

Classes and Objects

Concept of a Class

A class is an abstract model used to define a new data types. A class may contain any combination of encapsulated data (fields or member variables), operations that can be performed on data (methods) and accessors to data (properties). For example the class String contains an array of characters (data) and provide different operations (methods) that can be applied to its data like ToLowerCase(). It also has some properties like Length. A class in C# is declared using the keyword class and its members are enclosed in parenthesis

```
class MyClass {  
    ... fields, constants...           // for object-oriented programming  
    ... methods.....  
    ... constructors, destructors...  
  
    ... properties...                 // for component-based programming  
    ... events...  
  
    ... indexers...                   // for amenity  
    ... overloaded operators...  
  
    ... nested types (classes, interfaces, structs, enums, delegates)...  
}
```

Objects

An object is the concrete realization or instance built on the model specified by the class. An object is created in the memory using the keyword 'new' and is referenced by an identifier called a "reference" as shown below

```
MyClass myObjectReference = new MyClass();
```

In the statement above, an object of type MyClass has been created and which is referenced by an identifier (or handle) myObjectReference.

E.g. is instantiated instantiating (making an object) of Student class

```
Student theStudent = new Student();
```

Note that it is very similar to using implicit data types except for the object is created with the new operator while implicit data types are created using literals

```
int i;  
i = 4;
```

Class Members

i). Data Members

Data members are those members that contain the data for the class: -fields, constants, and events

Data members can be either static (associated with the class as a whole) or instance (each instance of the class has its own copy of the data). A class member is always an instance member unless it is explicitly declared as static.

- Fields are any variables associated with the class
- Constants can be associated with classes in the same way as variables and are declared as constant using the const keyword
- Events are class members that allow an object to notify a caller whenever something noteworthy happens, such as a field or property of the class changing, or some form of user interaction occurring. The client can have code known as an event handler that reacts to the event.

Fields

Fields are the data contained in the class. Fields may be implicit data types, objects of some other class, enumerations, structs or delegates. In the example below, we define a class named Student containing a student's name, age, marks in two subjects, total marks, obtained marks and a percentage.

```
class Student
```

```
{
    // fields contained in Student class
    string name;
    int age;
    int subject1;
    int subject2;
    int totalMarks = 200; // initialization
    int obtainedMarks;
    double percentage;
}
```

It is possible to initialize the fields with the initial values as in case of totalMarks as shown above. If you don't initialize the members of the class, they will be initialized with their default values.

Constant

They are the variables that represent constant values associated with the class and whose value cannot be changed during program execution. The const members are implicitly static and are accessed using class name e.g. Math.PI. Value must be set when const is defined as shown below in example below:

```
public class MyClass {
    public const string version= "1.0.0";
    public const string PI_I3= 3*Math.PI;
    public const string s= Math.Sin(Math.PI); //Error
}
```

Any attempt to change the value will result in compilation error

Readonly Fields

The readonly keyword gives a bit more flexibility than const, allowing for the case in which you might want a field to be constant but also need to carry out some calculations to determine its initial value i.e. they are initialized at run-time in its declaration or in a constructor. **The rule is that you can assign values to a readonly field inside a constructor, but not anywhere else.**

Notice that you don't have to assign a value to a readonly field in a constructor i.e. it can be left with the default value for its particular data type or whatever value you it was initialized at the time of its declaration. This applies to both static and instance readonly fields.

```
public class MyClass {
    public static readonly double d1 = Math.Sin(Math.PI);
    public readonly string s1;
    public MyClass(string s) { s1 = s; }
}
```

ii). Function Members

Function members are those members that provide some functionality for manipulating the data in the class. They include methods, properties, constructors, finalizers, operators, and indexers.

Methods are functions that are associated with a particular class and define operations performed on the data. They can be either instance methods or static methods, which provide more generic functionality that doesn't require us to instantiate a class.

