Day 4 C#

Authorized & published by Summitworks Technologies Inc

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Agenda Day 4

- Access modifiers and Constructors
- OOPS Concepts
  - Introduction to OOPS
  - Class and Objects
  - Inheritance
    - Type casting of reference type
    - Static and Dynamic binding
    - Abstract Class
  - Encapsulation
  - Polymorphism
  - Method Overloading and Method Overriding
- Interface
- Collections
  - Generic Collections
  - Non-Generic Collections
- Delegates
- Anonymous methods and Lambda expressions
- Extension Methods
- More on Classes
  - Partial Class
  - Sealed Class
  - Static Class

Access Modifiers

Access Modifiers are the keywords which are used to define an accessibility level for all types and type members.

By specifying an access level for all types and type members, we can control that whether they can be accessed in other classes or in current assembly or in other assemblies based on our requirements.

Following are the different type of access modifiers available in C# programming language.

- Public
- Private
- Protected
- Internal

Access Modifiers

Using these four access modifiers, we can specify a following six levels of accessibility for all types and type members based on our requirements:

Access Modifier	Description
public	It is used to specifies that access is not restricted.
private	It is used to specifies that access is limited to the containing type.
protected	It is used to specifies that access is limited to the containing type or types derived from the containing class.
internal	It is used to specifies that access is limited to current assembly.
protected internal	It is used to specifies that access is limited to the current assembly or types derived from the containing class.
private protected	It is used to specifies that access is limited to the containing class or types derived from the containing class within the current assembly.

Public Access Modifier

- **Public** modifier is used to specify that access is not restricted, so the defined type or member can be accessed by any other code in current assembly or another assembly that references it.
- Example of defining a members with **public** modifier in c# programming language.

```
2 references
class User
{
    public string Name;
    public string Location;
    public int Age;
    1 reference
    public void GetUserDetails()
    {

        Console.WriteLine("Name: {0}", Name);
        Console.WriteLine("Location: {0}", Location);
        Console.WriteLine("Age: {0}", Age);
    }
}
0 references
```

Accessible in all other classes

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        User u = new User();
        u.Name = "Suresh Dasari";
        u.Location = "Hyderabad";
        u.Age = 32;
        u.GetUserDetails();
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }

}
```

Accessing the public properties in another class

Private Access Modifier

Private modifier is used to specify that access is limited to the containing type so the defined type or member can only be accessed by the code in same class or structure.

Example of defining a members with **private** modifier in c# programming language.

```
class User
{
    private string Name;
    private string Location;
    private int Age;
    private void GetUserDetails()
    {
        Console.WriteLine("Name: {0}", Name);
        Console.WriteLine("Location: {0}", Location);
        Console.WriteLine("Age: {0}", Age);
    }
}
```

Accessible with in the same class

```
class Program
{
    static void Main(string[] args)
    {
        User u = new User();
        // Complier Error
        // These are inaccessible due to private specifier
        u.Name = "Suresh Dasari";
        u.Location = "Hyderabad";
        u.Age = 32;
        u.GetUserDetails();
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
}
```

Protected Access Modifier

Protected modifier is used to specify that access is limited to the containing type or types derived from the containing class so the type or member can only be accessed by code in the same class or in a derived class.

Following is the example of defining a members with **protected** modifier in c# programming language.

```
class User
{
    protected string Name;
    protected string Location;
    protected int Age;
    protected void GetUserDetails()
    {
        Console.WriteLine("Name: {0}", Name);
        Console.WriteLine("Location: {0}", Location);
        Console.WriteLine("Age: {0}", Age);
    }
}
```

```
class Program : User
{
    static void Main(string[] args)
    {
        User u = new User();
        Program p = new Program();
        // Complier Error
        // protected members can only accessible with derived classes
        //u.Name = "Suresh Dasari";
        p.Name = "Suresh Dasari";
        p.Location = "Hyderabad";
        p.Age = 32;
        p.GetUserDetails();
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
}
```

Class Program inherits class User

Instantiating the Derived class

Accessing the protected properties of the User class using the derived class reference

Internal Access Modifier

Internal modifier is used to specify that access is limited to current assembly so the type or member can be accessed by any code in the same assembly, but not from other assembly.

Following is the example of defining a members with **internal** modifier in c# programming language.

