qqli1209 / **You-Dont-Know-JS**

forked from getify/You-Dont-Know-JS

Branch: master ▾ **You-Dont-Know-JS** / this & object prototypes / **ch4.md**

Find file | Copy path

ncpierson fix grammar in "this & object prototypes" chap. 4   cb5021a on 15 Feb 2016

9 contributors

479 lines (300 sloc) | 33.8 KB

You Don't Know JS: *this* & Object Prototypes

Chapter 4: Mixing (Up) "Class" Objects

Following our exploration of objects from the previous chapter, it's natural that we now turn our attention to "object oriented (OO) programming", with "classes". We'll first look at "class orientation" as a design pattern, before examining the mechanics of "classes": "instantiation", "inheritance" and "(relative) polymorphism".

We'll see that these concepts don't really map very naturally to the object mechanism in JS, and the lengths (mixins, etc.) many JavaScript developers go to overcome such challenges.

Note: This chapter spends quite a bit of time (the first half!) on heavy "objected oriented programming" theory. We eventually relate these ideas to real concrete JavaScript code in the second half, when we talk about "Mixins". But there's a lot of concept and pseudo-code to wade through first, so don't get lost -- just stick with it!

Class Theory

"Class/Inheritance" describes a certain form of code organization and architecture -- a way of modeling real world problem domains in our software.

OO or class oriented programming stresses that data intrinsically has associated behavior (of course, different depending on the type and nature of the data!) that operates on it, so proper design is to package up (aka, encapsulate) the data and the behavior together. This is sometimes called "data structures" in formal computer science.

For example, a series of characters that represents a word or phrase is usually called a "string". The characters are the data. But you almost never just care about the data, you usually want to *do things* with the data, so the behaviors that can apply *to* that data (calculating its length, appending data, searching, etc.) are all designed as methods of a `String` class.

Any given string is just an instance of this class, which means that it's a neatly collected packaging of both the character data and the functionality we can perform on it.

Classes also imply a way of *classifying* a certain data structure. The way we do this is to think about any given structure as a specific variation of a more general base definition.

Let's explore this classification process by looking at a commonly cited example. A *car* can be described as a specific implementation of a more general "class" of thing, called a *vehicle*.

We model this relationship in software with classes by defining a `Vehicle` class and a `Car` class.

The definition of `Vehicle` might include things like propulsion (engines, etc.), the ability to carry people, etc., which would all be the behaviors. What we define in `Vehicle` is all the stuff that is common to all (or most of) the different types of vehicles (the "planes, trains, and automobiles").

It might not make sense in our software to re-define the basic essence of "ability to carry people" over and over again for each different type of vehicle. Instead, we define that capability once in `Vehicle`, and then when we define `Car`, we simply indicate that it "inherits" (or "extends") the base definition from `Vehicle`. The definition of `Car` is said to specialize the general `Vehicle` definition.

While `Vehicle` and `Car` collectively define the behavior by way of methods, the data in an instance would be things like the unique VIN of a specific car, etc.

And thus, classes, inheritance, and instantiation emerge.

Another key concept with classes is "polymorphism", which describes the idea that a general behavior from a parent class can be overridden in a child class to give it more specifics. In fact, relative polymorphism lets us reference the base behavior from the overridden behavior.

Class theory strongly suggests that a parent class and a child class share the same method name for a certain behavior, so that the child overrides the parent (differentially). As we'll see later, doing so in your JavaScript code is opting into frustration and code brittleness.

"Class" Design Pattern

You may never have thought about classes as a "design pattern", since it's most common to see discussion of popular "OO Design Patterns", like "Iterator", "Observer", "Factory", "Singleton", etc. As presented this way, it's almost an assumption that OO classes are the lower-level mechanics by which we implement all (higher level) design patterns, as if OO is a given foundation for *all* (proper) code.

Depending on your level of formal education in programming, you may have heard of "procedural programming" as a way of describing code which only consists of procedures (aka, functions) calling other functions, without any higher abstractions. You may have been taught that classes were the *proper* way to transform procedural-style "spaghetti code" into well-formed, well-organized code.

