**//---------------------------------------------------------- Classes -----------------------------------------------------------//**

/\* A Software models real world things (goods, cars, spheres) or abstract concepts like vectors, list, stacks etc as Objects.\*/  
// All those real world or abstract Objects have two distinct characteristics - State and Behaviour  
// State - characteristics which define the condition of object at a given moment / general time  
// State - represented by values of data members / fields / member variables  
// Behaviour - specific distinctive actions that can be done by the objects   
// Behaviour – modelled or implemented by function members/ methods  
// Class - User defined data type with template for field data & members which operate on this data(contructors,properties,methods,events,etc)  
// The data members of class - field and properties are collectively called Attributes.  
/\* Properties - special elements which control access to fields, extend the functionality of the fields by giving the ability of extra data management when extracting and recording it in the class fields (using get , set). In other words, properties are the way to describe the **characteristics** of a given class. Usually, the value of the characteristics is kept in the fields of the object. Similar to the fields, the properties may be held by certain object or to be shared among the rest of the objects. \*/   
// Objects - Variables of this user-defined data type/class; called Instances of the class; object creation: Instantiation  
// The values in the fields, reflect the specific state of the given object; other types of fields, called **static**, are shared among all the objects.  
// The state is specific to instance/object but behaviour is common to all instances of the class.  
// **Constructor** – used for **initializing & creating new objects**.

class EmptyMan // class - keyword, followed by valid C# identifier  
{  
 // State, Behaviour are all optional  
} // Empty class has many uses: Marker interfaces, Type parameters in generics, Singleton pattern, Placeholder Code, etc

public partial class Molecules // Syntax: Access-modifier Type-modifier class ClassName  
{  
 public int radius; // Instance Variable: All objects will have their own copy/different value of this field.  
 public float velocity;  
 public float charge;  
 public string name;  
 protected double VolumeConstant = 4 \* Math.PI / 3; //   
 public void PrintState()  
 {  
 Console.WriteLine("This {0} molecule is moving with a speed {1}",name,velocity);  
 }  
 public void RadiusChange(int delta) => radius += delta;  
}

Properties  
// **Properties** : way to **encapsulate** fields (or class’ data) in a class by providing controlled access to them (expose data of a class to outside while keeping control) by defining how values are accessed and modified (how data is read, written, or modified/ compute the value of a private field). Helps in maintaining the integrity of the data. Properties provide a cleaner and more intuitive syntax than traditional getter and setter methods.  
public class Patient  
{   
 private string name; // Private field; cannot be accessed/ modified directly from outside ; sensitive information   
 public string Name // Property Name Is declared (public) which can now be used to access, modify the private field name  
 {  
 get { return name; } // Using get *accessor* to return value in private field name  
 set // set *accessor* used to assign value; may include additional logic, like validation, formatting, events etc  
 {   
 if (!string.IsNullOrEmpty(**value**)) // Customizing set method using a check/validation  
 {  
 name = **value**; // **value** is a keyword  
 }  
 }   
 }  
 }  
Patient patient = new Patient();  
patient.Name = “Harry”; // Setting the value via property, since name field is private – **value** of set method takes “Harry” as argument  
string data = patient.Name; // Calls get

// The **get{}** and **set{}** in a property is not a class in itself, but rather a special kind of method (more like helper methods) known as an *accessor* – meant to provide controlled access to the field. The accessors are tightly coupled to the property they belong to. They don’t exist independently and can’t be accessed or instantiated separately. They only make sense within the context of the property.  
// **get{}** accessor is like a method that retrieves the value of the property - doesn’t take any parameters and returns the type of the property.  
// **set{}** accessor is like a method that assigns a value to the property- has an *implicit parameter* called **value**, which represents the value being assigned.  
// The *get and set methods are called implicitly* when you access or assign a value to the property. The C# syntax for properties omits the () to make property access look cleaner and more like accessing a field, even though methods are being called behind the scenes. By omitting () after get and set, properties look like fields when accessed, even though they are backed by methods. (Syntactic Sugar)  
// **Behind the Scenes :** When you write get; set; within a property, C# **automatically generates a backing field** for you. This is known as an *auto-implemented property*. For example, within a class MyClass:   
public int MyProperty { get; set; }  
is equivalent to writing(what compiler does under the hood probably)-  
private int \_myProperty; // Implicit backing field  
public int MyProperty //Compiler creates a private, **anonymous backing field** to store the property’s value.  
{  
 get { return \_myProperty; } //Generates the get and set methods that interact with this backing field.  
 set { \_myProperty = value; }   
}  
So when user creates object, something like this-  
MyClass obj = new MyClass();   
obj.MyProperty = 10; // Calls set   
int value = obj.MyProperty; // Calls get

