You Don't Know JS Yet: Get Started - 2nd Edition

Chapter 2: Surveying JS

The best way to learn JS is to start writing JS.

To do that, you need to know how the language works, and that's what we'll focus on here. Even if you've programmed in other languages before, take your time getting comfortable with JS, and make sure to practice each piece.

This chapter is not an exhaustive reference on every bit of syntax of the JS language. It's also not intended to be a complete "intro to JS" primer.

Instead, we're just going to survey some of the major topic areas of the language. Our goal is to get a better *feel* for it, so that we can move forward writing our own programs with more confidence. We'll revisit many of these topics in successively more detail as you go through the rest of this book, and the rest of the series.

Please don't expect this chapter to be a quick read. It's long and there's plenty of detail to chew on. Take your time.

TIP:

If you're still getting familiar with JS, I suggest you reserve plenty of extra time to work through this chapter. Take each section and ponder and explore the topic for awhile. Look through existing JS programs and compare what you see in them to the code and explanations (and opinions!) presented here. You will get a lot more out of the rest of the book and series with a solid foundation of JS's *nature*.

Each File is a Program

Almost every website (web application) you use is comprised of many different JS files (typically with the .js file extension). It's tempting to think of the whole thing (the application) as one program. But JS sees it differently.

In JS, each standalone file is its own separate program.

The reason this matters is primarily around error handling. Since JS treats files as programs, one file may fail (during parse/compile or execution) and that will not necessarily prevent the next file from being processed. Obviously, if your application depends on five .js files, and one of them fails, the overall application will probably only partially operate, at best. It's important to ensure that each file works properly, and that to whatever extent possible, they handle failure in other files as gracefully as possible.

It may surprise you to consider separate .js files as separate JS programs. From the perspective of your usage of an application, it sure seems like one big program. That's because the execution of the application allows these individual *programs* to cooperate and act as one program.

NOTE:

Many projects use build process tools that end up combining separate files from the project into a single file to be delivered to a web page. When this happens, JS treats this single combined file as the entire program.

The only way multiple standalone .js files act as a single program is by sharing their state (and access to their public functionality) via the "global scope." They mix together in this global scope namespace, so at runtime they act as as whole.

Since ES6, JS has also supported a module format in addition to the typical standalone JS program format. Modules are also file-based. If a file is loaded via module-loading mechanism such as an `import` statement or a `<script type=module>` tag, all its code is treated as a single module.

Though you wouldn't typically think about a module—a collection of state and publicly exposed methods to operate on that state—as a standalone program, JS does in fact still treat each module separately. Similar to how "global scope" allows standalone files to mix together at runtime, importing a module into another allows runtime interoperation between them.

Regardless of which code organization pattern (and loading mechanism) is used for a file (standalone or module), you should still think of each file as its own (mini) program, which may then cooperate with other (mini) programs to perform the functions of your overall application.

Values

The most fundamental unit of information in a program is a value. Values are data. They're how the program maintains state. Values come in two forms in JS: **primitive** and **object**.

Values are embedded in programs using *literals*:

```
greeting("My name is Kyle.");
```

In this program, the value `"My name is Kyle."` is a primitive string literal; strings are ordered collections of characters, usually used to represent words and sentences.

I used the double-quote `"` character to *delimit* (surround, separate, define) the string value. But I could have used the single-quote `'` character as well. The choice of which quote character is entirely stylistic. The important thing, for the sake of code readability and maintainability, is to pick one and to use it consistently throughout the program.

Another option to delimit a string literal is to use the back-tick `` ` `` character. However, this choice is not merely stylistic; there's a behavioral difference as well. Consider:

```
console.log("My name is ${ firstName }.");
// My name is ${ firstName }.

console.log('My name is ${ firstName }.');
// My name is ${ firstName }.

console.log(`My name is ${ firstName }.`);
// My name is Kyle.
```

Assuming this program has already defined a variable `firstName` with the string value `"Kyle"`, the `` ` `` -delimited string then resolves the variable expression (indicated with `${ .. }`) to its current value. This is called **interpolation**.

The back-tick `` ` `` -delimited string can be used without including interpolated expressions, but that defeats the whole purpose of that alternate string literal syntax:

```
console.log(
    `Am I confusing you by omitting interpolation?`
);
// Am I confusing you by omitting interpolation?
```

The better approach is to use `"` or `'` (again, pick one and stick to it!) for strings *unless you need* interpolation; reserve `` ` `` only for strings that will include interpolated expressions.