```
class User
{
    internal string Name;
    internal string Location;
    internal int Age;
    internal void GetUserDetails()
    {
        Console.WriteLine("Name: {0}", Name);
        Console.WriteLine("Location: {0}", Location);
        Console.WriteLine("Age: {0}", Age);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        User u = new User();
        u.Name = "Suresh Dasari";
        u.Name = "Suresh Dasari";
        u.Location = "Hyderabad";
        u.Age = 32;
        u.GetUserDetails();
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
    }
}
```

Internal properties are accessible within the same assembly

Internal Access Modifier

If you add another file, the class *Complex* will not be accessible in that namespace and compiler gives an error.

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

namespace xyz {

class text {

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

Cannot access the internal type in another namespace

Protected Internal Access Modifier

Access is limited to the current assembly or the derived types of the containing class. It means access is granted to any class which is derived from the containing class within or outside the current Assembly.

Following is the example of defining a members with **protected internal** modifier in c# programming language.

```
// 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;
    }
}
```

```
// 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);
    }
}
}
```

Private Protected Access Modifier

The **private protected** modifier is available from version **7.2** and it is used to specify that access is limited to the containing class or types derived from the containing class within the current assembly, so the type or member can be accessed by code in the same class or in a derived class within the base class assembly.

Following is the example of defining a members with **private protected** modifier in c# programming language.

```
// C# Program to show use of
// the private protected
// Accessibility Level
using System;

namespace PrivateProtectedAccessModifier {

class Parent {

    // Member is declared as private protected
    private protected int value;

    // value is Accessible only inside the class
    public void setValue(int v)
    {
        value = v;
    }
    public int getValue()
    {
        return value;
    }
}
```

```
class Child : Parent {

    public void showValue()
    {
        // Trying to access value
        // Inside a derived class

        Console.WriteLine("Value = " + value);
        // value is accesible
    }
}
```

```
// Driver Code
class Program {

    // Main Method
    static void Main(string[] args)
    {
        Parent obj = new Parent();

        // obj.value = 5;
        // Also gives an error

        // Use public functions to assign
        // and use value of the member 'value'
        obj.setValue(4);
        Console.WriteLine("Value = " + obj.getValue());
    }
}
}
```

Introduction to OOPS concepts

- Object-Oriented Programming (OOP) is a programming structure where programs are organized around objects as opposed to action and logic.
- It is a programming methodology that uses Objects to build a system or web applications using programming languages like C#, Vb.net etc.
- Objects plays a very important role because it hides the implementation details and exposed only the needed functionalities and related stuff that is required to adopt it.
- We can access class properties and methods by creating class object.
- OOPS contains list of elements that are very helpful to make object oriented programming stronger.
  - Encapsulation
  - Abstraction
  - Inheritance and
  - Polymorphism

Class

- A class is a collection of method and variables.
- It is a blueprint that defines the data and behavior of a type.
- If a class does not declare a static, you are required to create an instance/object of a class stating what operation can be performed on that object.
- Objects are instances of a class.
- The methods and variables that constitute a class are called members of the class.

Class with example

- We can define a class using the class keyword and the class body enclosed by a pair of curly braces, as shown in the following example:

```
public class Bike
{
    //your code goes here..
}
```

- The keyword class is preceded by the access level.
- The public access modifier allows anyone can create objects from this class.
- The name of the class follows the keyword class.
- The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods and events on a class are collectively referred to as class members.

Methods

- Method is an object's behavior.

For example, if you consider "Bike" as a class then its behavior is to get bike color, engine, mileage etc. Here is the example of Method:

```
public class Bike
{
    //here is some properties of class Bike
    public string color;
    public string engine;
    public int mileage;

    //here is some behavior/methods of class Bike
    public string GetColor()
    {
        return "red";
    }
    public int GetMileage()
    {
        return 65;
    }
}
```

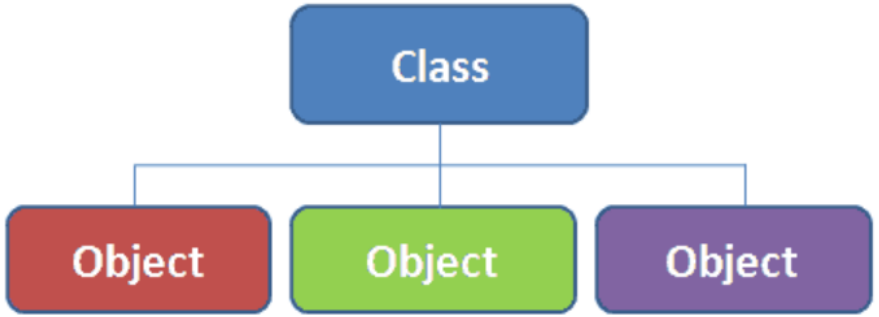
In above example *GetColor()* and *GetMileage()* are the methods considered as a object's behavior.

What is an Object?

- An object is a real-world entity/runtime entity that keeps together property states and behaviors
- An object can be created by using the new keyword to allocate memory for the class in heap, the object is called an instance and its address will be stored in the object in stack memory.
- When an object is created without the new operator, a memory will not be allocated in the heap, in other words, an instance will not be created and the object in the stack contains the value null.
- It is simple to create an object and access the class. We can create object of class via single line of code as shown below.

```
//It also considered as an "Instance of a Bike Class"
Bike objBike = new Bike();

//Accessing Bike class methods
objBike.GetColor();
objBike.GetMileage();
```



Encapsulation

- Encapsulation is a process to hide internal details to an object
- In object oriented programming methodology it prevents access to implementation details.
- Encapsulation is a technique used to protect the information in an object from another object. Hide the data for security such as making the variables private and expose the property to access the private data that will be public.
- Encapsulation is implemented by using access specifiers.
- An access specifier defines the scope and visibility of a class member. Available access specifiers are public, private, protected, internal etc.

Encapsulation with an example

How we can achieve Encapsulation?

We can achieve Encapsulation by using ***private*** access modifier as shown in below example method.

```
class Bike
{
    //here is some properties of class Bike
    public string color ="Red";

    public int mileage = 100;

    private string formula ="a*b";

    //here is some behavior/methods of class Bike
    2 references
    public string GetColor()
    {
        return color;
    }
    2 references
    public int GetMileage()
    {

        return mileage;
    }
    1 reference
    private string GetEngineMakeFormula()
    {

        return formula;
    }
```

Encapsulating the property formula

```
public class Program
{
    public static void Main(string[] args)
    {
        Bike objBike = new Bike();
        Console.WriteLine("Bike mileage is : " + objBike.GetMileage()); //accessible outside "Bike"
        Console.WriteLine("Bike color is : " + objBike.GetColor()); //accessible outside "Bike"
        //we can't call this method as it is inaccessible outside "Bike"
        //objBike.GetEngineMakeFormula(); //commented because we can't access it
        Console.Read();
    }
}
```

Abstraction

Abstraction is the process of providing only essential information to the outside real world and hiding overall background details to present an object. It relies on the separation of interface and implementation.

For example, with “Bike” as an example, we have no access to the piston directly, we can use *start button* to run the piston. Just imagine if a bike manufacturer allows direct access to piston, it would be very difficult to control actions on the piston. That’s the reason why a bike provider separates its internal implementation from its external interface.

Abstraction with an example

```
class Bike
{
    //here is some properties of class Bike
    public string color ="Red";

    public int mileage = 100;

    private string formula ="a*b";