Methods

A method may take some input values through its parameters and may return a value of a particular data type. The signature of the method takes the form

```
<return type> <name of method>(<data type> <identifier>, <data type> <identifier>,...)
{
    // body of the method
}
```

Accessing the members of a class

The members of a class are accessed using dot '.' operator against the reference of the object as shown below:

```

Student theStudent = new Student();
theStudent.subject1=93;
theStudent.CalculateTotal();
Console.WriteLine(theStudent.obtainedMarks);

```

Example: - Write a C# program that has a Student class with some related fields, methods and then instantiate it in the Main() method.

```

using System;
// Defining a class to store and manipulate students information
class Student
{
    // fields
    string name;
    int age;
    int subject1;
    int subject2;
    int totalMarks = 200;
    int obtainedMarks;
    double percentage;

    // methods
    void CalculateTotalMarks()
    {
        obtainedMarks = subject1+subject2;
    }
    void CalculatePercentage()
    {
        percentage = (double) obtainedMarks / totalMarks * 100;
    }
    // Main method or entry point of program
    static void Main()
    {
        // creating new instance of Student
        Student stud1 = new Student();
        // setting the values of fields
        stud1.name = "Frank";
        stud1.age = 23;
        stud1.subject1 = 80;
        stud1.subject2 = 99;
        // calling methods
        stud1.CalculateTotalMarks();
        stud1.CalculatePercentage();

        Student stud2 = new Student();
        stud2.name = "Elsie";
        stud2.age = 20;
        stud2.subject1 = 77;
        stud2.subject2 = 100;
        stud2.CalculateTotalMarks();
        stud2.CalculatePercentage();

        Console.WriteLine("{0} of {1} years age got {2}% marks",
            stud1.name, stud1.age, stud1.percentage);
    }
}

```

```

        Console.WriteLine("{0} of {1} years age got {2}% marks",
            stud2.name, stud2.age, stud2.percentage);
    }
}

```

Access Modifiers or Accessibility Levels

In the Student class, everyone has access to each of the fields and methods. So if one wants to change the totalMarks from 200 to say 100, resulting in the percentages getting beyond 100%, which in most cases we like to restrict. C# provides access modifiers or accessibility levels for this purpose, i.e., restricting access to a particular member. The following is a list of five access modifiers that can be applied to any member of the class.

Access Modifier	Description
private	private members can only be accessed within the class that contains them This type of members are accessible from any class derived from that class, or any class within the same assembly i.e. either protected or internal applies.
protected internal	Members are accessible only in the same assembly and invisible outside it.
internal	Can be accessed from a containing class and types inherited from the containing class
protected	public members are not restricted to anyone.
public	

In Object Oriented Programming (OOP) it is always advised and recommended to mark all your fields as private and allow the user of your class to access only certain methods by making them public. For example, we may change our student class by marking all the fields private and the three methods in the class public as shown below.

```

class Student
{
    // fields
    private string name;
    private int age;
    private int subject1;
    private int subject2;
    private int totalMarks = 200;
    private int obtainedMarks;
    private double percentage;

    // methods
    public void CalculateTotalMarks()
    {
        obtainedMarks = subject1+subject2;
    }
    public void CalculatePercentage()
    {
        percentage = (double) obtainedMarks / totalMarks * 100;
    }
}

```

If you don't mark any member of class with an access modifier, it will be treated as a private member i.e. the default access modifier is private. If no protection level is specified for top level classes, they are treated as internal.

You can also apply access modifiers to other types in C# such as the class, interface, struct, enum, delegate and

event. For top-level types (types not bound by any other type except namespace) like class, interface, struct and enum you can only use public and internal access modifiers. Note also that you cannot apply access modifiers to namespaces.

Properties

In C# Properties are used to assign values to all the fields in a class declared as private through their references. In languages such as Java and C++, in order to access the private fields of a class, you need to use public methods called getters (to retrieve the value) and setters (to assign the value)

E.g. Consider the following

```
private string name;
```

the getters and setters would be used as follows

```
// getter to name field
public string GetName()
{
    return name;
}
// setter to name field
public void SetName(string theName)
{
    name = theName;
}
```

This approach gives you a lot of control over how fields of your classes should be accessed and dealt in a program. But, the problem is that you need to define two methods and prefix them with Get or Set.

C# provides the built in support for these getters and setters in the form of properties. Properties are context sensitive constructs used to read, write or compute private fields of class and to achieve control over how the fields can be accessed.