Other than strings, JS programs often contain other primitive literal values such as booleans and numbers:

```
while (false) {
    console.log(3.141592);
}
```

`while` represents a loop type, a way to repeat operations *while* its condition is true.

In this case, the loop will never run (and nothing will be printed), because we used the `false` boolean value as the loop conditional. `true` would have resulted in a loop that keeps going forever, so be careful!

The number `3.141592` is, as you may know, an approximation of mathematical PI to the first six digits. Rather than embed such a value, however, you would typically use the predefined `Math.PI` value for that purpose. Another variation on numbers is the `bigint` (big-integer) primitive type, which is used for storing arbitrarily large numbers.

Numbers are most often used in programs for counting steps, such as loop iterations, and accessing information in numeric positions (i.e., an array index). We'll cover arrays/objects in a little bit, but as an example, if there was an array called `names`, we could access the element in its second position like this:

```
console.log(`My name is ${ names[1] }.`);
// My name is Kyle.
```

We used `1` for the element in the second position, instead of `2`, because like in most programming languages, JS array indices are 0-based (`0` is the first position).

In addition to strings, numbers, and booleans, two other *primitive* values in JS programs are `null` and `undefined`. While there are differences between them (some historic and some contemporary), for the most part both values serve the purpose of indicating *emptiness* (or absence) of a value.

Many developers prefer to treat them both consistently in this fashion, which is to say that the values are assumed to be indistinguishable. If care is taken, this is often possible. However, it's safest and best to use only `undefined` as the single empty value, even though `null` seems attractive in that it's shorter to type!

```
while (value != undefined) {
    console.log("Still got something!");
}
```

The final primitive value to be aware of is a symbol, which is a special-purpose value that behaves as a hidden unguessable value. Symbols are almost exclusively used as special keys on objects:

```
hitchhikersGuide[ Symbol("meaning of life") ];
// 42
```

You won't encounter direct usage of symbols very often in typical JS programs. They're mostly used in low-level code such as in libraries and frameworks.

Arrays And Objects

Besides primitives, the other value type in JS is an object value.

As mentioned earlier, arrays are a special type of object that's comprised of an ordered and numerically indexed list of data:

```
var names = [ "Frank", "Kyle", "Peter", "Susan" ];

names.length;
// 4

names[0];
// Frank

names[1];
// Kyle
```

JS arrays can hold any value type, either primitive or object (including other arrays). As we'll see toward the end of Chapter 3, even functions are values that can be held in arrays or objects.

NOTE:

Functions, like arrays, are a special kind (aka, sub-type) of object. We'll cover functions in more detail in a bit.

Objects are more general: an unordered, keyed collection of any various values. In other words, you access the element by a string location name (aka "key" or "property") rather than by its numeric position (as with arrays). For example:

```
var me = {
    first: "Kyle",
    last: "Simpson",
    age: 39,
    specialties: [ "JS", "Table Tennis" ]
};

console.log(`My name is ${ me.first }.`);
```

Here, `me` represents an object, and `first` represents the name of a location of information in that object (value collection). Another syntax option that accesses information in an object by its property/key uses the square-brackets `[ ]`, such as `me["first"]`.

Value Type Determination

For distinguishing values, the `typeof` operator tells you its built-in type, if primitive, or `"object"` otherwise:

```
typeof 42;                  // "number"
typeof "abc";               // "string"
typeof true;                // "boolean"
typeof undefined;           // "undefined"
typeof null;                // "object" -- oops, bug!
typeof { "a": 1 };          // "object"
typeof [1,2,3];             // "object"
typeof function hello(){};  // "function"
```

WARNING:

`typeof null` unfortunately returns `"object"` instead of the expected `"null"`. Also, `typeof` returns the specific `"function"` for functions, but not the expected `"array"` for arrays.

Converting from one value type to another, such as from string to number, is referred to in JS as "coercion." We'll cover this in more detail later in this chapter.

Primitive values and object values behave differently when they're assigned or passed around. We'll cover these details in Appendix A, "Values vs References."

Declaring and Using Variables

To be explicit about something that may not have been obvious in the previous section: in JS programs, values can either appear as literal values (as many of the preceding examples illustrate), or they can be held in variables; think of variables as just containers for values.

