results in the literal value whereas the constructor form results in an *Object that contains the value*.

The constructor form is useful—required really—for specific types of environment Objects like the aforementioned `Date` and `Error` among others. Custom types also leverage the constructor form.

The takeaway is that if you remember to use the `null`, `undefined`, `true`, `360`, `'word'`, `{}`, and `[]` literal forms and then use the constructor form for everything else, you'll be set.

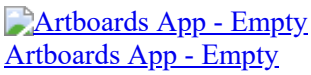
#

Primitive Values

Since primitive values are so fundamental to JavaScript, let's explore each of them in a little more detail. Below is a small program that will be further referenced as an example use of each primitive. Comments are intentionally absent so you can practice the *thinking in three zoom levels* technique. Pretend that the code is running in a browser that:

1. can display multiple artboards on a surface
2. has a "Create Artboard" button (`<button id='create'>Create Artboard</button>`)
3. has a "Delete Artboard" button (`<button id='delete'>Delete Artboard</button>`)
4. has downloaded, compiled, and is executing our program (`<script src='assets/js/artboards.js'></script>`)

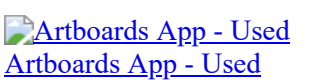
It could look something like this when first loaded:


Artboards App - Empty

Here is this JavaScript code of our `artboards.js` file:

```
var createArtboardButton = document.getElementById('create');
var deleteArtboardButton = document.getElementById('delete');
var artboards = [];
var artboardInFocus;

function setupEventListeners() {
    createArtboardButton.addEventListener('click', onCreateArtboardButtonClick);
    deleteArtboardButton.addEventListener('click', onDeleteArtboardButtonClick);
}

function updateArtboardInFocus(artboard) {
    artboardInFocus = artboard;
}

function deleteArtboardInFocus() {
    var artboardInFocusIndex = artboards.indexOf(artboardInFocus);
    artboards.splice(artboardInFocusIndex, 1);
    artboardInFocus.removeSelfFromSurface();
    artboardInFocus = null;
}

function onCreateArtboardButtonClick() {
    // Artboard is a custom object type, we'll learn about these later
    var artboard = new Artboard();
    artboards.push(artboard);
    artboard.addSelfToSurface();
    updateArtboardInFocus(artboard);
}

function onDeleteArtboardButtonClick() {
    if (artboardInFocus === null) {
        alert('No artboard to delete. None of the ' + artboards.length + ' artboards are in focus.');
    } else if (artboardInFocus === undefined) {
        alert('No artboard to delete. Try creating one first.');
    } else {
        deleteArtboardInFocus();
    }
}

setupEventListeners();
```

Once the app has been used, where artboards have been added, it could look something like this:


Artboards App - Used

#

Null

The special `null` value denotes the *explicit* absence of a value. This special value *is not* automatically assigned in JavaScript. It must intentionally be assigned to a keyword by a coder. Though `null` represents the absence of a value, it is technically a value itself. A little weird I know. This is what makes it "special".

In the `artboards.js` code above we use `null` in the `deleteArtboardInFocus` function:

```
artboardInFocus = null;
```

and in the `onDeleteArtboardButtonClick` function:

```
if (artboardInFocus === null) {
    alert('No artboard to delete. None of the ' + artboards.length + ' artboards are in focus.');
}
```

In the first example we are intentionally assigning `null` to `artboardInFocus`. Put another way, we are saying we don't want an artboard to be in focus right now. By using `null` in the second snippet we are conditionally testing a more intentional path (code flow) for the engine to execute code.

#

Undefined

The special `undefined` value denotes the *implicit* absence of a value. This special value *is* automatically assigned in JavaScript. It is the default value for `var`iable declarations. Additionally, it is the value returned when a nonexistent keyword is accessed. These two aspects make it "special".

In the `artboards.js` code above `undefined` is automatically used in the `var`iable declarations section:

```
var artboardInFocus;
```

and additionally in the `onDeleteArtboardButtonClick` function:

```
else if (artboardInFocus === undefined) {
    alert('No artboard to delete. Try creating one first.');
}
```

If the first interaction with the program is to click the "Delete Canvas" button then the above `alert` code would run. If we did not check for `null` and `undefined` prior to executing `deleteArtboardInFocus()` we'd get an `Error`. This would happen because we can't delete an artboard that does not exist. The above examples illustrate why the `null` and `undefined` values are useful.

The observant designer will wonder why the "Delete Artboard" button is interactive if there is not a valid artboard to delete. A better design would leverage the interaction design principle *progressive disclosure* and only show or enable the "Delete Artboard" button when an artboard became focused. I highlight this idea to illustrate the importance of designers and developers working together. Remember, design is the accumulation of decisions and these decisions impact designers and developers just as they do end-users.

#

Boolean

The Boolean type denotes one of two values: `true` or `false`. Remember the bit? This is JavaScript's formal approach to the same goal of defining one of two states. The bit's `0` is the Boolean's `false`. Its `1` is the Boolean's `true`.

In the `artboards.js` code above the `onDeleteArtboardButtonClick` function *implies* the use of a Boolean value in two places. Can you spot them before reading on?

Here is a more explicit approach using additional `var`iables:

```
function onDeleteArtboardButtonClick() {
    var isArtboardInFocusNull = artboardInFocus === null;
    var isArtboardInFocusUndefined = artboardInFocus === undefined;

    if (isArtboardInFocusNull) {
        alert('No artboard to delete. None of the ' + artboards.length + ' artboards are in focus.');
    } else if (isArtboardInFocusUndefined) {
        alert('No artboard to delete. Try creating one first.');
    } else {
```

```
        deleteArtboardInFocus();
    }
}
```

Both the original snippet and this updated snippet accomplish the same goal. Use whichever approach makes more sense to you. We'll cover the `if` statement in detail in the *Statements* section later, but you should be able to intuit what is happening.

The takeaway is that Boolean values are fundamental to controlling code flow.

#

Number

The Number type denotes numbers. Impressive I know. These numbers can be whole (`-360`, `0`, and `360`) or fractions (`-.36`, `.36`, and `3.6`). They can be negative or positive too. There are technically limits to a number's minimum and maximum value in JavaScript, but for our subset approach we can ignore them. If you ever need to work with extreme whole numbers (positives or negatives in the quadrillions) or similarly extreme fractions then feel free to dig deeper. Thought so.

In the `artboards.js` code above we use a Number twice in the `deleteArtboardInFocus` function:

```
var artboardInFocusIndex = artboards.indexOf(artboardInFocus);
artboards.splice(artboardInFocusIndex, 1);
```

and once in the `onDeleteArtboardButtonClick` function:

```
alert('No artboard to delete. None of the ' + artboards.length + ' artboards are in focus.');
```

The first snippet uses an evaluated number assigned to `artboardInFocusIndex` in addition to the literal `1` value. The two lines of code work together to:

1. find *where* in the `artboards` array the `artboardInFocus` is
2. use the Array's built-in `splice` function to remove that artboard (the `artboardInFocus`)

The second snippet uses the evaluated `artboards.length` value to get the number of total artboards that exist. This allows us to display an up-to-date message using the correct artboards count number as a String value.

#

String

The String type denotes `"one or more characters wrapped in double quotes"` or `'single quotes'`. There are eight examples of String values being used in the `artboard.js` code above.

Strings are useful for defining names, event types, and messages among other things. Concrete examples of this are the use of `'create'` and `'delete'`, `'click'`, and the `alert` strings respectively. It is worth noting that double quoted and single quoted strings are valuable in different scenarios:

- `"The artboard's size is too small."`
- `'"The artboard is too small," she said.'`

There is one specific example from the `onDeleteArtboardButtonClick` function I'd like to call out:

```
alert('No artboard to delete. None of the ' + artboards.length + ' artboards are in focus.');
```

Since `artboards.length` is a number, you might be wondering how a number and strings are added together? The answer is with `+`, the String Concatenation Operator. This will soon be explored in more detail in the *Operators* section.

The takeaway is that the String type prevents the engine from processing its characters as keywords or other value types. For example, `'null'`, `'undefined'`, `'true'`, `'false'`, and `'360'` are all String values because they are wrapped in quotes. If we removed the quotes they would instead be examples of the `null`, `undefined`, Boolean, Boolean, and Number types respectively.

