Alma Sotelo-Rodriguez

Explain Programming with Classes:

Firstly, the OOP model is based in the concept of Objects created from templates called Classes. This programing approach has many more advantages than the traditional procedural or functional programing technics. The OOP implements the following features:

**Abstraction**

In order to reduce complexity and increase the efficiency, some no relevant code for the specific task, is hidden by separating the code in classes. Each class contains only what is related to that class (data and responsibilities), inclusive, it may contain another class.

By doing this, the programmer can more easily debug, modify or maintain the whole program. However, it is important do not obsessively create classes, because that can lead to a very disjointed program that might result in a disadvantage instead.

I used Abstraction in the Journal program. Here I had classes for example the Entry class, where I included only code for the Entry, such the date, the prompt, the message and the display. This template of an Entry was used many times until the user had no more entries to enter. I later modified the Journal program to accommodate the Exceed Core by focusing just on the code inside Main and coded additional features for the program. Easy read and handle!

**Encapsulation**

When implementing Encapsulation, we are also using abstraction; organizing the code in the way we separate related data and tasks by functionality where each class can communicate each other, but at the same time protect its own members and procedures, by setting how its code is accessed. In Encapsulation, we usually use contractors with parameters to indirectly initialize objects.

With Encapsulation, the programmers can protect data from unwanted access or modification, the grouped tasks can be re-used, which makes the program more efficient and easier to understand and modify.

I used Encapsulation in the ScriptureMemorizer program; kept member variables private inside its classes, and used public methods of the class instead to export what was needed outside. For example, the status of whether the word was hidden or not. I later escalated this program up to make it randomly reverse the status of words to Show() in the case the user enter “Help”. For that purpose, I just added another method very similar to the existing HideRandomWords and reversed its functionality. Pretty easy! Because the method was in an encapsulated structure where its changes did not affect the code of other parts. Organized and secure!

**Inheritance**

With Inheritance we can use whatever is contained in one another existing class to re-use the code. This gives a lot of flexibility to the programmers. With proper access protocols, we can access variables and methods outside the class, we can re-use code allowing us reduce lines of code, and with this we can easier debug, understand, modify the code.

I used Inheritance in the Homework program. Assignment was the “parent” class because Math and Writing were also Assignments. Math and Writing inherited the what was inside Assignment which was the GetSummary(), in addition to have access to the student’s name and the topic, needed all those to accomplish the display required from Math and Writing. We could easily come later and add another assignment, for instance, History that inherits Assignment class, without the need to modify what is already coded. Very much code saver!

**Polymorphism**

With Polymorphism we can transform the behavior of the objects! based in the existing context at the runtime. We use override to specify our own method implementation and thus changes the object behavior. For example, if I have an object of certain type, let say, mySimpleGoal of SimpleGoal type, I can treat it as myEternalGoal object (of EternalGoal type), as if they were the same type. After implementing inside each SimpleGoal and EternalGoal classes, the method inherited from their base that don’t work here, I can have both objects to respond differently to the same call RecordEvent(). This makes the code extremely flexible and adaptable because the same operation can be done in different ways by different classes. I can later add another class, like IntelectualGoal, inheriting the Goal class, then override the methos that do not work for this new class, and implement them with specific code. My newIntelectualGoal object can be throw it to the game as if it were one of the other objects. Super flexible!