    //here is some behavior/methods of class Bike
    2 references
    public string GetColor()
    {
        return color;
    }
    2 references
    public int GetMileage()
    {

        return mileage;
    }
    1 reference
    private string GetEngineMakeFormula()
    {

        return formula;
    }
    public string DisplayEngineFormula()
    {
        return GetEngineMakeFormula();
    }
}
```

Accessing the private member in another method is Abstraction

```
public class Program
{
    public static void Main(string[] args)
    {
        Bike objBike = new Bike();
        Console.WriteLine("Bike mileage is : " + objBike.GetMileage()); //accessible outside "Bike"
        Console.WriteLine("Bike color is : " + objBike.GetColor()); //accessible outside "Bike"
        //we can't call this method as it is inaccessible outside "Bike"
        //objBike.GetEngineMakeFormula(); //commented because we can't access it
        Console.WriteLine("Bike color is : " + objBike.DisplayMakeFormula()); //accessible outside
        Console.Read();
    }
}
```

Inheritance

- Inheritance is a feature of object-oriented programming. It allows code reusability when a class includes property of another class it is known as inheritance.
- Inheritance in OOP allows us to create a new class using an existing one by extending one class to another.

```
2 references
public class Base
{
    0 references
    public Base()
    {
        Console.WriteLine("Constructor of Base Class");
    }
    1 reference
    public void DisplayMessage()
    {
        Console.WriteLine("Parent class method");
    }
}
3 references
public class Child : Base
{
    1 reference
    public Child()
    {
        Console.WriteLine("Constructor of Child class");
    }

}
```

```
class Program
{
    0 references
    static void Main(string[] args)
    {

        Child objChild = new Child();
         //Child class don't have DisplayMessage() method but we inherited from "Base" class
        objChild.DisplayMessage();
        Console.ReadKey();
    }
}
```

Polymorphism

- The word *Polymorphism* means having many forms.
- Polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
- It allows you to invoke methods of a derived class through base class reference during runtime.
- In polymorphism, we will declare methods with same name and different parameters in same class or methods with same name and same parameters in different classes.
- Polymorphism has the ability to provide the different implementation of methods that are implemented with the same name.

Types of Polymorphism – Method Overloading

There are two types of polymorphism:

1. Compile Time Polymorphism (Early Binding or Overloading or static binding):

   Compile time polymorphism means we will declare a method with same name and different parameter/signature because of this we will perform different tasks with same method name in the same class is called compile time polymorphism.

   Advantage: Execution will be fast because everything about the method is known to compiler during compilation.
   Disadvantage: It has lack of flexibility.

Example of Method Overriding

```
namespace ConsoleApplication5
{
    public class PrentClass
    {
        public void sum(int a, int b)
        {
            Console.WriteLine(a + b);
        }

        public void sum(int a, int b, int c)
        {
            Console.WriteLine(a + b + c);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PrentClass obj = new PrentClass();
            obj.sum(10, 20);
            obj.sum(10, 20, 20);
            Console.ReadLine();
        }
    }
}
```

Method Overloading

Types of Polymorphism – Method Overriding

2. Runtime polymorphism (Late Binding or Overriding or dynamic binding):

- Runtime polymorphism means we will declare a method with same name and same parameter/signature in the different class is called runtime polymorphism.
- In the runtime polymorphism we can override a method in base class by creating a similar function in derived class this can be achieved by using inheritance principle and using virtual & override keywords. We can declare methods with the **virtual** keyword then you can override those methods using **override** keyword in the derived class.

Example of Method Overriding

```
namespace ConsoleApplication5
{
    public class PrentClass
    {
        public virtual void Display()
        {
            Console.WriteLine("Prent class method");
        }
    }
    public class childClass : PrentClass
    {
        public override void Display()
        {
            Console.WriteLine("child class method");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            PrentClass obj = new childClass();
            obj.Display();
            Console.ReadLine();
        }
    }
}
```

Method Overriding

Interfaces

- An interface only contains declarations of method, events and properties
- An interface cannot include private members
- Interface members are public by default
- Interface cannot contains fields, operator, constructors
- We can define interface using the keyword interface
- To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member
- If a class inherits from an interface, it must provide implementation for all interface members
- A class or struct can implement multiple interfaces.
- Interface can inherit from other interfaces.
- We cannot create an instance for an interface but an interface reference variable can point to a derived class object

Interfaces

We can define interface using the **interface** keyword, other classes can implement ILog by providing an implementation of the Ilog() method.