Variables have to be declared (created) to be used. There are various syntax forms that declare variables (aka, "identifiers"), and each form has different implied behaviors.

For example, consider the `var` statement:

```
var myName = "Kyle";
var age;
```

The `var` keyword declares a variable to be used in that part of the program, and optionally allows an initial assignment of a value.

Another similar keyword is `let`:

```
let myName = "Kyle";
let age;
```

The `let` keyword has some differences to `var`, with the most obvious being that `let` allows a more limited access to the variable than `var`. This is called "block scoping" as opposed to regular or function scoping.

Consider:

```
var adult = true;

if (adult) {
    var myName = "Kyle";
    let age = 39;
    console.log("Shhh, this is a secret!");
}

console.log(myName);
// Kyle

console.log(age);
// Error!
```

The attempt to access `age` outside of the `if` statement results in an error, because `age` was block-scoped to the `if`, whereas `myName` was not.

Block-scoping is very useful for limiting how widespread variable declarations are in our programs, which helps prevent accidental overlap of their names.

But `var` is still useful in that it communicates "this variable will be seen by a wider scope (of the whole function)". Both declaration forms can be appropriate in any given part of a program, depending on the circumstances.

NOTE:

It's very common to suggest that `var` should be avoided in favor of `let` (or `const`!), generally because of perceived confusion over how the scoping behavior of `var` has worked since the beginning of JS. I believe this to be overly restrictive advice and ultimately unhelpful. It's assuming you are unable to learn and use a feature properly in combination with other features. I believe you *can* and *should* learn any features available, and use them where appropriate!

A third declaration form is `const`. It's like `let` but has an additional limitation that it must be given a value at the moment it's declared, and cannot be re-assigned a different value later.

Consider:

```
const myBirthday = true;
let age = 39;

if (myBirthday) {
    age = age + 1;    // OK!
    myBirthday = false;  // Error!
}
```

The `myBirthday` constant is not allowed to be re-assigned.

`const` declared variables are not "unchangeable", they just cannot be re-assigned. It's ill-advised to use `const` with object values, because those values can still be changed even though the variable can't be re-assigned. This leads to potential confusion down the line, so I think it's wise to avoid situations like:

```
const actors = [
    "Morgan Freeman", "Jennifer Aniston"
];

actors[2] = "Tom Cruise";   // OK :(
actors = [];                // Error!
```

The best semantic use of a `const` is when you have a simple primitive value that you want to give a useful name to, such as using `myBirthday` instead of `true`. This makes programs easier to read.

> **TIP:**
>
> If you stick to using `const` only with primitive values, you avoid any confusion of re-assignment (not allowed) vs. mutation (allowed)! That's the safest and best way to use `const`.

Besides `var` / `let` / `const`, there are other syntactic forms that declare identifiers (variables) in various scopes. For example:

```
function hello(myName) {
    console.log(`Hello, ${ myName }.`);
}

hello("Kyle");
// Hello, Kyle.
```

The identifier `hello` is created in the outer scope, and it's also automatically associated so that it references the function. But the named parameter `myName` is created only inside the function, and thus is only accessible inside that function's scope. `hello` and `myName` generally behave as `var`-declared.

Another syntax that declares a variable is a `catch` clause:

```
try {
    someError();
}
catch (err) {
    console.log(err);
}
```

The `err` is a block-scoped variable that exists only inside the `catch` clause, as if it had been declared with `let`.

Functions

The word "function" has a variety of meanings in programming. For example, in the world of Functional Programming, "function" has a precise mathematical definition and implies a strict set of rules to abide by.

In JS, we should consider "function" to take the broader meaning of another related term: "procedure." A procedure is a collection of statements that can be invoked one or more times, may be provided some inputs, and may give back one or more outputs.

From the early days of JS, function definition looked like:

```
function awesomeFunction(coolThings) {
    // ..
    return amazingStuff;
}
```

This is called a function declaration because it appears as a statement by itself, not as an expression in another statement. The association between the identifier `awesomeFunction` and the function value happens during the compile phase of the code, before that code is executed.

In contrast to a function declaration statement, a function expression can be defined and assigned like this:

```
// let awesomeFunction = ..
// const awesomeFunction = ..
var awesomeFunction = function(coolThings) {
    // ..
    return amazingStuff;
};
```

This function is an expression that is assigned to the variable `awesomeFunction`. Different from the function declaration form, a function expression is not associated with its identifier until that statement during runtime.

