UNDERSTANDMENTS & CONCLUSIONS

NANDINI BHATTACHARYA

MODULE – 4

20. Enumerations

- An **enum** (or enumeration type) is used to assign constant names to a group of numeric integer values. It makes constant values more readable.
- An enum is defined using the enum keyword, directly inside a namespace, class, or structure. All the constant names can be declared inside the curly brackets and separated by a comma.

```
enum Enum_variable
{
        string_1...,
        string_2...,
        .
        .
}
```

- If values are not assigned to enum members, then the compiler will assign integer values to each member starting with zero by default. The first member of an enum will be 0, and the value of each successive enum member is increased by 1.
- You can assign different values to enum member. A change in the default value of an enum member will automatically assign incremental values to the other members sequentially.

```
enum Enum_variable
{
        string_1...; //0
        string_2...=6, //6
        string_3..., //7
        string_4...,  //8
        .
}
```

- The enum can be of any numeric data type such as byte, short, ushort, int, long, or ulong. However, an enum cannot be a string type. Specify the type after enum name as :type
- Explicit casting is required to convert from an enum type to its underlying integral type. E.g. enum to int

21. Handling Exceptions

- Exceptions are the unexpected events occurring during the execution of program. Their response to be given is not known by program. They are handled by exception objects which call the exception handler code. The execution of exception handler is exception handling.
- **try** - Used to define a try block. This block holds the code that may throw an exception.
- **catch** - Used to define a catch block. This block catches the exception thrown by the try block.
- **finally** - Used to define the finally block. This block holds the default code.
- **throw** - Used to throw an exception manually.

- C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the System.Exception class.
- Some of the exception classes derived from the System.Exception class are the System.ApplicationException and System.SystemException classes.
- The System.ApplicationException class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.
- The System.SystemException class is the base class for all predefined system exception.

Exception Class	Description
ArgumentException	Raised when a non-null argument that is passed to a method is invalid.
ArgumentNullException	Raised when null argument is passed to a method.
ArgumentOutOfRangeException	Raised when the value of anargument is outside the range of valid values.
DivideByZeroException	Raised when an integer value is divide by zero.
FileNotFoundException	Raised when a physical file does not exist at the specified location.
FormatException	Raised when a value is not in an appropriate format to be converted from a string by a conversion method such as Parse.
IndexOutOfRangeException	Raised when an array index is outside the lower or upper bounds of an array or collection.
InvalidOperationException	Raised when a method call isinvalidin an object's current state.
KeyNotFoundException	Raised when the specified key for accessing a member in a collection is not exists.
NotSupportedException	Raised when a method or operation is not supported.
NullReferenceException	Raised when program access members of null object.
OverflowException	Raised when an arithmetic,casting, or conversion operation results in an overflow.
OutOfMemoryException	Raised when a program does notget enough memory to execute the code.
StackOverflowException	Raised when a stack in memory overflows.
TimeoutException	The time interval allotted to an operation has expired.

23. Basic file operations

- File Handling refers to the various operations like creating the file, reading from the file, writing to the file, appending the file, etc.
- There are two basic operations which are mostly used in file handling: reading and writing of the file.
- In C#, System.IO namespace contains classes which handle input and output streams and provide information about file and directory structure

23.1 Write into a file StreamWriter Class - The StreamWriter class implements TextWriter for writing character to stream in a particular format. The class contains the following method which is mostly used.

Methods	Description
Close	Closes the current StreamWriter object and the underlying stream
Flush	Clears all buffers for the current writer and causes any buffered data to be written to the underlying stream
Write	Writes to the stream
WriteLine	Writes data specified by the overloaded parameters, followed by end of line

23.2 Read into a file StreamReader Class - The StreamReader class implements TextReader for reading character from the stream in a particular format. The class contains the following methods which are mostly used.

Methods	Description
Close	Closes the object of StreamReader class and the underlying stream, and release any system resources associated with the reader
Peek	Returns the next available character but doesn't consume it
Read	Reads the next character or the next set of characters from the stream
ReadLine	Reads a line of characters from the current stream and returns data as a string
Seek	Allows the read/write position to be moved to any position with the file

Method	Usage
AppendText	Creates a StreamWriter that appends UTF-8 encoded text to an existing file, or to a new file if the specified file does not exist.
Copy	Copies an existing file to a new file. Overwriting a file of the same name is not allowed.
Create	Creates or overwrites a file in the specified path.
Exists	Determines whether the specified file exists.
Move	Moves a specified file to a new location, providing the option to specify a new file name.
Delete	Delete the specified file.
Open	Opens a FileStream on the specified path with read/write access.
ReadAllLines	Opens a text file, reads all lines of the file, and then closes the file.
ReadAllText	Opens a text file, reads all lines of the file, and then closes the file.
WriteAllText	Creates a new file, writes the specified string to the file, and then closes

the file. If the target file already exists, it is overwritten.