Of course, if you have experience with "functional programming" (Monads, etc.), you know very well that classes are just one of several common design patterns. But for others, this may be the first time you've asked yourself if classes really are a fundamental foundation for code, or if they are an optional abstraction on top of code.

Some languages (like Java) don't give you the choice, so it's not very *optional* at all -- everything's a class. Other languages like C/C++ or PHP give you both procedural and class-oriented syntaxes, and it's left more to the developer's choice which style or mixture of styles is appropriate.

JavaScript "Classes"

Where does JavaScript fall in this regard? JS has had *some* class-like syntactic elements (like `new` and `instanceof`) for quite awhile, and more recently in ES6, some additions, like the `class` keyword (see Appendix A).

But does that mean JavaScript actually *has* classes? Plain and simple: **No.**

Since classes are a design pattern, you *can*, with quite a bit of effort (as we'll see throughout the rest of this chapter), implement approximations for much of classical class functionality. JS tries to satisfy the extremely pervasive *desire* to design with classes by providing seemingly class-like syntax.

While we may have a syntax that looks like classes, it's as if JavaScript mechanics are fighting against you using the *class design pattern*, because behind the curtain, the mechanisms that you build on are operating quite differently. Syntactic sugar and (extremely widely used) JS "Class" libraries go a long way toward hiding this reality from you, but sooner or later you will face the fact that the *classes* you have in other languages are not like the "classes" you're faking in JS.

What this boils down to is that classes are an optional pattern in software design, and you have the choice to use them in JavaScript or not. Since many developers have a strong affinity to class oriented software design, we'll spend the rest of this chapter exploring what it takes to maintain the illusion of classes with what JS provides, and the pain points we experience.

Class Mechanics

In many class-oriented languages, the "standard library" provides a "stack" data structure (push, pop, etc.) as a `Stack` class. This class would have an internal set of variables that stores the data, and it would have a set of publicly accessible behaviors ("methods") provided by the class, which gives your code the ability to interact with the (hidden) data (adding & removing data, etc.).

But in such languages, you don't really operate directly on `Stack` (unless making a **Static** class member reference, which is outside the scope of our discussion). The `Stack` class is merely an abstract explanation of what *any* "stack" should do, but it's not itself *a* "stack". You must **instantiate** the `Stack` class before you have a concrete data structure *thing* to operate against.

Building

The traditional metaphor for "class" and "instance" based thinking comes from a building construction.

An architect plans out all the characteristics of a building: how wide, how tall, how many windows and in what locations, even what type of material to use for the walls and roof. She doesn't necessarily care, at this point, *where* the building will be built, nor does she care *how many* copies of that building will be built.

She also doesn't care very much about the contents of the building -- the furniture, wall paper, ceiling fans, etc. -- only what type of structure they will be contained by.

The architectural blue-prints she produces are only *plans* for a building. They don't actually constitute a building we can walk into and sit down. We need a builder for that task. A builder will take those plans and follow them, exactly, as he *builds* the building. In a very real sense, he is *copying* the intended characteristics from the plans to the physical building.

Once complete, the building is a physical instantiation of the blue-print plans, hopefully an essentially perfect *copy*. And then the builder can move to the open lot next door and do it all over again, creating yet another *copy*.

The relationship between building and blue-print is indirect. You can examine a blue-print to understand how the building was structured, for any parts where direct inspection of the building itself was insufficient. But if you want to open a door, you have to go to the building itself -- the blue-print merely has lines drawn on a page that *represent* where the door should be.

A class is a blue-print. To actually *get* an object we can interact with, we must build (aka, "instantiate") something from the class. The end result of such "construction" is an object, typically called an "instance", which we can directly call methods on and access any public data properties from, as necessary.

This object is a *copy* of all the characteristics described by the class.