It's extremely important to note that in JS, functions are values that can be assigned (as shown in this snippet) and passed around. In fact, JS functions are a special type of the object value type. Not all languages treat functions as values, but it's essential for a language to support the functional programming pattern, as JS does.

JS functions can receive parameter input:

```
function greeting(myName) {
    console.log(`Hello, ${ myName }!`);
}

greeting("Kyle");   // Hello, Kyle!
```

In this snippet, `myName` is called a parameter, which acts as a local variable inside the function. Functions can be defined to receive any number of parameters, from none upward, as you see fit. Each parameter is assigned the argument value that you pass in that position (`"Kyle"` , here) of the call.

Functions also can return values using the `return` keyword:

```
function greeting(myName) {
    return `Hello, ${ myName }!`;
}

var msg = greeting("Kyle");

console.log(msg);   // Hello, Kyle!
```

You can only `return` a single value, but if you have more values to return, you can wrap them up into a single object/array.

Since functions are values, they can be assigned as properties on objects:

```
var whatToSay = {
    greeting() {
        console.log("Hello!");
    },
    question() {
        console.log("What's your name?");
    },
    answer() {
        console.log("My name is Kyle.");
    }
};

whatToSay.greeting();
// Hello!
```

In this snippet, references to three functions (`greeting()` , `question()` , and `answer()`) are included in the object held by `whatToSay` . Each function can be called by accessing the property to retrieve the function reference value. Compare this straightforward style of defining functions on an object to the more sophisticated `class` syntax discussed later in this chapter.

There are many varied forms that `function` s take in JS. We dig into these variations in Appendix A, "So Many Function Forms."

Comparisons

Making decisions in programs requires comparing values to determine their identity and relationship to each other. JS has several mechanisms to enable value comparison, so let's take a closer look at them.

Equal...ish

The most common comparison in JS programs asks the question, "Is this X value *the same as* that Y value?" What exactly does "the same as" really mean to JS, though?

For ergonomic and historical reasons, the meaning is more complicated than the obvious *exact identity* sort of matching. Sometimes an equality comparison intends *exact* matching, but other times the desired comparison is a bit broader, allowing *closely similar* or *interchangeable* matching. In other words, we must be aware of the nuanced differences between an **equality** comparison and an **equivalence** comparison.

If you've spent any time working with and reading about JS, you've certainly seen the so-called "triple-equals" `===` operator, also described as the "strict equality" operator. That seems rather straightforward, right? Surely, "strict" means strict, as in narrow and *exact*.

Not *exact*ly.

Yes, most values participating in an `===` equality comparison will fit with that *exact same* intuition. Consider some examples:

```
3 === 3.0;              // true
"yes" === "yes";        // true
null === null;          // true
false === false;        // true

42 === "42";            // false
"hello" === "Hello";    // false
true === 1;             // false
0 === null;             // false
"" === null;            // false
null === undefined;     // false
```

NOTE:

Another way `===`'s equality comparison is often described is, "checking both the value and the type". In several of the examples we've looked at so far, like `42 === "42"`, the *type* of both values (number, string, etc.) does seem to be the distinguishing factor. There's more to it than that, though. **All** value comparisons in JS consider the type of the values being compared, not *just* the `===` operator. Specifically, `===` disallows any sort of type conversion (aka, "coercion") in its comparison, where other JS comparisons *do* allow coercion.

But the `===` operator does have some nuance to it, a fact many JS developers gloss over, to their detriment. The `===` operator is designed to *lie* in two cases of special values: `NaN` and `-0`. Consider:

```
NaN === NaN;            // false
0 === -0;               // true
```

In the case of `NaN`, the `===` operator *lies* and says that an occurrence of `NaN` is not equal to another `NaN`. In the case of `-0` (yes, this is a real, distinct value you can use intentionally in your programs!), the `===` operator *lies* and says it's equal to the regular `0` value.

Since the *lying* about such comparisons can be bothersome, it's best to avoid using `===` for them. For `NaN` comparisons, use the `Number.isNaN(..)` utility, which does not *lie*. For `-0` comparison, use the `Object.is(..)` utility, which also does not *lie*. `Object.is(..)` can also be used for non-*lying* `NaN` checks, if you prefer. Humorously, you could think of `Object.is(..)` as the "quadruple-equals" `====`, the really-really-strict comparison!