// Note that **set {}** may not be always used with a property – in case the properties are read-only ; in other words the properties would be set once during construction (using constructor ) and cannot be altered afterward. So only **get{…}** would be provided.  
// For example: Once a matrix has been created with a certain rows and columns, the number of rows and columns should not be changed further. Also, any include additional logic for validation, formatting, condition checking, etc would then be performed within constructor itself. This **design choice** helps protect the integrity of the object and **adheres to principles of encapsulation and immutability**.

Indexers  
//Indexing refers to accessing elements within a collection (a data structure that can hold multiple elements) or array using an index -numerical value that represents the position of an element within the collection, starting from zero.  
//C# allows you to **define indexers in your own classes**, which ***enable instances of your class to be indexed like arrays***.  
// Note: The indexer does not apply to the instances of the class (say A, B, C if you create multiple objects of the class). Instead**, *it applies to the elements within a single instance of the class.***// The purpose of an indexer is to allow you to treat an object of the class as though it were a collection, such as an array or a list, where you can access the elements inside that object using an index.  
// Indexer Syntax within some generic class SampleCollection<T> -   
private T[] arr = new T[100]; //Declare an array of type T(int, float,etc)  
public T this[int i] // Public access modifier allows the indexer to be accessed from outside the class.  
{ // The return type of the indexer corresponds to the type specified by T- use int, float, array ,etc in place of T(placeholder).  
 get { return arr[i]; } // **this[int i]** :- ‘**this**’ keyword declares an indexer; **[int i]** part - the indexer will take **integer** index **i** as a parameter.  
 set { arr[i] = value; } // get and set methods for the indexer; indexers must have at least one accessors  
}

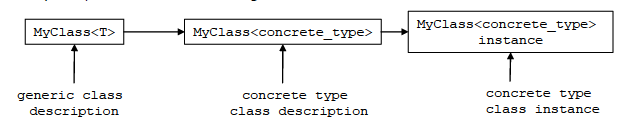
Generics  
// To apply the functionality encapsulated within a class on objects of different types (not known at the time of declaration) , C# provides Generics.  
// Generics allow us to **declare parameters of this class, by indicating an unknown type** that the class will work eventually with.  
// When we instantiate our *generic class*, we **replace** the *unknown with a particular type*. Accordingly, the newly created object will only work with objects of the type that we assign at time of initialization – can be any data type that compiler recognizes( class, structure, enumeration or another generic class.)  
// Generics still provide type safety, reusability without sacrificing performance or type checking.  
// **Typifying** a class (creating a generic class) means to add to the declaration of class a paramenter(replacement) of the unknown type. When class is instantiated, this parameter is then replaced with the name of the some specific type.  
// Add **<T>** to the declaration after class name. (use descriptive name in place of T). For multiple *different* unknown types add **<T,U,V>** .

