Lecture No 5

Muhammad Siddique

Outline

- Definition of Class
- Constructors
- Destructors
- Encapsulation (Methods)
- Static class
- Static methods

Definition of Class

- A class definition starts with the keyword class followed by the class name; and the class body enclosed by a pair of curly braces.
- Access specifiers specify the access rules for the members as well as the class itself.
- Data type specifies the type of variable, and return type specifies the data type of the data the method returns, if any.
- To access the class members, you use the dot (.) operator.
- The dot operator links the name of an object with the name of a member.

Definition of Class (Example)

```
using System;

namespace BoxApplication {
   class Box {
      public double length;   // Length of a box
      public double breadth;  // Breadth of a box
      public double height;   // Height of a box
   }
   class Boxtester {
      static void Main(string[] args) {
         Box Box1 = new Box();   // Declare Box1 of type Box
         Box Box2 = new Box();   // Declare Box2 of type Box
         double volume = 0.0;    // Store the volume of a box here

         // box 1 specification
         Box1.height = 5.0;
         Box1.length = 6.0;
         Box1.breadth = 7.0;

         // box 2 specification
         Box2.height = 10.0;
         Box2.length = 12.0;
         Box2.breadth = 13.0;

         // volume of box 1
         volume = Box1.height * Box1.length * Box1.breadth;
         Console.WriteLine("Volume of Box1 : {0}",  volume);

         // volume of box 2
         volume = Box2.height * Box2.length * Box2.breadth;
         Console.WriteLine("Volume of Box2 : {0}", volume);
         Console.ReadKey();
      }
   }
}
```

Output

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Encapsulation

- It is defined 'as the process of enclosing one or more items within a physical or logical package'.
- Encapsulation, in object oriented programming methodology, prevents access to implementation details.
- Abstraction and encapsulation are related features in object oriented programming.
- Abstraction allows making relevant information visible and encapsulation enables a programmer to implement the desired level of abstraction.
- Encapsulation is implemented by using access specifiers. An access specifier defines the scope and visibility of a class member.

Access Specifier Types

- Following are the types of access specifier are as follows:
- Private (Only functions of the same class can access its private members).
- Public (Allows a class to expose its member variables and member functions to other functions and objects).
- Protected (Will discuss later in topic of inheritance).
- Internal (Will discuss later in topic of inheritance).
- Protected internal (Will discuss later in topic of inheritance).

Private (Example-1)

```
using System;

namespace BoxApplication {
   class Box {
      private double length;   // Length of a box
      private double breadth;  // Breadth of a box
      private double height;   // Height of a box

      public void setLength( double len ) {
         length = len;
      }
      public void setBreadth( double bre ) {
         breadth = bre;
      }
      public void setHeight( double hei ) {
         height = hei;
      }
      public double getVolume() {
         return length * breadth * height;
      }
   }
   class Boxtester {
      static void Main(string[] args) {
         Box Box1 = new Box();    // Declare Box1 of type Box
         Box Box2 = new Box();
         double volume;

         // Declare Box2 of type Box
         // box 1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);

         // box 2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
         Box2.setHeight(10.0);

         // volume of box 1
         volume = Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}" ,volume);

         // volume of box 2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);

         Console.ReadKey();
      }
   }
}
```

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

Private (Example-2)

```
using System;

namespace RectangleApplication {
   class Rectangle {
      //member variables
      private double length;
      private double width;

      public void Acceptdetails() {
         Console.WriteLine("Enter Length: ");
         length = Convert.ToDouble(Console.ReadLine());
         Console.WriteLine("Enter Width: ");
         width = Convert.ToDouble(Console.ReadLine());
      }
      public double GetArea() {
         return length * width;
      }
      public void Display() {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle

   class ExecuteRectangle {
      static void Main(string[] args) {
         Rectangle r = new Rectangle();
         r.Acceptdetails();
         r.Display();
         Console.ReadLine();
      }
   }
}
```

```
Enter Length:
4.4
Enter Width:
3.3
Length: 4.4
Width: 3.3
Area: 14.52
```