Syntax

```
<access modifier> <data type> <name of property>
{
    get
    {
        // some optional statements
        return <some private
field>; }
    set
    {
        // some optional statements;
        <some private field> =
value; }
}
```

Example

Consider the following private field name

```
private string name;
```

You can define a property for this providing both getters and setters as follows:

```
public string Name
{
    get
    {
        return name;
    }
    set
    {
```

```

        name=value;
    }
}

```

A property called 'Name' with both getter and setter in the form of get { } and set { } blocks has been created. Note that it is a convention to name the property the same as the corresponding field but with first letter in uppercase. As properties are accessors to certain fields, they are mostly marked as public while the corresponding field is mostly private

The **value** is a keyword and contains the value passed when a property is called. In the program above, use the property as follows

```

theStudent.Name = "Frank";
string myName = theStudent.Name;

```

Properties are context sensitive. When we write

```

theStudent.Name = "Frank";

```

The compiler notices that the property Name is on the left hand side of assignment operator, so it will call the set { } block of the properties passing "Frank" as a value. In the next line, when you write

```

string myName = theStudent.Name;

```

the compiler notices that the property Name is on the right hand side of the assignment operator, hence it will call the get { } block of property Name which will return the contents of the private field.

Example of Complete Program

Create a C# program for a rectangle abstract data type. The class has field's **length** and **width** and methods to calculate the **area** and **perimeter** of rectangle. It has properties **set** and **get** for both the length and width. Instantiate at least one object to demonstrate your program is working.

```

using System;
class Rectangle{
    private double length;
    private double width;
    public double Length{
        set{
            length=value;
        }
        get{
            return length;
        }
    }
    public double Width{
        set{
            width=value;
        }
        get{
            return width;
        }
    }
    public double Area() {
        return length*width;
    }
    public double Perimeter() {
        return 2*(length+width);
    }
}
class TestRectangle{

```

```

static int Main() {
    Rectangle rec=new Rectangle();
    double length, width;
    Console.WriteLine("Enter the length and width of the
rectangle");
    length=Double.Parse(Console.ReadLine());
    width=Double.Parse(Console.ReadLine());
    rec.Length=length;
    rec.Width=width;
    Console.WriteLine("For rectangle of length: " + rec.Length +
        " and width: " + rec.Width + "\n has an area of: " +
        rec.Area() + " and a perimeter of: " + rec.Perimeter());
    return 0;
}
}

```




Generally Properties are used for the following reasons:

- i). Properties allow the specification of read-only and write-only fields i.e. a property can omit either a get clause (read-only property) or a set clause (write-only property)
- ii). Properties are used to validate a field when it is assigned a value (setter method) and when its value is accessed (getter method).
- iii). Properties are especially useful in component oriented programming.
- iv). Properties help to build reliable and robust software.

Constructors

Constructors are a special kind of method. A Constructor has the following characteristics: 

It has the same name as its containing class and no return type

-  It is automatically called when a new instance or object of a class is created
-  The constructor contains initialization code for each object, like assigning default values to the fields.
-  Constructor method is declared as public because it is used to create objects from outside the class in which it is declared. We can also declare a constructor method as private, but then such a constructor cannot be used to create objects.

Example

```

using System;
class Person
{
    // field
    private string
    name; // constructor
    public Person()
    {
        name = "unknown";
        Console.WriteLine("Constructor called...");
    }
    // property
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}

```

In the Person class above, there is a private field name, a public constructor which initializes the name field with string "unknown" and a public property to read/write the private field name.

The following is another class Test which contains the Main() method and which instantiates the Person class

```

class Test
{

```

```

public static void Main()
{
    Person thePerson = new Person();
    Console.WriteLine("The name of person in the object is " +
        thePerson.Name);
    thePerson.Name = "Frank";
    Console.WriteLine("The name of person in the object is " +
        thePerson.Name);
}
}

```

Note that the constructor is called just as we create a new instance of Person class and initialize the field name with string "unknown".

```
Person thePerson = new Person();
```

That is why constructor is usually made public. If you make your constructor private, no one would be able to make an object of your class outside of it.

Constructors with Parameters

Constructors which take parameters. This technique helps to assign initial value to an object at the time of its creation as shown below:

```

class Person
{
    private string name;
    public Person(string theName)
    {
        name = theName;
        Console.WriteLine("Constructor called...");
    }
}

```

Now, the object of class Person can only be created by passing a string into the constructor.

```
Person thePerson = new Person("Frank");
```

Default Constructor

If no constructor has been specified for a given class, the compiler generates a parameter-less default constructor. If a constructor has been specified for a given class, no default constructor is generated:

```

class C{ int x; }
C c = new C(); // ok

class C{ int x;
public C( int y ) { x = y; } } C
c1 = new C(); // error
C c2 = new C(3); // ok

```

Static Constructors

C# supports two types of constructors: static constructor and instance constructor. Whereas an instance constructor is called every time an object of that class is created, the static constructor is called only once before any object of the class is created. They are used to initialize any static data members of a class and they cannot have parameters

Syntax:

```

class Abc
{
    static Abc()
    {
        .....
    }
}

```



```
}
```

Notice that any class can have only one static constructor and useful for doing any housekeeping work that needs to be done once

Copy Constructors

A copy constructor creates an object by copying variables from another object. It is called by creating an object of the required type and passing it the object to be copied.