#

Complex Values

Any value that isn't one of the five primitive values is a complex value. The only complex value type in JavaScript is the `Object`. An object is considered either *basic* or *specific*. In either case, each of its properties (attached identifiers) can contain a primitive *or* complex value. In contrast, a primitive can be *only* one of the five primitive types (`null`, `undefined`, `Boolean`, `Number`, or `String`).

#

Object

`Object`s in JavaScript are what empower coders with a massive amount of creative freedom. Since we can create custom identifiers and assign each a primitive or complex value, we can represent virtually anything. We can model real-world concepts just as easily as fantasy concepts. We can mix and match to our desire. We can use existing specific objects like `Date`, `Error`, and `Math` or create our own like `Color`, `Pen`, and `Artboard`. This range of freedom is what enables the wide variety of interactive games, tools, and software we love, to exist.

Do you remember the first two of four ideas I aimed to instill in the *Breaking Barriers - Bits and Bytes* section? Here is a refresher:

1. Computers, these complex machines, rely solely on extremely basic concepts
2. There is no magic in coding, just simple ideas stacked atop each other

These same two ideas that applied to bits manifest in `Object`s too. Instead of the binary nature of bits we're working with six value types. Any value of a given type can then be attached to an `Object` using an identifier. Admittedly this is a little more challenging, but basic concepts and ideas stacked atop each other are still at the core.

The distinction between a basic and specific object is simple. A basic `Object` is only ever one that is created via:

- object literal form (`{}`)
- object constructor form with the `Object` keyword (`new Object()`)

Both are examples of the *Object object*. Sounds funny I know. `Object` is the most basic type of object where all others are specific and built on top of it.

Specific objects are almost always created with the constructor form where the `Object` keyword is replaced. Here are three examples of the construction of specific objects:

1. `new Date();`
2. `new Error();`
3. `new Array();`

Below are three more examples, but of *custom* objects. Take note that the `Color`, `Pen`, and `Artboard` keywords need a custom function value associated also. Their constructor functions would need to exist in third-party or custom components. Otherwise, as you learned in the `undefined` section above, the engine will have no idea what they mean and instead default to `undefined`. This situation would subsequently result in an `Error` because `new undefined()` is invalid.

1. `new Color();`
2. `new Pen();`
3. `new Artboard();`

The core takeaway is you should use the object literal form for creating basic `Object`s and the constructor form for specific objects.

#

Array

As mentioned previously, the `Array` also has a literal form. It is `[]`. It is a specific type of `Object`. It is so common that it gets its own literal form like `Object` does. The `Object` and `Array` data structures are so common and useful that they have literal forms (creation shortcuts essentially).

`Array`s are nowhere near as flexible as `Object`s however. They are simply a list container. An `Array`'s flexibility manifests as the ability to order any amount of any of the six value types. Here are example arrays containing each value type (excluding the `undefined` and `null` types):

- `[ true, false, true ]`
- `[ 1, 2, 3 ]`
- `[ 'Page 1', 'Page 2', 'Page 3' ]`
- `[ {}, {}, {} ]`

The takeaway is that `Object`s are useful for organizing and modeling a *tree* of real-world, fantasy, or a combination of values using meaningful identifiers. Sequence doesn't matter. `Array`s are useful for organizing a *list* of those same primitive or complex values. Sequence does matter.

#

Copy vs. Reference

Custom keywords—declared with the help of the reserved `var` and `function` keywords—are how we declare our own named shortcuts to values. These values are one of the six types mentioned above where `function` is a specific type of `Object` (the `Function` object). Additionally, the aforementioned `Date`, `Error`, and `Math` are also specific types of `Object`s. Any value that isn't a primitive value is either a basic or specific `Object` value.

When a primitive value is assigned or associated to a keyword, the keyword holds a *copy* of the value. A complex value assigned or associated to a keyword is instead a *reference* to the value. So primitive values are always copies and complex values are always references. This distinction is important because references—unlike copies—enable the sharing of:

- functionality
- structured data

We already know that sharing functionality—sharing `function`s and the `Object`'s that contain them—provides us a simple and reusable way to do work. This shared functionality is what gives us APIs. Code would not be able to talk to other code otherwise. Data and specifically structured data have not been explicitly covered yet however. Let's do that now.

Data is simply any primitive or complex value. Structured data is always a complex value. It parents other values as either an:

- `Object`
- `Array`

Each allows the organization of data—primitive or complex values—using a particular structure. These structures are:

- `Object` as tree
- `Array` as list

The literal form of each is `{}` and `[]` respectively. An object literal uses *braces* and an array literal uses *brackets*. Braces are curved and brackets are straight. These visual differences are clues reminding you which structure belongs to which literal form:

- `Object` / "O" is curved / non-linear / braces / `{}`
- `Array` / "A" is straight / linear / brackets / `[]`

Let's explore some examples of code to really drive home `Object`s and `Array`s. First, here is an example of a few custom `Object`s in literal form. They are each assigned (using the `=` operator) to a keyword—using `var`—for easier understanding and later reuse. Take note that the `name`, `color`, and `thickness` custom identifier keywords of each `pen`, `highlighter`, and `paintbrush` object have their values associated with `:` instead of assigned with `=`. I wish `=` was used for simplicity and consistency, but I didn't design JavaScript. For object literals, this is how their keywords are assigned values.

```
var pen = {
    name: 'Pen',
    color: '#000000',
    thickness: 1
};
var highlighter = {
    name: 'Highlighter',
    color: '#FFFF00',
    thickness: 3
};
var paintbrush = {
```

```
    name: 'Paintbrush',
    color: '#0000FF',
    thickness: 16
};
```

Now here is an example of a custom `Array` in literal form. The list is assigned using the `=` operator to a keyword—again using `var`—for easier understanding and later reuse.

```
var drawingTools = [pen, highlighter, paintbrush];
```

The code for each drawing tool type makes the most sense when structured as a tree—an `Object`. When organizing all the drawing tools, it makes the most sense to structure them as a list—an `Array`. Over time you will learn to intuit when to use which type, but this distinction should help immensely. Even so, refactoring is always an option if a better model for organizing presents itself as the code you author evolves.

On their own, all four `var`iable declarations above are not all that useful. An `Object` is most useful when code can *access* its nested keywords and thus its nested values. We can then *operate* on those values to create new ones. Similarly, an `Array` is most useful when code can *iterate* its values. We'll cover what it means to iterate an `Array`'s values in the *Functions* section. Accessing and operating on an object's nested keyword values transitions us right into learning about operators.

#

Operators

Expressions are synonymous with values. Values by themselves are useful, but they are more so when they can be *operated* on. Put another way, being able to assign, combine, access, and create new values is useful.

JavaScript provides certain reserved keywords and special characters that fall into this operator category. There are over fifty but with our subset approach we only care about sixteen. You know most of them already from elementary math class. Bonus. Additionally, there is an extremely useful character that's not technically an operator but might as well be. We'll start there. Welcome to the *dot* (.).

#

Dot Notation

The dot is not an operator, but a notation. This is just fancy talk. We'll consider it an operator.

The dot allows us to access an object's nested keywords and thus its nested values. You have already seen this in action through virtually every code snippet up to this point. Refresher time:

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

makeBackgroundBlack();
```

There are three uses of the dot in the familiar snippet above. In each subsequent use a specific keyword of a nested object is accessed. Accessing a specific keyword of an object—and thus its value—is exactly what the dot is for.

If you recall from the *Interactive Code* chapter, a JavaScript program at execution time is just a tree of function executions and thus a tree of scopes. Since a function is also a certain type of `Object` (the `Function` Object) a JavaScript program is also a *tree of objects*.

A JavaScript program is a tree of:

- Objects
- Functions
- Scopes

The dot character is powerful because it allows us to *navigate objects*. By navigating objects, we can search for and access specific and nested values to work with. The moment a dot provides access to a nested keyword value is the moment operators become useful.

#

Assignment Operator

The most commonly used operator is the *assignment* operator. It is the equal (=) sign. You already know what it does, but you are likely used to seeing it work with numbers only. In JavaScript, you assign keyword identifiers specific values with it. These values can be primitive (like `360`, `'360'`, and `true`) or complex (like `{}` and `[]`). They can be literal expressions (like `360`) or evaluated expressions too (like `300 + 60`). In all cases, the engine works to provide a single value that is then assigned to a specific keyword identifier. Simple. Here we go again:

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

makeBackgroundBlack();
```

In the above snippet, the engine's work is simple. After `makeBackgroundBlack()` is called, the value of focus is the literal hex color String `'#000000'`. It is assigned to the `backgroundColor` keyword of the `style` object of the `body` object of the `document` object. As an aside, we now know the `backgroundColor` keyword is on an object nested three levels deep in the program tree.