Public (Example)

```
using System;

namespace RectangleApplication {
   class Rectangle {
      //member variables
      internal double length;
      internal double width;

      double GetArea() {
         return length * width;
      }
      public void Display() {
         Console.WriteLine("Length: {0}", length);
         Console.WriteLine("Width: {0}", width);
         Console.WriteLine("Area: {0}", GetArea());
      }
   }//end class Rectangle
```

```
   class ExecuteRectangle {
      static void Main(string[] args) {
         Rectangle r = new Rectangle();
         r.length = 4.5;
         r.width = 3.5;
         r.Display();
         Console.ReadLine();
      }
   }
}
```

```
Length: 4.5
Width: 3.5
Area: 15.75
```

Constructor

- A class constructor is a special member function of a class that is executed whenever we create new objects of that class.
- A constructor has exactly the same name as that of class and it does not have any return type.
- A constructor with no parameters called default Constructor.
- If a constructor have parameters, such constructors are called parameterized constructors.

Constructor (Example)

```
using System;

namespace LineApplication {
   class Line {
      private double length;   // Length of a line

      public Line() {
         Console.WriteLine("Object is being created");
      }
      public void setLength( double len ) {
         length = len;
      }
      public double getLength() {
         return length;
      }
      static void Main(string[] args) {
         Line line = new Line();

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

```
Object is being created
Length of line : 6
```

Destructor

- A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope.
- A destructor has exactly the same name as that of the class with a prefixed tilde (~) and it can neither return a value nor can it take any parameters.
- Destructor can be very useful for releasing memory resources before exiting the program. Destructors cannot be inherited or overloaded.

Destructor (Example)

```
using System;

namespace LineApplication {
   class Line {
      private double length;   // Length of a line

      public Line() {   // constructor
         Console.WriteLine("Object is being created");
      }
      ~Line() {   //destructor
         Console.WriteLine("Object is being deleted");
      }
      public void setLength( double len ) {
         length = len;
      }
      public double getLength() {
         return length;
      }
      static void Main(string[] args) {
         Line line = new Line();

         // set line length
         line.setLength(6.0);
         Console.WriteLine("Length of line : {0}", line.getLength());
      }
   }
}
```

```
Object is being created
Length of line : 6
Object is being deleted
```

Static Members of a Class

- We can define class members as static using the static keyword.
- When we declare a member of a class as static, it means no matter how many objects of the class are created, there is only one copy of the static member.
- The keyword static implies that only one instance of the member exists for a class.
- Static variables are used for defining constants because their values can be retrieved by invoking the class without creating an instance of it.
- Static variables can be initialized outside the member function or class definition.
- You can also initialize static variables inside the class definition.

Static Members of a Class (Example)

```
using System;

namespace StaticVarApplication {
   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }
      public int getNum() {
         return num;
      }
   }
   class StaticTester {
      static void Main(string[] args) {
         StaticVar s1 = new StaticVar();
         StaticVar s2 = new StaticVar();
         s1.count();
         s1.count();
         s1.count();

         s2.count();
         s2.count();
         s2.count();

         Console.WriteLine("Variable num for s1: {0}", s1.getNum());
         Console.WriteLine("Variable num for s2: {0}", s2.getNum());
         Console.ReadKey();
      }
   }
}
```

```
Variable num for s1: 6
Variable num for s2: 6
```

Static Function (Example)

```
using System;

namespace StaticVarApplication {
   class StaticVar {
      public static int num;

      public void count() {
         num++;
      }
      public static int getNum() {
         return num;
      }
   }
```

```
   class StaticTester {
      static void Main(string[] args) {
         StaticVar s = new StaticVar();

         s.count();
         s.count();
         s.count();

         Console.WriteLine("Variable num: {0}", StaticVar.getNum());
         Console.ReadKey();
      }
   }
}
```

```
Variable num: 3
```

Inheritance in C#

- Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and speeds up implementation time.
- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.
- The idea of inheritance implements the IS-A relationship. For example, mammal IS A animal, dog IS-A mammal hence dog IS-A animal as well, and so on.

Inheritance in C#

Base and Derived Classes

- A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.
- The syntax used in C# for creating derived classes is as follows:

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

class <derived_class> : <base_class> {
    ...
}
```