You likely wouldn't expect to walk into a building and find, framed and hanging on the wall, a copy of the blue-prints used to plan the building, though the blue-prints are probably on file with a public records office. Similarly, you don't generally use an object instance to directly access and manipulate its class, but it is usually possible to at least determine *which class* an object instance comes from.

It's more useful to consider the direct relationship of a class to an object instance, rather than any indirect relationship between an object instance and the class it came from. **A class is instantiated into object form by a copy operation.**
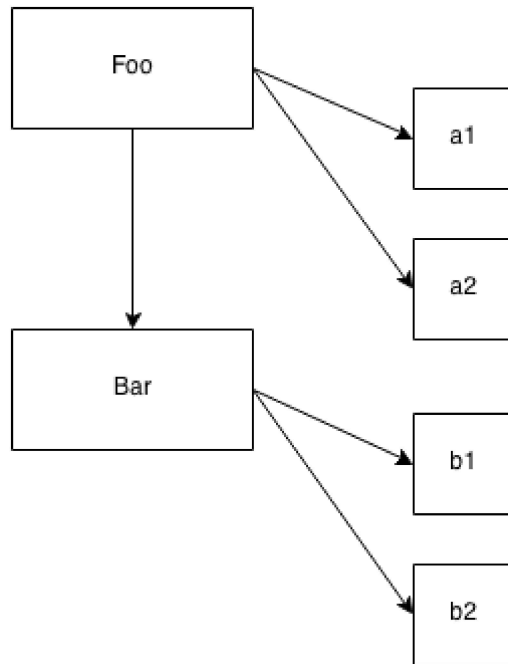
As you can see, the arrows move from left to right, and from top to bottom, which indicates the copy operations that occur, both conceptually and physically.

Constructor

Instances of classes are constructed by a special method of the class, usually of the same name as the class, called a *constructor*. This method's explicit job is to initialize any information (state) the instance will need.

For example, consider this loose pseudo-code (invented syntax) for classes:

```
class CoolGuy {
	specialTrick = nothing
```

```
        CoolGuy( trick ) {
                specialTrick = trick
        }

        showOff() {
                output( "Here's my trick: ", specialTrick )
        }
    }
```

To *make* a `CoolGuy` instance, we would call the class constructor:

```
Joe = new CoolGuy( "jumping rope" )

Joe.showOff() // Here's my trick: jumping rope
```

Notice that the `CoolGuy` class has a constructor `CoolGuy()`, which is actually what we call when we say `new CoolGuy(..)`. We get an object back (an instance of our class) from the constructor, and we can call the method `showOff()`, which prints out that particular `CoolGuy`s special trick.

Obviously, jumping rope makes Joe a pretty cool guy.

The constructor of a class *belongs* to the class, almost universally with the same name as the class. Also, constructors pretty much always need to be called with `new` to let the language engine know you want to construct a *new* class instance.

Class Inheritance

In class-oriented languages, not only can you define a class which can be instantiated itself, but you can define another class that **inherits** from the first class.

The second class is often said to be a "child class" whereas the first is the "parent class". These terms obviously come from the metaphor of parents and children, though the metaphors here are a bit stretched, as you'll see shortly.

When a parent has a biological child, the genetic characteristics of the parent are copied into the child. Obviously, in most biological reproduction systems, there are two parents who co-equally contribute genes to the mix. But for the purposes of the metaphor, we'll assume just one parent.

Once the child exists, he or she is separate from the parent. The child was heavily influenced by the inheritance from his or her parent, but is unique and distinct. If a child ends up with red hair, that doesn't mean the parent's hair *was* or automatically *becomes* red.

In a similar way, once a child class is defined, it's separate and distinct from the parent class. The child class contains an initial copy of the behavior from the parent, but can then override any inherited behavior and even define new behavior.

It's important to remember that we're talking about parent and child **classes**, which aren't physical things. This is where the metaphor of parent and child gets a little confusing, because we actually should say that a parent class is like a parent's DNA and a child class is like a child's DNA. We have to make (aka "instantiate") a person out of each set of DNA to actually have a physical person to have a conversation with.