Assignment is useful for one fundamental reason. What do you think it is?

Without assignment, we would never be able to *change or save values*. This would mean new values could be created during execution time, but no other code would be able to use them. That would make for lame games, tools, and software. Thank you =.

#

Arithmetic Operators

We won't spend much time on the arithmetic operators as you already know about them. They are:

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division

These four operators in combination with the assignment operator give us coders a ton of power. For example, we gain the core ability to animate and otherwise move, resize, scale, and transform visual elements to our desire. In fact, here is a primitive example using HTML (structure), CSS (style), and JavaScript (behavior):

HTML

```
<div id='the-brick' class='brick'>I'm a brick</div>
```

CSS

```
.brick {
    background-color: #FF0000;
    display: inline-block;
}
```

JavaScript

```
var theBrick = document.getElementById('the-brick');
var angle = 0;

function updateBrickRotation() {
    angle = angle + 1;
    theBrick.style.transform = 'rotate(' + angle + 'deg)';
}

setInterval(updateBrickRotation, 16);
```

Take a moment to envision what the combination of snippets results in before reading on. Spoiler alert, it's a spinning brick. This example illustrates how we can use the arithmetic and assignment operators together to make changes over time.

Remember the *Interactive Code - Frame Rate* section? Animation is simply one or more changes that impact visual elements between each rendered frame. With code we can make and apply these changes to visual elements over time. It's as simple as that.

#

String Concatenation Operator

The line of code in the example above that is *applying* the rotation change is:

```
theBrick.style.transform = 'rotate(' + angle + 'deg)';
```

We've seen this String Concatenation Operator (`+`) before in our `artboards.js` file:

```
alert('No artboard to delete. None of the ' + artboards.length + ' artboards are in focus.');
```

Yes, the `+` is *both* an arithmetic operator *and* the String Concatenation Operator. When used only with numbers, it is the arithmetic operator and when Strings are involved, it is the String Concatenation Operator.

If we pretend that the `angle` value is `45` and `artboards.length` is `3`, then our snippets become:

```
theBrick.style.transform = 'rotate(' + '45' + 'deg)';
```

and

```
alert('No artboard to delete. None of the ' + '3' + ' artboards are in focus.');
```

After the concatenation operator does its work (adding Strings together), the end result of each snippet is:

```
theBrick.style.transform = 'rotate(45deg)';
```

and

```
alert('No artboard to delete. None of the 3 artboards are in focus.');
```

When expressions are evaluated and a resulting value remains (specific Strings in our two examples above), this is the moment in time they are useful. What if we only wanted to make changes over time *given a certain condition*? This is where the comparison operators come in.

#

Comparison Operators

These comparison operators are vital for controlling *code flow*. Put another way, only certain code executes at a given moment in time based on one or more conditions. We'll update the brick JavaScript snippet from above to extend our animation example:

```
var theBrick = document.getElementById('the-brick');
var isClockwise = true;
var angle = 0;

function toggleIsClockwise() {
    isClockwise = !isClockwise;
}

function updateBrickRotation() {
    if (isClockwise === true) {
        angle = angle + 1;
    } else if (isClockwise === false) {
        angle = angle - 1;
    }

    theBrick.style.transform = 'rotate(' + angle + 'deg)';
}

setInterval(updateBrickRotation, 16);
setInterval(toggleIsClockwise, 2000);
```

We added four new pieces:

1. `isClockwise` identifier
2. `toggleIsClockwise()` function
3. `if` statement
4. `setInterval(toggleIsClockwise, 2000)` function call

How do you think the above additions change the program? There is a new operator (`!`) added that we'll soon cover, but still try to guess how the program changes.

Here is added context for each addition:

1. `isClockwise` identifier (to track clockwise rotation over time)
2. `toggleIsClockwise()` function (to alternate `isClockwise`)
3. `if` statement (to control code flow)
4. `setInterval(toggleIsClockwise, 2000)` function call (to execute `toggleIsClockwise` every 2000 milliseconds)

You nailed it if you guessed that the changes make the brick's rotation toggle between a clockwise and counter-clockwise rotation every two seconds.

The main comparison operator in use is the `===`, but there are six in total:

- `===` Strict equality
- `!==` Strict inequality
- `>` Greater than
- `>=` Greater than or equal to
- `<` Less than
- `<=` Less than or equal to

It is worth noting that the `===` and `!==` are useful with Strings just as they are with Numbers. In fact the strict equality and strict inequality operators are useful for comparing *all* types of values. In all cases, the expression—the *resulting value of the comparison*—is a Boolean value. If the Boolean is `true` the code flows that way. If `false`, the code flow skips that way.

With these six operator options you can validate certain conditions and thus control code flow. You'll notice that in the example above and in all cases we care about, the comparison operators are used with variations of `if` statements. We've yet to explicitly cover these statements in hopes that you can intuit what they do on your own. We will cover them in the following *Statements* section however to solidify your understanding.

Of the four additions in the previous code example, the one that should be the most odd has to do with the `toggleIsClockwise` function and specifically its body:

```
isClockwise = !isClockwise;
```

This should be the case due to the uncovered `!` operator. It is one of three logical operators.

#

Logical Operators

The logical operators are useful for controlling code flow just like the comparison operators. As such, they too operate on Boolean values. They are:

- `!` NOT
- `&&` AND
- `||` OR

Think of the NOT operator as a shortcut to a function that *flips* the Boolean value it is attached to. Here is the example shortcut function:

```
function flipTheBoolean(booleanArgument) {
    if (booleanArgument === true) {
        return false;
    } else if (booleanArgument === false) {
        return true;
    }
}
```

Put another way, by flipping a Boolean we produce its opposite. A `true` becomes `false` and a `false` becomes `true`. Pretty simple. The `!` is just odd looking. If we refresh back to our `toggleIsClockwise` function body:

```
isClockwise = !isClockwise;
```

We can instead envision:

```
isClockwise = flipTheBoolean(isClockwise);
```

So naturally, when `isClockwise` is `true` its flipped value is `false` and vice versa. In our example, this new flipped value is then assigned via the assignment operator to the `isClockwise` keyword identifier. It may seem odd to use the `isClockwise` keyword identifier twice on the same line and you are right. It becomes less odd in time and especially after we dig deeper in the *Statements* section.

The AND (`&&`) and OR (`||`) operators are useful for working with *more than one* Boolean at a time. An example helps illustrate this:

```
var red = '#FF0000';
var green = '#00FF00';
var blue = '#0000FF';
var currentColor = red;

function randomizeBackgroundColor() {
    var randomNumber = Math.random();
    var isRed = randomNumber >= 0 && randomNumber < .33;
    var isGreen = randomNumber >= .33 && randomNumber < .66;
    var isBlue = randomNumber >= .66 && randomNumber <= 1;

    if (isRed) {
        currentColor = red;
    } else if (isGreen) {
        currentColor = green;
    } else if (isBlue) {
        currentColor = blue;
    }

    document.body.style.backgroundColor = currentColor;
}

setInterval(randomizeBackgroundColor, 1000);
```

What happens in the above snippet during execution time?

You nailed it if you said every second the `document`'s `backgroundColor` gets randomly updated to either red, green, or blue. We use the `&&` operator to work with two Boolean values in each `isRed`, `isGreen`, and `isBlue` assignment. Then `currentColor` is assigned a color based on the code flow. For example, the color blue is assigned to `currentColor` only if `isBlue` is `true`. It only becomes `true` if the `randomNumber` is greater than or equal to `.66` *and* `randomNumber` is less than or equal to `1`. In that scenario both `isRed` and `isGreen` will be `false` where the code flowed past each conditional check. That's code flow in action using the `&&` operator.

Let's look at code flow in action using the `||` operator:

```
var volume = 0;

function updateVolume(keyEvent) {
    var isArrowUpPressed = keyEvent.key === 'ArrowUp';
    var isArrowRightPressed = keyEvent.key === 'ArrowRight';
    var isIncrease = isArrowUpPressed || isArrowRightPressed;

    if (isIncrease) {
        volume = volume + 1;
    }
}

document.addEventListener('keydown', updateVolume);
```

What happens in the above snippet during execution time?