Inheritance in C#

Example:

```
using System;

namespace InheritanceApplication {
   class Shape {
      public void setWidth(int w) {
         width = w;
      }
      public void setHeight(int h) {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Derived class
   class Rectangle: Shape {
      public int getArea() {
         return (width * height);
      }
   }
```

```
   class RectangleTester {
      static void Main(string[] args) {
         Rectangle Rect = new Rectangle();

         Rect.setWidth(5);
         Rect.setHeight(7);

         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Total area: 35
```

Access Specifier Types (Remaining)

- Following are the types of access specifier are as follows:
- Protected *(Any protected members of parent class can be accessed from child class but the values cannot be modified).*
- Internal *(Any class is a part of its namespace and is accessible throughout it).*
- Protected internal *(the member 'value' is declared as protected internal therefore it is accessible throughout the class Parent and also in any other class in the same assembly like ABC. It is also accessible inside another class derived from Parent, namely Child which is inside another assembly.).*

Protected (Example)

```
// C# Program to show the use of
// protected Access Modifier
using System;

namespace protectedAccessModifier {

class X {

    // Member x declared
    // as protected
    protected int x;

    public X()
    {
        x = 10;
    }
}

// class Y inherits the
// class X
class Y : X {

    // Members of Y can access 'x'
    public int getX()
    {
        return x;
    }
}
```

```
class Program {

    static void Main(string[] args)
    {
        X obj1 = new X();
        Y obj2 = new Y();

        // Displaying the value of x
        Console.WriteLine("Value of x is : {0}", obj2.getX());
    }
}
}
```

Output:

```
Value of x is : 10
```

Internal (Example-1)

```
namespace internalAccessModifier {

// Declare class Complex as internal
internal class Complex {

    int real;
    int img;

    public void setData(int r, int i)
    {
        real = r;
        img = i;
    }

    public void displayData()
    {
        Console.WriteLine("Real = {0}", real);
        Console.WriteLine("Imaginary = {0}", img);
    }
}
```

```
// Driver Class
class Program {

    // Main Method
    static void Main(string[] args)
    {
        // Instantiate the class Complex
        // in separate class but within
        // the same assembly
        Complex c = new Complex();

        // Accessible in class Program
        c.setData(2, 1);
        c.displayData();
    }
}
}
```

Output:

```
Real = 2
Imaginary = 1
```

Note: In the same code if you add another file, the class *Complex* will not be accessible in that namespace and compiler gives an error.

Internal (Example-2)

```
// C# program inside file xyz.cs
// separate nampespace named xyz
using System;

namespace xyz {

class text {

    // Will give an error during compilation
    Complex c1 = new Complex();
    c1.setData(2, 3);
}
}
```

Output:

```
error CS1519
```

Protected Internal (Example)

```
// Inside file parent.cs
using System;

public class Parent {

    // Declaring member as protected internal
    protected internal int value;
}

class ABC {

    // Trying to access
    // value in another class
    public void testAccess()
    {
        // Member value is Accessible
        Parent obj1 = new Parent();
        obj1.value = 12;
    }
}
```

Output
Value= 12

```
// Inside file GFg.cs
using System;

namespace GFG {

class Child : Parent {

    // Main Method
    public static void Main(String[] args)
    {
        // Accessing value in another assembly
        Child obj3 = new Child();

        // Member value is Accessible
        obj3.value = 9;
        Console.WriteLine("Value = " + obj3.value);
    }
}
}
```

Output
Value= 9

Multiple Inheritance

C# does not support multiple inheritance. However, you can use interfaces to implement multiple inheritance.

Interfaces: An interface (e.g. vehicle) is defined as a syntactical contract that all the classes inheriting the interface should follow.

- The interface defines the 'what' part of the syntactical contract and the deriving classes define the 'how' part of the syntactical contract.
- Interfaces define properties, methods, and events, which are the members of the interface.
- Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Multiple Inheritance

- Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities (discussed in later slides).

