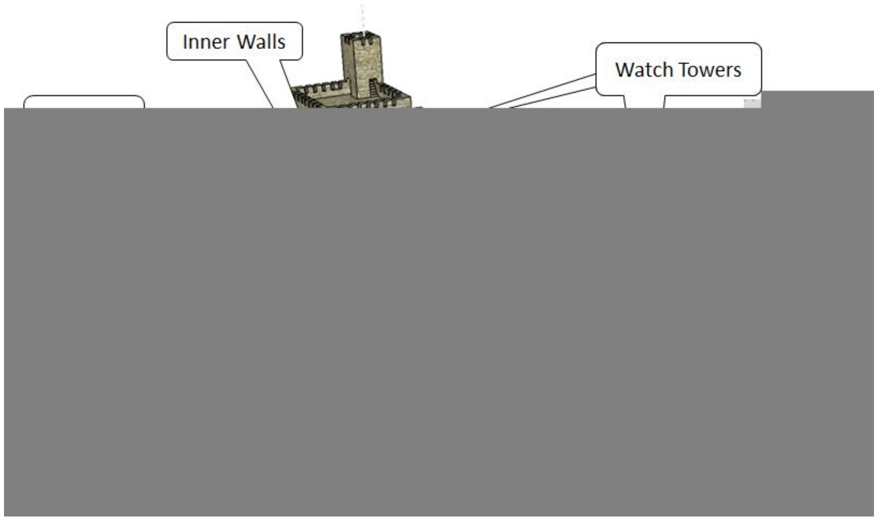
[0540] The Pillars of OOP

Encapsulation, Inheritance, Polymorphism. Encapsulation, Inheritance, Polymorphism. You'll hear those words over and over again when studying OOP. And while they sound fancy, they are really just describing very natural things that programmers need to do.

Encapsulation

We've already talked about this one. I love it. It is very commonly used. It provides instant "no-brainer" benefits. The only downside is that it sometimes gets monotonous to code (as we will see). When you hear the term "Encapsulation" think POJO+Getters+Setters. And think of this picture:



[redacted]

Encapsulated data is like gold treasure in a castle keep. You have to get by the guards at the entrances (the Getters and Setters) in order to see it (or take it!).

Polymorphism

Polymorphism is the ability to call the same method on a collection of related (but different) objects and have that method do different things depending on which object you made the call against. The easiest example of this that you are already familiar with is the ToString() method. All C# objects have the ToString() method because they are all subclasses of the Object method. Most data types override the default ToString() method (which isn't really that helpful) and implement their own (which uses specific knowledge of the class to generate more meaningful strings). That's polymorphism in action.

Inheritance

Like GOTO before it, Inheritance has fallen out of favor with the programming community because it can easily lead to code that is very hard to debug, maintain and incrementally improve. Before we can understand why, we need to be clear on what inheritance is and what it isn't.

Inheritance is an OOP feature that allows you to create a hierarchy of related classes that share common characteristics. Inheritance lets you take a general class (for example a Shape) and use it as the starting point for creating a more specific class (say a Square class or a Circle class) that automatically has all of the characteristics of the general class plus whatever additional items turn it into the more specific thing.

The starting class (Shape) is called the "base class" (or parent class) and the more specific class (Circle) is called the "subclass." When done correctly, the subclass has an "is-a" relationship with the base class. For example, a Circle *is a* Shape.

Inheritance offers the promise of massive amounts of code reuse. The more functionality you add to your base classes, the more reusable features your subclasses automatically have access to. That's VERY COOL!

The problem comes from the fact that if you build lots of code that assumes a certain, specific class hierarchy, you can't really change that hierarchy without breaking much of the code that depends upon it. Essentially, inheritance hierarchies are very "brittle" meaning that the slightest change can have a huge impact.

While there can be cases where inheritance hierarchies work, those cases are usually extremely general or very, very strictly-defined. The fact that every class in C# is a subclass of the Object class is an example of inheritance, but it is a very limited example because the methods implemented by Object (such as ToString()) are very general purpose. On the other end of the spectrum would be an inheritance hierarchy that you created just for your own use using a subclassing scheme that you fully control (for example the Shape/Circle hierarchy). Unfortunately, most real-world programs live in the middle of that spectrum where the hierarchy is not fully defined and evolves over time as more aspects of the program are discovered. The benefits of Inheritance often turn into huge hassles in such environments.

As an example, imagine that you created a fully functional Shape class hierarchy based on 2-dimensional shapes like Square, Circle, etc. and you used those classes in lots of programs. Now imagine what would happen if you suddenly needed to convert the Shape class to 3-dimensions. Would all of your previous programs break? Unless you were super-careful, the answer is "Probably." And the Circle/Shape stuff is very simple compared to many of the hierarchies found in the real world.

Composition - the "New Kid" on the Block

Fortunately, there is a different way to create a related hierarchy of classes that is more flexible than an inheritance hierarchy. Instead of using inheritance syntax, programmers add a member variable of the "subclass's" type to the "base" class. This technique - called "composition" - provides most of the benefits on inheritance but in a much more flexible way as we will see later in the class.

It is now considered to be a "Best Practice" to **"favor composition over inheritance."**