**Implementing Stack using Generics:**

public class Stack<T> // Note the Syntax: modifiers **class** classname<Placeholder>  
{   
 private T[] elements; //We can use the unknown type for declaring fields   
 private int size;   
 private int capacity;  
 public Stack(int capacity) // Note : **Properties and Constructors of the class cannot have their own generic type**  
 { // Only the generic types defined at class level can be used in body or parameter list of a constructor.  
 this.capacity = capacity;  
 elements = new T[capacity];  
 size = 0;  
 }  
 public T number {get; set;} // Properties can be tied only to class’ type parameters if the class is generic.  
 // This would be **invalid** in C# : public T1 Property<T1> { get; set; } ; Similarly invalid for Constructors.  
 public void Push(T element) // We can use the unknown type **T** for declaring **Method parameters**  
 {  
 if (size >= capacity)  
 {  
 throw new InvalidOperationException("Stack Overflow");  
 }  
 elements[size++] = element; // Push the given element value at end of stack; then increment the size variable   
 }  
 public T Pop() // We can use the unknown type **T** as **return type** of Method   
 {  
 if ( size == 0)  
 {  
 throw new InvalidOperationException("Stack Underflow");  
 }  
 return elements[--size]; // We pop the last element, but first we decrement size by 1, since indexing starts from 0  
 }  
 public T Peek()   
 {  
 if ( size == 0)  
 {  
 throw new InvalidOperationException("Stack is empty!");  
 }  
 return elements[size - 1];  
 }  
 public bool IsEmpty()  
 {  
 return size == 0;  
 }  
 public int Size()  
 {  
 return size;  
 }  
}  
// **Methods** can be **Generic**: We can parameterize/ typify the method – when its type of parameters are unknown.  
public static void Swap<K>(ref K first,ref K second) //Use unknown type **K** for parameters in the parameter’s list of method  
{ // We swap values of two variables irrespective of their type; will work on integers as well as strings,etc  
 K temp = first; // Here keyword **ref** is given to exchange the values of two references.   
 first = second; //So, the method will use the same reference that was given by the calling method.   
 second = temp; //This way, all changes on this variable made by our method, will remain after the method exits.  
}  
//Within Main() method, to call this Generic Method/ or invoke this generic method with concrete type:  
Swap<string>(ref a, ref b); //We would need to create an instance and then call Swap on it, if the declaration did not have static .  
Swap(ref a, ref b); // In calling a generic method, we can skip explicit declaration of a specific type (in our example <string>), because the **compiler will detect it automatically**, recognizing the type of the given parameters.  
  
// Keep track of the scope of the unknown type within the method and that of the containing generic class, in case the symbols are same.

public class CommonOperations<T> // Compiler throws a message CS0693(not error):   
{ //parameter 'T' has the same name as type parameter from outer type CommonOperations<T>'   
 public void Swap<T>(ref T a, ref T b) // Action scope of the unknown type T, defined in declaration of the method, overlaps   
{ … } // the scope of action of the unknown type T, in class declaration

// So, unknown type in the declaration of the generic method needs to be different than the parameter for the unknown type in the class declaration if you are dealing with a different variable within method.

// After declaration of a Generic class, compiler translates code to an intermediate language, containing information that class is generic, i.e. it works with undefined types until now.   
// At runtime, when someone tries to work with our generic class and tries to use it with a specific type, a new **description of the class** is created (specific type class description in the diagram above), which is identical to the generic class, with the difference that where it has been used **T**, now is replaced by a specific type.  
// The interesting thing here is that to create this object, the description of the class, which was created in the meantime (specific type class description), will be used. Instantiating of a generic class by given specific types of its parameters is called "**specialization of the type**" or "**extension of generic class**".

Enumerations  
// Enumeration is like class but differs from it in the sense that we can declare only constants within it. It is declared using the keyword enum; and can have access modifiers like public, private and internal. Constants are separated by commas within enumeration block. Enums in C# are inherently static; you do not and cannot declare them as static(throws compilation error)  
// Since the constants are tied to the type itself and not to an instance of the type, enum is implicitly static. We can access the members of enum directly through enum type without needing to instantiate anything.  
enum Days   
{  
 Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday // Each constant listed here is of 'type' this enum-Days  
}  
public enum CoffeeSize  
{ // We assign the values to each constant during declaration  
 Small = 100, Normal = 150, Double = 300 // We can change the numerical value of constants in an enumeration  
}   
  
class Coffee  
{  
 private CoffeeSize size; // note we define a field variable of type enum Coffeesize  
 // An enumerated variable can have as a value any one of the constants listed but cannot have null.

private AddOn extraItem; // Instance variable of enumerated type

// Sometime enumeration should and can be declared within a class for better encapsulation

public enum AddOn // Note there is no static modifier;

{

Burger = 1, ChocoLava =2, Custard=3, None =4 // We can always add new constants with time

}

public double CoffeePrice()

{

switch (this.size)

{

case CoffeeSize.Small:

return 0.20;

case CoffeeSize.Normal:

return 0.50;

case CoffeeSize.Double:

return 0.80;

default:

throw new InvalidOperationException("Unsupported Size");