```
interface ILog
{
                    void Log(string msgToLog);
}
```

```
class ConsoleLog: ILog
{
                public void Log(string msgToPrint)
    {
                Console.WriteLine(msgToPrint);
    }
}

class FileLog :ILog
{
                public void Log(string msgToPrint)
    {
                File.AppendText(@"C:\Log.txt").Write(msgToPrint);
    }
}
```

Interfaces

In the code the consoleLog class implements the Ilog interface to log the string on the console, whereas the FileLog class implements the Ilog to log the string into a text file.

Instantiate Object:

```
ILog log = new ConsoleLog();

//Or

ILog log = new FileLog();
```

Explicit Implementation:

We can implement interfaces explicitly by prefixing the interface name with the method name.

```
class ConsoleLog: ILog
{
        public void ILog.Log(string msgToPrint) // explicit implementatio
    {
        Console.WriteLine(msgToPrint);
    }
}
```

OOPS Concepts- Encapsulation

- Encapsulation is a way of hiding the data into a single unit called class, by doing this we can protect our data

- Using access modifiers we can achieve encapsulating our data from the outside world

Private If we create any function or variable inside the class as private then it will not be accessible to outside the class.

Protected members are visible only in derived classes

Internal members are visible only in derived classes that are located in the same assembly as the base class. They are not visible in derived classes located in a different assembly from the base class

Public members are visible in derived classes and are part of the derived class public interface

OOPS Concepts- Inheritance

Inheritance:

- Inheritance is a concept in which you define parent classes and child classes
- It allows you to define a child class that reuses (inherits), extends, or modifies the behavior of a parent class
- The class whose members are inherited is called the *base class*
- The class that inherits the members of the base class is called the *derived class*
- C# and .NET support *single inheritance* only
- inheritance is transitive, which allows you to define an inheritance hierarchy for a set of types
- Not all members of a base class are inherited by derived classes
- The following members are not inherited from the base class
    - Static constructors, which initialize the static data of a class
    - Instance constructors
    - Finalizers, which are called by the runtime's garbage collector to destroy instances of a class.

OOPS Concepts- Inheritance

- Let’s see an example for Inheritance

```
public class Animal
{
    public void Greet()
    {
        Console.WriteLine("Hello, I'm some sort of animal!");
    }
}

public class Dog : Animal
{

}
```

```
Animal animal = new Animal();
animal.Greet();
Dog dog = new Dog();
dog.Greet();
```

```
public class Animal
{
    public virtual void Greet()
    {
        Console.WriteLine("Hello, I'm some sort of animal!");
    }
}

public class Dog : Animal
{
    public override void Greet()
    {
        Console.WriteLine("Hello, I'm a dog!");
    }
}
```

```
public override void Greet()
{
    base.Greet();
    Console.WriteLine("Yes I am - a dog!");
}
```

OOPS Concepts- Inheritance

- A member's accessibility affects its visibility for derived classes as follows:
  - Private members are visible only in derived classes that are nested in their base class

Example:

```
public class A
{
    private int value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return this.value;
        }
    }
}

public class C : A
{
//     public int GetValue()
//     {
//         return this.value;
//     }
}

public class Example
{
     public static void Main(string[] args)
     {
          var b = new A.B();
          Console.WriteLine(b.GetValue());
     }
}
// The example displays the following output:
//        10
```

OOPS Concepts- Inheritance

- Derived classes can also *override* inherited members by providing an alternate implementation by marking the member in the base class using the virtual keyword
- The base keyword is used to access an overridden function member from the subclass and calling a base class constructor

Abstract Class

- Abstract modifier in a class declaration indicates that the class is intended only to be a base class of other classes
- A class declared as abstract can never be instantiated only its concrete subclasses can be instantiated
- An abstract class may contain abstract methods and accessors
- A class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors

Abstract methods have the following features:

- An abstract method is implicitly a virtual method and provides only the method declaration

```
public abstract void MyMethod();
```

- The implementation is provided by a method override, which is a member of a sub or derived class.

Polymorphism

- It allows you to invoke methods of derived class through base class reference during runtime
- It has the ability for classes to provide different implementations of methods that are called through the same name

Polymorphism is of two types:

- Compile time polymorphism/Overloading
- Runtime polymorphism/Overriding

Compile time polymorphism:

- Compile time polymorphism is method and operator overloading which is also called early binding
- The methods performs different task at the different input parameters

Runtime polymorphism:

- Runtime time polymorphism is done using inheritance and virtual functions
- Method overriding is called runtime polymorphism which is also called late binding

Method Overloading

- It allows the programmer to define several methods with the same name, as long as they take a different set of parameters. When you use the classes of the .NET framework, you will soon realize that method overloading is used all over the place. A good example of this,

```
class SillyMath
{
    public static int Plus(int number1, int number2)
    {
        return Plus(number1, number2, 0);
    }

    public static int Plus(int number1, int number2, int number3)
    {
        return number1 + number2 + number3;
    }
}
```

```
class SillyMath
{
    public static int Plus(int number1, int number2)
    {
        return Plus(number1, number2, 0);
    }

    public static int Plus(int number1, int number2, int number3)
    {
        return Plus(number1, number2, number3, 0);
    }

    public static int Plus(int number1, int number2, int number3, int number4)
    {
        return number1 + number2 + number3 + number4;
    }
}
```

Collections

- Collections are similar to Arrays, it provides a more flexible way of working with a group of objects
- The group of objects can grow and shrink dynamically as the needs of the application
- Collections are Key value pairs so that you can retrieve the object quickly by using the key
- A collection is a class, so you must declare an instance of the class before you can add elements to that collection
- Collections are of two types
  - Non-Generic Collection:
    - The non-generic collections (ArrayList, Hashtable, SortedList, Queue etc.) store elements internally in 'object' arrays which, can of course, store any type of data.
  - Generic Collection:
    - Generic collections (List<T>, Dictionary<T, U>, SortedList<T, U>, Queue<T> etc) store elements internally in arrays of their actual types and so no boxing or casting is ever required

Collections- Non Generic

ArrayList:

- ArrayList can store elements of any datatype
- ArrayList resizes automatically as you add the elements
- ArrayList values must be cast to appropriate data types before using it
- ArrayList can contain multiple null and duplicate items
- ArrayList can be accessed using foreach or for loop or indexer
- Using the new keyword we create an object to access the elements of the array list

Adding element to the ArrayList

```
ArrayList arryList1 = new ArrayList();
arryList1.Add(1);
arryList1.Add("Two");
arryList1.Add(3);
arryList1.Add(4.5);
```

Collections- Non Generic

Pass ArrayList as an argument

```
    //
    // Create an ArrayList and add two ints.
    //
    ArrayList list = new ArrayList();
    list.Add(5);
    list.Add(7);
    //
    // Use ArrayList with method.
    //
    Example(list);
}

static void Example(ArrayList list)
{
    foreach (int i in list)
    {
        Console.WriteLine(i);
    }
}
```

Combining two ArrayList

```
ArrayList arryList1 = new ArrayList();
arryList1.Add(1);
arryList1.Add("Two");
arryList1.Add(3);
arryList1.Add(4.5);

ArrayList arryList2 = new ArrayList();
arryList2.Add(100);
arryList2.Add(200);