You nailed it if you said that the `volume` increases when the up or right arrow keys are pressed. Nothing happens if any other keys are pressed as our code flow check is only using `isIncrease` (which was assigned with the help of `||`).

This program rocks a little too hard because the volume can't be turned down. Think how you might fix that before reading on.

Answer time:

```
var volume = 0;

function updateVolume(keyEvent) {
    var isArrowUpPressed = keyEvent.key === 'ArrowUp';
    var isArrowDownPressed = keyEvent.key === 'ArrowDown';
    var isArrowLeftPressed = keyEvent.key === 'ArrowLeft';
    var isArrowRightPressed = keyEvent.key === 'ArrowRight';
    var isIncrease = isArrowUpPressed || isArrowRightPressed;
    var isDecrease = isArrowDownPressed || isArrowLeftPressed;

    if (isIncrease) {
        volume = volume + 1;
    } else if (isDecrease) {
        volume = volume - 1;
    }
}

document.addEventListener('keydown', updateVolume);
```

Each of the `&&` and `||` examples above only illustrate their use with two Booleans. This is intentional. As mentioned earlier however, they are useful with *more than one* Boolean. Two is not the limit. It is recommended however to keep multiple uses at a minimum to reduce complexity. This is especially true for us regarding our 80/20 approach.

#

New Operator

If you recall the from the *Keywords - Reserved Keywords* section, the `new` keyword was introduced. It was and still is a *helper for creating unique instances*. An *instance* is simply a unique version of a specific Object. The specific Object can be built-in or custom:

- `var date = new Date();` (built-in)
- `var artboard = new Artboard();` (custom)

By having unique instances we can dynamically create *new* objects for use in our program. Let's build on the `artboard.js` example that we introduced in the *Expressions* section. Here it is again for reference:

```
var createArtboardButton = document.getElementById('create');
var deleteArtboardButton = document.getElementById('delete');
var artboards = [];
var artboardInFocus;

function setupEventListeners() {
    createArtboardButton.addEventListener('click', onCreateArtboardButtonClick);
    deleteArtboardButton.addEventListener('click', onDeleteArtboardButtonClick);
}

function updateArtboardInFocus(artboard) {
    artboardInFocus = artboard;
}

function deleteArtboardInFocus() {
    var artboardInFocusIndex = artboards.indexOf(artboardInFocus);
    artboards.splice(artboardInFocusIndex, 1);
    artboardInFocus.removeSelfFromSurface();
    artboardInFocus = null;
}

function onCreateArtboardButtonClick() {
    var artboard = new Artboard();
    artboards.push(artboard);
    artboard.addSelfToSurface();
    updateArtboardInFocus(artboard);
}

function onDeleteArtboardButtonClick() {
    if (artboardInFocus === null) {
        alert('No artboard to delete. None of the ' + artboards.length + ' artboards are in focus.');
```

```
    } else if (artboardInFocus === undefined) {
        alert('No artboard to delete. Try creating one first.');
    } else {
        deleteArtboardInFocus();
    }
}

setupEventListeners();
```

You will notice that there is no `Artboard` function. If it was a built-in type then there would be no problem. Since it is a custom type we do. An error will result when `onCreateArtboardButtonClick()` executes due to a `click` on the `createArtboardButton`. We need to declare and define this `Artboard` function so our program will work. If we do not, then the code evaluates to `new undefined();`. This is an error as `undefined` is not a function and thus cannot be called.

We will add our `Artboard` function just under the `var`iable declarations. Again, in time you'll intuit where and how to best group your functions to organize your code.

```
function Artboard() {
    var artboardElement;
    var api = {
            addSelfToSurface: add,
            removeSelfFromSurface: remove
        };

    function initialize() {
        artboardElement = document.createElement('div');
        artboardElement.classList.add('artboard');
    }

    function add() {
        document.body.appendChild(artboardElement);
    }

    function remove() {
        document.body.removeChild(artboardElement);
    }

    initialize();

    return api;
}
```

Now new artboards can be created by simply calling:

```
var artboard = new Artboard();
```

This is exactly what the `new` operator is useful for. Since we are explicitly `return`ing an Object literal, we could instead:

```
var artboard = Artboard();
```

This would be the more right and better approach. For our purposes however feel free to use the `new` operator and think of it as your helper for getting unique instances of built-in, third-party, and custom types of Objects.

We have covered quite a few code snippets up to this point and every single one of them is comprised of at least one *statement*. We'll cover statements in detail now as you have gone long enough using your intuition without validation.

#

Statements

If you recall from the *Programming and Visual Design - Elements and Elements* section, we learned that a statement in programming is similar to the line in visual design:

> For example connecting two points forms the second visual element, a *line*. Similarly, connecting expressions forms the second programming element, a *statement*. Each are the building blocks of visual design and programming respectively.

Additionally, in the *Mindset* section earlier we explored *Thinking in Three Zoom Levels* and specifically *Zoom Level 2 - Statement Pattern*. No statement patterns have been explicitly covered yet in favor of letting your intuition guide you. Now is the time to cover them however. There are over twenty-five, but we only care about six:

Primary

- Variable Declaration
- Function Declaration
- Expression

Secondary

- If
- Return
- Debugger

We use the term *statement pattern* as a statement is simply a particular pattern of characters and keywords. You will recognize each of the above six as they've been used in previous code snippets. Now you will learn each statement pattern by name for improved understanding and communication.

The order of each statement above is intentional as this is the general order you should expect to see them in code. Consider the primary statements to always be present in a program where the secondary ones are less guaranteed.

Anytime you see one or more `var`iable declarations, you can consider them as the start of some meaningful group of code. Following these declarations, you should see one or more `function` declarations. The `var`iables exist to denote that these keywords—and ultimately their values—will be useful to the declared `function`s when executed. A program typically starts by at least one `function` executing via `()`. This is one example of an expression statement.

These `function` declarations that follow will almost always use the prior `var`iable declarations. We say almost always as the program is interactive. If a user doesn't take certain actions or a certain trigger doesn't occur, some functions simply will not execute. We author them so they are prepared to however.

The rest of the statements help further control code flow, return values of interest from `function`s, and help with debugging.

If it hasn't been obvious in the snippets up to this point, three general things happen in order:

1. Declare `var`iables for reuse in `function`s
2. Declare `function`s to do work (often using the `var`iables)
3. Execute one or more `function`s with `()`

By mapping each statement type to the above three steps we have:

1. Variable Declaration
2. Function Declaration
3. Expression, If, Return, and Debugger

When thinking about programming in these terms, programming is much simpler to reason about. Deeply understanding these six statements and their general three step order gives you a lot of power. As we dig deeper into each, your exposure to and knowledge of keywords, expressions, and operators will come full circle.

#

Variable Declaration Statement

The goal of the Variable Declaration Statement is to *declare a `var`iable*. This statement is useful because it allows us to identify a value container by name for later reuse. Again, if we couldn't save values, our programs wouldn't be able to do much. Here is an example of a `var`iable declaration:

```
var maximumRotation;
```

The anatomy of this type of statement consists of three parts in sequence:

1. `var` keyword
2. custom keyword naming the `var`iable
3. `;` character

You are familiar with the first two, but the `;` has not been covered. What do you think it means? In comparing the above statement to an English sentence, what do you think the `;` would be equivalent to?

If you said the period, you are correct. The `;` simply denotes that the coder intentionally ends the statement. So in our example above we are telling the engine:

"Hey engine, please create a `var`iable named `maximumRotation`."

The statement pattern is valid (no `Error`s result) so the engine does what we say. Nice. In most snippets we've covered thus far we make the engine do a little more work than just a `var`iable declaration:

```
var maximumRotation = 360;
```

You guessed it:

"Hey engine, please create a `var`iable named `maximumRotation` and while you are at it, assign it the value `360`."

The engine sees this as a valid statement too and does what we say. There is a small caveat that we need to understand however. If you recall from the *Interactive Code - Authoring, Compiling, and Executing* section we learned that the compile step *transforms our static authored code into code that can be executed*. As a result of compilation, this last snippet actually becomes two:

```
var maximumRotation = undefined;

maximumRotation = 360;
```

When we talked about `undefined` earlier being the default, now you see how.