Let's set aside biological parents and children, and look at inheritance through a slightly different lens: different types of vehicles. That's one of the most canonical (and often groan-worthy) metaphors to understand inheritance.

Let's revisit the `Vehicle` and `Car` discussion from earlier in this chapter. Consider this loose pseudo-code (invented syntax) for inherited classes:

```
class Vehicle {
    engines = 1

    ignition() {
        output( "Turning on my engine." )
    }

    drive() {
        ignition()
        output( "Steering and moving forward!" )
    }
```

```
    }

    class Car inherits Vehicle {
        wheels = 4

        drive() {
            inherited:drive()
            output( "Rolling on all ", wheels, " wheels!" )
        }
    }

    class SpeedBoat inherits Vehicle {
        engines = 2

        ignition() {
            output( "Turning on my ", engines, " engines." )
        }

        pilot() {
            inherited:drive()
            output( "Speeding through the water with ease!" )
        }
    }
```

Note: For clarity and brevity, constructors for these classes have been omitted.

We define the `Vehicle` class to assume an engine, a way to turn on the ignition, and a way to drive around. But you wouldn't ever manufacture just a generic "vehicle", so it's really just an abstract concept at this point.

So then we define two specific kinds of vehicle: `Car` and `SpeedBoat` . They each inherit the general characteristics of `Vehicle` , but then they specialize the characteristics appropriately for each kind. A car needs 4 wheels, and a speed boat needs 2 engines, which means it needs extra attention to turn on the ignition of both engines.

Polymorphism

`Car` defines its own `drive()` method, which overrides the method of the same name it inherited from `Vehicle` . But then, `Car` s `drive()` method calls `inherited:drive()` , which indicates that `Car` can reference the original pre-overridden `drive()` it inherited. `SpeedBoat` s `pilot()` method also makes a reference to its inherited copy of `drive()` .

This technique is called "polymorphism", or "virtual polymorphism". More specifically to our current point, we'll call it "relative polymorphism".

Polymorphism is a much broader topic than we will exhaust here, but our current "relative" semantics refers to one particular aspect: the idea that any method can reference another method (of the same or different name) at a higher level of the inheritance hierarchy. We say "relative" because we don't absolutely define which inheritance level (aka, class) we want to access, but rather relatively reference it by essentially saying "look one level up".

In many languages, the keyword `super` is used, in place of this example's `inherited:` , which leans on the idea that a "super class" is the parent/ancestor of the current class.

Another aspect of polymorphism is that a method name can have multiple definitions at different levels of the inheritance chain, and these definitions are automatically selected as appropriate when resolving which methods are being called.

We see two occurrences of that behavior in our example above: `drive()` is defined in both `Vehicle` and `Car` , and `ignition()` is defined in both `Vehicle` and `SpeedBoat` .

Note: Another thing that traditional class-oriented languages give you via `super` is a direct way for the constructor of a child class to reference the constructor of its parent class. This is largely true because with real classes, the constructor belongs to the class. However, in JS, it's the reverse -- it's actually more appropriate to think of the "class" belonging to the constructor (the `Foo.prototype...` type references). Since in JS the relationship between child and parent exists only between the two `.prototype` objects of the respective constructors, the constructors themselves are not directly related, and thus there's no simple way to relatively reference one from the other (see Appendix A for ES6 `class` which "solves" this with `super`).

An interesting implication of polymorphism can be seen specifically with `ignition()` . Inside `pilot()` , a relative-polymorphic reference is made to (the inherited) `Vehicle` s version of `drive()` . But that `drive()` references an `ignition()` method just by name (no relative reference).

Which version of `ignition()` will the language engine use, the one from `Vehicle` or the one from `SpeedBoat`? **It uses the `SpeedBoat` version of `ignition()`.** If you *were* to instantiate `Vehicle` class itself, and then call its `drive()`, the language engine would instead just use `Vehicle`s `ignition()` method definition.

Put another way, the definition for the method `ignition()` *polymorphs* (changes) depending on which class (level of inheritance) you are referencing an instance of.

