DECEMBER 18, 2020 / #OBJECT ORIENTED PROGRAMMING

The Four Pillars of Object-Oriented Programming

Kealan Parr

JavaScript is a multi-paradigm language and can be written following different programming paradigms. A programming paradigm is essentially a bunch of rules that you follow when writing code, to help you solve a particular problem.

The four pillars of object-oriented programming are:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Let's take a closer look at each of them.

Abstraction in Object-Oriented Programming

To abstract something away means to hide away the implementation details inside something – sometimes a prototype, sometimes a function. So when you call the function you don't have to understand exactly what it is doing.

If you had to understand every single function in a big codebase you would never code anything. It would take months to finish reading through it all.

You can create a reusable, simple to understand, and easily changeable codebase by abstracting away certain details. Let me give you an example:

```
    } else if (type instanceof NavBar) {
        // Implementation example
    } else {
        // Implementation example
    }
}
```

This is not abstracted away at all.

Can you see in the example how you have to implement exactly what you need for your custom use-case?

Every new API you need to hit needs a new `if` block, and it's own custom code. This isn't abstracted away as you need to worry about the implementation for every new type you add. It isn't reusable, and is a maintenance nightmare.

How about something like the below?

```
hitApi('www.kealanparr.com', HTTPMethod.Get)
```

You now can just pass a URL to your function and what HTTP method you want to use and you're done.

You don't have to worry about how the function works. It's dealt with. This massively helps code reuse! And makes your code a lot more maintainable, too.

multiple places, with multiple senses.

Here's a good final example of **Abstraction**: imagine if you were creating a machine to make coffee for your users. There could be two approaches:

How to Create it With Abstraction

- Have a button with the title "Make coffee"

How to Create it Without Abstraction

- Have a button with the title "Boil the water"
- Have a button with the title "Add the cold water to the kettle"
- Have a button with the title "Add 1 spoon of ground coffee to a clean cup"
- Have a button with the title "Clean any dirty cups"
- And all the other buttons

It's a very simple example, but the first approach *abstracts* away the logic into the machine. But the second approach forces the user to understand how to make coffee and essentially make their own.

Encapsulation in Object-Oriented Programming

The definition of encapsulation is "the action of enclosing something in or as if in a capsule". Removing access to parts of your code and making things private is exactly what **Encapsulation** is all about (often times, people refer to it as data hiding).

Encapsulation means that each object in your code should control its own state. State is the current "snapshot" of your object. The keys, the methods on your object, Boolean properties and so on. If you were to reset a Boolean or delete a key from the object, they're all changes to your state.

Limit what pieces of your code can access. Make more things inaccessible, if they aren't needed.

Private properties are achieved in JavaScript by using closures. Here's an example below:

```
var Dog = (function () {

    // Private
    var play = function () {
        // play implementation
    };

    // Private
```

```
    // Public
    var makeNoise = function () {
        return 'Bark bark!';
    };

    return {
        makeNoise: makeNoise,
        name: name
    };
})();
```

The first thing we did was create a function that immediately gets called (called an **Immediately Invoked Function Expression**, or IIFE for short). This created an object that anyone can access but hid away some of the details. You can't call `play` and you can't access `breed` as we didn't expose it in the final object with the return.

This particular pattern above is called the **Revealing Module Pattern**, but it's just an example of how you can achieve **Encapsulation.**

I want to focus more on the idea of **Encapsulation** (as it is more important than just learning one pattern and counting **Encapsulation** as totally complete now).

Reflect, and think more about how you can hide away your data and code, and separate it out. Modularising and having clear responsibilities is key to **Object Orientation**.

coupled to one another via the global variable.

- You will likely override the variables if the name get's reused, which can lead to bugs or unpredictable behaviour.
- You will likely end up with **Spaghetti Code** – code that's hard to reason through and follow what is reading and writing to your variables and changing state.

Encapsulation can be applied by separating out long lines of code into smaller separate functions. Separate out those functions into modules. We hide away the data in a place nothing else needs access to, and cleanly expose what is needed.

That is **Encapsulation** is a nutshell. Binding your data to something, whether it's a class, object, module or function, and doing your best to keep it as private as you reasonably can.

Inheritance in Object-Oriented Programming

Inheritance lets one object acquire the properties and methods of another object. In JavaScript this is done by **Prototypal Inheritance**.

Reusability is the main benefit here. We know sometimes that multiple places need to do the same thing, and they need to do everything the same except for one small part. This is a problem inheritance can solve.

