You Don't Know JS: Up & Going

Chapter 1: Into Programming

Welcome to the *You Don't Know JS* (*YDKJS*) series.

Up & Going is an introduction to several basic concepts of programming – of course we lean toward JavaScript (often abbreviated JS) specifically – and how to approach and understand the rest of the titles in this series. Especially if you're just getting into programming and/or JavaScript, this book will briefly explore what you need to get *up and going*.

This book starts off explaining the basic principles of programming at a very high level. It's mostly intended if you are starting *YDKJS* with little to no prior programming experience, and are looking to these books to help get you started along a path to understanding programming through the lens of JavaScript.

Chapter 1 should be approached as a quick overview of the things you'll want to learn more about and practice to get *into programming*. There are also many other fantastic programming introduction resources that can help you dig into these topics further, and I encourage you to learn from them in addition to this chapter.

Once you feel comfortable with general programming basics, Chapter 2 will help guide you to a familiarity with JavaScript's flavor of programming. Chapter 2 introduces what JavaScript is about, but again, it's not a comprehensive guide – that's what the rest of the *YDKJS* books are for!

If you're already fairly comfortable with JavaScript, first check out Chapter 3 as a brief glimpse of what to expect from *YDKJS*, then jump right in!

Code

Let's start from the beginning.

A program, often referred to as *source code* or just *code*, is a set of special instructions to tell the computer what tasks to perform. Usually code is saved in a text file, although with JavaScript you can also type code directly into a developer console in a browser, which we'll cover shortly.

The rules for valid format and combinations of instructions is called a *computer language*, sometimes referred to as its *syntax*, much the same as English tells you how to spell words and how to create valid sentences using words and punctuation.

Statements

In a computer language, a group of words, numbers, and operators that performs a specific task is a *statement*. In JavaScript, a statement might look as follows:

```
a = b * 2;
```

The characters `a` and `b` are called *variables* (see "Variables"), which are like simple boxes you can store any of your stuff in. In programs, variables hold values (like the number `42`) to be used by the program. Think of them as symbolic placeholders for the values themselves.

By contrast, the `2` is just a value itself, called a *literal value*, because it stands alone without being stored in a variable.

The `=` and `*` characters are *operators* (see "Operators") – they perform actions with the values and variables such as assignment and mathematic multiplication.

Most statements in JavaScript conclude with a semicolon (`;`) at the end.

The statement `a = b * 2;` tells the computer, roughly, to get the current value stored in the variable `b`, multiply that value by `2`, then store the result back into another variable we call `a`.

Programs are just collections of many such statements, which together describe all the steps that it takes to perform your program's purpose.

Expressions

Statements are made up of one or more *expressions*. An expression is any reference to a variable or value, or a set of variable(s) and value(s) combined with operators.

For example:

```
a = b * 2;
```

This statement has four expressions in it:

- `2` is a *literal value expression*
- `b` is a *variable expression*, which means to retrieve its current value
- `b * 2` is an *arithmetic expression*, which means to do the multiplication
- `a = b * 2` is an *assignment expression*, which means to assign the result of the `b * 2` expression to the variable `a` (more on assignments later)

A general expression that stands alone is also called an *expression statement*, such as the following:

```
b * 2;
```

This flavor of expression statement is not very common or useful, as generally it wouldn't have any effect on the running of the program – it would retrieve the value of `b` and multiply it by `2`, but then wouldn't do anything with that result.

A more common expression statement is a *call expression* statement (see "Functions"), as the entire statement is the function call expression itself:

```
alert( a );
```

Executing a Program

How do those collections of programming statements tell the computer what to do? The program needs to be *executed*, also referred to as *running the program*.

Statements like `a = b * 2` are helpful for developers when reading and writing, but are not actually in a form the computer can directly understand. So a special utility on the computer (either an *interpreter* or a *compiler*) is used to translate the code you write into commands a computer can understand.

For some computer languages, this translation of commands is typically done from top to bottom, line by line, every time the program is run, which is usually called *interpreting* the code.

For other languages, the translation is done ahead of time, called *compiling* the code, so when the program *runs* later, what's running is actually the already compiled computer instructions ready to go.

It's typically asserted that JavaScript is *interpreted*, because your JavaScript source code is processed each time it's run. But that's not entirely accurate. The JavaScript engine actually *compiles* the program on the fly and then immediately runs the compiled code.

Note: For more information on JavaScript compiling, see the first two chapters of the *Scope & Closures* title of this series.

Try It Yourself

This chapter is going to introduce each programming concept with simple snippets of code, all written in JavaScript (obviously!).

It cannot be emphasized enough: while you go through this chapter – and you may need to spend the time to go over it several times – you should practice each of these concepts by typing the code yourself. The easiest way to do that is to open up the developer tools console in your nearest browser (Firefox, Chrome, IE, etc.).

Tip: Typically, you can launch the developer console with a keyboard shortcut or from a menu item. For more detailed information about launching and using the

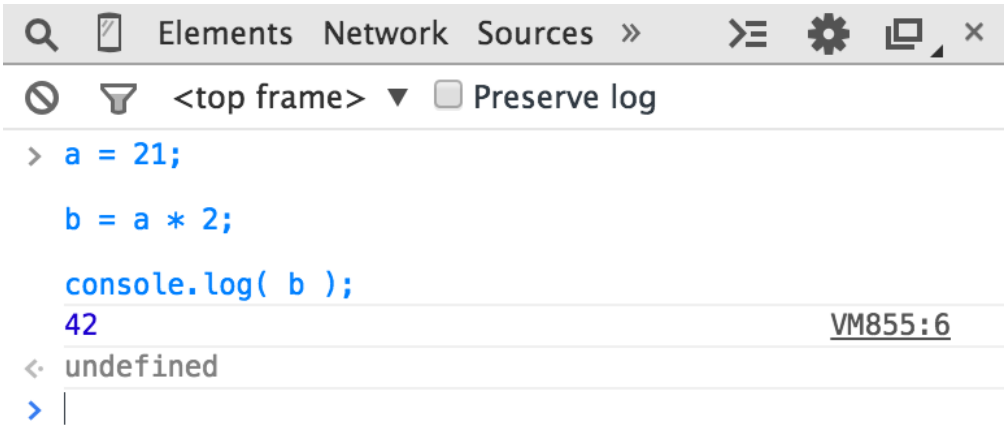
console in your favorite browser, see "Mastering The Developer Tools Console" (http://blog.teamtreehouse.com/mastering-developer-tools-console). To type multiple lines into the console at once, use `<shift> + <enter>` to move to the next new line. Once you hit `<enter>` by itself, the console will run everything you've just typed.

Let's get familiar with the process of running code in the console. First, I suggest opening up an empty tab in your browser. I prefer to do this by typing `about:blank` into the address bar. Then, make sure your developer console is open, as we just mentioned.

Now, type this code and see how it runs:

```
a = 21;

b = a * 2;

console.log( b );
```

Typing the preceding code into the console in Chrome should produce something like the following:



Go on, try it. The best way to learn programming is to start coding!

Output

In the previous code snippet, we used `console.log(..)`. Briefly, let's look at what that line of code is all about.

You may have guessed, but that's exactly how we print text (aka *output* to the user) in the developer console. There are two characteristics of that statement that we should explain.

First, the `log( b )` part is referred to as a function call (see "Functions"). What's happening is we're handing the `b` variable to that function, which asks it to take the value of `b` and print it to the console.

Second, the `console.` part is an object reference where the `log(..)` function is located. We cover objects and their properties in more detail in Chapter 2.

Another way of creating output that you can see is to run an `alert(..)` statement. For example:

```
alert( b );
```

If you run that, you'll notice that instead of printing the output to the console, it shows a popup "OK" box with the contents of the `b` variable. However, using `console.log(..)` is generally going to make learning about coding and running your programs in the console easier than using `alert(..)`, because you can output many values at once without interrupting the browser interface.

For this book, we'll use `console.log(..)` for output.

Input

While we're discussing output, you may also wonder about *input* (i.e., receiving information from the user).

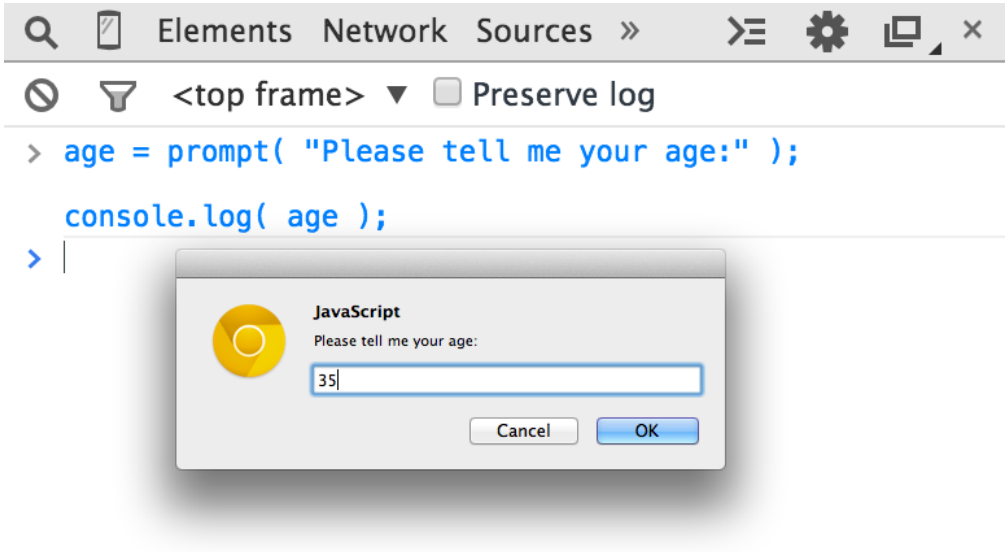
The most common way that happens is for the HTML page to show form elements (like text boxes) to a user that they can type into, and then using JS to read those values into your program's variables.

But there's an easier way to get input for simple learning and demonstration purposes such as what you'll be doing throughout this book. Use the `prompt(..)` function:

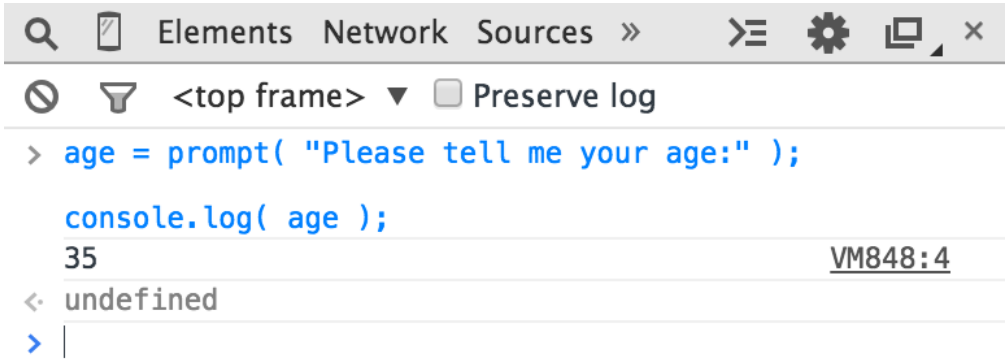
```
age = prompt( "Please tell me your age:" );

console.log( age );
```

As you may have guessed, the message you pass to `prompt(..)` – in this case, `"Please tell me your age:"` – is printed into the popup.