Example:

```
using System;
namespace CopyConstructor
{
    class Rectangle
    {
        private int length, breadth;
        public Rectangle(int x, int y)
        {
            length = x;
            breadth = y;
        }
        public Rectangle(Rectangle r)//copy constructor
        {
            length = r.length;
            breadth = r.breadth;
        }
        public void display()
        {
            Console.WriteLine("Length = " + length);
            Console.WriteLine("Breadth = " + breadth);
        }
    } // end of class Rectangle
    class Program
    {
        public static void Main()
        {
            Rectangle rec1 = new Rectangle(5, 10);
            Console.WriteLine("Values of first object");
            rec1.display();
            Rectangle rec2 = new Rectangle(rec1);
            Console.WriteLine("Values of second object");
            rec2.display();
        }
    }
}
```

Finalize() Method of Object Class

Each class in C# is automatically (implicitly) inherited from the Object class which contains a method Finalize(). This method is guaranteed to be called when your object is garbage collected (removed from memory).

Destructors

Destructor are just the opposite of constructor and is called automatically when the object is about to be destructed (when garbage collector is about to destroy your object). It has the same name as the containing class but prefixes it with the ~ (tilde) sign and has no return type.

Unlike C++, C# destructors are non-deterministic i.e. they are not guaranteed to be called at a specific time,

although, they are guaranteed to be called before shutdown. In C# you cannot directly call the destructor
Destructors are declared as shown below:

```
class Person
{
    // constructor
    public Person()
    {

    }
    // destructor
    ~Person()
    {
        // put resource freeing code here.
    }
}
```

The C# compiler internally converts the destructor to the Finalize() method. Destructors are not very common in C# programming practices (that is why Java dropped the idea of destructors) also they slow down the garbage collection.

this Reference

Each object has a reference **this** which points to itself. Conventionally, the parameters to constructors and other methods are named the same as the name of the fields they refer to and are distinguished only by using **this** reference.

```
using System;
class Rectangle{
    private int length,width;
    public Rectangle(int length, int width){
        this.length=length;
        this.width=width;
    }
    ~Rectangle() {
        Console.WriteLine("Destroyed {0}", this);
    }
    public int Area(){
        return length*width;
    }
    static void Main(){
        Rectangle rectangle=new Rectangle(10,20);
        Console.WriteLine("The Area of Rectangle is: {0}",
            rectangle.Area());
    }
}
```

In this example, when we use length or width, we actually get the variables passed in the method which overshadow the instance members (fields) with same name. Hence, the need of **this** reference. This is an extremely useful, widely and commonly used construct

Static Members of the class

This are members belong to the whole class rather than to individual object and they are also referred as class variables and class methods.

For example, if you have a static phoneNumber field in your Student class, then there will be the single instance of this field and all the objects of this class will share this single field. Changes made by one object to phoneNumber will be realized by the other object. Static members are defined using keyword static as shown

below

```
class Student
{
    public static int  phoneNumber;
    public int rollNumber;
}
```

Static members are accessed with the name of class rather than reference to objects.

```
class Test
{
    public static void Main()
    {
        Student st1 = new Student();
        Student st2 = new Student();
        st1.rollNumber = 3;
        st2.rollNumber = 5;
        Student.phoneNumber = 4929067;
    }
}
```

Static methods are very useful while programming. The WriteLine() and ReadLine() methods are static methods of Console class and also the Main method in C# is declared static and can be called without making any instance of the class. They are also useful when you want to cache data that should be available to all objects of the class. You can use static fields, methods, properties and constructors.