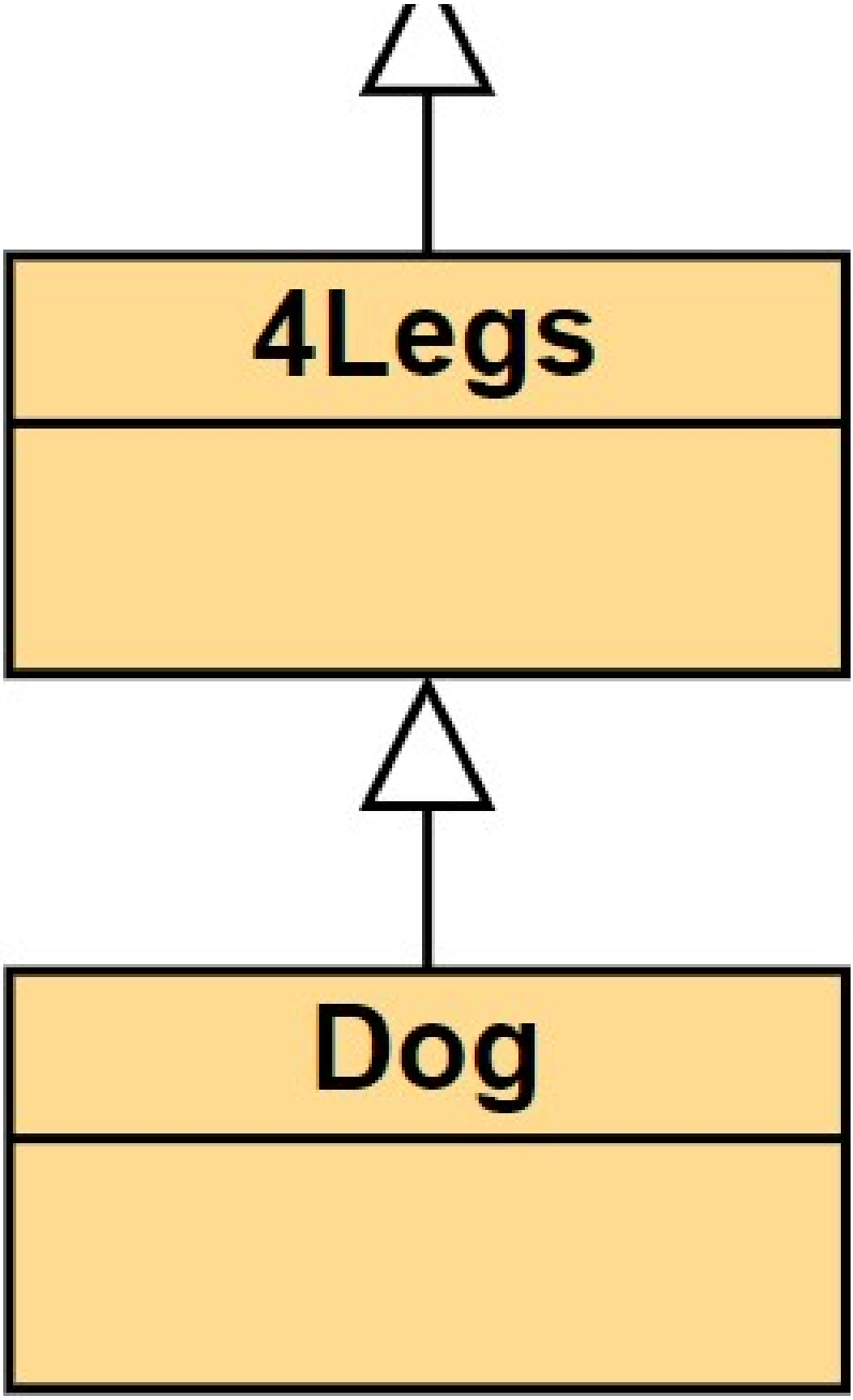
is. For example, does the `Bird` type extend from the `DieselEngine` type?

Keep your inheritance simple to understand and predictable. Don't inherit from somewhere completely unrelated because there's one method or property you need. Inheritance doesn't fix that particular problem well.

When using inheritance, you should require most of the functionality (you don't always need absolutely everything).

Developers have a principle called the **Liskov Substitution principle**. It states that if you can use a parent class (let's call it `ParentType`) anywhere you use a child (let's call it `ChildType`) – and `ChildType` inherits from the `ParentType` – then you pass the test.

The main reason you would fail this test, is if the `ChildType` is removing things from the parent. If `ChildType` removed methods it inherited from the parent, it'd lead to `TypeError`'s where things are undefined that you are expecting not to be.



Inheritance chain is the term used to describe the flow of inheritance from the base object's prototype (the one that everything else inherits from) to the "end" of the inheritance chain (the last type that is inheriting – **Dog** in the above example).

Do your best to keep your inheritance chains clean and sensible. You can easily end up coding an anti-patterns when using **Inheritance** (called the **Fragile base anti-pattern**). This happens where your base prototypes are considered "fragile" because you make a "safe" change to the base object and then start to break all your children.

Polymorphism in Object-Oriented Programming

Polymorphism means "the condition of occurring in several different forms." That's exactly what the fourth and final pillar is concerned with – types in the same inheritance chains being able to do different things.

If you have used inheritance correctly you can now reliably use parents like their children. When two types share an inheritance chain, they can be used interchangeably with no errors or assertions in your code.

From the last diagram, we might have a base prototype that is called `Animal` which defines `makeNoise`. Then every type extending from

```
// Let's set up an Animal and Dog example
function Animal(){}
function Dog(){}

Animal.prototype.makeNoise = function(){
    console.log("Base noise");
};

// Most animals we code up have 4. This can be overridden if needed
Animal.prototype.legs = 4;

Dog.prototype = new Animal();

Dog.prototype.makeNoise = function(){
    console.log("Woof woof");
};

var animal = new Animal();
var dog = new Dog();

animal.makeNoise(); // Base noise
dog.makeNoise();    // Woof woof- this was overridden
dog.legs;           // 4! This was inherited
```

`Dog` extends from `Animal` and can make use of the default `legs` property. But it's also able to do its own implementation of making its own noise.

The real power of polymorphism is sharing behaviours, and allowing custom overrides.

I hope this has explained what the four pillars of object-oriented programming are, and how they lead to cleaner and more robust code.

I share my writing on Twitter if you enjoyed this article and want to see more.

Kealan Parr

Senior Software Engineer working on mobile, web and iOS.
