[0502] POJOs (aka POCOs)

What's a POJO? Why isn't this in our book?

POJO stands for "Plain Old Java Object" - an expression that refers to Objects whose primary role is the safe storage and easy retrieval of data. In the .NET world, these things sometimes called POCOs - "Plain Old CLR Objects" - the concept is the same. Due to sheer habit, I will probably call these POJO's in this class. Don't let the word "Java" confuse you.

POJOs are all about the first pillar of Object-Oriented Programming: Encapsulation.

Encapsulation is the process of taking one or more (usually related) data fields and saving them inside of an object and then protecting those fields so that they can only be read and/or changed by authorized code. If the code that does the protecting of the data is very simple and straightforward, then the object is referred to as a POJO / POCO.

For example, let's say that you need to create a program that sorts jellybeans. Each jellybean has 2 characteristics that we care about: 1.) color and 2.) size. Those two characteristics become "fields" in our new POJO. (They are also referred to as "member variables" and/or "properties" depending on context.)

So, initially, our proto-POJO looks like this:

```
class JellyBean
{
   Color beanColor;
   int size;
};
```

Seems straightforward enough. However, that class that is essentially the same as the old structs that C and Pascal had. It will store the data we need stored, but the data is not protected from invalid modification. The first problem is that the fields are not explicitly protected from direct access by other programs. In order to do that, the fields need the "private" modifier.

Side Note: Despite what your textbook shows you, fields inside of classes should NEVER be made public. Never, ever, ever, ever, ever! If I see a public member variable in any part of an assignment, I will deduct a significant number of points. Public fields are the sign of a very lazy programmer who doesn't understand the benefits of encapsulation or how to write maintainable code.

There's another problem that our proto-POJO has. There's nothing preventing the size field from being set to an invalid value. Assuming that Color is an enum, it will probably always be valid however we need to find a way to ensure that size is also always valid.

Clever programmers might try making size a uint instead of an int. While that would eliminate the negative values, we'd still have to deal with zero. And what if we want to add an upper limit to size?

What if we decide that a 10-pound jellybean is the largest confection we are able to handle? uint's are not the full solution that we need. Encapsulation is.

Here's a better version of the JellyBean POJO that starts to use encapsulation concepts:

```
class JellyBean
{
   private Color beanColor;
   private int size;

   public Color GetBeanColor()
   {
      return beanColor;
   }

   public int GetSize()
   {
      return size;
   }

   public void SetSize(int size)
   {
      if (size > 0 && size < 1000)
         this.size = size;
   }

}
```

First, notice that both the color and size fields are now marked "private." This prevents other code from accessing them directly. Always mark your POJO's member variables as private - that's the whole point of a POJO!

By marking the fields as private, we are "locking the data gates" for our object. Now, we need to re-open the gates, but in a controlled, validated manner. That is what the methods in our POJO do.

Getters and Setters

The methods in a old-school POJO often start with either "Get_____" or "Set_____" followed by the field's name. These methods are often called "Getters and Setters." (The book calls them Accessors and Mutators. I don't.) Getters and Setters function as "bodyguards" for the private data fields that they correspond to. For example, a getter might first check to see if a program's user is authorized to see its data before returning it. And just like I'm showing in this simple example, a setter often will make sure the new data is valid before saving it in the underlying field.

Note: As I show in my example, it is very common for a Setter's parameter to have the same name as the underlying field. When that happens, you have to add "this," to the front of the field's name to distinguish it from the Setter's parameter.

A POJO does not need (and usually should not have) Getters and Setters for every field it contains. A good example of this is the concept of immutable POJOs. (We talked about immutability before: **[0401] Why Immutability is A Good Thing (https://lwtech.instructure.com/courses/1841516/pages/0401-why-**

immutability-is-a-good-thing)) To make a POJO immutable, you simply don't define any Setters. Without setters, the POJO can only contain the value it gets when it is first created - which is exactly what immutability means!

Occasionally, POJOs can have Setters that are more tailored to the way that an object works in the real world and thus have a different name. For instance, a Person object might have a "setter" called "HaveBirthday()" that increments the Person's age by 1. This would ensure that a Person's age cannot be radically altered with a completely new value.