This may seem like overly deep academic detail. But understanding these details is necessary to properly contrast similar (but distinct) behaviors in JavaScript's `[[Prototype]]` mechanism.

When classes are inherited, there is a way **for the classes themselves** (not the object instances created from them!) to *relatively* reference the class inherited from, and this relative reference is usually called `super`.

Remember this figure from earlier:
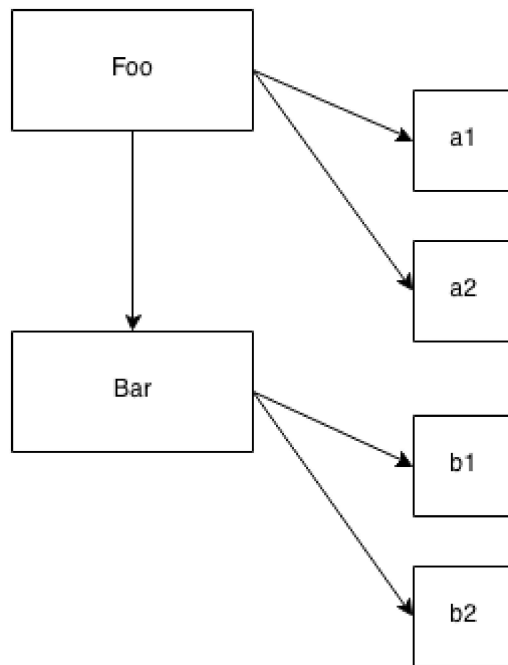
Notice how for both instantiation (`a1`, `a2`, `b1`, and `b2`) *and* inheritance (`Bar`), the arrows indicate a copy operation.

Conceptually, it would seem a child class `Bar` can access behavior in its parent class `Foo` using a relative polymorphic reference (aka, `super`). However, in reality, the child class is merely given a copy of the inherited behavior from its parent class. If the child "overrides" a method it inherits, both the original and overridden versions of the method are actually maintained, so that they are both accessible.

Don't let polymorphism confuse you into thinking a child class is linked to its parent class. A child class instead gets a copy of what it needs from the parent class. **Class inheritance implies copies.**
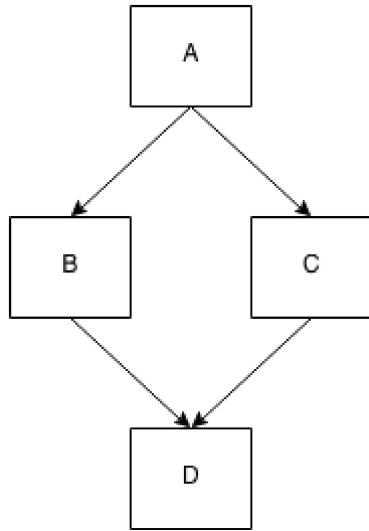
Multiple Inheritance

Recall our earlier discussion of parent(s) and children and DNA? We said that the metaphor was a bit weird because biologically most offspring come from two parents. If a class could inherit from two other classes, it would more closely fit the parent/child metaphor.

Some class-oriented languages allow you to specify more than one "parent" class to "inherit" from. Multiple-inheritance means that each parent class definition is copied into the child class.

On the surface, this seems like a powerful addition to class-orientation, giving us the ability to compose more functionality together. However, there are certainly some complicating questions that arise. If both parent classes provide a method called `drive()`, which version would a `drive()` reference in the child resolve to? Would you always have to manually specify which parent's `drive()` you meant, thus losing some of the gracefulness of polymorphic inheritance?

There's another variation, the so called "Diamond Problem", which refers to the scenario where a child class "D" inherits from two parent classes ("B" and "C"), and each of those in turn inherits from a common "A" parent. If "A" provides a method `drive()`, and both "B" and "C" override (polymorph) that method, when `D` references `drive()`, which version should it use (`B:drive()` or `C:drive()`)?

These complications go even much deeper than this quick glance. We address them here only so we can contrast to how JavaScript's mechanisms work.