This should look similar to the following:



Once you submit the input text by clicking "OK," you'll observe that the value you typed is stored in the `age` variable, which we then *output* with `console.log(..)`:



To keep things simple while we're learning basic programming concepts, the examples in this book will not require input. But now that you've seen how to use `prompt(..)`, if you want to challenge yourself you can try to use input in your explorations of the examples.

Operators

Operators are how we perform actions on variables and values. We've already seen two JavaScript operators, the `=` and the `*`.

The `*` operator performs mathematic multiplication. Simple enough, right?

The `=` equals operator is used for *assignment* – we first calculate the value on the *right-hand side* (source value) of the `=` and then put it into the variable that we specify on the *left-hand side* (target variable).

Warning: This may seem like a strange reverse order to specify assignment. Instead of `a = 42`, some might prefer to flip the order so the source value is on the left and the target variable is on the right, like `42 -> a` (this is not valid JavaScript!). Unfortunately, the `a = 42` ordered form, and similar variations, is quite prevalent in modern programming languages. If it feels unnatural, just spend some time rehearsing that ordering in your mind to get accustomed to it.

Consider:

```
a = 2;
b = a + 1;
```

Here, we assign the `2` value to the `a` variable. Then, we get the value of the `a` variable (still `2`), add `1` to it resulting in the value `3`, then store that value in the `b` variable.

While not technically an operator, you'll need the keyword `var` in every program, as it's the primary way you *declare* (aka *create*) *var*iables (see "Variables").

You should always declare the variable by name before you use it. But you only need to declare a variable once for each *scope* (see "Scope"); it can be used as many times after that as needed. For example:

```
var a = 20;

a = a + 1;
a = a * 2;

console.log( a );   // 42
```

Here are some of the most common operators in JavaScript:

- Assignment: `=` as in `a = 2`.
- Math: `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division), as in `a * 3`.
- Compound Assignment: `+=`, `-=`, `*=`, and `/=` are compound operators that combine a math operation with assignment, as in `a += 2` (same as `a = a + 2`).
- Increment/Decrement: `++` (increment), `--` (decrement), as in `a++` (similar to `a = a + 1`).
- Object Property Access: `.` as in `console.log()`.

  Objects are values that hold other values at specific named locations called properties. `obj.a` means an object value called `obj` with a property of

the name `a`. Properties can alternatively be accessed as `obj["a"]`. See Chapter 2.

- Equality: `==` (loose-equals), `===` (strict-equals), `!=` (loose not-equals), `!==` (strict not-equals), as in `a == b`.

  See "Values & Types" and Chapter 2.

- Comparison: `<` (less than), `>` (greater than), `<=` (less than or loose-equals), `>=` (greater than or loose-equals), as in `a <= b`.

  See "Values & Types" and Chapter 2.

- Logical: `&&` (and), `||` (or), as in `a || b` that selects either `a` *or* `b`.

  These operators are used to express compound conditionals (see "Conditionals"), like if either `a` *or* `b` is true.

Note: For much more detail, and coverage of operators not mentioned here, see the Mozilla Developer Network (MDN)'s "Expressions and Operators" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators).

Values & Types

If you ask an employee at a phone store how much a certain phone costs, and they say "ninety-nine, ninety-nine" (i.e., $99.99), they're giving you an actual numeric dollar figure that represents what you'll need to pay (plus taxes) to buy it. If you want to buy two of those phones, you can easily do the mental math to double that value to get $199.98 for your base cost.

If that same employee picks up another similar phone but says it's "free" (perhaps with air quotes), they're not giving you a number, but instead another kind of representation of your expected cost ($0.00) – the word "free."

When you later ask if the phone includes a charger, that answer could only have been either "yes" or "no."

In very similar ways, when you express values in a program, you choose different representations for those values based on what you plan to do with them.

These different representations for values are called *types* in programming terminology. JavaScript has built-in types for each of these so called *primitive* values:

- When you need to do math, you want a `number`.
- When you need to print a value on the screen, you need a `string` (one or more characters, words, sentences).
- When you need to make a decision in your program, you need a `boolean` (`true` or `false`).

Values that are included directly in the source code are called *literals*. `string` literals are surrounded by double quotes `"..."` or single quotes (`'...'`) – the only difference is stylistic preference. `number` and `boolean` literals are just presented as is (i.e., `42`, `true`, etc.).

Consider:

```
"I am a string";
'I am also a string';

42;

true;
false;
```

Beyond `string`/`number`/`boolean` value types, it's common for programming languages to provide *arrays*, *objects*, *functions*, and more. We'll cover much more about values and types throughout this chapter and the next.

Converting Between Types

If you have a `number` but need to print it on the screen, you need to convert the value to a `string`, and in JavaScript this conversion is called "coercion." Similarly, if someone enters a series of numeric characters into a form on an ecommerce page, that's a `string`, but if you need to then use that value to do math operations, you need to *coerce* it to a `number`.

JavaScript provides several different facilities for forcibly coercing between *types*. For example:

```
var a = "42";
var b = Number( a );

console.log( a );   // "42"
console.log( b );   // 42
```

Using `Number(..)` (a built-in function) as shown is an *explicit* coercion from any other type to the `number` type. That should be pretty straightforward.

But a controversial topic is what happens when you try to compare two values that are not already of the same type, which would require *implicit* coercion.

When comparing the string `"99.99"` to the number `99.99`, most people would agree they are equivalent. But they're not exactly the same, are they? It's the same value in two different representations, two different *types*. You could say they're "loosely equal," couldn't you?

To help you out in these common situations, JavaScript will sometimes kick in and *implicitly* coerce values to the matching types.

So if you use the `==` loose equals operator to make the comparison `"99.99" == 99.99`, JavaScript will convert the left-hand side `"99.99"` to its `number` equivalent `99.99`. The comparison then becomes `99.99 == 99.99`, which is of course `true`.

While designed to help you, implicit coercion can create confusion if you haven't taken the time to learn the rules that govern its behavior. Most JS developers never have, so the common feeling is that implicit coercion is confusing and harms programs with unexpected bugs, and should thus be avoided. It's even sometimes called a flaw in the design of the language.

However, implicit coercion is a mechanism that *can be learned*, and moreover *should be learned* by anyone wishing to take JavaScript programming seriously. Not only is it not confusing once you learn the rules, it can actually make your programs better! The effort is well worth it.

Note: For more information on coercion, see Chapter 2 of this title and Chapter 4 of the *Types & Grammar* title of this series.

Code Comments

The phone store employee might jot down some notes on the features of a newly released phone or on the new plans her company offers. These notes are only for the employee – they're not for customers to read. Nevertheless, these notes help the employee do her job better by documenting the hows and whys of what she should tell customers.

One of the most important lessons you can learn about writing code is that it's not just for the computer. Code is every bit as much, if not more, for the developer as it is for the compiler.

Your computer only cares about machine code, a series of binary 0s and 1s, that comes from *compilation*. There's a nearly infinite number of programs you could write that yield the same series of 0s and 1s. The choices you make about how to write your program matter – not only to you, but to your other team members and even to your future self.

You should strive not just to write programs that work correctly, but programs that make sense when examined. You can go a long way in that effort by choosing good names for your variables (see "Variables") and functions (see "Functions").

But another important part is code comments. These are bits of text in your program that are inserted purely to explain things to a human. The interpreter/compiler will always ignore these comments.

There are lots of opinions on what makes well-commented code; we can't really define absolute universal rules. But some observations and guidelines are quite useful:

- Code without comments is suboptimal.
- Too many comments (one per line, for example) is probably a sign of poorly written code.
- Comments should explain *why*, not *what*. They can optionally explain *how* if that's particularly confusing.

In JavaScript, there are two types of comments possible: a single-line comment and a multiline comment.

Consider:

```
// This is a single-line comment

/* But this is
       a multiline
             comment.
                    */
```

The `//` single-line comment is appropriate if you're going to put a comment right above a single statement, or even at the end of a line. Everything on the line after the `//` is treated as the comment (and thus ignored by the compiler), all the way to the end of the line. There's no restriction to what can appear inside a single-line comment.

Consider:

```
var a = 42;      // 42 is the meaning of life
```

The `/* .. */` multiline comment is appropriate if you have several lines worth of explanation to make in your comment.

Here's a common usage of multiline comments:

```
/* The following value is used because
   it has been shown that it answers
   every question in the universe. */
var a = 42;
```

It can also appear anywhere on a line, even in the middle of a line, because the `*/` ends it. For example:

```
var a = /* arbitrary value */ 42;

console.log( a );   // 42
```

The only thing that cannot appear inside a multiline comment is a `*/`, because that would be interpreted to end the comment.

You will definitely want to begin your learning of programming by starting off with the habit of commenting code. Throughout the rest of this chapter, you'll see I use comments to explain things, so do the same in your own practice. Trust me, everyone who reads your code will thank you!

Variables

Most useful programs need to track a value as it changes over the course of the program, undergoing different operations as called for by your program's intended tasks.

The easiest way to go about that in your program is to assign a value to a symbolic container, called a *variable* – so called because the value in this container can *vary* over time as needed.

In some programming languages, you declare a variable (container) to hold a specific type of value, such as `number` or `string`. *Static typing*, otherwise known as *type enforcement*, is typically cited as a benefit for program correctness by preventing unintended value conversions.

Other languages emphasize types for values instead of variables. *Weak typing*, otherwise known as *dynamic typing*, allows a variable to hold any type of value at any time. It's typically cited as a benefit for program flexibility by allowing a single variable to represent a value no matter what type form that value may take at any given moment in the program's logic flow.

JavaScript uses the latter approach, *dynamic typing*, meaning variables can hold values of any *type* without any *type* enforcement.

As mentioned earlier, we declare a variable using the `var` statement – notice there's no other *type* information in the declaration. Consider this simple program:

```
var amount = 99.99;

amount = amount * 2;

console.log( amount );      // 199.98

// convert `amount` to a string, and
// add "$" on the beginning
amount = "$" + String( amount );

console.log( amount );      // "$199.98"
```

The `amount` variable starts out holding the number `99.99`, and then holds the `number` result of `amount * 2`, which is `199.98`.

The first `console.log(..)` command has to *implicitly* coerce that `number` value to a `string` to print it out.

Then the statement `amount = "$" + String(amount)` *explicitly* coerces the `199.98` value to a `string` and adds a `"$"` character to the beginning. At this point, `amount` now holds the `string` value `"$199.98"`, so the second `console.log(..)` statement doesn't need to do any coercion to print it out.