24. Interface & inheritance

24.1 Interfaces

- An interface looks like a class, but has no implementation. The only thing it contains are declarations of events, indexers, methods and/or properties. The reason interfaces only provide declarations is because they are inherited by structs and classes,that must provide an implementation for each interface member declared.
- The entities that implement the interface must provide the implementation of declared functionalities.
- In C#, an interface can be defined using the **interface** keyword.
- You cannot apply access modifiers to interface members. All the members are public by default. If you use an access modifier in an interface, then the C# compiler will give a compile-time error "The modifier 'public/private/protected' is not valid for this item."
- Interfaces are the another way of abstraction, in which they contain only abstract methods and properties. They show “what” should be implemented and the derived classes show “how” it should be implemented.
- Interface cannot contain fields because they represent a particular implementation of data.
- Multiple inheritance is possible with the help of Interfaces.

Syntax of Interface Declaration –

```
interface <interface_name >

{

 // declare Events

 // declare indexers

 // declare methods

 // declare properties

}
```

Syntax of Implementing Interface –

```
<class class_name> : <interface_name>
```

24.2 Inheritance

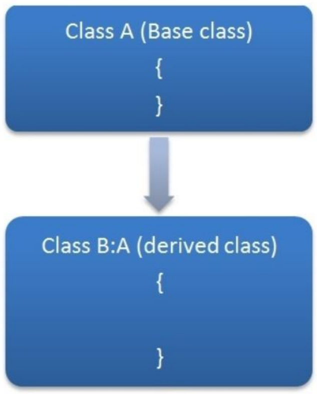
- Acquiring (taking) the properties of one class into another class is called **inheritance**. Inheritance provides reusability by allowing us to extend an existing class.
- The reason behind OOP programming is to promote the reusability of code and to reduce complexity in code and it is possible by using inheritance.
- The inheritance concept is based on a base class and derived class.
- **Base class/Super class** - is the class from which features are to be inherited into another class.
- **Derived class/ Subclass** - it is the class in which the base class features are inherited.

Syntax –

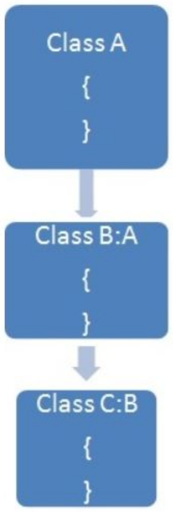
```
<acess-specifier> class <base_class> {
 ...
}
class <derived_class> : <base_class> {
 ...
```

}

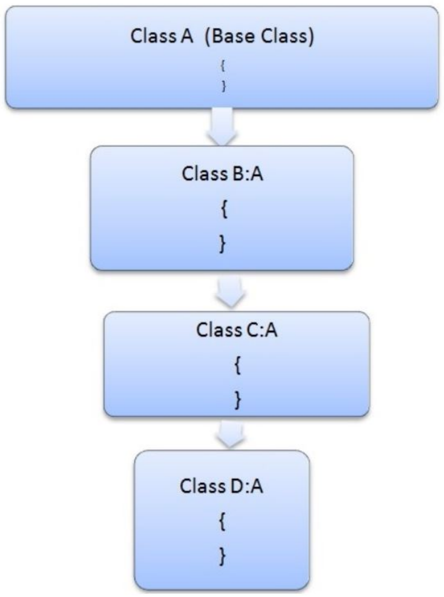
- **Classification of Inheritance –**
  Implementation Inheritance – This is an inheritance when a class is derived from another class.
  Interface Inheritance – This is an inheritance when a class is derived from an interface.
- **Single Inheritance** - In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B



- **Multilevel Inheritance**: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



- **Hierarchical Inheritance**: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In below image, class A serves as a base class for the derived class B, C, and D.


Class A (Base Class)
{
}
Class B:A
{
}
Class C:A
{
}
Class D:A
{
}