JavaScript is simpler: it does not provide a native mechanism for "multiple inheritance". Many see this is a good thing, because the complexity savings more than make up for the "reduced" functionality. But this doesn't stop developers from trying to fake it in various ways, as we'll see next.

Mixins

JavaScript's object mechanism does not *automatically* perform copy behavior when you "inherit" or "instantiate". Plainly, there are no "classes" in JavaScript to instantiate, only objects. And objects don't get copied to other objects, they get *linked together* (more on that in Chapter 5).

Since observed class behaviors in other languages imply copies, let's examine how JS developers **fake** the *missing* copy behavior of classes in JavaScript: mixins. We'll look at two types of "mixin": **explicit** and **implicit**.

Explicit Mixins

Let's again revisit our `Vehicle` and `Car` example from before. Since JavaScript will not automatically copy behavior from `Vehicle` to `Car`, we can instead create a utility that manually copies. Such a utility is often called `extend(..)` by many libraries/frameworks, but we will call it `mixin(..)` here for illustrative purposes.

```
// vastly simplified `mixin(..)` example:
function mixin( sourceObj, targetObj ) {
        for (var key in sourceObj) {
                // only copy if not already present
                if (!(key in targetObj)) {
                        targetObj[key] = sourceObj[key];
                }
        }

        return targetObj;
}

var Vehicle = {
        engines: 1,

        ignition: function() {
                console.log( "Turning on my engine." );
        },

        drive: function() {
                this.ignition();
                console.log( "Steering and moving forward!" );
        }
};

var Car = mixin( Vehicle, {
        wheels: 4,

        drive: function() {
                Vehicle.drive.call( this );
                console.log( "Rolling on all " + this.wheels + " wheels!" );
```

```
        }
    } );
```

Note: Subtly but importantly, we're not dealing with classes anymore, because there are no classes in JavaScript. `Vehicle` and `Car` are just objects that we make copies from and to, respectively.

`Car` now has a copy of the properties and functions from `Vehicle`. Technically, functions are not actually duplicated, but rather *references* to the functions are copied. So, `Car` now has a property called `ignition`, which is a copied reference to the `ignition()` function, as well as a property called `engines` with the copied value of `1` from `Vehicle`.

`Car` *already* had a `drive` property (function), so that property reference was not overridden (see the `if` statement in `mixin(..)` above).

"Polymorphism" Revisited

Let's examine this statement: `Vehicle.drive.call( this )`. This is what I call "explicit pseudo-polymorphism". Recall in our previous pseudo-code this line was `inherited:drive()`, which we called "relative polymorphism".

JavaScript does not have (prior to ES6; see Appendix A) a facility for relative polymorphism. So, **because both `Car` and `Vehicle` had a function of the same name: `drive()`**, to distinguish a call to one or the other, we must make an absolute (not relative) reference. We explicitly specify the `Vehicle` object by name, and call the `drive()` function on it.

But if we said `Vehicle.drive()`, the `this` binding for that function call would be the `Vehicle` object instead of the `Car` object (see Chapter 2), which is not what we want. So, instead we use `.call( this )` (Chapter 2) to ensure that `drive()` is executed in the context of the `Car` object.

Note: If the function name identifier for `Car.drive()` hadn't overlapped with (aka, "shadowed"; see Chapter 5) `Vehicle.drive()`, we wouldn't have been exercising "method polymorphism". So, a reference to `Vehicle.drive()` would have been copied over by the `mixin(..)` call, and we could have accessed directly with `this.drive()`. The chosen identifier overlap **shadowing** is *why* we have to use the more complex *explicit pseudo-polymorphism* approach.

In class-oriented languages, which have relative polymorphism, the linkage between `Car` and `Vehicle` is established once, at the top of the class definition, which makes for only one place to maintain such relationships.

But because of JavaScript's peculiarities, explicit pseudo-polymorphism (because of shadowing!) creates brittle manual/explicit linkage **in every single function where you need such a (pseudo-)polymorphic reference**. This can significantly increase the maintenance cost. Moreover, while explicit pseudo-polymorphism can emulate the behavior of "multiple inheritance", it only increases the complexity and brittleness.