- **Declaring Interfaces:** Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration:

```
public interface ITransactions {
    // interface members
    void showTransaction();
    double getAmount();
}
```

Multiple Inheritance using Interfaces

```
using System;

namespace InheritanceApplication {
   class Shape {
      public void setWidth(int w) {
         width = w;
      }
      public void setHeight(int h) {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Base class PaintCost
   public interface PaintCost {
      int getCost(int area);
   }

   // Derived class
   class Rectangle : Shape, PaintCost {
      public int getArea() {
         return (width * height);
      }
      public int getCost(int area) {
         return area * 70;
      }
   }
```

```
   class RectangleTester {
      static void Main(string[] args) {
         Rectangle Rect = new Rectangle();
         int area;

         Rect.setWidth(5);
         Rect.setHeight(7);
         area = Rect.getArea();

         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
         Console.ReadKey();
      }
   }
}
```

Output:

```
Total area: 35
Total paint cost: $2450
```

Polymorphism

- The word polymorphism means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.
- Polymorphism can be static or dynamic. In static polymorphism, the response to a function is determined at the compile time. In dynamic polymorphism, it is decided at run-time.
- Static Polymorphism: The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism.
  - Function overloading
  - Operator overloading

Static Polymorphism

Function Overloading:

- You can have multiple definitions for the same function name in the same scope.
- The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.
- You cannot overload function declarations that differ only by return type.

Static Polymorphism

Function Overloading: The following example shows using function print() to print different data types:

```
using System;

namespace PolymorphismApplication {
   class Printdata {
      void print(int i) {
         Console.WriteLine("Printing int: {0}", i );
      }
      void print(double f) {
         Console.WriteLine("Printing float: {0}" , f);
      }
      void print(string s) {
         Console.WriteLine("Printing string: {0}", s);
      }
      static void Main(string[] args) {
         Printdata p = new Printdata();

         // Call print to print integer
         p.print(5);

         // Call print to print float
         p.print(500.263);

         // Call print to print string
         p.print("Hello C++");
         Console.ReadKey();
      }
   }
}
```

Output:

```
Printing int: 5
Printing float: 500.263
Printing string: Hello C++
```

Static Polymorphism

Operator Overloading:

- You can redefine or overload most of the built-in operators available in C#.
- Thus a programmer can use operators with user-defined types as well.
- Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined.
- Similar to any other function, an overloaded operator has a return type and a parameter list.

Static Polymorphism

- For example, go through the following function:

```
public static Box operator+ (Box b, Box c) {
    Box box = new Box();
    box.length = b.length + c.length;
    box.breadth = b.breadth + c.breadth;
    box.height = b.height + c.height;
    return box;
}
```

- The above function implements the addition operator (+) for a user-defined class Box. It adds the attributes of two Box objects and returns the resultant Box object.

Static Polymorphism Operator Overloading (Example):

```
using System;

namespace OperatorOvlApplication {
   class Box {
      private double length;   // Length of a box
      private double breadth;  // Breadth of a box
      private double height;   // Height of a box

      public double getVolume() {
         return length * breadth * height;
      }
      public void setLength( double len ) {
         length = len;
      }
      public void setBreadth( double bre ) {
         breadth = bre;
      }
      public void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      public static Box operator+ (Box b, Box c) {
         Box box = new Box();
         box.length = b.length + c.length;
         box.breadth = b.breadth + c.breadth;
         box.height = b.height + c.height;
         return box;
      }
   }
```

```
   class Tester {
      static void Main(string[] args) {
         Box Box1 = new Box();    // Declare Box1 of type Box
         Box Box2 = new Box();    // Declare Box2 of type Box
         Box Box3 = new Box();    // Declare Box3 of type Box
         double volume = 0.0;     // Store the volume of a box here

         // box 1 specification
         Box1.setLength(6.0);
         Box1.setBreadth(7.0);
         Box1.setHeight(5.0);

         // box 2 specification
         Box2.setLength(12.0);
         Box2.setBreadth(13.0);
         Box2.setHeight(10.0);

         // volume of box 1
         volume = Box1.getVolume();
         Console.WriteLine("Volume of Box1 : {0}", volume);

         // volume of box 2
         volume = Box2.getVolume();
         Console.WriteLine("Volume of Box2 : {0}", volume);

         // Add two object as follows:
         Box3 = Box1 + Box2;

         // volume of box 3
         volume = Box3.getVolume();
         Console.WriteLine("Volume of Box3 : {0}", volume);
         Console.ReadKey();
      }
   }
}
```

Output:

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Static Polymorphism

Operator Overloading (Overload able and Non-Overload able Operators)

- The following table describes the overload ability of the operators in C#:

Sr.No.	Operators & Description
1	**+, -, !, ~, ++, --** These unary operators take one operand and can be overloaded.
2	**+, -, *, /, %** These binary operators take one operand and can be overloaded.
3	**==, !=, <, >, <=, >=** The comparison operators can be overloaded.
4	**&&, ||** The conditional logical operators cannot be overloaded directly.
5	**+=, -=, *=, /=, %=** The assignment operators cannot be overloaded.
6	**=, ., ?:, ->, new, is, sizeof, typeof** These operators cannot be overloaded.

