Programming Concepts!

Object-Oriented Programming

Sometimes abbreviated as OOP or OOD (Object-Oriented Design), this is a paradigm style of programming. This is one of several types of programming paradigms. Some others you might hear of are declarative (this is the style html uses), procedural (e.g. COBOL), functional, and a few others.

OOP takes the view of modeling the problem / real world (sometimes referred to as the domain), and utilizing the language's construct of an object to represent that domain in code. The other main aspect of OOP is that in objects, both behavior and data are co-located (as in both the data of an object and its behavior should all be located on the object itself and not scattered into various other possible locations). This has two main benefits: it can be easier cognitively to both create and maintain code in this fashion, and it promotes more efficient code reuse.

Classes vs Objects

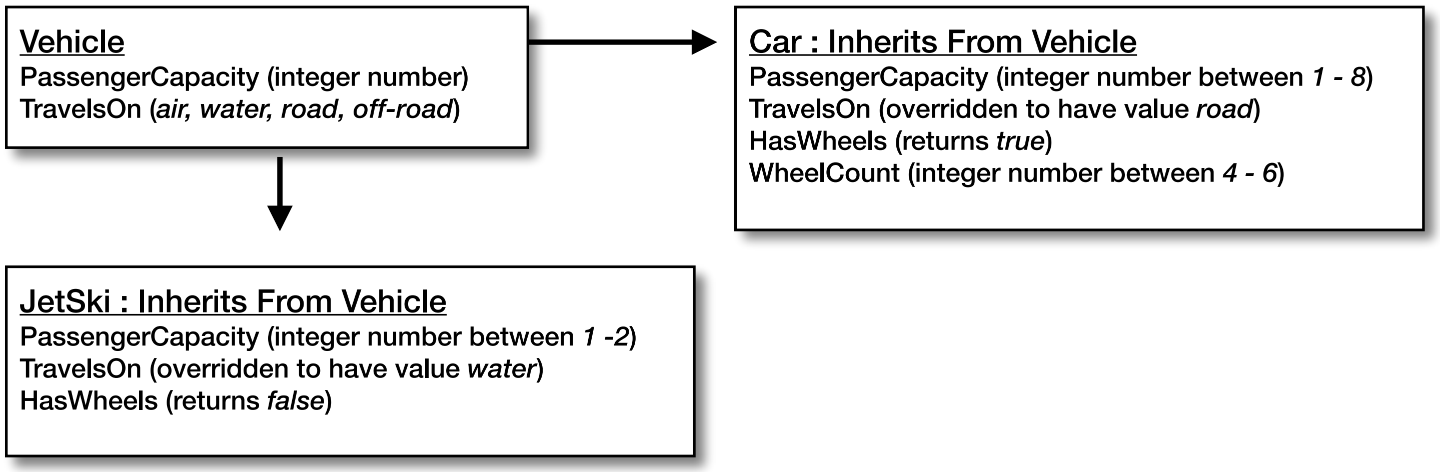
In OOP, it's worthwhile to note that the way we code objects is by writing something called a class. A class is the way in which we can declare the structure of our object. This would include writing our functions and exposing the data points that we want to make public for anyone who will be creating instances of our class. And that last part there, that is the distinction between a class and an object…and object is simply a live instance of a class. So think of a class as like a template, but an object is the instance of that template and this process can be repeated over and over with multiple objects being instantiated over and over from the same class. Or you can think of a class as a cookie-cutter whereas the object is the actual cookie that got made from your cookie-cutter. Sometimes you'll hear of programmers refer to objects as a "black-box" because another concept of OOP is for objects to hide its implementation. This means that an object is like a little machine that if you interact with one piece of it, there might be side-effects on the other pieces of it, and you don't have to care how that happened (nor should you).

For example, let's say you have an object called a MinecraftChest object that has a pair of functions called `Open()` and `Close()` along with a boolean property called `IsOpen`. When you call the function, the `IsOpen` boolean will return `true` whereas after you call the `Close()` function, the same `IsOpen` boolean will return `false`. The implementation of how this happened was done and we don't have to care how this happened, it just did and so we can write code that relies upon this behavior.

Example - Make Me A Transportation App!

So let's take the example of being tasked with making an app that will be used to facilitate transportation management (e.g. I need to keep track of all manner of vehicles, their maintenance statuses, take them in and out of service, etc.) We won't concern ourselves with all of the bootstrapping of all the details of this app, we'll just focus on some snippets of it regarding the kinds of objects we might need to create.

Let's say that this app needs to manage cars, motorcycles, and box trucks, but the managers say that they might also need to manage bicycles and other vehicle types. One of the features of languages that support OOP (like JavaScript) is called inheritance. Let's take this information and explore one way we might design this application. Consider the following illustration:


Vehicle
PassengerCapacity (integer number)
TravelsOn (*air, water, road, off-road*)
Car : Inherits From Vehicle
PassengerCapacity (integer number between *1 - 8*)
TravelsOn (overridden to have value *road*)
HasWheels (returns *true*)
WheelCount (integer number between *4 - 6*)
JetSki : Inherits From Vehicle
PassengerCapacity (integer number between *1 -2*)
TravelsOn (overridden to have value *water*)
HasWheels (returns *false*)

So this might be what it looks like in terms of code

```
class Vehicle {
    passengerCapacity = 1;
    get travelsOn() { return 'road'; }
    constructor(passengerCapacity) {
        this.passengerCapacity = passengerCapacity;
    }
}

class Car extends Vehicle {
    get hasWheels() { return true; }
    wheelCount = 2;
    constructor(passengerCapacity, wheelCount) {
        super(passengerCapacity);
        this.wheelCount = wheelCount;
    }
}

class JetSki extends Vehicle {
    get hasWheels() { return false; }
    get travelsOn() { return 'water'; }
    constructor(passengerCapacity) {
        super(passengerCapacity);
    }
}
const myCar = new Car(7, 4);
const myJetSki = new JetSki(4);
```