As we'll see in the Function Declaration Statement a similar thing happens. This aspect is one of the fundamental reasons we looked at authoring, compiling, and executing. Internalize the distinction of the two snippets due to the compiling step and you will be ahead of most in understanding JavaScript.

So our first snippet is our authored code. The second is the result of our compiled code. Now during execution, the *values* actually get assigned and operated on as the engine executes line by line and statement by statement.

#

Function Declaration Statement

The goal of the Function Declaration Statement is to *declare a `function`*. This statement is useful because it allows us to identify a `function` by name for later reuse. Again, `function`s allow us to encode work, often for instantaneous results. Here is an example of a `function` declaration:

```
function getMaximumRotation() {
    return maximumRotation;
}
```

The anatomy of this type of statement consists of four parts in sequence:

1. `function` keyword
2. custom keyword naming the `function`
3. `()` characters enclosing the optional *argument parameters*
4. `{}` characters enclosing the *`function` body*

So in our example above we are telling the engine:

"Hey engine, please create a `function` named `getMaximumRotation`. Set its value to the remainder of the statement, but you can ignore it until executed."

Below is another familiar example. It has a different name, it uses one argument parameter, and its body differs due it doing different work:

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

In this example we are telling the engine:

"Hey engine, please create a `function` named `changeBackgroundColor`. Set its value to the remainder of the statement, but you can ignore it until executed."

The engine only looks at the argument parameters and `function` body when the `function` is executed. Etch this in your brain. Only after a function is called via `()`—not when it is declared—does the engine do the work inside the body.

At this execution moment, a similar caveat as mentioned in the Variable Declaration Statement section is at play. Only the `changeBackgroundColor` declaration is impacted as only it has argument parameters. As a result of compilation, the `changeBackgroundColor('#FFFFFF')` executes as:

```
function changeBackgroundColor(newColor) {
    var newColor = undefined;

    newColor = '#FFFFFF';
    document.body.style.backgroundColor = newColor;
}
```

This `funtion` only expects one argument whose `var`iable name is predefined. Since `var`iables are `undefined` by default, that process still occurs. This would occur for each argument parameter. Only after that, if an argument is actually passed in does its value get assigned. If we instead called `changeBackgroundColor()` without an argument value, the result would be:

```
function changeBackgroundColor(newColor) {
    var newColor = undefined;

    document.body.style.backgroundColor = newColor;
}
```

The background color of the document would not update as `undefined` isn't a valid color. One approach to making this function more robust would be to add a `defaultColor` to fallback to:

```
function changeBackgroundColor(newColor) {
    var defaultColor = '#000000';

    if (newColor === undefined) {
        document.body.style.backgroundColor = defaultColor;
    } else {
        document.body.style.backgroundColor = newColor;
    }
}
```

Based off what we learned about the Variable Declaration Statement and Function Declaration Statement, the compiled result for a `changeBackgroundColor()` call is now:

```
function changeBackgroundColor(newColor) {
    var defaultColor = undefined;
    var newColor = undefined;

    defaultColor = '#000000';

    if (newColor === undefined) {
        document.body.style.backgroundColor = defaultColor;
    } else {
        document.body.style.backgroundColor = newColor;
    }
}
```

Where a `changeBackgroundColor('#FFFFFF')` call is:

```
function changeBackgroundColor(newColor) {
    var defaultColor = undefined;
    var newColor = undefined;

    defaultColor = '#000000';
    newColor = '#FFFFFF';

    if (newColor === undefined) {
        document.body.style.backgroundColor = defaultColor;
    } else {
        document.body.style.backgroundColor = newColor;
    }
}
```

The takeaway is that `var`iables are `undefined` by default where `function`s are predefined by the language, environment, or by a coder. In either case their declaration (due to compiling) exists at the top of the scope before any assignment or

`function` executions occur. Each time a `function` is executed, you can think of it as being `new`ed. We will explore this in more detail in the *Functions* section that soon follows.

#

Expression Statement

The goal of an Expression Statement is to *do work*. This statement is useful because it allows us to assign, combine, access, and create values. Here are a few examples of expression statements:

- `var maximumRotation = 360;`
- `return maximumRotation;`
- `changeBackgroundColor('#FFFFFF');`
- `newColor = '#FFFFFF';`
- `document.body.style.backgroundColor = newColor;`

There is no specific anatomy for this type of statement. We know expressions are synonymous with values, so consider an Expression Statement one that:

- uses values
- that isn't solely a declaration (`var`iable or `function`)

Put another way, Expression Statements are how work actually happens. A `function` execution occurs (which is itself an Expression Statement) to kick off that `function`'s work. The body of said `function` uses one or more Expression Statements—in combination with `var`iable declarations and other `function` declarations—to do work.

All three of the primary statements in concert make programs useful:

- Variable Declaration Statements for resuing values by name
- Function Declaration Statements for reusing patterns of work by name
- Expression Statements for using values (literal or evaluated)

#

If Statement

The goal of an If Statement is to *control code flow*. This statement is useful because it allows or prevents code from running conditionally. These conditions are validated with Booleans. Here is an example of an `if` statement:

```
if (isClockwise) {
    alert('isClockwise is true');
}
```

The anatomy of this type of statement consists of three parts in sequence:

1. `if` keyword
2. `()` characters enclosing one or more Boolean expressions
3. `{}` characters enclosing the conditional code to execute

There are two additional variations that leverage the `else` keyword and the `else if` keywords together. You can chain as many of these `else if` statements to the main `if` where the `else` is always last or non-existent. Here is an `if...else` example.

```
if (isClockwise) {
    alert('isClockwise is true');
} else {
    alert('isClockwise is false');
}
```

And here is an `if...else if` example:

```
if (isRed) {
    currentColor = red;
} else if (isGreen) {
    currentColor = green;
} else if (isBlue) {
    currentColor = blue;
}
```

Building a little further, here is an `if...else if...else` example:

```
if (isRed) {
    currentColor = red;
} else if (isGreen) {
    currentColor = green;
} else if (isBlue) {
    currentColor = blue;
} else {
    currentColor = defaultColor;
}
```

The takeaway is that variations of the `if`, `else if`, and `else` keywords are vital to controlling code flow.

#

Return Statement

The goal of a Return Statement is to *return a specific value from a `function`*. This statement is useful because it returns a value based on the `function`'s work. Here is an example of a `return` statement:

```
return maximumRotation;
```

The anatomy of this type of statement consists of three parts in sequence:

1. `return` keyword
2. an expression
3. `;` character

A `return` means that the expression value will be sent out of the `function`. No code after the `;` will run on this `function`'s execution. This approach is very useful for assigning a `var`iable the result of a `function`'s work. Here are two examples:

```
var date = new Date();
```

and

```
var artboard = new Artboard();
```

The takeaway is that the `return` statement is useful for providing a value that can be saved for later use.

#

Debugger Statement

The goal of a Debugger Statement is to *stop the executing program at an exact moment in time*. This statement is useful during debugging because it freezes code execution and automatically opens the browser's developer tools. Each browser's developer tools help a coder understand what certain values are during execution. We'll dive a bit deeper later in the *Debugging* section. Here is an example of a `debugger` statement:

```
debugger;
```

The anatomy of this type of statement consists of two parts in sequence:

1. `debugger` keyword
2. `;` character

The takeaway is that the `debugger` statement is useful for stopping execution at an exact moment in time to investigate values. This statement is only for coders. Remove it when your program is live for end-users.

#

Functions

A function is like a shape because it encloses scope just as a shape encloses space. This enclosure helps prevent clashing of identifiers between different scopes. At execution time, a JavaScript program is a nested tree of function executions.

Since each executing function encloses a scope, a JavaScript program is a nested tree of scopes. This tree changes because the stack of function executions grows and shrinks over time.

Functions are paramount to JavaScript programming for this organizational reason. They additionally serve two other fundamental use cases totaling three.

Function as:

- Organizational unit
- Instantiable unit
- Reusable work unit

Often times a function serves two or all three at once. Let's dive into each to take our understanding to the next level. First however, let's quickly refresh on the anatomy of a function as it is the same in all three use cases.

#

Anatomy of a Function

The anatomy of a function, as we learned in the *Function Declaration Statement* section, consists of four parts in sequence. We have also learned that the *Return Statement* is extremely useful in functions. What we have not yet covered is the fact that a `return` *always exists*.