There are deeper historical and technical reasons for these *lies*, but that doesn't change the fact that `===` is not actually *strictly exactly equal* comparison, in the *strictest* sense.

The story gets even more complicated when we consider comparisons of object values (non-primitives). Consider:

```
[ 1, 2, 3 ] === [ 1, 2, 3 ];    // false
{ a: 42 } === { a: 42 }         // false
(x => x * 2) === (x => x * 2)   // false
```

What's going on here?

It may seem reasonable to assume that an equality check considers the *nature* or *contents* of the value; after all, `42 === 42` considers the actual `42` value and compares it. But when it comes to objects, a content-aware comparison is generally referred to as "structural equality."

JS does not define `===` as *structural equality* for object values. Instead, `===` uses *identity equality* for object values.

In JS, all object values are held by reference (see "Values vs References" in Appendix A), are assigned and passed by reference-copy, **and** to our current discussion, are compared by reference (identity) equality. Consider:

```
var x = [ 1, 2, 3 ];

// assignment is by reference-copy, so
// y references the *same* array as x,
// not another copy of it.
var y = x;

y === x;              // true
y === [ 1, 2, 3 ];    // false
x === [ 1, 2, 3 ];    // false
```

In this snippet, `y === x` is true because both variables hold a reference to the same initial array. But the `=== [1,2,3]` comparisons both fail because `y` and `x`, respectively, are being compared to new *different* arrays `[1,2,3]`. The array structure and contents don't matter in this comparison, only the **reference identity**.

JS does not provide a mechanism for structural equality comparison of object values, only reference identity comparison. To do structural equality comparison, you'll need to implement the checks yourself.

But beware, it's more complicated than you'll assume. For example, how might you determine if two function references are "structurally equivalent"? Even stringifying to compare their source code text wouldn't take into account things like closure. JS doesn't provide structural equality comparison because it's almost intractable to handle all the corner cases!

Coercive Comparisons

Coercion means a value of one type being converted to its respective representation in another type (to whatever extent possible). As we'll discuss in Chapter 4, coercion is a core pillar of the JS language, not some optional feature that can reasonably be avoided.

But where coercion meets comparison operators (like equality), confusion and frustration unfortunately crop up more often than not.

Few JS features draw more ire in the broader JS community than the `==` operator, generally referred to as the "loose equality" operator. The majority of all writing and public discourse on JS condemns this operator as poorly designed and dangerous/bug-ridden when used in JS programs. Even the creator of the language himself, Brendan Eich, has lamented how it was designed as a big mistake.

From what I can tell, most of this frustration comes from a pretty short list of confusing corner cases, but a deeper problem is the extremely widespread misconception that it performs its comparisons without considering the types of its compared values.

The `==` operator performs an equality comparison similarly to how the `===` performs it. In fact, both operators consider the type of the values being compared. And if the comparison is between the same value type, both `==` and `===` **do exactly the same thing, no difference whatsoever.**

If the value types being compared are different, the `==` differs from `===` in that it allows coercion before the comparison. In other words, they both want to compare values of like types, but `==` allows type conversions *first*, and once the types have been converted to be the same on both sides, then `==` does the same thing as `===`. Instead of "loose equality," the `==` operator should be described as "coercive equality."

Consider:

```
42 == "42";             // true
1 == true;              // true
```

In both comparisons, the value types are different, so the `==` causes the non-number values (`"42"` and `true`) to be converted to numbers (`42` and `1`, respectively) before the comparisons are made.

Just being aware of this nature of `==`—that it prefers primitive numeric comparisons—helps you avoid most of the troublesome corner cases, such as staying away from a gotchas like `"" == 0` or `0 == false`.

You may be thinking, "Oh, well, I will just always avoid any coercive equality comparison (using `===` instead) to avoid those corner cases"! Eh, sorry, that's not quite as likely as you would hope.

There's a pretty good chance that you'll use relational comparison operators like `<`, `>` (and even `<=` and `>=`).

Just like `==`, these operators will perform as if they're "strict" if the types being relationally compared already match, but they'll allow coercion first (generally, to numbers) if the types differ.