}

}

public double AddOnPrice()

{

switch (this.extraItem)

{

case AddOn.Burger:

return 0.10;

case AddOn.ChocoLava:

return 0.15;

case AddOn.Custard:

return 0.20;

case AddOn.None:

return 0.0;

default:

throw new InvalidOperationException("Unsupported Size");

}

}

public Coffee(CoffeeSize consumeSize, AddOn extra = AddOn.None) // Constructor takes the enum type variable as parameter

{

this.size = consumeSize;

this.extraItem = extra;

}

public CoffeeSize Size // declare a property for obtaining size

{

get { return size; } // using get method to return size

}

public AddOn ExtraItem

{

get { return extraItem; }

}

}

public static void Main(string[] args)

{

Console.WriteLine(Days.Monday is Days); // Returns True

// Each constant, declared within an enum is associated with a certain integer-by default for this hidden integer

// representation, int is being used

// Each constant in one enum is actually a textual representation of an integer, by default this number is the constant's

// index in the list of constants of a particular enumeration type

Console.WriteLine((int)Days.Monday + " " + (int)Days.Sunday); // Returns 0 6

Console.WriteLine(Days.Sunday); // On print we get a textual representation given at the time of declaration

Coffee firstCoffee = new Coffee(CoffeeSize.Normal); // Using class Coffee to create an instance/object but with normal size chosen from enum

Console.WriteLine("The first order is {0} coffee of {1} ml.", firstCoffee.Size, (int)firstCoffee.Size);

// Note how we access the Size property from created instance, but cannot access 'size' due to its protection level-private

// But Size returns size of type enum CoffeeSize - gives textual description; to obtain numeric representation we use typecasting

Coffee SecondCoffee = new Coffee(CoffeeSize.Double, Coffee.AddOn.Burger);

Console.WriteLine("The second order is {0} coffee of size {1} ml, with add-on {2}", SecondCoffee.Size, (int)SecondCoffee.Size, SecondCoffee.ExtraItem);

Console.WriteLine("Coffee Sizes Available: Small = 100 ml, Normal = 150 ml, Double = 300 ml");

Console.Write("Please Enter your Coffee Order Size(in ml): ");

int orderSize = int.Parse(Console.ReadLine());

while (!Enum.IsDefined(typeof(CoffeeSize), orderSize))

{

Console.Write("Invalid Input. Please Input a Valid order size:");

orderSize = int.Parse(Console.ReadLine());

}

Console.WriteLine("Add on Items Available: Burger, ChocoLava, Custard");

Console.Write("Select Any one Add on Item. Enter 1 for Burger, 2 for ChocoLava, 3 for Custard, 4 for None:");

int orderAddOn = int.Parse(Console.ReadLine());

while (!Enum.IsDefined(typeof(Coffee.AddOn), orderAddOn))

{

Console.Write("Invalid Input. Please Input a Valid Add-on order number:");

orderAddOn = int.Parse(Console.ReadLine());

}

Coffee thirdOrder = new Coffee((CoffeeSize)orderSize, (Coffee.AddOn)orderAddOn);

Console.WriteLine("The third order is {0} coffee of size {1} ml and add on {2}", thirdOrder.Size, (int)thirdOrder.Size, thirdOrder.ExtraItem);

Console.WriteLine($"The price for your Coffee order is {thirdOrder.CoffeePrice() + thirdOrder.AddOnPrice()}.");

}

// When an exception is thrown, the program's execution jumps to the nearest appropriate exception handler (try-catch block). If no handler is found, the program terminates with an error message.

A structure is a value type that derives implicitly from [System.ValueType](https://docs.microsoft.com/en-us/dotnet/api/system.valuetype), which in turn is derived from [System.Object](https://docs.microsoft.com/en-us/dotnet/api/system.object). Like classes, structures define both data (the fields of the structure) and the operations that can be performed on that data (the methods of the structure). This means that you can call methods on structures, including the virtual methods defined on the [System.Object](https://docs.microsoft.com/en-us/dotnet/api/system.object) and [System.ValueType](https://docs.microsoft.com/en-us/dotnet/api/system.valuetype) classes, and any methods defined on the value type itself. In other words, structures can have fields, properties, and events, as well as static and nonstatic methods. You can create instances of structures, pass them as parameters, store them as local variables, or store them in a field of another value type or reference type. Structures can also implement interfaces.