There is actually a fifth part of the anatomy of a function, but it wasn't important to reveal until now. This fifth part is an automatically inserted `return undefined;` if a custom `return` isn't the last statement of the function body. This automatic insertion occurs just as `var = undefined;` does due to compilation. Here is the updated anatomy:

1. `function` keyword
2. custom keyword naming the `function`
3. `()` characters enclosing the optional *argument parameters*
4. `{}` characters enclosing the *function body*
5. `return undefined;` at end of *function body* if a custom `return` isn't defined there

Due to this anatomy, a function always has `return undefined;` as its last statement unless a custom `return` is defined instead. For example our familiar:

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

after compilation becomes:

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

In most cases, this does not matter as the same work gets done. What is nice about knowing the anatomy is that *all functions work like this*. Built-in ones, third-party ones, and custom ones. Internalizing the anatomy of a function empowers you to understand any function you encounter in addition to its `return` value. This fact in combination with the *thinking in three zoom levels* technique will help you read, understand, and author code. The name, argument parameters, and the work itself of functions will differ, but the pattern remains. Etch this pattern in your brain.

In *Interactive Code - Anatomy of HTML, CSS, and JavaScript - JavaScript* we learned that a function is structured in two parts:

1. Reference work
2. Core work

Reference work manifests as Variable Declaration and Function Declaration statements. The core work manifests as everything else. Our `makeBackgroundBlack` snippet doesn't need any reference work help so it just does core work. Functions are your friend.

#

Organizational Unit

All functions by default are an organizational unit as they enclose a scope (just as a shape encloses space). By separating and nesting functions coders can *intentionally* organize a program to their liking. This scope organization also allows identifiers (`variable` or `function`) of the same name to exist in different scopes without clashing. These identifiers must be declared in *different functions* for this to work. These are the core takeaways. Example time:

```
function PortraitArtboardElement() {
    var artboardElement = document.createElement('div');
    artboardElement.classList.add('portrait-artboard');
    return artboardElement;
}

function LandscapeArtboardElement() {
    var artboardElement = document.createElement('div');
    artboardElement.classList.add('landscape-artboard');
    return artboardElement;
}
```

In the example above we *organize* our code by declaring the `PortraitArtboardElement` and `LandscapeArtboardElement` functions. You will notice that the `var artboardElement` is declared twice, once in each function. This is not a problem however as each function encloses its own scope. Based on the example snippet below, how many `artboardElement` `var`iables do you think would exist in the program?

```
var portraitArtboardElement1 = new PortraitArtboardElement();
var portraitArtboardElement2 = new PortraitArtboardElement();
var landscapeArtboardElement1 = new LandscapeArtboardElement();
var landscapeArtboardElement2 = new LandscapeArtboardElement();
```

If you guessed four, you are correct. Each `var artboardElement` declaration is *unique to each function execution*. This transitions us to the *instantiable unit* use case.

#

Instantiable Unit

The instantiable unit use case is very common and useful when a function's purpose is to provide a *unique instance of a specific Object*. Built-in, third-party, and custom specific Objects can apply. We have seen a few examples of this:

- `var date = new Date();` (built-in)
- `var artboard = new Artboard();` (custom)
- `var portraitArtboardElement1 = new PortraitArtboardElement();` (custom)

In each example a unique instance is `return`ed and assigned to the respective `var`iable. From this point on, each can be further used in the program where the dot operator provides deeper access into the instance. This dot access manifests in one of two ways:

1. reading an identifier value
2. writing (updating) an identifier value using `=`

Without unique instances our programs would be extremely limited as you can imagine.

Take note of the intentional use of the `UpperCamelCase` naming convention. This is used to denote that a function `return`s a specific Object instance. You might now be wondering, what is the `lowerCamelCase` naming convention useful for? This transitions us to the *reusable work unit*.

#

Reusable Work Unit

The reusable work unit use case is also very common. It is useful for repetitive work. Additionally, the `lowerCamelCase` naming convention is used to distinguish it from the `UpperCamelCase` instantiable unit use case. We have seen numerous examples of this:

- `makeBackgroundBlack();`
- `changeBackgroundColor('#FFFFFF');`
- `toggleImageOpacity();`
- `getMaximumRotation();`
- `toggleIsClockwise();`

- `updateBrickRotation();`
- `updateArtboardInFocus(artboard);`
- `onCreateArtboardButtonClick();`

This type of function embodies the *Mindset - Don't Repeat Yourself* technique. By authoring functions to be flexible and reusable, more can be accomplished with less. Intuitively authoring functions this way takes time but it is a goal to strive for. Ultimately, these types of functions are useful anytime a trigger occurs where work should be done in response. All the function examples above fall in this category.

You may recall the brief mention of *iterating an* `Array` in the *Expressions* section. To iterate an `Array` is to *do work using each item*. This is a perfect use case for reusable work. In fact, iteration is so common that the `Array` type has built-in helper functions. There are five specifically that are extremely useful and worth learning. Here they are in alphabetical order:

- `filter()` - conditionally `return` each item
- `find()` - find the first item that meets a condition and `return` it
- `forEach()` - do work using each item
- `map()` - transform each item into a new value and `return` the new value
- `reduce()` - reduce all items to a single value and `return` that value

These five functions will be useful for most—if not all—of the code you author involving `Array`s. Let's look at examples of each to see how they are useful in practice. Take note that the `colors` and `widths` `Array` identifiers below are plural not singular. This is a best practice.

#

`filter()`

The `filter()` function is useful as it enables us to conditionally `return` each item. Additionally, this function creates a new `Array` automatically for us that contains only these filtered items. In the example below we iterate through each item—colors in this case—and `return` a Boolean value indicating our intention to keep (`true`) or not keep (`false`) the item. This new list of filtered colors is `return`ed from the `filter()` call. We go one step further and assign it to `colorsExcludingBlackAndWhite`.

```
var colors = ['#FFFFFF', '#FF0000', '#00FF00', '#0000FF', '#000000'];
var colorsExcludingBlackAndWhite;

function colorFilter(item) {
    if (item === '#FFFFFF' || item === '#000000') {
        return false;
    } else {
        return true;
    }
}

colorsExcludingBlackAndWhite = colors.filter(colorFilter);
```

Since a function is a value, *we can pass it as an argument to another function*. Mind blown.

#

`find()`

The `find()` function is useful as it enables us to find the first item that meets a condition and then `return` it. In the example below we iterate through each item—colors in this case—*until* the condition is `true` for a given item. When this occurs, that item is `return`ed from the `find()` call. We go one step further and assign it to `greenColor`.

```
var colors = ['#FFFFFF', '#FF0000', '#00FF00', '#0000FF', '#000000'];
var greenColor;

function findGreenColor(item) {
    return item === '#00FF00';
}

greenColor = colors.find(findGreenColor);
```

You guessed it, since a function is a value, we can pass it as an argument to another function.

#

forEach()

The `forEach()` function is useful as it enables us to do work using each item. In the example below we iterate through each item—colors in this case—and use it to set the `backgroundColor` of a newly created `<div>` element.

```
var colors = ['#FFFFFF', '#FF0000', '#00FF00', '#0000FF', '#000000'];

function createColorSwatch(item) {
    var swatchElement = document.createElement('div');
    swatchElement.classList.add('color-swatch');
    swatchElement.style.backgroundColor = item;
    document.body.appendChild(swatchElement);
}

colors.forEach(createColorSwatch);
```

In the case of `forEach()` we pass it the function identifier `createColorSwatch`. Then as part of `forEach()`'s work it calls `createColorSwatch` for each item in our `colors` array. The value—a color in this case—of each iteration is passed as the argument to `createColorSwatch`. Imagine if `colors` consisted of one-hundred colors instead of five. Iteration is powerful and extremely useful.

#

map()

The `map()` function is useful as it enables us to transform each item into a new value and `return` the new value. Additionally, this function creates a new `Array` automatically for us that contains these new values. In the example below we iterate through each item—colors in this case—and use it to create a transparent variation. This new list of transparent colors is `return`ed from the `map()` call. We go one step further and assign it to `transparentColors`. Though we didn't below, we could go a step further still with a `transparentColors.forEach(createColorSwatch);` call. Reuse in action.

```
var colors = ['#FFFFFF', '#FF0000', '#00FF00', '#0000FF', '#000000'];
var transparentColors;

function halveOpacity(item) {
    var transparentColor = item + '80';
    return transparentColor;
}

transparentColors = colors.map(halveOpacity);
```

Again, since a function is a value, we can pass it as an argument to another function. It will take time to get used to this powerful idea.