Since static methods may be called without any reference to object, you cannot use instance members inside static methods or properties, however you may call a static member from a non-static context since static members belong to the class and are present irrespective of the existence of even a single object. This is illustrated below:

```
class Student
{
    public static int  phoneNumber;
    public int rollNumber;
    public void DoWork()
    {
        //legal, static method called in non-static context
        MyMethod();
    }
    public static void MyMethod()
    {
        //legal, static field used in static context
        phoneNumber++;
        //illegal, non-static field used in static context
        rollNumber++;
        //illegal, non-static method used in static context
        DoWork();
    }
}
```

Static classes

Static classes are commonly used to implement a Singleton Pattern. All of the methods, properties, and fields of a static class are also static and can thus be used without instantiating the static class

```
public static class Writer
{
```

```

        public static void Write()
        {
            System.Console.WriteLine("Text");
        }
    }
    public class Sample
    {
        public static void Main()
        {
            Writer.Write();
        }
    }
}

```

Caution when using static members

Don't put too many static methods in your class as it is against the object oriented design principles and makes your class less extensible i.e. static methods can't be overridden which means it cannot be used polymorphically.

Method and Constructor Overloading

It is possible to have more than one method with the same name and return type but with a different number and type of arguments (parameters). This is called method overloading.

Example:

```

using System;
public class ShapeArea{
    double Area(int width,int height){
        return width*height;
    }
    double Area(int radius){
        return Math.PI*radius*radius;
    }
    static void Main (){
        ShapeArea shape=new ShapeArea();
        int d1=10,d2=20;
        Console.WriteLine("The circle's area = {0}",
            shape.Area(d1));
        Console.WriteLine("The Rectangle's area = {0}",
            shape.Area(d1,d2));
    }
}

```

In the ShapeArea class there are two methods with the name Area(). The return type of all these is double but all differ from each other in parameter list. When Area() is called, the compiler will decide (on the basis of the types and number of parameters being passed) which one of these methods to actually call. Notice the WriteLine() method of Console class has nineteen different overloaded forms.

Overloading Constructors

Since constructors are a special type of method, we can overload constructors.

Example

Write a C# program that has a class called circle from which two circles, circle1 and circle2 are created in such way that circle2 does not pass any argument to the constructor yet its radius value is initialized to 10 and circle1 is instantiated with the value 5. The class should have a method to calculate the area of the two circles which is displayed in an appropriate format.

```

using System;
class Circle{
    private double radius;
    public Circle() {
        radius = 10.0; }
    public Circle(double r) {
        radius = r;
    }
    public double Area (){
        return Math.PI*Math.Pow(radius,2);
    }
}
class TestCircle{
    static void Main(){
        Circle circle1 = new Circle();
        Circle circle2 = new Circle(5.0);
        Console.WriteLine("The circle one area is: {0}",
            circle1.Area());
        Console.WriteLine("The circle two area is :{0}",
            circle2.Area());
    }
}

```

Generally, overloading methods and constructors gives your program a lot of flexibility and reduces a lot of complexity that would otherwise be produced if we had to use different name for these methods.

Passing Objects to Methods

Objects are implicitly passed by reference. This means that only a copy of the reference is passed to the methods during method invocation. Hence, if we initialize an array (which is an object in C#) and pass it to some method where the array gets changed, then this changed effect would be visible after the method has been terminated in the calling method.

```

class Test
{
    public static void Main()
    {
        int [] nums = {2, 4, 8};
        DoWork(nums);
        int count =0;
        foreach(int num in nums)
            Console.WriteLine("The value of a[{0}] is {1}", count++,
                num);
    }
    public static void DoWork(int [] numbers)
    {
        for(int i=0; i<numbers.Length; i++)
            numbers[i]++;
    }
}

```

Some more about references and objects

A reference is just a pointer or handle to the object in memory. It is possible to create an object without giving its handle to any reference:

```
new Student();
```

The above statement will create an object of Student class without any reference pointing to it. An object is actually eligible to be garbage collected when there is no reference to point it. So, in the above case, the new Student object will instantly be eligible to be garbage collected after its creation. Unreferenced objects as can be used as follows

```
int pc = ( new Student(87, 94, 79) ).CalculatePercentage();
```

In the above statement, the newly created unreferenced object will be eligible to be garbage collected just after the method CalculatePercentage() completes its execution. The above statement is similar to the following

```
Student theStudent = new Student(87, 94, 79);
int pc = theStudent.CalculatePercentage();
theStudent = null;
```

We assigned null to theStudent so the object above will be destroyed after method call terminates as in the case of previous example.

Array of Objects

The elements of an array can be object references. When an array of objects is constructed, each element initially stores a special reference value null.