JavaScript developers will note the flexibility of using the `amount` variable for each of the `99.99`, `199.98`, and the `"$199.98"` values. Static-typing enthusiasts would prefer a separate variable like `amountStr` to hold the final `"$199.98"` representation of the value, because it's a different type.

Either way, you'll note that `amount` holds a running value that changes over the course of the program, illustrating the primary purpose of variables: managing program *state*.

In other words, *state* is tracking the changes to values as your program runs.

Another common usage of variables is for centralizing value setting. This is more typically called *constants*, when you declare a variable with a value and intend for that value to *not change* throughout the program.

You declare these *constants*, often at the top of a program, so that it's convenient for you to have one place to go to alter a value if you need to. By convention, JavaScript variables as constants are usually capitalized, with underscores `_` between multiple words.

Here's a silly example:

```
var TAX_RATE = 0.08;     // 8% sales tax

var amount = 99.99;

amount = amount * 2;

amount = amount + (amount * TAX_RATE);

console.log( amount );              // 215.9784
console.log( amount.toFixed( 2 ) ); // "215.98"
```

Note: Similar to how `console.log(..)` is a function `log(..)` accessed as an object property on the `console` value, `toFixed(..)` here is a function that can be accessed on `number` values. JavaScript `number`s aren't automatically formatted for dollars – the engine doesn't know what your intent is and there's no type for currency. `toFixed(..)` lets us specify how many decimal places we'd like the `number` rounded to, and it produces the `string` as necessary.

The `TAX_RATE` variable is only *constant* by convention – there's nothing special in this program that prevents it from being changed. But if the city raises the sales tax rate to 9%, we can still easily update our program by setting the `TAX_RATE` assigned value to `0.09` in one place, instead of finding many occurrences of the value `0.08` strewn throughout the program and updating all of them.

The newest version of JavaScript at the time of this writing (commonly called "ES6") includes a new way to declare *constants*, by using `const` instead of `var`:

```
// as of ES6:
const TAX_RATE = 0.08;
```

```
var amount = 99.99;

// ..
```

Constants are useful just like variables with unchanged values, except that constants also prevent accidentally changing value somewhere else after the initial setting. If you tried to assign any different value to `TAX_RATE` after that first declaration, your program would reject the change (and in strict mode, fail with an error – see "Strict Mode" in Chapter 2).

By the way, that kind of "protection" against mistakes is similar to the static-typing type enforcement, so you can see why static types in other languages can be attractive!

Note: For more information about how different values in variables can be used in your programs, see the *Types & Grammar* title of this series.

Blocks

The phone store employee must go through a series of steps to complete the checkout as you buy your new phone.

Similarly, in code we often need to group a series of statements together, which we often call a *block*. In JavaScript, a block is defined by wrapping one or more statements inside a curly-brace pair `{ .. }`. Consider:

```
var amount = 99.99;

// a general block
{
    amount = amount * 2;
    console.log( amount );  // 199.98
}
```

This kind of standalone `{ .. }` general block is valid, but isn't as commonly seen in JS programs. Typically, blocks are attached to some other control statement, such as an `if` statement (see "Conditionals") or a loop (see "Loops"). For example:

```
var amount = 99.99;

// is amount big enough?
if (amount > 10) {          // <-- block attached to `if`
    amount = amount * 2;
    console.log( amount );  // 199.98
}
```

We'll explain `if` statements in the next section, but as you can see, the `{ .. }` block with its two statements is attached to `if (amount > 10)`; the statements inside the block will only be processed if the conditional passes.

Note: Unlike most other statements like `console.log(amount);`, a block statement does not need a semicolon (`;`) to conclude it.

Conditionals

"Do you want to add on the extra screen protectors to your purchase, for $9.99?" The helpful phone store employee has asked you to make a decision. And you may need to first consult the current *state* of your wallet or bank account to answer that question. But obviously, this is just a simple "yes or no" question.

There are quite a few ways we can express *conditionals* (aka decisions) in our programs.

The most common one is the `if` statement. Essentially, you're saying, "*If* this condition is true, do the following...". For example:

```
var bank_balance = 302.13;
var amount = 99.99;

if (amount < bank_balance) {
    console.log( "I want to buy this phone!" );
}
```

The `if` statement requires an expression in between the parentheses `( )` that can be treated as either `true` or `false`. In this program, we provided the expression `amount < bank_balance`, which indeed will either evaluate to `true` or `false` depending on the amount in the `bank_balance` variable.

You can even provide an alternative if the condition isn't true, called an `else` clause. Consider:

```
const ACCESSORY_PRICE = 9.99;

var bank_balance = 302.13;
var amount = 99.99;

amount = amount * 2;

// can we afford the extra purchase?
if ( amount < bank_balance ) {
    console.log( "I'll take the accessory!" );
    amount = amount + ACCESSORY_PRICE;
}
// otherwise:
```

```
else {
    console.log( "No, thanks." );
}
```

Here, if `amount < bank_balance` is `true`, we'll print out `"I'll take the accessory!"` and add the `9.99` to our `amount` variable. Otherwise, the `else` clause says we'll just politely respond with `"No, thanks."` and leave `amount` unchanged.

As we discussed in "Values & Types" earlier, values that aren't already of an expected type are often coerced to that type. The `if` statement expects a `boolean`, but if you pass it something that's not already `boolean`, coercion will occur.

JavaScript defines a list of specific values that are considered "falsy" because when coerced to a `boolean`, they become `false` – these include values like `0` and `""`. Any other value not on the "falsy" list is automatically "truthy" – when coerced to a `boolean` they become `true`. Truthy values include things like `99.99` and `"free"`. See "Truthy & Falsy" in Chapter 2 for more information.

Conditionals exist in other forms besides the `if`. For example, the `switch` statement can be used as a shorthand for a series of `if..else` statements (see Chapter 2). Loops (see "Loops") use a *conditional* to determine if the loop should keep going or stop.

Note: For deeper information about the coercions that can occur implicitly in the test expressions of *conditionals*, see Chapter 4 of the *Types & Grammar* title of this series.

Loops

During busy times, there's a waiting list for customers who need to speak to the phone store employee. While there's still people on that list, she just needs to keep serving the next customer.

Repeating a set of actions until a certain condition fails – in other words, repeating only while the condition holds – is the job of programming loops; loops can take different forms, but they all satisfy this basic behavior.

A loop includes the test condition as well as a block (typically as `{ .. }`). Each time the loop block executes, that's called an *iteration*.

For example, the `while` loop and the `do..while` loop forms illustrate the concept of repeating a block of statements until a condition no longer evaluates to `true`:

```
while (numOfCustomers > 0) {
    console.log( "How may I help you?" );

    // help the customer...
```

```
    numOfCustomers = numOfCustomers - 1;
}

// versus:

do {
    console.log( "How may I help you?" );

    // help the customer...

    numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

The only practical difference between these loops is whether the conditional is tested before the first iteration (`while`) or after the first iteration (`do..while`).

In either form, if the conditional tests as `false`, the next iteration will not run. That means if the condition is initially `false`, a `while` loop will never run, but a `do..while` loop will run just the first time.

Sometimes you are looping for the intended purpose of counting a certain set of numbers, like from `0` to `9` (ten numbers). You can do that by setting a loop iteration variable like `i` at value `0` and incrementing it by `1` each iteration.

Warning: For a variety of historical reasons, programming languages almost always count things in a zero-based fashion, meaning starting with `0` instead of `1`. If you're not familiar with that mode of thinking, it can be quite confusing at first. Take some time to practice counting starting with `0` to become more comfortable with it!

The conditional is tested on each iteration, much as if there is an implied `if` statement inside the loop.

We can use JavaScript's `break` statement to stop a loop. Also, we can observe that it's awfully easy to create a loop that would otherwise run forever without a `break`ing mechanism.

Let's illustrate:

```
var i = 0;

// a `while..true` loop would run forever, right?
while (true) {
    // stop the loop?
    if ((i <= 9) === false) {
        break;
    }

    console.log( i );
```

```
    i = i + 1;
}
// 0 1 2 3 4 5 6 7 8 9
```

Warning: This is not necessarily a practical form you'd want to use for your loops. It's presented here for illustration purposes only.

While a `while` (or `do..while`) can accomplish the task manually, there's another syntactic form called a `for` loop for just that purpose:

```
for (var i = 0; i <= 9; i = i + 1) {
    console.log( i );
}
// 0 1 2 3 4 5 6 7 8 9
```

As you can see, in both cases the conditional `i <= 9` is `true` for the first 10 iterations (`i` of values `0` through `9`) of either loop form, but becomes `false` once `i` is value `10`.

The `for` loop has three clauses: the initialization clause (`var i=0`), the conditional test clause (`i <= 9`), and the update clause (`i = i + 1`). So if you're going to do counting with your loop iterations, `for` is a more compact and often easier form to understand and write.

There are other specialized loop forms that are intended to iterate over specific values, such as the properties of an object (see Chapter 2) where the implied conditional test is just whether all the properties have been processed. The "loop until a condition fails" concept holds no matter what the form of the loop.

Functions

The phone store employee probably doesn't carry around a calculator to figure out the taxes and final purchase amount. That's a task she needs to define once and reuse over and over again. Odds are, the company has a checkout register (computer, tablet, etc.) with those "functions" built in.

Similarly, your program will almost certainly want to break up the code's tasks into reusable pieces, instead of repeatedly repeating yourself repetitiously (pun intended!). The way to do this is to define a `function`.

A function is generally a named section of code that can be "called" by name, and the code inside it will be run each time. Consider:

```
function printAmount() {
    console.log( amount.toFixed( 2 ) );
}

var amount = 99.99;
```

```
printAmount(); // "99.99"

amount = amount * 2;

printAmount(); // "199.98"
```

Functions can optionally take arguments (aka parameters) – values you pass in. And they can also optionally return a value back.

```
function printAmount(amt) {
    console.log( amt.toFixed( 2 ) );
}

function formatAmount() {
    return "$" + amount.toFixed( 2 );
}

var amount = 99.99;

printAmount( amount * 2 );      // "199.98"

amount = formatAmount();
console.log( amount );          // "$99.99"
```

The function `printAmount(..)` takes a parameter that we call `amt`. The function `formatAmount()` returns a value. Of course, you can also combine those two techniques in the same function.

Functions are often used for code that you plan to call multiple times, but they can also be useful just to organize related bits of code into named collections, even if you only plan to call them once.

Consider:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // calculate the new amount with the tax
    amt = amt + (amt * TAX_RATE);

    // return the new amount
    return amt;
}

var amount = 99.99;

amount = calculateFinalPurchaseAmount( amount );

console.log( amount.toFixed( 2 ) );     // "107.99"
```

Although `calculateFinalPurchaseAmount(..)` is only called once, organizing its behavior into a separate named function makes the code that uses its logic (the `amount = calculateFinal...` statement) cleaner. If the function had more statements in it, the benefits would be even more pronounced.