#

reduce()

The `reduce()` function is useful as it enables us reduce all items to a single value and then `return` that value. In the example below we iterate through each item—numbers in this case—and `return` the current reduced value. This current reduced value is reused again as `previousItem` in the next iteration. This repeats until all values have been iterated over. Put another way, we iterate through each `item` and combine it with `previousItem` into a single value. This reduced value is `return`ed from the `reduce()` call. We go one step further and assign it to `totalWidth`.

```
var widths = [100, 300, 100];
var totalWidth;

function widthReducer(previousItem, item) {
    return previousItem + item;
}

totalWidth = widths.reduce(widthReducer);
```

The `Number 500` is the value of `totalWidth` now. One more time. A function is a value and we can pass it as an argument to another function.

It will take time for these five `Array` functions to become second nature. Look them up to remind yourself how they work and to understand what argument parameters are expected. Remember that you are not alone as professionals—like beginners—need to reference resources. The takeaway is that if you need to iterate, look to these five functions for help.

#

Scope

We know that in order for a function to do its work, it must be called using `()` with any expected arguments in-between. The first example we saw of this was the last line of:

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

makeBackgroundBlack();
```

During execution time two questions surface regarding the snippet:

1. How is `document` accessed if not declared in `makeBackgroundBlack`'s scope?
2. How did the snippet start executing in the first place?

To answer these questions we need to understand the *scope chain* and *execution context*. Understanding these will bring full circle our final *Zoom Level 3 - Value Resolution* technique.

#

Scope Chain

A literal value is obviously one of the primitive or complex types. This is easy to resolve. Resolving an *identifier's* value is not obvious. As such, the scope chain is useful for *resolving identifier values*. We briefly covered how it works in the *Interactive Code - Anatomy of HTML, CSS, and JavaScript - JavaScript* section. Now we give it a name and dive a little deeper.

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 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

This process defines the scope chain. With it, the found identifier's value is used in place of the identifier itself. The identifier value is now resolved. Winning. We know a JavaScript program is a nested tree of scopes. The scope chain refers to the process of walking up this scope tree in search of an identifier and thus its value.

So to answer question one above the process is:

1. look for `document` in the current scope (`makeBackgroundBlack` scope)
    - not found
2. look for `document` in the current scope's parent (global scope)
    - `window` is the browser environment's global object
    - `window.document` is found
3. replace the initial statement's `window.document` with the found value
    - the value is a a specific `Object` defined by the environment
4. continue work using this found value

As we saw in the *Keywords - Non-Reserved Keywords - Environment* section, the value assigned to `window.document`—by the environment—is a specific `Object`. To refresh, here is the partial example:

```
window.document = {
    body: {
        style: {
            backgroundColor: ''
        }
    }
}
```

Here is another example to showcase the same scope chain process using a custom identifier and a nesting of `function`s:

```
var someIdentifier = 'Winning!';

function one() {

    function two() {

        function three() {
            console.log(someIdentifier);
        }

        three();
    }

    two();
}

one();
```

The identifier `someIdentifier` does not exist in `three`'s scope (nor does `console` for that matter). It does not exist in `two`'s or `one`'s either (same with `console`). It does however exist in the global scope (`console` does too). As a result, `console.log('Winning!')` executes as `someIdentifer` resolves to `Winning!` where `console` resolves to an environment object.

The takeaway is that a child scope can look to a parent scope for an identifier. It does not work the other way around. Additionally, the nesting of scopes has no limit of depth. Thank you scope chain.

What happens if an identifier is *not* found? In these scenarios we have an error.

```
function doSomeWork() {
    console.log(someUndeclaredIdentifier);
    console.log('This never executes :(');
}

doSomeWork();
```

Our program breaks when the above snippet is executed. Additionally, no further statements after the error will be executed. Our program is broken. This is not good.

It is important for us to use our editor, `debugger`, and a browser's developer tools to prevent errors. With our developer tools open we would see `Uncaught ReferenceError: someUndeclaredIdentifier is not defined`. We will explain this and other common `Error`s in the *Errors* section.

#

Execution Context

To answer the *how did the snippet start executing in the first place?* question we need to more deeply understand *execution context*. We have and will continue to just use the term scope however.

We know a function encloses a scope just as a shape encloses space. A scope only ever exists when a function gets *executed* though. So the code:

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

Is only a *single* Function Declaration Statement with *no scope* where each execution of it results in a *new scope*. Our understanding of *Interactive Code - Authoring, Compiling, and Executing* is of use here.

At authoring time we declare and define this `changeBackgroundColor function` *knowing* that each call of it during execution time will result in a new scope. Each execution—each new scope—manifests as an addition to the stack. The stack *grows*. When the top most `function` on the stack `return`s, this is the moment our stack *shrinks*. The event loop becomes unblocked once all the `function` executions have `return`ed.

For each execution the `newColor` identifier's value may be different. It differs based on the argument value passed in. This fact ensures the snippet below results in four distinct scopes where each `changeBackgroundColor` call has a different `newValue` as a result. The same `document` value is used due to the scope chain. Pretty sweet.

```
changeBackgroundColor('#FF0000');
changeBackgroundColor('#00FF00');
changeBackgroundColor('#0000FF');
```

There are four and not three scopes in play above. What is the fourth? If you guessed the global scope then you are correct. This leads us to finally answering *how did the snippet start executing in the first place?* Do you remember the `<script>` tag introduced in the *Interactive Code - Anatomy of HTML, CSS, and JavaScript - JavaScript* section? Bingo.

Each `<script>` is essentially a function call. The stack grows. Once the last line of the `<script>`'s code is executed, it `return`s. The stack shrinks.

It's been implied that all snippets thus far exist in a particular `.js` file loaded by the `<script>` tag's `src` attr. Our above snippet can be thought of as:

```
function global1() {
    changeBackgroundColor('#FF0000');
    changeBackgroundColor('#00FF00');
    changeBackgroundColor('#0000FF');
}

global1();
```

We can load many different `<script>` tags allowing us to further organize our code in unique `.js` files. Declaring and calling `global1` is done by the engine. The stack grows and shrinks just as if we defined and called `global1` ourselves.

The takeaway is that the scope chain is used to resolve identifier values. Understanding it is a requirement for the computer to run our program. When us coders understand it, we know how to substitute an identifier with the proper value. When we understand execution context, we see how scopes exist and how code gets executed in the first place. Thank you `<script>` tag.

#

Errors

Errors suck. They are inevitable however. Even the most seasoned professionals encounter them. A professional's knowledge and experience simply increases his or her chance of a quick resolution.

To help you more quickly resolve errors, we'll focus on the subset that you are most likely to encounter. There are three specific types of `Error`s worth focusing on:

- `SyntaxError`
- `ReferenceError`
- `TypeError`

`SyntaxError`s relate to typos and they are experienced during authoring or compile time. The other two types—`ReferenceError` and `TypeError`—fundamentally revolve around `undefined` or `null` and they are experienced during execution time. Let's dig a little deeper into each so you have a concrete understanding of what is happening when you encounter them.

#

`SyntaxError`

When you encounter a `SyntaxError`, a specific code statement will likely have:

- invalid identifier(s) or character(s)
- missing required character(s)

In either case, a typo exists. The code will accidentally contain one or more unknown characters in the first case. As for the second case, an expected character will be missing.

Below are a few examples, each with an associated comment. The comment denotes the specific error details that would display in a browser's developer tools console during execution time. We'll explore these tools in the *Debugging* section later in this chapter.

```
// Uncaught SyntaxError: Unexpected identifier
vars color = '#FFFFFF';
```

What is wrong with the above code snippet? The `vars` should be `var`. The compiler program doesn't know what `vars` is and throws a `SyntaxError` because it doesn't know what to do. We could have a snippet like `var vars = 'vars';` because `vars` in this example would be a valid `var`iable identifier name and `'vars'` a valid string value. Typos matter.