Sealed Classes

Sealed Classes:

- Sealed classes are used to restrict the users from inheriting the class.
- A class can be sealed by using the sealed keyword.
- The keyword tells the compiler that the class is sealed, and therefore, cannot be extended.
- No class can be derived from a sealed class.
- The following is the syntax of a sealed class :

```
sealed class class_name
{
    // data members
    // methods
    .
    .
    .

}
```

Sealed Classes

Sealed Classes:

- A method can also be sealed, and in that case, the method cannot be overridden.
- However, a method can be sealed in the classes in which they have been inherited.
- If you want to declare a method as sealed, then it has to be declared as virtual in its base class.

Sealed Classes

```
// C# code to define
// a Sealed Class
using System;

// Sealed class
sealed class SealedClass {

    // Calling Function
    public int Add(int a, int b)
    {
        return a + b;
    }
}

class Program {

    // Main Method
    static void Main(string[] args)
    {

        // Creating an object of Sealed Class
        SealedClass slc = new SealedClass();

        // Performing Addition operation
        int total = slc.Add(6, 4);
        Console.WriteLine("Total = " + total.ToString());
    }
}
```

Output:
Total = 10

Now, if it is tried to inherit a class from a sealed class then an error will be produced stating that " It cannot be derived from a **Sealed class**.

```
// C# code to show restrictions
// of a Sealed Class
using System;

class Bird {

}

// Creating a sealed class
sealed class Test : Bird {
}

// Inheriting the Sealed Class
class Example : Test {
}

// Driver Class
class Program {

    // Main Method
    static void Main()
    {
    }
}
```

Output:

Error CS0509
'Example' : cannot derive from sealed type 'Test'

Dynamic Polymorphism

- C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it.
- Abstract classes: these classes (e.g. Human) contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.
- Here are the rules about abstract classes:

  1. You cannot create an instance of an abstract class
  2. You cannot declare an abstract method outside an abstract class
  3. When a class is declared sealed, it cannot be inherited, abstract classes cannot be declared sealed.

Dynamic Polymorphism

Virtual Functions:

- When you have a function defined in a class that you want to be implemented in an inherited class(es), you use virtual functions.
- The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.
- Dynamic polymorphism is implemented by:
  - Abstract classes
  - Virtual functions

Dynamic Polymorphism

```
using System;

namespace PolymorphismApplication {
   class Shape {
      protected int width, height;

      public Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      public virtual int area() {
         Console.WriteLine("Parent class area :");
         return 0;
      }
   }
   class Rectangle: Shape {
      public Rectangle( int a = 0, int b = 0): base(a, b) {

      }
      public override int area () {
         Console.WriteLine("Rectangle class area :");
         return (width * height);
      }
   }
   class Triangle: Shape {
      public Triangle(int a = 0, int b = 0): base(a, b) {
      }
      public override int area() {
         Console.WriteLine("Triangle class area :");
         return (width * height / 2);
      }
   }
   class Caller {
      public void CallArea(Shape sh) {
         int a;
         a = sh.area();
         Console.WriteLine("Area: {0}", a);
      }
   }
   class Tester {
      static void Main(string[] args) {
         Caller c = new Caller();
         Rectangle r = new Rectangle(10, 7);
         Triangle t = new Triangle(10, 5);

         c.CallArea(r);
         c.CallArea(t);
         Console.ReadKey();
      }
   }
}
```

Output:

```
Rectangle class area:
Area: 70
Triangle class area:
Area: 25
```

LINQ

- The acronym LINQ stands for Language Integrated Query.
- Microsoft's query language is fully integrated and offers easy data access from in-memory objects, databases, XML documents, and many more.
- It is through a set of extensions, LINQ ably integrates queries in C# and Visual Basic.
- Developers across the world have always encountered problems in querying data because of the lack of a defined path and need to master a multiple of technologies like SQL, Web Services, XQuery, etc.