The result of such approaches is usually more complex, harder-to-read, *and* harder-to-maintain code. **Explicit pseudo-polymorphism should be avoided wherever possible**, because the cost outweighs the benefit in most respects.

Mixing Copies

Recall the `mixin(..)` utility from above:

```js
// vastly simplified `mixin()` example:
function mixin( sourceObj, targetObj ) {
        for (var key in sourceObj) {
                // only copy if not already present
                if (!(key in targetObj)) {
                        targetObj[key] = sourceObj[key];
                }
        }

        return targetObj;
}
```

Now, let's examine how `mixin(..)` works. It iterates over the properties of `sourceObj` (`Vehicle` in our example) and if there's no matching property of that name in `targetObj` (`Car` in our example), it makes a copy. Since we're making the copy after the initial object exists, we are careful to not copy over a target property.

If we made the copies first, before specifying the `Car` specific contents, we could omit this check against `targetObj`, but that's a little more clunky and less efficient, so it's generally less preferred:

```js
// alternate mixin, less "safe" to overwrites
function mixin( sourceObj, targetObj ) {
        for (var key in sourceObj) {
                targetObj[key] = sourceObj[key];
        }

        return targetObj;
}

var Vehicle = {
        // ...
};

// first, create an empty object with
// Vehicle's stuff copied in
var Car = mixin( Vehicle, { } );

// now copy the intended contents into Car
mixin( {
        wheels: 4,

        drive: function() {
                // ...
        }
}, Car );
```

Either approach, we have explicitly copied the non-overlapping contents of `Vehicle` into `Car`. The name "mixin" comes from an alternate way of explaining the task: `Car` has `Vehicle`s contents **mixed-in**, just like you mix in chocolate chips into your favorite cookie dough.

As a result of the copy operation, `Car` will operate somewhat separately from `Vehicle`. If you add a property onto `Car`, it will not affect `Vehicle`, and vice versa.

Note: A few minor details have been skimmed over here. There are still some subtle ways the two objects can "affect" each other even after copying, such as if they both share a reference to a common object (such as an array).

Since the two objects also share references to their common functions, that means that **even manual copying of functions (aka, mixins) from one object to another doesn't** *actually emulate* **the real duplication from class to instance that occurs in class-oriented languages**.

JavaScript functions can't really be duplicated (in a standard, reliable way), so what you end up with instead is a **duplicated reference** to the same shared function object (functions are objects; see Chapter 3). If you modified one of the shared **function objects** (like `ignition()`) by adding properties on top of it, for instance, both `Vehicle` and `Car` would be "affected" via the shared reference.

Explicit mixins are a fine mechanism in JavaScript. But they appear more powerful than they really are. Not much benefit is *actually* derived from copying a property from one object to another, **as opposed to just defining the properties twice**, once on each object. And that's especially true given the function-object reference nuance we just mentioned.

If you explicitly mix-in two or more objects into your target object, you can **partially emulate** the behavior of "multiple inheritance", but there's no direct way to handle collisions if the same method or property is being copied from more than one source. Some developers/libraries have come up with "late binding" techniques and other exotic work-arounds, but fundamentally these "tricks" are *usually* more effort (and lesser performance!) than the pay-off.

Take care only to use explicit mixins where it actually helps make more readable code, and avoid the pattern if you find it making code that's harder to trace, or if you find it creates unnecessary or unwieldy dependencies between objects.

If it starts to get *harder* **to properly use mixins than before you used them**, you should probably stop using mixins. In fact, if you have to use a complex library/utility to work out all these details, it might be a sign that you're going about it the harder way, perhaps unnecessarily. In Chapter 6, we'll try to distill a simpler way that accomplishes the desired outcomes without all the fuss.

Parasitic Inheritance

A variation on this explicit mixin pattern, which is both in some ways explicit and in other ways implicit, is called "parasitic inheritance", popularized mainly by Douglas Crockford.