The program below creates and initializes an array of twenty even numbers. The program instantiates twenty one separate objects - one for the array and one each for the twenty elements.

```
using System;
class EvenNumbers {
    private int number, square;
    public EvenNumbers(int number, int square) {
        this.number = number;
        this.square = square;
    }
    public int Number{
        get{
            return number;
        }
    }
    public int Square{
        get{
            return square;
        }
    }
}
class Test {
    static void Main() {
        EvenNumbers[] numbers = new EvenNumbers[20];
        for (int i = 0; i < numbers.Length; i++)
            numbers[i] = new EvenNumbers(i, i*2+2);
        Console.WriteLine("Number\tSquare");
        for (int i = 0; i < numbers.Length; i++)
            Console.WriteLine("{0}\t\t{1}", numbers[i].Number,
                               numbers[i].Square);
    }
}
```

Indexers (smart arrays)

This is a C# construct that can be used to access collections contained by a class using the [] syntax for arrays. Declaration of behavior of an indexer is similar to a property since you use get and set accessors for defining an indexer. However, properties return or set a specific data member, whereas indexers returns or sets a particular value from the object instance. Indexer is also declared using the keyword this

Syntax

A one dimensional indexer has the following syntax:

```
element-type this[int index]
{
    // The get accessor.
    get
    {
        return coll[index];
    }
    // The set accessor.
    set
    {
        coll[index] = value;
    }
}
```

The following are differences between indexers and properties:

- i). A property can be static member, whereas an indexer is always an instance member.
- ii). A get accessor of a property corresponds to a method with no parameters, whereas for indexer it corresponds to same formal parameter list as the indexer.
- iii). A set accessor of a property corresponds to a method with a single parameter named value, whereas for indexer it corresponds to same formal parameter list as the indexer, plus the parameter named value.

```
using System;
using System.Collections;
class MyCities
{
    private string []data = new string[4];
    public string this [int index]
    {
        get
        { return data[index];}
        set
        { data[index] = value;}
    }
}
class TestMyCities
{
    public static void Main()
    {
        MyCities mc = new MyCities();
        mc[0] = "Nairobi";
        mc[1] = "Nakuru";
        mc[2] = "Mombasa";
        mc[3] = "Kisumu";
        for(int i=0; i<4; i++)
            Console.WriteLine(mc[i]);
    }
}
```

Structures (struct)

This is a composite data type, consisting of a number of elements (or members) of other types and used they used to create lightweight objects. The variables which make up a struct are called members (fields in C#), and are accessed using a dot notation. Structures are mostly used when only a data container is required for a collection of value type variables. Generally, **structs** are similar to **classes** since they can have constructors, methods, and can even implement interfaces, but there are also important differences as shown below

Class	Struct
Reference type	Value type
Can inherit from any non-sealed reference type	No inheritance (inherits only from System.ValueType)
Can have a destructor	No destructor
Can have user-defined parameter-less constructor	No user-defined parameter-less constructor

Constructors for Structs

You can define constructors for structs in exactly the same way that you can for classes, except that you are not permitted to define parameterless constructor i.e. for every struct the compiler generates a parameter-less default constructor, even if there are other constructors. The default constructor initializes all fields to their default value.

Instantiating the struct

A struct can be instantiated in three ways:

- i). Using the new keyword and calling the default no-argument constructor. ii).
- Using the new keyword and calling a custom or user defined constructor.
- iii). Without using the new keyword.

Example 1

```
using System;
struct Point
{
    public double x;
    public double y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
class Test
{
    static void Main()
    {
        Point pt = new Point();
        Point pt1 = new Point(15, 20);
        Point pt2; // instantiation without using the new keyword
        pt2.x = 6;
        pt2.y = 3;
        Console.WriteLine("pt = {0}, {1}", pt.x, pt.y);
        Console.WriteLine("pt1 = {0}, {1}", pt1.x, pt1.y);
        Console.WriteLine("pt2 = {0}, {1}",
            pt2.x, pt2.y);    }
```



```
}
```

Example 2

```
using System;
struct Person
{
    public string name;
    public int height;
    public string occupation;
    public DateTime dob;
}
public class TestStruct
{
    public static void Main()
    {
        Person person = new Person {name = "Your name", height = 182,
        occupation = "Programmer",dob=new DateTime(1990,8,20)};
        Console.WriteLine("Name: " + person.name);
        Console.WriteLine("Height: " + person.height);
        Console.WriteLine("Occupation: " + person.occupation);
        Console.WriteLine("Age: " + (DateTime.Now.Year-
        person.dob.Year));
    }
}
```