LINQ

- Introduced in Visual Studio 2008 and designed by Anders Hejlsberg, LINQ (Language Integrated Query) allows writing queries even without the knowledge of query languages like SQL, XML etc. LINQ queries can be written for diverse data types.

LINQ - Example

```
using System;
using System.Linq;

class Program {
    static void Main() {

        string[] words = {"hello", "wonderful", "LINQ", "beautiful", "world"};

        //Get only short words
        var shortWords = from word in words where word.Length <= 5 select word;

        //Print each word out
        foreach (var word in shortWords) {
            Console.WriteLine(word);
        }

        Console.ReadLine();
    }
}
```

Output:

```
hello
LINQ
world
```

Lambda Expression

- The term ‘Lambda expression’ has derived its name from ‘lambda’ calculus which in turn is a mathematical notation applied for defining functions.
- Lambda expressions as a LINQ equation’s executable part translate logic in a way at run time so it can pass on to the data source conveniently.
- However, lambda expressions are not just limited to find application in LINQ only.
- These expressions are expressed by the following syntax.

```
(Input parameters) ⇒ Expression or statement block
```

Lambda Expression

- Here is an example of a lambda expression:
- y ⇒ y * y
- The above expression specifies a parameter named y and that value of y is squared.
- However, it is not possible to execute a lambda expression in this form.
- Example of a lambda expression in C# is shown in next slide.

Lambda Expression - Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace lambdaexample {
    class Program {

        delegate int del(int i);
        static void Main(string[] args) {

            del myDelegate = y => y * y;
            int j = myDelegate(5);
            Console.WriteLine(j);
            Console.ReadLine();
        }
    }
}
```

Output:

Threads

- A thread is defined as the execution path of a program.
- Each thread defines a unique flow of control.
- If your application involves complicated and time-consuming operations, then it is often helpful to set different execution paths or threads, with each thread performing a particular job.
- The life cycle of a thread starts when an object of the System.Threading.Thread class is created and ends when the thread is terminated or completes execution.
- The following are the various states in the life cycle of a thread –
- **The Un-started State** - It is the situation when the instance of the thread is created but the Start method is not called.

Threads

The Ready State: It is the situation when the thread is ready to run and waiting CPU cycle.

The Not Runnable State: - A thread is not executable, when

- Sleep method has been called
- Wait method has been called
- Blocked by I/O operations

The Dead State: It is the situation when the thread completes execution or is aborted.

Threads - Example

The following is an example showing how to create a thread:

```
using System;
using System.Threading;

namespace Demo {
   class Program {
      public static void ThreadFunc() {
         Console.WriteLine("Child thread starts");
      }

      static void Main(string[] args) {
         ThreadStart childref = new ThreadStart(ThreadFunc);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadKey();
      }
   }
}
```

Output

```
In Main: Creating the Child thread
Child thread starts
```

Multithreading

- Threads are **lightweight processes**.
- One common example of use of thread is implementation of concurrent programming by modern operating systems.
- Use of threads saves wastage of CPU cycle and increase efficiency of an application.
- So far we wrote the programs where a single thread runs as a single process which is the running instance of the application.
- However, this way the application can perform one job at a time.
- To make it execute more than one task at a time, it could be divided into smaller threads.

Multithreading

- **The Main Thread:**
- As told earlier, the **System.Threading.Thread** class is used for working with threads.
- It allows creating and accessing individual threads in a multithreaded application.
- The first thread to be executed in a process is called the **main** thread.
- When a C# program starts execution, the main thread is automatically created.
- The threads created using the **Thread** class are called the child threads of the main thread.
- You can access a thread using the **CurrentThread** property of the Thread class.

Multithreading - Example

```
using System;
using System.Threading;

namespace MultithreadingApplication {
   class MainThreadProgram {
      static void Main(string[] args) {
         Thread th = Thread.CurrentThread;
         th.Name = "MainThread";

         Console.WriteLine("This is {0}", th.Name);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
This is MainThread
```