Scope

If you ask the phone store employee for a phone model that her store doesn't carry, she will not be able to sell you the phone you want. She only has access to the phones in her store's inventory. You'll have to try another store to see if you can find the phone you're looking for.

Programming has a term for this concept: *scope* (technically called *lexical scope*). In JavaScript, each function gets its own scope. Scope is basically a collection of variables as well as the rules for how those variables are accessed by name. Only code inside that function can access that function's *scoped* variables.

A variable name has to be unique within the same scope – there can't be two different `a` variables sitting right next to each other. But the same variable name `a` could appear in different scopes.

```
function one() {
    // this `a` only belongs to the `one()` function
    var a = 1;
    console.log( a );
}

function two() {
    // this `a` only belongs to the `two()` function
    var a = 2;
    console.log( a );
}

one();      // 1
two();      // 2
```

Also, a scope can be nested inside another scope, just like if a clown at a birthday party blows up one balloon inside another balloon. If one scope is nested inside another, code inside the innermost scope can access variables from either scope.

Consider:

```
function outer() {
    var a = 1;

    function inner() {
        var b = 2;
```

```
        // we can access both `a` and `b` here
        console.log( a + b );   // 3
    }

    inner();

    // we can only access `a` here
    console.log( a );           // 1
}

outer();
```

Lexical scope rules say that code in one scope can access variables of either that scope or any scope outside of it.

So, code inside the `inner()` function has access to both variables `a` and `b`, but code in `outer()` has access only to `a` – it cannot access `b` because that variable is only inside `inner()`.

Recall this code snippet from earlier:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // calculate the new amount with the tax
    amt = amt + (amt * TAX_RATE);

    // return the new amount
    return amt;
}
```

The `TAX_RATE` constant (variable) is accessible from inside the `calculateFinalPurchaseAmount(..)` function, even though we didn't pass it in, because of lexical scope.

Note: For more information about lexical scope, see the first three chapters of the *Scope & Closures* title of this series.

Practice

There is absolutely no substitute for practice in learning programming. No amount of articulate writing on my part is alone going to make you a programmer.

With that in mind, let's try practicing some of the concepts we learned here in this chapter. I'll give the "requirements," and you try it first. Then consult the code listing below to see how I approached it.

- Write a program to calculate the total price of your phone purchase. You will keep purchasing phones (hint: loop!) until you run out of money in

your bank account. You'll also buy accessories for each phone as long as your purchase amount is below your mental spending threshold.
- After you've calculated your purchase amount, add in the tax, then print out the calculated purchase amount, properly formatted.
- Finally, check the amount against your bank account balance to see if you can afford it or not.
- You should set up some constants for the "tax rate," "phone price," "accessory price," and "spending threshold," as well as a variable for your "bank account balance."
- You should define functions for calculating the tax and for formatting the price with a "$" and rounding to two decimal places.
- **Bonus Challenge:** Try to incorporate input into this program, perhaps with the `prompt(..)` covered in "Input" earlier. You may prompt the user for their bank account balance, for example. Have fun and be creative!

OK, go ahead. Try it. Don't peek at my code listing until you've given it a shot yourself!

Note: Because this is a JavaScript book, I'm obviously going to solve the practice exercise in JavaScript. But you can do it in another language for now if you feel more comfortable.

Here's my JavaScript solution for this exercise:

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
    return amount * TAX_RATE;
}

function formatAmount(amount) {
    return "$" + amount.toFixed( 2 );
}

// keep buying phones while you still have money
while (amount < bank_balance) {
    // buy a new phone!
    amount = amount + PHONE_PRICE;

    // can we afford the accessory?
    if (amount < SPENDING_THRESHOLD) {
        amount = amount + ACCESSORY_PRICE;
```

```
    }
}

// don't forget to pay the government, too
amount = amount + calculateTax( amount );

console.log(
    "Your purchase: " + formatAmount( amount )
);
// Your purchase: $334.76

// can you actually afford this purchase?
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// You can't afford this purchase. :(
```

Note: The simplest way to run this JavaScript program is to type it into the developer console of your nearest browser.

How did you do? It wouldn't hurt to try it again now that you've seen my code. And play around with changing some of the constants to see how the program runs with different values.

Review

Learning programming doesn't have to be a complex and overwhelming process. There are just a few basic concepts you need to wrap your head around.

These act like building blocks. To build a tall tower, you start first by putting block on top of block on top of block. The same goes with programming. Here are some of the essential programming building blocks:

- You need *operators* to perform actions on values.
- You need values and *types* to perform different kinds of actions like math on `numbers` or output with `strings`.
- You need *variables* to store data (aka *state*) during your program's execution.
- You need *conditionals* like `if` statements to make decisions.
- You need *loops* to repeat tasks until a condition stops being true.
- You need *functions* to organize your code into logical and reusable chunks.

Code comments are one effective way to write more readable code, which makes your program easier to understand, maintain, and fix later if there are problems.

Finally, don't neglect the power of practice. The best way to learn how to write code is to write code.

I'm excited you're well on your way to learning how to code, now! Keep it up. Don't forget to check out other beginner programming resources (books, blogs, online training, etc.). This chapter and this book are a great start, but they're just a brief introduction.

The next chapter will review many of the concepts from this chapter, but from a more JavaScript-specific perspective, which will highlight most of the major topics that are addressed in deeper detail throughout the rest of the series.

You Don't Know JS: Up & Going

Chapter 2: Into JavaScript

In the previous chapter, I introduced the basic building blocks of programming, such as variables, loops, conditionals, and functions. Of course, all the code shown has been in JavaScript. But in this chapter, we want to focus specifically on things you need to know about JavaScript to get up and going as a JS developer.

We will introduce quite a few concepts in this chapter that will not be fully explored until subsequent *YDKJS* books. You can think of this chapter as an overview of the topics covered in detail throughout the rest of this series.

Especially if you're new to JavaScript, you should expect to spend quite a bit of time reviewing the concepts and code examples here multiple times. Any good foundation is laid brick by brick, so don't expect that you'll immediately understand it all the first pass through.

Your journey to deeply learn JavaScript starts here.

Note: As I said in Chapter 1, you should definitely try all this code yourself as you read and work through this chapter. Be aware that some of the code here assumes capabilities introduced in the newest version of JavaScript at the time of this writing (commonly referred to as "ES6" for the 6th edition of ECMAScript – the official name of the JS specification). If you happen to be using an older, pre-ES6 browser, the code may not work. A recent update of a modern browser (like Chrome, Firefox, or IE) should be used.

Values & Types

As we asserted in Chapter 1, JavaScript has typed values, not typed variables. The following built-in types are available:

- `string`
- `number`
- `boolean`
- `null` and `undefined`
- `object`
- `symbol` (new to ES6)

JavaScript provides a `typeof` operator that can examine a value and tell you what type it is:

```
var a;
typeof a;                   // "undefined"

a = "hello world";
typeof a;                   // "string"

a = 42;
typeof a;                   // "number"

a = true;
typeof a;                   // "boolean"

a = null;
typeof a;                   // "object" -- weird, bug

a = undefined;
typeof a;                   // "undefined"

a = { b: "c" };
typeof a;                   // "object"
```

The return value from the `typeof` operator is always one of six (seven as of ES6! - the "symbol" type) string values. That is, `typeof "abc"` returns `"string"`, not `string`.

Notice how in this snippet the `a` variable holds every different type of value, and that despite appearances, `typeof a` is not asking for the "type of `a`", but rather for the "type of the value currently in `a`." Only values have types in JavaScript; variables are just simple containers for those values.

`typeof null` is an interesting case, because it errantly returns `"object"`, when you'd expect it to return `"null"`.

Warning: This is a long-standing bug in JS, but one that is likely never going to be fixed. Too much code on the Web relies on the bug and thus fixing it would cause a lot more bugs!

Also, note `a = undefined`. We're explicitly setting `a` to the `undefined` value, but that is behaviorally no different from a variable that has no value set yet,

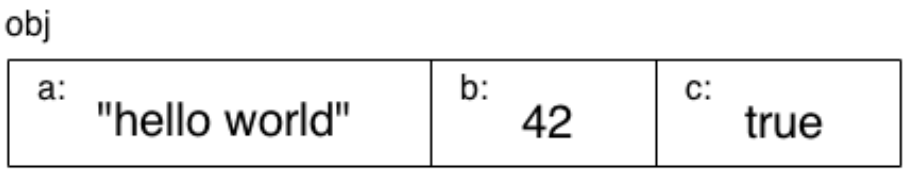
like with the `var a;` line at the top of the snippet. A variable can get to this "undefined" value state in several different ways, including functions that return no values and usage of the `void` operator.

Objects

The `object` type refers to a compound value where you can set properties (named locations) that each hold their own values of any type. This is perhaps one of the most useful value types in all of JavaScript.

```
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;      // "hello world"
obj.b;      // 42
obj.c;      // true

obj["a"];   // "hello world"
obj["b"];   // 42
obj["c"];   // true
```

It may be helpful to think of this `obj` value visually:



Properties can either be accessed with *dot notation* (i.e., `obj.a`) or *bracket notation* (i.e., `obj["a"]`). Dot notation is shorter and generally easier to read, and is thus preferred when possible.

Bracket notation is useful if you have a property name that has special characters in it, like `obj["hello world!"]` – such properties are often referred to as *keys* when accessed via bracket notation. The `[ ]` notation requires either a variable (explained next) or a `string` *literal* (which needs to be wrapped in `" .. "` or `' .. '`).

Of course, bracket notation is also useful if you want to access a property/key but the name is stored in another variable, such as:

```
var obj = {
    a: "hello world",
```

```
    b: 42
};

var b = "a";

obj[b];          // "hello world"
obj["b"];        // 42
```

Note: For more information on JavaScript `objects`, see the *this & Object Prototypes* title of this series, specifically Chapter 3.

There are a couple of other value types that you will commonly interact with in JavaScript programs: *array* and *function*. But rather than being proper built-in types, these should be thought of more like subtypes – specialized versions of the `object` type.

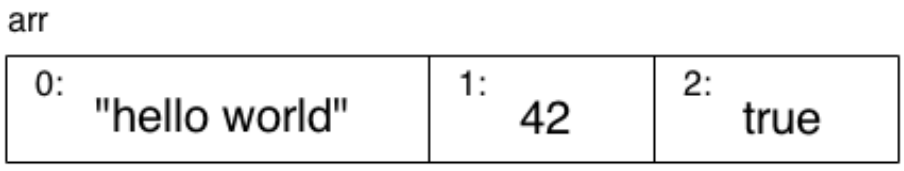
Arrays

An array is an `object` that holds values (of any type) not particularly in named properties/keys, but rather in numerically indexed positions. For example:

```
var arr = [
    "hello world",
    42,
    true
];

arr[0];          // "hello world"
arr[1];          // 42
arr[2];          // true
arr.length;      // 3

typeof arr;      // "object"
```

Note: Languages that start counting at zero, like JS does, use `0` as the index of the first element in the array.