Consider:

```
var arr = [ "1", "10", "100", "1000" ];
for (let i = 0; i < arr.length && arr[i] < 500; i++) {
    // will run 3 times
}
```

The `i < arr.length` comparison is "safe" from coercion because `i` and `arr.length` are always numbers. The `arr[i] < 500` invokes coercion, though, because the `arr[i]` values are all strings. Those comparisons thus become `1 < 500`, `10 < 500`, `100 < 500`, and `1000 < 500`. Since that fourth one is false, the loop stops after its third iteration.

These relational operators typically use numeric comparisons, except in the case where **both** values being compared are already strings; in this case, they use alphabetical (dictionary-like) comparison of the strings:

```
var x = "10";
var y = "9";

x < y;      // true, watch out!
```

There's no way to get these relational operators to avoid coercion, other than to just never use mismatched types in the comparisons. That's perhaps admirable as a goal, but it's still pretty likely you're going to run into a case where the types *may* differ.

The wiser approach is not to avoid coercive comparisons, but to embrace and learn their ins and outs.

Coercive comparisons crop up in other places in JS, such as conditionals (`if`, etc.), which we'll revisit in Appendix A, "Coercive Conditional Comparison."

How We Organize in JS

Two major patterns for organizing code (data and behavior) are used broadly across the JS ecosystem: classes and modules. These patterns are not mutually exclusive; many programs can and do use both. Other programs will stick with just one pattern, or even neither!

In some respects, these patterns are very different. But interestingly, in other ways, they're just different sides of the same coin. Being proficient in JS requires understanding both patterns and where they are appropriate (and not!).

Classes

The terms "object-oriented," "class-oriented," and "classes" are all very loaded full of detail and nuance; they're not universal in definition.

We will use a common and somewhat traditional definition here, the one most likely familiar to those with backgrounds in "object-oriented" languages like C++ and Java.

A class in a program is a definition of a "type" of custom data structure that includes both data and behaviors that operate on that data. Classes define how such a data structure works, but classes are not themselves concrete values. To get a concrete value that you can use in the program, a class must be *instantiated* (with the `new` keyword) one or more times.

Consider:

```
class Page {
    constructor(text) {
        this.text = text;
    }

    print() {
        console.log(this.text);
    }
}

class Notebook {
    constructor() {
        this.pages = [];
    }

    addPage(text) {
        var page = new Page(text);
        this.pages.push(page);
    }

    print() {
        for (let page of this.pages) {
            page.print();
        }
    }
}

var mathNotes = new Notebook();
mathNotes.addPage("Arithmetic: + - * / ...");
mathNotes.addPage("Trigonometry: sin cos tan ...");

mathNotes.print();
// ..
```

In the `Page` class, the data is a string of text stored in a `this.text` member property. The behavior is `print()`, a method that dumps the text to the console.

For the `Notebook` class, the data is an array of `Page` instances. The behavior is `addPage(..)`, a method that instantiates new `Page` pages and adds them to the list, as well as `print()` (which prints out all the pages in the notebook).

The statement `mathNotes = new Notebook()` creates an instance of the `Notebook` class, and `page = new Page(text)` is where instances of the `Page` class are created.

Behavior (methods) can only be called on instances (not the classes themselves), such as `mathNotes.addPage(..)` and `page.print()`.

The `class` mechanism allows packaging data (`text` and `pages`) to be organized together with their behaviors (e.g., `addPage(..)` and `print()`). The same program could have been built without any `class` definitions, but it would likely have been much less organized, harder to read and reason about, and more susceptible to bugs and subpar maintenance.

Class Inheritance

Another aspect inherent to traditional "class-oriented" design, though a bit less commonly used in JS, is "inheritance" (and "polymorphism"). Consider:

```
class Publication {
    constructor(title,author,pubDate) {
        this.title = title;
        this.author = author;
        this.pubDate = pubDate;
    }

    print() {
        console.log(`
            Title: ${ this.title }
            By: ${ this.author }
            ${ this.pubDate }
        `);
    }
}
```

This `Publication` class defines a set of common behavior that any publication might need.

