I –Enumerations and Composition

[DRAFT - ~95% Complete]

# LOG Review

These are some very brief points regarding the concepts and skills listed in the LOGs for this topic.

**General Programming Concepts and Terms**

* **Define the term Enumeration**
  + An enumeration is a simple program-defined data type with a limited set of possible values. Enumerations represent information that is “conceptual” in nature (as opposed to information that is textual, numeric, or complex).  
    For example, a coin has two sides that are commonly referred to as HEADS and TAILS. Rather than creating two constant strings with the values “Heads” and “Tails” and a String variable called FaceShowing, you can use an enumeration.
  + There are two parts to enumerations:
    - The name of the enumeration, which is the name of the new data type; this is sometimes referred to as the “enumerated data type”.
    - A set of enumerated values, which represent all of the possible values for a variable of that enumerated type. These values act like “constants” that can be used to distinguish different “states” for the conceptual data.
* **List the benefits of using enumerations**
  + ***Type Safety*** – Enumerations allow a program to be “type safe”, meaning that there is less likelihood of creating logical errors or other mistakes through the use of constants or literal values.
  + ***Code Readability*** – Enumerations make code more readable than constants or literal values (like integers).
  + ***Compatibility with Switch Statements*** – Enumerations are compatible with switch statements, making them a great option for describing distinct values for matching against different cases. Because enumerations are like constants, and are treated as integral data types by the compiler, they are often the best choice when working with case structures.
  + ***Useful as Flags*** – Because the programmer can associate specific integer values to the enumerated values, it is possible to use an enumerated data type to create complex “flags” for representing distinct “states” that can be combined together.  
    For example, if you had an enumerated type such as the following, you could combine two states together in a variable’s value to represent distinct but compatible states, such as ACTIVE and MEDIUM\_PRIORITY.  
    enum Status  
    {  
     INACTIVE = 0  
     ACTIVE = 1  
     LOW\_PRIORITY = 2  
     MEDIUM\_PRIORITY = 4  
     HIGH\_PRIORITY = 8  
    }
* **Describe where and when enumerations are used**
  + Enumerations are used wherever a clearly defined set of values can be “classified” and used to represent conceptual information.
  + Enumerations should be used whenever you might be inclined to use a set of two or more constants to represent distinct “states” or values for comparing against a variable.  
      
    For example, if you want to represent the two sides of a coin, it is better to use an enumeration for the coin’s HEADS and TAILS sides, rather than declaring some String or integer constants. The problem with using strings or simple integers to represent possible coin faces is that any variable that is defined as a string or integer can have values that won’t match either of the declared constants, as in this example:  
      
    const string Heads = “Heads”;  
    const string Tails = “Tails”;  
    // There is nothing preventing “invalid” values for the following coin face...  
    string theFaceShowingOnTheCoin = “Sailboat”;  
      
    An enumerated type called CoinFace with the values HEADS and TAILS is better, because any variable declared of type CoinFace can only have the values defined in the enumeration.  
      
    // The only other possible value that could be assigned is  
    // CoinFace.TAILS  
    CoinFace theFaceShowingOnTheCoin = CoinFace.HEADS;
* **Compare and contrast enumerations and classes**
  + Enumerations and classes both define new data types that can be used by our program to represent custom data. However, while classes are useful in describing *complex* data types (that is, data types with assorted fields and that can perform various tasks), enumerations are only used to create new data types that are *simple* in nature. Enumerations only define a set of possible values, and each variable defined by an enumeration can only hold one of those possible values at any given time; there is no inner complexity to enumerations, as there is with classes.
* **Use enumerations to define simple data types that are conceptual in nature and that have a limited set of possible “values”**
  + *Demonstrate your ability to meet this learning outcome guide by completing the related assignments.*

**OOP Basics**

* **Describe the idea of Composition**
  + Composition refers to the situation where one or more of the fields in a class are objects. All classes are capable of having objects as fields, rather than just using the simple primitive types that come with the language.  
    For example, if a class called Address represents some mailing address, another class, such as Student, Company, School, or Business, could have a field whose data type is Address.
* **List the benefits of Composition**
  + Composition is an aspect of modern programming languages that makes it possible to create new data types (classes) of ever increasing complexity.
  + Composition also makes it possible to get better code re-use, because classes can be more distinct, rather than overlapping. For example, both Students and Employees can have Addresses, so it is not necessary to code individual fields in the Student and Employee class for each part of an address; rather, a common type (called Address) can be shared by both classes.  
    public class Employee {  
     // ... other fields, followed by these Address fields:  
     public string street { get; set; }  
     public string city { get; set; }  
     public string province { get; set; }  
     // ... etc.  
    }  
      
    public class Student {  
     // ... other fields, followed by these Address fields:  
     public string street { get; set; }  
     public string city { get; set; }  
     public string province { get; set; }  
     // ... etc.  
    }  
      
    These could be coded like this:  
    public class Employee {  
     // ... other fields, followed by these Address fields:  
     public Address mailingAddress { get; set; }  
    }  
      
    class Student {  
     // ... other fields, followed by these Address fields:  
     public Address homeAddress { get; set; }  
    }
  + [advanced benefit] - Composition promotes the “Has-A” approach to designing objects rather than the “Is-A” design approach that makes use of inheritance. This “Has-A” approach gives programmers more flexibility in their coding and promotes the use of Design Patterns.