It may be helpful to think of `arr` visually:



Because arrays are special objects (as `typeof` implies), they can also have properties, including the automatically updated `length` property.

You theoretically could use an array as a normal object with your own named properties, or you could use an `object` but only give it numeric properties (`0`, `1`, etc.) similar to an array. However, this would generally be considered improper usage of the respective types.

The best and most natural approach is to use arrays for numerically positioned values and use `objects` for named properties.

Functions

The other `object` subtype you'll use all over your JS programs is a function:

```
function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;          // "function"
typeof foo();        // "number"
typeof foo.bar;      // "string"
```

Again, functions are a subtype of `objects` – `typeof` returns `"function"`, which implies that a `function` is a main type – and can thus have properties, but you typically will only use function object properties (like `foo.bar`) in limited cases.

Note: For more information on JS values and their types, see the first two chapters of the *Types & Grammar* title of this series.

Built-In Type Methods

The built-in types and subtypes we've just discussed have behaviors exposed as properties and methods that are quite powerful and useful.

For example:

```
var a = "hello world";
var b = 3.14159;

a.length;               // 11
a.toUpperCase();        // "HELLO WORLD"
b.toFixed(4);           // "3.1416"
```

The "how" behind being able to call `a.toUpperCase()` is more complicated than just that method existing on the value.

Briefly, there is a `String` (capital `S`) object wrapper form, typically called a "native," that pairs with the primitive `string` type; it's this object wrapper that defines the `toUpperCase()` method on its prototype.

When you use a primitive value like `"hello world"` as an `object` by referencing a property or method (e.g., `a.toUpperCase()` in the previous snippet), JS automatically "boxes" the value to its object wrapper counterpart (hidden under the covers).

A `string` value can be wrapped by a `String` object, a `number` can be wrapped by a `Number` object, and a `boolean` can be wrapped by a `Boolean` object. For the most part, you don't need to worry about or directly use these object wrapper forms of the values – prefer the primitive value forms in practically all cases and JavaScript will take care of the rest for you.

Note: For more information on JS natives and "boxing," see Chapter 3 of the *Types & Grammar* title of this series. To better understand the prototype of an object, see Chapter 5 of the *this & Object Prototypes* title of this series.

Comparing Values

There are two main types of value comparison that you will need to make in your JS programs: *equality* and *inequality*. The result of any comparison is a strictly `boolean` value (`true` or `false`), regardless of what value types are compared.

Coercion

We talked briefly about coercion in Chapter 1, but let's revisit it here.

Coercion comes in two forms in JavaScript: *explicit* and *implicit*. Explicit coercion is simply that you can see obviously from the code that a conversion from one type to another will occur, whereas implicit coercion is when the type conversion can happen as more of a non-obvious side effect of some other operation.

You've probably heard sentiments like "coercion is evil" drawn from the fact that there are clearly places where coercion can produce some surprising results. Perhaps nothing evokes frustration from developers more than when the language surprises them.

Coercion is not evil, nor does it have to be surprising. In fact, the majority of cases you can construct with type coercion are quite sensible and understandable, and can even be used to *improve* the readability of your code. But we won't go much further into that debate – Chapter 4 of the *Types & Grammar* title of this series covers all sides.

Here's an example of *explicit* coercion:

```
var a = "42";

var b = Number( a );
```

```
a;                  // "42"
b;                  // 42 -- the number!
```

And here's an example of *implicit* coercion:

```
var a = "42";

var b = a * 1;  // "42" implicitly coerced to 42 here

a;                  // "42"
b;                  // 42 -- the number!
```

Truthy & Falsy

In Chapter 1, we briefly mentioned the "truthy" and "falsy" nature of values: when a non-`boolean` value is coerced to a `boolean`, does it become `true` or `false`, respectively?

The specific list of "falsy" values in JavaScript is as follows:

- `""` (empty string)
- `0`, `-0`, `NaN` (invalid `number`)
- `null`, `undefined`
- `false`

Any value that's not on this "falsy" list is "truthy." Here are some examples of those:

- `"hello"`
- `42`
- `true`
- `[ ]`, `[ 1, "2", 3 ]` (arrays)
- `{ }`, `{ a: 42 }` (objects)
- `function foo() { .. }` (functions)

It's important to remember that a non-`boolean` value only follows this "truthy"/"falsy" coercion if it's actually coerced to a `boolean`. It's not all that difficult to confuse yourself with a situation that seems like it's coercing a value to a `boolean` when it's not.

Equality

There are four equality operators: `==`, `===`, `!=`, and `!==`. The `!` forms are of course the symmetric "not equal" versions of their counterparts; *non-equality* should not be confused with *inequality*.

The difference between `==` and `===` is usually characterized that `==` checks for value equality and `===` checks for both value and type equality. However, this is inaccurate. The proper way to characterize them is that `==` checks for value

equality with coercion allowed, and `===` checks for value equality without allowing coercion; `===` is often called "strict equality" for this reason.

Consider the implicit coercion that's allowed by the `==` loose-equality comparison and not allowed with the `===` strict-equality:

```
var a = "42";
var b = 42;

a == b;          // true
a === b;         // false
```

In the `a == b` comparison, JS notices that the types do not match, so it goes through an ordered series of steps to coerce one or both values to a different type until the types match, where then a simple value equality can be checked.

If you think about it, there's two possible ways `a == b` could give `true` via coercion. Either the comparison could end up as `42 == 42` or it could be `"42" == "42"`. So which is it?

The answer: `"42"` becomes `42`, to make the comparison `42 == 42`. In such a simple example, it doesn't really seem to matter which way that process goes, as the end result is the same. There are more complex cases where it matters not just what the end result of the comparison is, but *how* you get there.

The `a === b` produces `false`, because the coercion is not allowed, so the simple value comparison obviously fails. Many developers feel that `===` is more predictable, so they advocate always using that form and staying away from `==`. I think this view is very shortsighted. I believe `==` is a powerful tool that helps your program, *if you take the time to learn how it works.*

We're not going to cover all the nitty-gritty details of how the coercion in `==` comparisons works here. Much of it is pretty sensible, but there are some important corner cases to be careful of. You can read section 11.9.3 of the ES5 specification (http://www.ecma-international.org/ecma-262/5.1/) to see the exact rules, and you'll be surprised at just how straightforward this mechanism is, compared to all the negative hype surrounding it.

To boil down a whole lot of details to a few simple takeaways, and help you know whether to use `==` or `===` in various situations, here are my simple rules:

- If either value (aka side) in a comparison could be the `true` or `false` value, avoid `==` and use `===`.
- If either value in a comparison could be of these specific values (`0`, `""`, or `[]` – empty array), avoid `==` and use `===`.
- In *all* other cases, you're safe to use `==`. Not only is it safe, but in many cases it simplifies your code in a way that improves readability.

What these rules boil down to is requiring you to think critically about your code and about what kinds of values can come through variables that get compared

for equality. If you can be certain about the values, and `==` is safe, use it! If you can't be certain about the values, use `===`. It's that simple.

The `!=` non-equality form pairs with `==`, and the `!==` form pairs with `===`. All the rules and observations we just discussed hold symmetrically for these non-equality comparisons.

You should take special note of the `==` and `===` comparison rules if you're comparing two non-primitive values, like `objects` (including `function` and `array`). Because those values are actually held by reference, both `==` and `===` comparisons will simply check whether the references match, not anything about the underlying values.

For example, `array`s are by default coerced to `string`s by simply joining all the values with commas (`,`) in between. You might think that two `array`s with the same contents would be `==` equal, but they're not:

```
var a = [1,2,3];
var b = [1,2,3];
var c = "1,2,3";

a == c;     // true
b == c;     // true
a == b;     // false
```

Note: For more information about the `==` equality comparison rules, see the ES5 specification (section 11.9.3) and also consult Chapter 4 of the *Types & Grammar* title of this series; see Chapter 2 for more information about values versus references.

Inequality

The `<`, `>`, `<=`, and `>=` operators are used for inequality, referred to in the specification as "relational comparison." Typically they will be used with ordinally comparable values like `numbers`. It's easy to understand that `3 < 4`.

But JavaScript `string` values can also be compared for inequality, using typical alphabetic rules (`"bar" < "foo"`).

What about coercion? Similar rules as `==` comparison (though not exactly identical!) apply to the inequality operators. Notably, there are no "strict inequality" operators that would disallow coercion the same way `===` "strict equality" does.

Consider:

```
var a = 41;
var b = "42";
var c = "43";
```

```
a < b;      // true
b < c;      // true
```

What happens here? In section 11.8.5 of the ES5 specification, it says that if both values in the `<` comparison are `strings`, as it is with `b < c`, the comparison is made lexicographically (aka alphabetically like a dictionary). But if one or both is not a `string`, as it is with `a < b`, then both values are coerced to be `numbers`, and a typical numeric comparison occurs.

The biggest gotcha you may run into here with comparisons between potentially different value types – remember, there are no "strict inequality" forms to use – is when one of the values cannot be made into a valid number, such as:

```
var a = 42;
var b = "foo";

a < b;      // false
a > b;      // false
a == b;     // false
```

Wait, how can all three of those comparisons be `false`? Because the `b` value is being coerced to the "invalid number value" `NaN` in the `<` and `>` comparisons, and the specification says that `NaN` is neither greater-than nor less-than any other value.

The `==` comparison fails for a different reason. `a == b` could fail if it's interpreted either as `42 == NaN` or `"42" == "foo"` – as we explained earlier, the former is the case.

Note: For more information about the inequality comparison rules, see section 11.8.5 of the ES5 specification and also consult Chapter 4 of the *Types & Grammar* title of this series.

Variables

In JavaScript, variable names (including function names) must be valid *identifiers*. The strict and complete rules for valid characters in identifiers are a little complex when you consider nontraditional characters such as Unicode. If you only consider typical ASCII alphanumeric characters, though, the rules are simple.

An identifier must start with `a-z`, `A-Z`, `$`, or `_`. It can then contain any of those characters plus the numerals `0-9`.

Generally, the same rules apply to a property name as to a variable identifier. However, certain words cannot be used as variables, but are OK as property names. These words are called "reserved words," and include the JS keywords (`for`, `in`, `if`, etc.) as well as `null`, `true`, and `false`.

Note: For more information about reserved words, see Appendix A of the *Types & Grammar* title of this series.

Function Scopes

You use the `var` keyword to declare a variable that will belong to the current function scope, or the global scope if at the top level outside of any function.

Hoisting

Wherever a `var` appears inside a scope, that declaration is taken to belong to the entire scope and accessible everywhere throughout.

Metaphorically, this behavior is called *hoisting*, when a `var` declaration is conceptually "moved" to the top of its enclosing scope. Technically, this process is more accurately explained by how code is compiled, but we can skip over those details for now.