Now let's consider more specific types of publication, like `Book` and `BlogPost`:

```
class Book extends Publication {
    constructor(bookDetails) {
        super(
            bookDetails.title,
            bookDetails.author,
            bookDetails.publishedOn
        );
        this.publisher = bookDetails.publisher;
        this.ISBN = bookDetails.ISBN;
    }

    print() {
        super.print();
        console.log(`
            Publisher: ${ this.publisher }
            ISBN: ${ this.ISBN }
        `);
    }
}

class BlogPost extends Publication {
    constructor(title,author,pubDate,URL) {
        super(title,author,pubDate);
        this.URL = URL;
    }

    print() {
        super.print();
        console.log(this.URL);
    }
}
```

Both `Book` and `BlogPost` use the `extends` clause to *extend* the general definition of `Publication` to include additional behavior. The `super(..)` call in each constructor delegates to the parent `Publication` class's constructor for its initialization work, and then they do more specific things according to their respective publication type (aka, "sub-class" or "child class").

Now consider using these child classes:

```
var YDKJS = new Book({
    title: "You Don't Know JS",
    author: "Kyle Simpson",
    publishedOn: "June 2014",
    publisher: "O'Reilly",
    ISBN: "123456-789"
});

YDKJS.print();
// Title: You Don't Know JS
// By: Kyle Simpson
// June 2014
// Publisher: O'Reilly
// ISBN: 123456-789

var forAgainstLet = new BlogPost(
    "For and against let",
    "Kyle Simpson",
    "October 27, 2014",
    "https://davidwalsh.name/for-and-against-let"
);

forAgainstLet.print();
// Title: For and against let
// By: Kyle Simpson
// October 27, 2014
// https://davidwalsh.name/for-and-against-let
```

Notice that both child class instances have a `print()` method, which was an override of the *inherited* `print()` method from the parent `Publication` class. Each of those overridden child class `print()` methods call `super.print()` to invoke the inherited version of the `print()` method.

The fact that both the inherited and overridden methods can have the same name and co-exist is called *polymorphism*.

Inheritance is a powerful tool for organizing data/behavior in separate logical units (classes), but allowing the child class to cooperate with the parent by accessing/using its behavior and data.

Modules

The module pattern has essentially the same goal as the class pattern, which is to group data and behavior together into logical units. Also like classes, modules can "include" or "access" the data and behaviors of other modules, for cooperation's sake.

But modules have some important differences from classes. Most notably, the syntax is entirely different.

Classic Modules

ES6 added a module syntax form to native JS syntax, which we'll look at in a moment. But from the early days of JS, modules was an important and common pattern that was leveraged in countless JS programs, even without a dedicated syntax.

The key hallmarks of a *classic module* are an outer function (that runs at least once), which returns an "instance" of the module with one or more functions exposed that can operate on the module instance's internal (hidden) data.

Because a module of this form is *just a function*, and calling it produces an "instance" of the module, another description for these functions is "module factories".

Consider the classic module form of the earlier `Publication`, `Book`, and `BlogPost` classes:

```
function Publication(title,author,pubDate) {
    var publicAPI = {
        print() {
            console.log(`
                Title: ${ title }
                By: ${ author }
                ${ pubDate }
            `);
        }
    };

    return publicAPI;
}

function Book(bookDetails) {
    var pub = Publication(
        bookDetails.title,
        bookDetails.author,
        bookDetails.publishedOn
    );

    var publicAPI = {
        print() {
            pub.print();
            console.log(`
                Publisher: ${ bookDetails.publisher }
                ISBN: ${ bookDetails.ISBN }
            `);
        }
    };

    return publicAPI;
}

function BlogPost(title,author,pubDate,URL) {
    var pub = Publication(title,author,pubDate);

    var publicAPI = {
        print() {
            pub.print();
            console.log(URL);
        }
    };

    return publicAPI;
}
```

Comparing these forms to the `class` forms, there are more similarities than differences.

The `class` form stores methods and data on an object instance, which must be accessed with the `this.` prefix. With modules, the methods and data are accessed as identifier variables in scope, without any `this.` prefix.

With `class`, the "API" of an instance is implicit in the class definition—also, all data and methods are public. With the module factory function, you explicitly create and return an object with any publicly exposed methods, and any data or other unreferenced methods remain private inside the factory function.

There are other variations to this factory function form that are quite common across JS, even in 2020; you may run across these forms in different JS programs: AMD (Asynchronous Module Definition), UMD (Universal Module Definition), and CommonJS (classic Node.js-style modules). The variations are minor (not quite compatible). However, all of these forms rely on the same basic principles.