Value types also differ from classes in several respects. First, although they implicitly inherit from [System.ValueType](https://docs.microsoft.com/en-us/dotnet/api/system.valuetype), they cannot directly inherit from any type. Similarly, all value types are sealed, which means that no other type can be derived from them. They also do not require constructors.

**#**Objects in OOP combine data and the means for their processing in one. They correspond to objects in real world and contain data and actions:

**Class:  
#**The **class** defines abstract characteristics of objects. It provides a structure for objects or a pattern which we use to describe the nature of something (some object). **Classes are building blocks of OOP** and are inseparably related to the **objects**. Furthermore, each object is an **instance** of exactly one specific class.

We are going to give as an **example a class and an object**, which is its instance. We have a **class Dog** and an **object Lassie**, which is an instance of the class **Dog** (we say it is an object of type **Dog**). The class **Dog** describes the characteristics of all dogs whereas **Lassie** is a certain dog.   
  
**Class** in the OOP is called a definition (**specification**) of a given type of objects from the real-world. The class represents a pattern, which describes the different states and behavior of the certain objects (the copies), which are created from this class (pattern).

**Object** is a copy created from the definition (specification) of a given class, also called an **instance**. When one object is created by the description of one class we say **the object is from type "name of the class"**.  
**#**The class defines the **characteristics of an object** (which we are going to call **attributes**) and its **behavior** (actions that can be performed by the object). The attributes of the class are defined as its own variables in its body (called **member variables**). The behavior of objects is modeled by the definition of **methods** in classes.

**#**In the context of such behavior the object consists of two things: current **state** and **behavior** defined in the class of the object. The state is specific for the instance (the object), but the behavior is common for all objects which are instances of this class.

A **class** in C# is defined by the keyword **class**, followed by an identifier (name) of the class and a set of data members and methods in a separate code block.

**Classes** in C# can contain the following elements:

- **Fields** – member-variables from a certain type;   
- **Properties** – these are a special type of elements, which extend the functionality of the fields by giving the ability of extra data management when extracting and recording it in the class fields;   
- **Methods** – they implement the manipulation of the data.

public class Cat

{

// Field name

private string name;

// Field color

private string color;

public string Name

{

// Getter of the property "Name"

get

{

return this.name;

}

// Setter of the property "Name"

set

{

this.name = value;

}

}

public string Color

{

// Getter of the property "Color"

get

{

return this.color;

}

// Setter of the property "Color"   
set

{

this.color = value;

}

}

// Default constructor

public Cat()

{

this.name = "Unnamed";

this.color = "gray";

}

// Constructor with parameters

public Cat(string name, string color)

{

this.name = name;

this.color = color;

}

// Method SayMiau

public void SayMiau()

{

Console.WriteLine("Cat {0} said: Miauuuuuu!", name);

}

}   
The example class **Cat** defines the **properties Name** and **Color**, which keep their values in the hidden (private) **fields name** and **color**. Furthermore, two **constructors** are defined for creating instances of the class **Cat**, respectively with and without parameters, and a **method** of the class **SayMiau()**.  
After the example class is defined we can now use it in the following way:   
static void Main()

{

Cat firstCat = new Cat();

firstCat.Name = "Tony";

firstCat.SayMiau();

Cat secondCat = new Cat("Pepy", "red");

secondCat.SayMiau();

Console.WriteLine("Cat {0} is {1}.",

secondCat.Name, secondCat.Color);

}

Calling the method **Console.WriteLine(…)** of the class **System.Console** is an example of usage of a **system class** in C#. We call system classes the classes defined in **standard libraries** for building applications with C# for example the classes **String**, **Environment** and **Math .**It is important to know that the implementation of the logic in classes is **encapsulated** (hidden) inside them. For the programmer it is important what they do, not how they do it and for this reason a great part of the classes is not publicly available (**public**). With system classes the implementation is often not available at all to the programmer. Thus, new **layers of abstraction** are created which is one of the basic principles in OOP.