Consider:

```
var a = 2;

foo();                  // works because `foo()`
                        // declaration is "hoisted"

function foo() {
    a = 3;

    console.log( a );   // 3

    var a;              // declaration is "hoisted"
                        // to the top of `foo()`
}

console.log( a );   // 2
```

Warning: It's not common or a good idea to rely on variable *hoisting* to use a variable earlier in its scope than its `var` declaration appears; it can be quite confusing. It's much more common and accepted to use *hoisted* function declarations, as we do with the `foo()` call appearing before its formal declaration.

Nested Scopes

When you declare a variable, it is available anywhere in that scope, as well as any lower/inner scopes. For example:

```
function foo() {
    var a = 1;

    function bar() {
        var b = 2;

        function baz() {
            var c = 3;

            console.log( a, b, c ); // 1 2 3
        }

        baz();
        console.log( a, b );        // 1 2
    }

    bar();
    console.log( a );               // 1
}

foo();
```

Notice that `c` is not available inside of `bar()`, because it's declared only inside the inner `baz()` scope, and that `b` is not available to `foo()` for the same reason.

If you try to access a variable's value in a scope where it's not available, you'll get a `ReferenceError` thrown. If you try to set a variable that hasn't been declared, you'll either end up creating a variable in the top-level global scope (bad!) or getting an error, depending on "strict mode" (see "Strict Mode"). Let's take a look:

```
function foo() {
    a = 1;  // `a` not formally declared
}

foo();
a;          // 1 -- oops, auto global variable :(
```

This is a very bad practice. Don't do it! Always formally declare your variables.

In addition to creating declarations for variables at the function level, ES6 *lets* you declare variables to belong to individual blocks (pairs of `{ .. }`), using the `let` keyword. Besides some nuanced details, the scoping rules will behave roughly the same as we just saw with functions:

```
function foo() {
    var a = 1;

    if (a >= 1) {
```

```
        let b = 2;

        while (b < 5) {
            let c = b * 2;
            b++;

            console.log( a + c );
        }
    }
}

foo();
// 5 7 9
```

Because of using `let` instead of `var`, `b` will belong only to the `if` statement and thus not to the whole `foo()` function's scope. Similarly, `c` belongs only to the `while` loop. Block scoping is very useful for managing your variable scopes in a more fine-grained fashion, which can make your code much easier to maintain over time.

Note: For more information about scope, see the *Scope & Closures* title of this series. See the *ES6 & Beyond* title of this series for more information about `let` block scoping.

Conditionals

In addition to the `if` statement we introduced briefly in Chapter 1, JavaScript provides a few other conditionals mechanisms that we should take a look at.

Sometimes you may find yourself writing a series of `if..else..if` statements like this:

```
if (a == 2) {
    // do something
}
else if (a == 10) {
    // do another thing
}
else if (a == 42) {
    // do yet another thing
}
else {
    // fallback to here
}
```

This structure works, but it's a little verbose because you need to specify the `a` test for each case. Here's another option, the `switch` statement:

```
switch (a) {
    case 2:
        // do something
        break;
    case 10:
        // do another thing
        break;
    case 42:
        // do yet another thing
        break;
    default:
        // fallback to here
}
```

The `break` is important if you want only the statement(s) in one `case` to run. If you omit `break` from a `case`, and that `case` matches or runs, execution will continue with the next `case`'s statements regardless of that `case` matching. This so called "fall through" is sometimes useful/desired:

```
switch (a) {
    case 2:
    case 10:
        // some cool stuff
        break;
    case 42:
        // other stuff
        break;
    default:
        // fallback
}
```

Here, if `a` is either `2` or `10`, it will execute the "some cool stuff" code statements.

Another form of conditional in JavaScript is the "conditional operator," often called the "ternary operator." It's like a more concise form of a single `if..else` statement, such as:

```
var a = 42;

var b = (a > 41) ? "hello" : "world";

// similar to:

// if (a > 41) {
//     b = "hello";
// }
// else {
//     b = "world";
```

```
// }
```

If the test expression (`a > 41` here) evaluates as `true`, the first clause (`"hello"`) results, otherwise the second clause (`"world"`) results, and whatever the result is then gets assigned to `b`.

The conditional operator doesn't have to be used in an assignment, but that's definitely the most common usage.

Note: For more information about testing conditions and other patterns for `switch` and `? :`, see the *Types & Grammar* title of this series.

Strict Mode

ES5 added a "strict mode" to the language, which tightens the rules for certain behaviors. Generally, these restrictions are seen as keeping the code to a safer and more appropriate set of guidelines. Also, adhering to strict mode makes your code generally more optimizable by the engine. Strict mode is a big win for code, and you should use it for all your programs.

You can opt in to strict mode for an individual function, or an entire file, depending on where you put the strict mode pragma:

```
function foo() {
    "use strict";

    // this code is strict mode

    function bar() {
        // this code is strict mode
    }
}

// this code is not strict mode
```

Compare that to:

```
"use strict";

function foo() {
    // this code is strict mode

    function bar() {
        // this code is strict mode
    }
}

// this code is strict mode
```

One key difference (improvement!) with strict mode is disallowing the implicit auto-global variable declaration from omitting the `var`:

```
function foo() {
    "use strict";   // turn on strict mode
    a = 1;          // `var` missing, ReferenceError
}

foo();
```

If you turn on strict mode in your code, and you get errors, or code starts behaving buggy, your temptation might be to avoid strict mode. But that instinct would be a bad idea to indulge. If strict mode causes issues in your program, almost certainly it's a sign that you have things in your program you should fix.

Not only will strict mode keep your code to a safer path, and not only will it make your code more optimizable, but it also represents the future direction of the language. It'd be easier on you to get used to strict mode now than to keep putting it off – it'll only get harder to convert later!

Note: For more information about strict mode, see the Chapter 5 of the *Types & Grammar* title of this series.

Functions As Values

So far, we've discussed functions as the primary mechanism of *scope* in JavaScript. You recall typical `function` declaration syntax as follows:

```
function foo() {
    // ..
}
```

Though it may not seem obvious from that syntax, `foo` is basically just a variable in the outer enclosing scope that's given a reference to the `function` being declared. That is, the `function` itself is a value, just like `42` or `[1,2,3]` would be.

This may sound like a strange concept at first, so take a moment to ponder it. Not only can you pass a value (argument) *to* a function, but *a function itself can be a value* that's assigned to variables, or passed to or returned from other functions.

As such, a function value should be thought of as an expression, much like any other value or expression.

Consider:

```
var foo = function() {
    // ..
```

```
};

var x = function bar(){
    // ..
};
```

The first function expression assigned to the `foo` variable is called *anonymous* because it has no `name`.

The second function expression is *named* (`bar`), even as a reference to it is also assigned to the `x` variable. *Named function expressions* are generally more preferable, though *anonymous function expressions* are still extremely common.

For more information, see the *Scope & Closures* title of this series.

Immediately Invoked Function Expressions (IIFEs)

In the previous snippet, neither of the function expressions are executed – we could if we had included `foo()` or `x()`, for instance.

There's another way to execute a function expression, which is typically referred to as an *immediately invoked function expression* (IIFE):

```
(function IIFE(){
    console.log( "Hello!" );
})();
// "Hello!"
```

The outer `( .. )` that surrounds the `(function IIFE(){ .. })` function expression is just a nuance of JS grammar needed to prevent it from being treated as a normal function declaration.

The final `()` on the end of the expression – the `})();` line – is what actually executes the function expression referenced immediately before it.

That may seem strange, but it's not as foreign as first glance. Consider the similarities between `foo` and `IIFE` here:

```
function foo() { .. }

// `foo` function reference expression,
// then `()` executes it
foo();

// `IIFE` function expression,
// then `()` executes it
(function IIFE(){ .. })();
```

As you can see, listing the `(function IIFE(){ .. })` before its executing `()` is essentially the same as including `foo` before its executing `()`; in both cases, the function reference is executed with `()` immediately after it.

Because an IIFE is just a function, and functions create variable *scope*, using an IIFE in this fashion is often used to declare variables that won't affect the surrounding code outside the IIFE:

```
var a = 42;

(function IIFE(){
    var a = 10;
    console.log( a );    // 10
})();

console.log( a );        // 42
```

IIFEs can also have return values:

```
var x = (function IIFE(){
    return 42;
})();

x;  // 42
```

The `42` value gets `return`ed from the `IIFE`-named function being executed, and is then assigned to `x`.

Closure

Closure is one of the most important, and often least understood, concepts in JavaScript. I won't cover it in deep detail here, and instead refer you to the *Scope & Closures* title of this series. But I want to say a few things about it so you understand the general concept. It will be one of the most important techniques in your JS skillset.

You can think of closure as a way to "remember" and continue to access a function's scope (its variables) even once the function has finished running.

Consider:

```
function makeAdder(x) {
    // parameter `x` is an inner variable

    // inner function `add()` uses `x`, so
    // it has a "closure" over it
    function add(y) {
        return y + x;
    };
```

```
    return add;
}
```

The reference to the inner `add(..)` function that gets returned with each call to the outer `makeAdder(..)` is able to remember whatever `x` value was passed in to `makeAdder(..)`. Now, let's use `makeAdder(..)`:

```
// `plusOne` gets a reference to the inner `add(..)`
// function with closure over the `x` parameter of
// the outer `makeAdder(..)`
var plusOne = makeAdder( 1 );

// `plusTen` gets a reference to the inner `add(..)`
// function with closure over the `x` parameter of
// the outer `makeAdder(..)`
var plusTen = makeAdder( 10 );

plusOne( 3 );         // 4  <-- 1 + 3
plusOne( 41 );        // 42 <-- 1 + 41

plusTen( 13 );        // 23 <-- 10 + 13
```

More on how this code works:

1. When we call `makeAdder(1)`, we get back a reference to its inner `add(..)` that remembers `x` as `1`. We call this function reference `plusOne(..)`.
2. When we call `makeAdder(10)`, we get back another reference to its inner `add(..)` that remembers `x` as `10`. We call this function reference `plusTen(..)`.
3. When we call `plusOne(3)`, it adds `3` (its inner `y`) to the `1` (remembered by `x`), and we get `4` as the result.
4. When we call `plusTen(13)`, it adds `13` (its inner `y`) to the `10` (remembered by `x`), and we get `23` as the result.

Don't worry if this seems strange and confusing at first – it can be! It'll take lots of practice to understand it fully.

But trust me, once you do, it's one of the most powerful and useful techniques in all of programming. It's definitely worth the effort to let your brain simmer on closures for a bit. In the next section, we'll get a little more practice with closure.

Modules

The most common usage of closure in JavaScript is the module pattern. Modules let you define private implementation details (variables, functions) that are

hidden from the outside world, as well as a public API that *is* accessible from the outside.