```
// Uncaught SyntaxError: Invalid or unexpected token
var color = #FFFFFF;
```

What is wrong with the above code snippet? If you said our color definition isn't a valid string value then you nailed it. Our `#FFFFFF` must become `'#FFFFFF'` or `"#FFFFFF"` to become valid. Again, typos matter.

Do yourself a favor and mentally link a `SyntaxError` to a typo. From here you will have at least identified the core problem so you may more quickly resolve the error. Debugging tools will further help you.

#

ReferenceError

When you encounter a `ReferenceError`, a specific code statement will likely have:

- referenced an undeclared identifier
- referenced an identifier outside the current scope chain

In either case, an identifier is inaccessible. The scope chain will be traversed in the first case, but the identifier will not be found—because it was never declared. As for the second case, the identifier may have been declared but its declaration exists outside the current scope chain.

Below are a few examples, each with an associated comment as before.

```
// Uncaught ReferenceError: selectedColor is not defined
var colorInFocus = selectedColor;
```

What is wrong with the above code snippet? The syntax is correct, so no issue there. However, `selectedColor` does not exist in the scope chain. It is an undeclared identifier. Declarations matter.

```
function rollDice() {
  var sideCount = 6;
  var randomDiceValue = Math.ceil(Math.random() * sideCount);
  return randomDiceValue;
}

// Uncaught ReferenceError: sideCount is not defined
console.log('The ' + sideCount + '-sided dice roll result is ' + rollDice());
```

What is wrong with the above code snippet? You nailed it if you recognized that `sideCount` is only accessible in the `rollDice() function`'s scope. The `sideCount` identifier declaration exists, but it is not in the scope chain of the `console.log` statement referencing it. Declarations—and the scope in which they are declared—matter.

Refresh with the *Functions - Scope Chain* section if your understanding of how the scope chain works is still a little fuzzy.

#

TypeError

When you encounter a `TypeError`, a specific code statement will likely have:

- attempted to access a nested identifier of an `undefined` or `null` object
- called an identifier with `()` as if it referenced a `function`

In either case, a basic or specific `Object` type is misused. The code will reference an identifier of an `undefined` or `null` value in the first case. Each of these special values are literal values that don't contain nested identifiers. As for the second case, an identifier will be used as a `function` when it is not one.

As before, below are a few examples, each with an associated comment.

```
var blackObject = { name: 'black', color: '#000000' };
var whiteObject;
```

```
// Uncaught TypeError: Cannot read property 'name' of undefined
console.log("Oreo cookies are " + blackObject.name + " and " + whiteObject.name);
```

What is wrong with the above code snippet? As you may recall, our `whiteObject` is `undefined` by default. We forgot to assign it a value. Accessing a nested identifier—`name` in this example—of the `undefined` value is an error. The same is true if our `whiteObject`'s assigned value was `null`. Using types properly matters.

```
var blackObject = { name: 'black', color: '#000000' };
var whiteObject = { name: 'white', color: '#FFFFFF' };

// Uncaught TypeError: blackObject.name is not a function
console.log("Oreo cookies are " + blackObject.name() + " and " + whiteObject.name;
```

What is wrong with the above code snippet? We updated our `whiteObject` by assigning it a value to resolve our issue in the previous snippet. However, we are now accidentally calling `blackObject.name` as if it is a `function`. It is a string, not a `function` though so we get an error as a result. Using types properly matters.

#

Debugging

Each comment associated with the error examples above was provided for context. It doesn't show up in the code otherwise. To see `Errors`, you need to use debugging tools.

Most browsers have a built-in subprogram called something like *developer tools*. Unsurprisingly these tools are for us coders. With them, any website's underlying HTML, CSS, and JavaScript code is explorable. This is a big deal and extremely useful. These tools help us understand how a particular website or web app works. It is advantageous to use the tools while quickly iterating during the author, compile, and execution time cycle. They provide another view into our code that isn't just the rendered output of our creation.

Of the many built-in developer tools, there are three that will be most useful to you. There exact names vary among browsers, but they'll closely align to the following:

1. Elements - view the HTML structure in real-time
2. Console - view `console.log` output and `Error`s in real-time
3. Sources - view the source code of files

All three are super useful, but the console is where you'll see a `SyntaxError`, `ReferenceError`, or `TypeError`. A clean console is the goal. It is best practice to remove your `console.log` statements after you've confirmed they produce the expected output.

The `debugger` statement is another useful debugging tool that was previously mentioned. It is useful because it opens the developer tools at the exact line of code it exists on. Additionally, execution freezes—until it is resumed—so state can be examined at an exact moment in time. Super powerful.

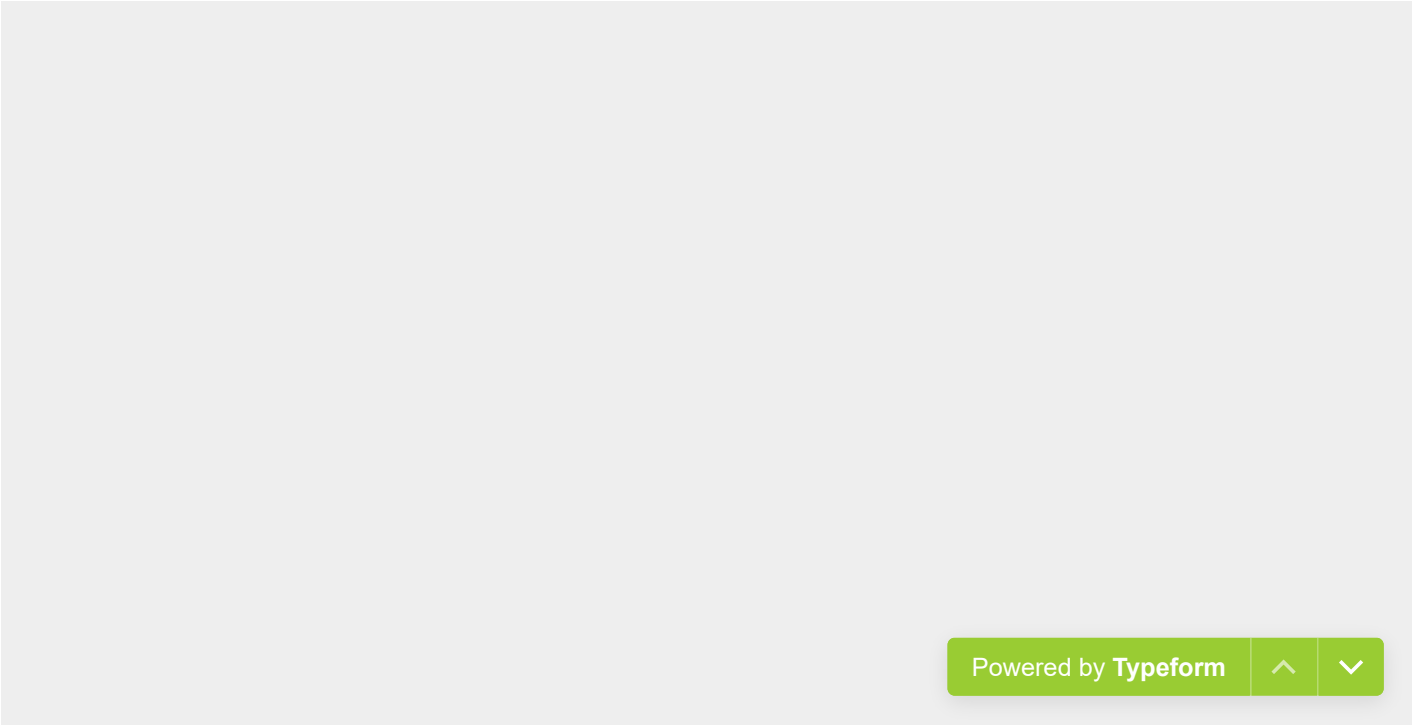
Lastly, your editor is another useful tool that is immensely empowering. It improves authoring time, but it can be additionally useful for debugging. Some common features of editors include:

- code-completion - help prevent typos and maintain code consistency
- color coding - aid the scanning and reading of code
- linter - help prevent typos and maintain code consistency
- snippets - help prevent typos and maintain code consistency

All of them additionally speed up authoring time.

Though code can be authored using a basic text editor, it's worth using a sophisticated text editor and other visual code generation tools. They give you super powers in comparison.

Errors still suck, but you are better equipped to resolve them now.



Chapter 4 Interactive Code

Chapter 6 Deconstructing Designs