Object Oriented Programming Principles

Sam Huffer, 27 September 2017

Hang on, what's object oriented programming?

Well, it's programming focused around objects. Yes, that tells you bugger all about any of it, but if you keep reading, you'll find out.

Many people, myself included, get introduced to programming through procedural programming, where code is split into chunks called procedures and functions. The program will have various variables scattered throughout it as needed, to hold all the data that the program will use. Object oriented programming (or OOP) works in much the same way, but with a few bonuses that make it conceptually less abstract than blocks of badly arranged code floating in the middle of nowhere. These bonuses are objects.

Objects, objects, and more objects

In OOP, an object is a single thing that both knows things and can do things, encapsulating data and functionality in one entity. It has fields that sit there and act all smug and knowledgeable, holding an object's data. And it has methods, its procedures and functions that you go to when you need stuff done for you. For example, a programmer (hopefully) knows various programming techniques, and has the arms and fingers to type everything up and tell the computer how to make the program work.

Not all of an object is visible from the outside though. Some fields and methods are private and accessible only by the object that has them, while some are public and are available to everyone. This is comparable to how you'd usually give out your name to anyone who asked, but if they asked for your credit card number, you'd look at them like they were crazy, and ignore them. Part of an object encapsulating data and functionality is splitting them between what should be publicly available, and what should be its deepest, darkest secrets, hidden down inside it! Though maybe not that extreme…

Why so abstract?

But first, we've got some thinking to do. What, did you think you could just snap your fingers and make objects out of thin air? (Yes, we all wish we could do that, it'd make life so much easier…) The first thing when dealing with any programming problem is to identify the roles that will need to be taken care of, the responsibilities that each will need to manage, and how parts of the program will need to collaborate and work together.

For example, take the Muppets. You need the Muppets themselves, the puppeteers (or muppeteers?), the voice actors, etc. (or is that all politics? Eh, same difference.) These are some of the roles that need to be managed. The Muppets need to be movable, the puppeteers have the responsibility of manipulating said Muppets and knowing how they ought to be manipulated for a particular performance, and the voice actors (yes, Muppets don't actually speak; that's also how ventriloquism works, genius) will be responsible for knowing their lines and providing the Muppets' voices; without them, you wouldn't hear Kermit calling out "It's the Muppet Show!" every time you tuned in.

This process of abstraction involves brainstorming all of these roles and responsibilities for a particular program, and mapping them to classes that describe each role, and the fields and methods that will take responsibility for knowing and doing things. These classes will then be used as the blueprints for different objects when implemented. Abstraction also involves deciding how different objects will interact and collaborate, how cohesive (small and focused, rather than being large,

sprawling, and managing everything like a control freak) a class will be, and the degree of coupling (or interdependence) there will be between one object and another.

It's all in the DNA

Usually when you devise a class of objects, they'll do things their own way and won't take advice from anyone else. A lot of the time, however, you'll have objects that are similar and do the same sorts of things, but which will be different for other purposes. This is where OOP's use of inheritance comes in useful. With inheritance, a class can take a parent class, adopt the fields and methods it inherits from it as its basis, and expand upon them and modify them, becoming a more specialised child of the original class.

Sometimes you will want a class of objects to inherit not just from a parent class. In the real world, you might inherit your height and hair colour from your parents, but you might choose to adopt a few quirks or views held by your friends. As opposed to the parent class providing you with the former attributes, the latter could be provided to a class by an interface, a non-class entity that can be inherited from, that takes the role of the quirky friend in providing the basis for the quirks and knowledge that are inherited from outside of the parent class.

Most of the time, parents and children will have disagreements about preferences or how they like things done (e.g. the parents might like cooking a meal a certain way, while the child might prefer to add a few extra spices, or do the pasta differently). The same principle applies to parent and child classes. Many of the methods that a child class inherits, they'll just leave them as is. But often they will want to override this one or that, and do it a different way to suit their own style. Many parent classes will be designed to allow for such adaptation.

This ability to override or fill in inherited methods as necessary gives rise to one of the strengths of OOP: polymorphism. The same inherited method can be overridden by different sibling classes in different ways, similar to the differences in drawing a piece of artwork. One person might go about it by drawing cartoony stick figures, while another might go all out with detail, shading, etc. This capacity for sibling classes to do the same thing but in different ways allows for greater flexibility with how objects are designed, and gives designers the ability to make classes distinct but still fairly interchangeable.

Assembling a rover

To reinforce your concepts of abstraction, encapsulation, inheritance, and polymorphism, let's apply them all, in order, to a single example. One of the programming tasks I've had to deal with is a rover simulator, where the user will be wandering around a planet collecting specimens for scientists back home.

The first part of this task involved going through the process of abstracting all the roles that objects would need to fill (e.g. motors, drills, radars, batteries, etc.), determining the things that each object would be responsible for knowing and doing (e.g. moving, drilling, scanning, providing power, etc.) and outlining them in a diagram. Once done, these plans were taken and used to code the different objects, encapsulating in each the functionality and knowledge that they would need to do their jobs, and making some parts available to all, and others hidden from unwelcome Nosey Parkers.

Part of the abstraction and coding processes involved determining which classes would inherit from others. For this program, the drill, radar, motor, and solar panel would all inherit from the device class, gaining some fields and methods from it. In hindsight, the device class could have necessitated another feature or two from its child classes in order to maximise their interchangeability and the elegance of their use. However, each device was still sufficiently steeped in polymorphism to allow acceptable interchangeability and avoid the program resembling a mismanaged economy.

That is what object oriented programming is

Now you know the basics behind object oriented programs. To recap, the programmer will first go through a process of abstraction, thinking of all the different pieces that a program will need. Then they'll use these plans to code objects that encapsulate information and functionality in a single thing. Said objects might inherit parts of themselves from parent objects, and different siblings might override a particular part of the parent object in different ways, making them polymorphous. Now go and juggle some objects in a program (maybe literally juggling balls in a juggling program, perhaps?)