Consider:

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // do the rest of the login work
    }

    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
}

// create a `User` module instance
var fred = User();

fred.login( "fred", "12Battery34!" );
```

The `User()` function serves as an outer scope that holds the variables `username` and `password`, as well as the inner `doLogin()` function; these are all private inner details of this `User` module that cannot be accessed from the outside world.

Warning: We are not calling `new User()` here, on purpose, despite the fact that probably seems more common to most readers. `User()` is just a function, not a class to be instantiated, so it's just called normally. Using `new` would be inappropriate and actually waste resources.

Executing `User()` creates an *instance* of the `User` module – a whole new scope is created, and thus a whole new copy of each of these inner variables/functions. We assign this instance to `fred`. If we run `User()` again, we'd get a new instance entirely separate from `fred`.

The inner `doLogin()` function has a closure over `username` and `password`, meaning it will retain its access to them even after the `User()` function finishes running.

`publicAPI` is an object with one property/method on it, `login`, which is a reference to the inner `doLogin()` function. When we return `publicAPI` from `User()`, it becomes the instance we call `fred`.

At this point, the outer `User()` function has finished executing. Normally, you'd

think the inner variables like `username` and `password` have gone away. But here they have not, because there's a closure in the `login()` function keeping them alive.

That's why we can call `fred.login(..)` – the same as calling the inner `doLogin(..)` – and it can still access `username` and `password` inner variables.

There's a good chance that with just this brief glimpse at closure and the module pattern, some of it is still a bit confusing. That's OK! It takes some work to wrap your brain around it.

From here, go read the *Scope & Closures* title of this series for a much more in-depth exploration.

`this` Identifier

Another very commonly misunderstood concept in JavaScript is the `this` identifier. Again, there's a couple of chapters on it in the *this & Object Prototypes* title of this series, so here we'll just briefly introduce the concept.

While it may often seem that `this` is related to "object-oriented patterns," in JS `this` is a different mechanism.

If a function has a `this` reference inside it, that `this` reference usually points to an `object`. But which `object` it points to depends on how the function was called.

It's important to realize that `this` *does not* refer to the function itself, as is the most common misconception.

Here's a quick illustration:

```
function foo() {
    console.log( this.bar );
}

var bar = "global";

var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

// --------
```

```
foo();                  // "global"
obj1.foo();             // "obj1"
foo.call( obj2 );       // "obj2"
new foo();              // undefined
```

There are four rules for how `this` gets set, and they're shown in those last four lines of that snippet.

1. `foo()` ends up setting `this` to the global object in non-strict mode – in strict mode, `this` would be `undefined` and you'd get an error in accessing the `bar` property – so `"global"` is the value found for `this.bar`.
2. `obj1.foo()` sets `this` to the `obj1` object.
3. `foo.call(obj2)` sets `this` to the `obj2` object.
4. `new foo()` sets `this` to a brand new empty object.

Bottom line: to understand what `this` points to, you have to examine how the function in question was called. It will be one of those four ways just shown, and that will then answer what `this` is.

Note: For more information about `this`, see Chapters 1 and 2 of the *this & Object Prototypes* title of this series.

Prototypes

The prototype mechanism in JavaScript is quite complicated. We will only glance at it here. You will want to spend plenty of time reviewing Chapters 4-6 of the *this & Object Prototypes* title of this series for all the details.

When you reference a property on an object, if that property doesn't exist, JavaScript will automatically use that object's internal prototype reference to find another object to look for the property on. You could think of this almost as a fallback if the property is missing.

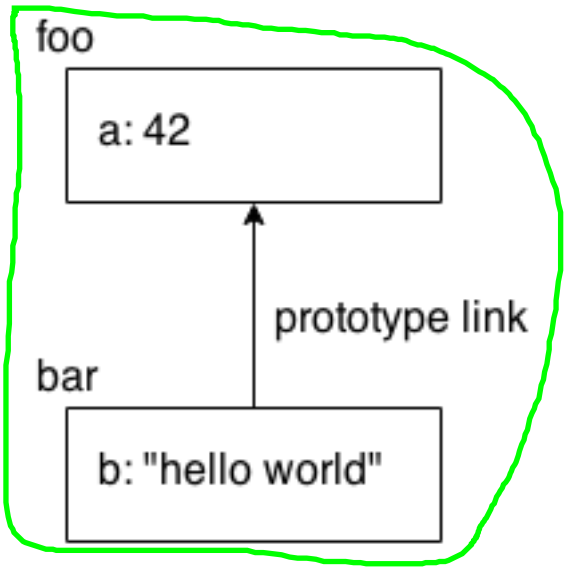
The internal prototype reference linkage from one object to its fallback happens at the time the object is created. The simplest way to illustrate it is with a built-in utility called `Object.create(..)`.

Consider:

```
var foo = {
    a: 42
};

// create `bar` and link it to `foo`
var bar = Object.create( foo );

bar.b = "hello world";
```

```
bar.b;      // "hello world"
bar.a;      // 42 <-- delegated to `foo`
```

It may help to visualize the `foo` and `bar` objects and their relationship:



The `a` property doesn't actually exist on the `bar` object, but because `bar` is prototype-linked to `foo`, JavaScript automatically falls back to looking for `a` on the `foo` object, where it's found.

This linkage may seem like a strange feature of the language. The most common way this feature is used – and I would argue, abused – is to try to emulate/fake a "class" mechanism with "inheritance."

But a more natural way of applying prototypes is a pattern called "behavior delegation," where you intentionally design your linked objects to be able to *delegate* from one to the other for parts of the needed behavior.

Note: For more information about prototypes and behavior delegation, see Chapters 4-6 of the *this & Object Prototypes* title of this series.

Old & New

Some of the JS features we've already covered, and certainly many of the features covered in the rest of this series, are newer additions and will not necessarily be available in older browsers. In fact, some of the newest features in the specification aren't even implemented in any stable browsers yet.

So, what do you do with the new stuff? Do you just have to wait around for years or decades for all the old browsers to fade into obscurity?

That's how many people think about the situation, but it's really not a healthy approach to JS.

There are two main techniques you can use to "bring" the newer JavaScript stuff to the older browsers: polyfilling and transpiling.

Polyfilling

The word "polyfill" is an invented term (by Remy Sharp) (https://remysharp.com/2010/10/08/what-is-a-polyfill) used to refer to taking the definition of a newer feature and producing a piece of code that's equivalent to the behavior, but is able to run in older JS environments.

For example, ES6 defines a utility called `Number.isNaN(..)` to provide an accurate non-buggy check for `NaN` values, deprecating the original `isNaN(..)` utility. But it's easy to polyfill that utility so that you can start using it in your code regardless of whether the end user is in an ES6 browser or not.

Consider:

```
if (!Number.isNaN) {
    Number.isNaN = function isNaN(x) {
        return x !== x;
    };
}
```

The `if` statement guards against applying the polyfill definition in ES6 browsers where it will already exist. If it's not already present, we define `Number.isNaN(..)`.

Note: The check we do here takes advantage of a quirk with `NaN` values, which is that they're the only value in the whole language that is not equal to itself. So the `NaN` value is the only one that would make `x !== x` be `true`.

Not all new features are fully polyfillable. Sometimes most of the behavior can be polyfilled, but there are still small deviations. You should be really, really careful in implementing a polyfill yourself, to make sure you are adhering to the specification as strictly as possible.

Or better yet, use an already vetted set of polyfills that you can trust, such as those provided by ES5-Shim (https://github.com/es-shims/es5-shim) and ES6-Shim (https://github.com/es-shims/es6-shim).

Transpiling

There's no way to polyfill new syntax that has been added to the language. The new syntax would throw an error in the old JS engine as unrecognized/invalid.

So the better option is to use a tool that converts your newer code into older code equivalents. This process is commonly called "transpiling," a term for transforming + compiling.

Essentially, your source code is authored in the new syntax form, but what you deploy to the browser is the transpiled code in old syntax form. You typically insert the transpiler into your build process, similar to your code linter or your minifier.

You might wonder why you'd go to the trouble to write new syntax only to have it transpiled away to older code – why not just write the older code directly?

There are several important reasons you should care about transpiling:

- The new syntax added to the language is designed to make your code more readable and maintainable. The older equivalents are often much more convoluted. You should prefer writing newer and cleaner syntax, not only for yourself but for all other members of the development team.
- If you transpile only for older browsers, but serve the new syntax to the newest browsers, you get to take advantage of browser performance optimizations with the new syntax. This also lets browser makers have more real-world code to test their implementations and optimizations on.
- Using the new syntax earlier allows it to be tested more robustly in the real world, which provides earlier feedback to the JavaScript committee (TC39). If issues are found early enough, they can be changed/fixed before those language design mistakes become permanent.

Here's a quick example of transpiling. ES6 adds a feature called "default parameter values." It looks like this:

```
function foo(a = 2) {
    console.log( a );
}

foo();      // 2
foo( 42 );  // 42
```

Simple, right? Helpful, too! But it's new syntax that's invalid in pre-ES6 engines. So what will a transpiler do with that code to make it run in older environments?

```
function foo() {
    var a = arguments[0] !== (void 0) ? arguments[0] : 2;
    console.log( a );
}
```

As you can see, it checks to see if the `arguments[0]` value is `void 0` (aka `undefined`), and if so provides the `2` default value; otherwise, it assigns whatever was passed.

In addition to being able to now use the nicer syntax even in older browsers, looking at the transpiled code actually explains the intended behavior more

clearly.

You may not have realized just from looking at the ES6 version that `undefined` is the only value that can't get explicitly passed in for a default-value parameter, but the transpiled code makes that much more clear.

The last important detail to emphasize about transpilers is that they should now be thought of as a standard part of the JS development ecosystem and process. JS is going to continue to evolve, much more quickly than before, so every few months new syntax and new features will be added.

If you use a transpiler by default, you'll always be able to make that switch to newer syntax whenever you find it useful, rather than always waiting for years for today's browsers to phase out.

There are quite a few great transpilers for you to choose from. Here are some good options at the time of this writing:

- Babel (https://babeljs.io) (formerly 6to5): Transpiles ES6+ into ES5
- Traceur (https://github.com/google/traceur-compiler): Transpiles ES6, ES7, and beyond into ES5

Non-JavaScript

So far, the only things we've covered are in the JS language itself. The reality is that most JS is written to run in and interact with environments like browsers. A good chunk of the stuff that you write in your code is, strictly speaking, not directly controlled by JavaScript. That probably sounds a little strange.

The most common non-JavaScript JavaScript you'll encounter is the DOM API. For example:

```
var el = document.getElementById( "foo" );
```

The `document` variable exists as a global variable when your code is running in a browser. It's not provided by the JS engine, nor is it particularly controlled by the JavaScript specification. It takes the form of something that looks an awful lot like a normal JS `object`, but it's not really exactly that. It's a special `object`, often called a "host object."

Moreover, the `getElementById(..)` method on `document` looks like a normal JS function, but it's just a thinly exposed interface to a built-in method provided by the DOM from your browser. In some (newer-generation) browsers, this layer may also be in JS, but traditionally the DOM and its behavior is implemented in something more like C/C++.

Another example is with input/output (I/O).

Everyone's favorite `alert(..)` pops up a message box in the user's browser window. `alert(..)` is provided to your JS program by the browser, not by the

JS engine itself. The call you make sends the message to the browser internals and it handles drawing and displaying the message box.

The same goes with `console.log(..)`; your browser provides such mechanisms and hooks them up to the developer tools.

This book, and this whole series, focuses on JavaScript the language. That's why you don't see any substantial coverage of these non-JavaScript JavaScript mechanisms. Nevertheless, you need to be aware of them, as they'll be in every JS program you write!

Review

The first step to learning JavaScript's flavor of programming is to get a basic understanding of its core mechanisms like values, types, function closures, `this`, and prototypes.

Of course, each of these topics deserves much greater coverage than you've seen here, but that's why they have chapters and books dedicated to them throughout the rest of this series. After you feel pretty comfortable with the concepts and code samples in this chapter, the rest of the series awaits you to really dig in and get to know the language deeply.

The final chapter of this book will briefly summarize each of the other titles in the series and the other concepts they cover besides what we've already explored.

You Don't Know JS: Up & Going

Chapter 3: Into YDKJS

What is this series all about? Put simply, it's about taking seriously the task of learning *all parts of JavaScript*, not just some subset of the language that someone called "the good parts," and not just whatever minimal amount you need to get your job done at work.

Serious developers in other languages expect to put in the effort to learn most or all of the language(s) they primarily write in, but JS developers seem to stand out from the crowd in the sense of typically not learning very much of the language. This is not a good thing, and it's not something we should continue to allow to be the norm.

The *You Don't Know JS* (*YDKJS*) series stands in stark contrast to the typical approaches to learning JS, and is unlike almost any other JS books you will read. It challenges you to go beyond your comfort zone and to ask the deeper

"why" questions for every single behavior you encounter. Are you up for that challenge?

I'm going to use this final chapter to briefly summarize what to expect from the rest of the books in the series, and how to most effectively go about building a foundation of JS learning on top of *YDKJS*.

Scope & Closures

Perhaps one of the most fundamental things you'll need to quickly come to terms with is how scoping of variables really works in JavaScript. It's not enough to have anecdotal fuzzy *beliefs* about scope.

The *Scope & Closures* title starts by debunking the common misconception that JS is an "interpreted language" and therefore not compiled. Nope.

The JS engine compiles your code right before (and sometimes during!) execution. So we use some deeper understanding of the compiler's approach to our code to understand how it finds and deals with variable and function declarations. Along the way, we see the typical metaphor for JS variable scope management, "Hoisting."

This critical understanding of "lexical scope" is what we then base our exploration of closure on for the last chapter of the book. Closure is perhaps the single most important concept in all of JS, but if you haven't first grasped firmly how scope works, closure will likely remain beyond your grasp.

One important application of closure is the module pattern, as we briefly introduced in this book in Chapter 2. The module pattern is perhaps the most prevalent code organization pattern in all of JavaScript; deep understanding of it should be one of your highest priorities.

this & Object Prototypes

Perhaps one of the most widespread and persistent mistruths about JavaScript is that the `this` keyword refers to the function it appears in. Terribly mistaken.

The `this` keyword is dynamically bound based on how the function in question is executed, and it turns out there are four simple rules to understand and fully determine `this` binding.

Closely related to the `this` keyword is the object prototype mechanism, which is a look-up chain for properties, similar to how lexical scope variables are found. But wrapped up in the prototypes is the other huge miscue about JS: the idea of emulating (fake) classes and (so-called "prototypal") inheritance.

Unfortunately, the desire to bring class and inheritance design pattern thinking to JavaScript is just about the worst thing you could try to do, because while

the syntax may trick you into thinking there's something like classes present, in fact the prototype mechanism is fundamentally opposite in its behavior.

What's at issue is whether it's better to ignore the mismatch and pretend that what you're implementing is "inheritance," or whether it's more appropriate to learn and embrace how the object prototype system actually works. The latter is more appropriately named "behavior delegation."

This is more than syntactic preference. Delegation is an entirely different, and more powerful, design pattern, one that replaces the need to design with classes and inheritance. But these assertions will absolutely fly in the face of nearly every other blog post, book, and conference talk on the subject for the entirety of JavaScript's lifetime.

The claims I make regarding delegation versus inheritance come not from a dislike of the language and its syntax, but from the desire to see the true capability of the language properly leveraged and the endless confusion and frustration wiped away.

But the case I make regarding prototypes and delegation is a much more involved one than what I will indulge here. If you're ready to reconsider everything you think you know about JavaScript "classes" and "inheritance," I offer you the chance to "take the red pill" (*Matrix* 1999) and check out Chapters 4-6 of the *this & Object Prototypes* title of this series.

Types & Grammar

The third title in this series primarily focuses on tackling yet another highly controversial topic: type coercion. Perhaps no topic causes more frustration with JS developers than when you talk about the confusions surrounding implicit coercion.

By far, the conventional wisdom is that implicit coercion is a "bad part" of the language and should be avoided at all costs. In fact, some have gone so far as to call it a "flaw" in the design of the language. Indeed, there are tools whose entire job is to do nothing but scan your code and complain if you're doing anything even remotely like coercion.

But is coercion really so confusing, so bad, so treacherous, that your code is doomed from the start if you use it?

I say no. After having built up an understanding of how types and values really work in Chapters 1-3, Chapter 4 takes on this debate and fully explains how coercion works, in all its nooks and crevices. We see just what parts of coercion really are surprising and what parts actually make complete sense if given the time to learn.

But I'm not merely suggesting that coercion is sensible and learnable, I'm asserting that coercion is an incredibly useful and totally underestimated tool

that *you should be using in your code.* I'm saying that coercion, when used properly, not only works, but makes your code better. All the naysayers and doubters will surely scoff at such a position, but I believe it's one of the main keys to upping your JS game.

Do you want to just keep following what the crowd says, or are you willing to set all the assumptions aside and look at coercion with a fresh perspective? The *Types & Grammar* title of this series will coerce your thinking.

Async & Performance

The first three titles of this series focus on the core mechanics of the language, but the fourth title branches out slightly to cover patterns on top of the language mechanics for managing asynchronous programming. Asynchrony is not only critical to the performance of our applications, it's increasingly becoming *the* critical factor in writability and maintainability.

The book starts first by clearing up a lot of terminology and concept confusion around things like "async," "parallel," and "concurrent," and explains in depth how such things do and do not apply to JS.

Then we move into examining callbacks as the primary method of enabling asynchrony. But it's here that we quickly see that the callback alone is hopelessly insufficient for the modern demands of asynchronous programming. We identify two major deficiencies of callbacks-only coding: *Inversion of Control* (IoC) trust loss and lack of linear reason-ability.

To address these two major deficiencies, ES6 introduces two new mechanisms (and indeed, patterns): promises and generators.

Promises are a time-independent wrapper around a "future value," which lets you reason about and compose them regardless of if the value is ready or not yet. Moreover, they effectively solve the IoC trust issues by routing callbacks through a trustable and composable promise mechanism.

Generators introduce a new mode of execution for JS functions, whereby the generator can be paused at `yield` points and be resumed asynchronously later. The pause-and-resume capability enables synchronous, sequential looking code in the generator to be processed asynchronously behind the scenes. By doing so, we address the non-linear, non-local-jump confusions of callbacks and thereby make our asynchronous code sync-looking so as to be more reason-able.

But it's the combination of promises and generators that "yields" our most effective asynchronous coding pattern to date in JavaScript. In fact, much of the future sophistication of asynchrony coming in ES7 and later will certainly be built on this foundation. To be serious about programming effectively in an async world, you're going to need to get really comfortable with combining promises and generators.

If promises and generators are about expressing patterns that let our programs run more concurrently and thus get more processing accomplished in a shorter period, JS has many other facets of performance optimization worth exploring.

Chapter 5 delves into topics like program parallelism with Web Workers and data parallelism with SIMD, as well as low-level optimization techniques like ASM.js. Chapter 6 takes a look at performance optimization from the perspective of proper benchmarking techniques, including what kinds of performance to worry about and what to ignore.

Writing JavaScript effectively means writing code that can break the constraint barriers of being run dynamically in a wide range of browsers and other environments. It requires a lot of intricate and detailed planning and effort on our parts to take a program from "it works" to "it works well."

The *Async & Performance* title is designed to give you all the tools and skills you need to write reasonable and performant JavaScript code.

ES6 & Beyond

No matter how much you feel you've mastered JavaScript to this point, the truth is that JavaScript is never going to stop evolving, and moreover, the rate of evolution is increasing rapidly. This fact is almost a metaphor for the spirit of this series, to embrace that we'll never fully *know* every part of JS, because as soon as you master it all, there's going to be new stuff coming down the line that you'll need to learn.

This title is dedicated to both the short- and mid-term visions of where the language is headed, not just the *known* stuff like ES6 but the *likely* stuff beyond.

While all the titles of this series embrace the state of JavaScript at the time of this writing, which is mid-way through ES6 adoption, the primary focus in the series has been more on ES5. Now, we want to turn our attention to ES6, ES7, and ...

Since ES6 is nearly complete at the time of this writing, *ES6 & Beyond* starts by dividing up the concrete stuff from the ES6 landscape into several key categories, including new syntax, new data structures (collections), and new processing capabilities and APIs. We cover each of these new ES6 features, in varying levels of detail, including reviewing details that are touched on in other books of this series.

Some exciting ES6 things to look forward to reading about: destructuring, default parameter values, symbols, concise methods, computed properties, arrow functions, block scoping, promises, generators, iterators, modules, proxies, weakmaps, and much, much more! Phew, ES6 packs quite a punch!

The first part of the book is a roadmap for all the stuff you need to learn to get ready for the new and improved JavaScript you'll be writing and exploring over

the next couple of years.

The latter part of the book turns attention to briefly glance at things that we can likely expect to see in the near future of JavaScript. The most important realization here is that post-ES6, JS is likely going to evolve feature by feature rather than version by version, which means we can expect to see these near-future things coming much sooner than you might imagine.

The future for JavaScript is bright. Isn't it time we start learning it!?

Review

The *YDKJS* series is dedicated to the proposition that all JS developers can and should learn all of the parts of this great language. No person's opinion, no framework's assumptions, and no project's deadline should be the excuse for why you never learn and deeply understand JavaScript.

We take each important area of focus in the language and dedicate a short but very dense book to fully explore all the parts of it that you perhaps thought you knew but probably didn't fully.

"You Don't Know JS" isn't a criticism or an insult. It's a realization that all of us, myself included, must come to terms with. Learning JavaScript isn't an end goal but a process. We don't know JavaScript, yet. But we will!