Here's how it can work:

```js
// "Traditional JS Class" `Vehicle`
function Vehicle() {
        this.engines = 1;
}
Vehicle.prototype.ignition = function() {
        console.log( "Turning on my engine." );
};
Vehicle.prototype.drive = function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
};

// "Parasitic Class" `Car`
function Car() {
        // first, `car` is a `Vehicle`
        var car = new Vehicle();

        // now, let's modify our `car` to specialize it
        car.wheels = 4;

        // save a privileged reference to `Vehicle::drive()`
        var vehDrive = car.drive;

        // override `Vehicle::drive()`
        car.drive = function() {
                vehDrive.call( this );
                console.log( "Rolling on all " + this.wheels + " wheels!" );
        };
```

```
        return car;
    }

    var myCar = new Car();

    myCar.drive();
    // Turning on my engine.
    // Steering and moving forward!
    // Rolling on all 4 wheels!
```

As you can see, we initially make a copy of the definition from the `Vehicle` "parent class" (object), then mixin our "child class" (object) definition (preserving privileged parent-class references as needed), and pass off this composed object `car` as our child instance.

Note: when we call `new Car()`, a new object is created and referenced by `Car`s `this` reference (see Chapter 2). But since we don't use that object, and instead return our own `car` object, the initially created object is just discarded. So, `Car()` could be called without the `new` keyword, and the functionality above would be identical, but without the wasted object creation/garbage-collection.

Implicit Mixins

Implicit mixins are closely related to *explicit pseudo-polymorphism* as explained previously. As such, they come with the same caveats and warnings.

Consider this code:

```
    var Something = {
        cool: function() {
            this.greeting = "Hello World";
            this.count = this.count ? this.count + 1 : 1;
        }
    };
```

```
    Something.cool();
    Something.greeting; // "Hello World"
    Something.count; // 1

    var Another = {
        cool: function() {
            // implicit mixin of `Something` to `Another`
            Something.cool.call( this );
        }
    };

    Another.cool();
    Another.greeting; // "Hello World"
    Another.count; // 1 (not shared state with `Something`)
```

With `Something.cool.call( this )`, which can happen either in a "constructor" call (most common) or in a method call (shown here), we essentially "borrow" the function `Something.cool()` and call it in the context of `Another` (via its `this` binding; see Chapter 2) instead of `Something`. The end result is that the assignments that `Something.cool()` makes are applied against the `Another` object rather than the `Something` object.

So, it is said that we "mixed in" `Something`s behavior with (or into) `Another`.

While this sort of technique seems to take useful advantage of `this` rebinding functionality, it is the brittle `Something.cool.call( this )` call, which cannot be made into a relative (and thus more flexible) reference, that you should **heed with caution**. Generally, **avoid such constructs where possible** to keep cleaner and more maintainable code.

Review (TL;DR)

Classes are a design pattern. Many languages provide syntax which enables natural class-oriented software design. JS also has a similar syntax, but it behaves **very differently** from what you're used to with classes in those other languages.

Classes mean copies.

When traditional classes are instantiated, a copy of behavior from class to instance occurs. When classes are inherited, a copy of behavior from parent to child also occurs.

Polymorphism (having different functions at multiple levels of an inheritance chain with the same name) may seem like it implies a referential relative link from child back to parent, but it's still just a result of copy behavior.

JavaScript **does not automatically** create copies (as classes imply) between objects.

The mixin pattern (both explicit and implicit) is often used to *sort of* emulate class copy behavior, but this usually leads to ugly and brittle syntax like explicit pseudo-polymorphism (`OtherObj.methodName.call(this, ...)`), which often results in harder to understand and maintain code.

Explicit mixins are also not exactly the same as class *copy*, since objects (and functions!) only have shared references duplicated, not the objects/functions duplicated themselves. Not paying attention to such nuance is the source of a variety of gotchas.

In general, faking classes in JS often sets more landmines for future coding than solving present *real* problems.