Consider also the usage (aka, "instantiation") of these module factory functions:

```
var YDKJS = Book({
    title: "You Don't Know JS",
    author: "Kyle Simpson",
    publishedOn: "June 2014",
    publisher: "O'Reilly",
    ISBN: "123456-789"
});

YDKJS.print();
// Title: You Don't Know JS
// By: Kyle Simpson
// June 2014
// Publisher: O'Reilly
// ISBN: 123456-789

var forAgainstLet = BlogPost(
    "For and against let",
    "Kyle Simpson",
    "October 27, 2014",
    "https://davidwalsh.name/for-and-against-let"
);

forAgainstLet.print();
// Title: For and against let
// By: Kyle Simpson
// October 27, 2014
// https://davidwalsh.name/for-and-against-let
```

The only observable difference here is the lack of using `new`, calling the module factories as normal functions.

ES Modules

ES modules (ESM), introduced to the JS language in ES6, are meant to serve much the same spirit and purpose as the existing *classic modules* just described, especially taking into account important variations and use cases from AMD, UMD, and CommonJS.

The implementation approach does, however, differ significantly.

First, there's no wrapping function to *define* a module. The wrapping context is a file. ESMs are always file-based; one file, one module.

Second, you don't interact with a module's "API" explicitly, but rather use the `export` keyword to add a variable or method to its public API definition. If something is defined in a module but not `export`ed, then it stays hidden (just as with *classic modules*).

Third, and maybe most noticeably different from previously discussed patterns, you don't "instantiate" an ES module, you just `import` it to use its single instance. ESMs are, in effect, "singletons," in that there's only one instance ever created, at first `import` in your program, and all other `import`s just receive a reference to that same single instance. If your module needs to support multiple instantiations, you have to provide a *classic module-style* factory function on your ESM definition for that purpose.

In our running example, we do assume multiple-instantiation, so these following snippets will mix both ESM and *classic modules*.

Consider the file `publication.js`:

```
function printDetails(title,author,pubDate) {
    console.log(`
        Title: ${ title }
        By: ${ author }
        ${ pubDate }
    `);
}

export function create(title,author,pubDate) {
    var publicAPI = {
        print() {
            printDetails(title,author,pubDate);
        }
    };

    return publicAPI;
}
```

To import and use this module, from another ES module like `blogpost.js`:

```
import { create as createPub } from "publication.js";

function printDetails(pub,URL) {
    pub.print();
    console.log(URL);
}

export function create(title,author,pubDate,URL) {
    var pub = createPub(title,author,pubDate);

    var publicAPI = {
        print() {
            printDetails(pub,URL);
        }
    };

    return publicAPI;
}
```

And finally, to use this module, we import into another ES module like `main.js`:

```
import { create as newBlogPost } from "blogpost.js";

var forAgainstLet = newBlogPost(
    "For and against let",
    "Kyle Simpson",
    "October 27, 2014",
    "https://davidwalsh.name/for-and-against-let"
);

forAgainstLet.print();
// Title: For and against let
// By: Kyle Simpson
// October 27, 2014
// https://davidwalsh.name/for-and-against-let
```

NOTE:

The `as newBlogPost` clause in the `import` statement is optional; if omitted, a top-level function just namec `create(..)` would be imported. In this case, I'm renaming it for readability's sake; its more generic factory name of `create(..)` becomes more semantically descriptive of its purpose as `newBlogPost(..)`.

As shown, ES modules can use *classic modules* internally if they need to support multiple-instantiation. Alternatively, we could have exposed a `class` from our module instead of a `create(..)` factory function, with generally the same outcome. However, since you're already using ESM at that point, I'd recommend sticking with *classic modules* instead of `class`.

If your module only needs a single instance, you can skip the extra layers of complexity: `export` its public methods directly.

The Rabbit Hole Deepens

As promised at the top of this chapter, we just glanced over a wide surface area of the main parts of the JS language. Your head may still be spinning, but that's entirely natural after such a firehose of information!

Even with just this "brief" survey of JS, we covered or hinted at a ton of details you should carefully consider and ensure you are comfortable with. I'm serious when I suggest: re-read this chapter, maybe several times.

In the next chapter, we're going to dig much deeper into some important aspects of how JS works at its core. But before you follow that rabbit hole deeper, make sure you've taken adequate time to fully digest what we've just covered here.