//adding entire arryList2 into arryList1
arryList1.AddRange(arryList2);
```

Generic Collection

- A generic collection gets all the benefit of generics. It doesn't need to do boxing and unboxing while storing or retrieving items and so performance is improved
- Generics denotes with angel bracket <>
- Compiler applies specified type for generics at compile time
- Generics can be applied to interface, abstract class, method, static method, property, event, delegate and operator
- Generic are type safe. You get compile time errors if you try to use a different type of data than the one specified in the definition
- Generics performs faster by not doing boxing & unboxing

Generic Collection

The following are widely used generic collections

Generic Collections	Description
List<T>	Generic List<T> contains elements of specified type. It grows automatically as you add elements in it.
Dictionary<TKey,TValue>	Dictionary<TKey,TValue> contains key-value pairs.
SortedList<TKey,TValue>	SortedList stores key and value pairs. It automatically adds the elements in ascending order of key by default.
Hashset<T>	Hashset<T> contains non-duplicate elements. It eliminates duplicate elements.
Queue<T>	Queue<T> stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an Enqueue() method to add values and a Dequeue() method to retrieve values from the collection.
Stack<T>	Stack<T> stores the values as LIFO (Last In First Out). It provides a Push() method to add a value and Pop() & Peek() methods to retrieve values.

Generic Collection

Generic List<T>

- List<T> stores elements of the specified type and it grows automatically
- List<T> can store multiple null and duplicate elements
- List<T> can be access using indexer, for loop or foreach statement
- List<T> is ideal for storing and retrieving large number of elements
- List<T> can be assigned to **IList<T>** or **List<T>** type of variable. It provides more helper method When assigned to List<T> variable

Generic Collection

Add Elements into List:

```
public static void Main()
{
    IList<int> intList = new List<int>();
    intList.Add(10);
    intList.Add(20);
    intList.Add(30);
    intList.Add(40);

    Console.WriteLine(intList.Count);

    IList<string> strList = new List<string>();
    strList.Add("one");
    strList.Add("two");
    strList.Add("three");
    strList.Add("four");
    strList.Add("four");
    strList.Add(null);
    strList.Add(null);
```

Add elements using object initializer syntax

```
IList<int> intList = new List<int>(){ 10, 20, 30, 40 };

//Or

IList<Student> studentList = new List<Student>() {
                new Student(){ StudentID=1, StudentName="Bill"},
                new Student(){ StudentID=2, StudentName="Steve"},
                new Student(){ StudentID=3, StudentName="Ram"},
                new Student(){ StudentID=1, StudentName="Moin"}
            };
```

Generic Collection

- The AddRange() method adds all the elements from another collection

```
IList<int> intList1 = new List<int>();
intList1.Add(10);
intList1.Add(20);
intList1.Add(30);
intList1.Add(40);

List<int> intList2 = new List<int>();
intList2.Add(50);
intList2.Add(60);
intList2.Add(70);
intList2.Add(80);

intList2.AddRange(intList1);
Console.WriteLine(intList2.Count);
```

Generic Collection

Accessing elements from the List using For loop

```
List<int> intList2 = new List<int>();
intList2.Add(50);
intList2.Add(60);
intList2.Add(70);
intList2.Add(80);
intList2.AddRange(intList1);

Console.WriteLine(intList2.Count);

for(int i = 0; i<= intList2.Count; i++)
{
    Console.WriteLine(i);
}
```

Accessing elements from List using foreach

```
//Accessing elements using foreach loop
List<int> primes = new List<int>(new int[] { 2, 3, 5 });

// See if List contains 3.
foreach (int number in primes)
{
    if (number == 3) // Will match once.
    {
        Console.WriteLine("Contains 3");
    }
}
```

Generic Collection

Copy Array to the List

```
//Coping Array to the List

int[] arr = new int[3]; // New array with 3 elements.
arr[0] = 2;
arr[1] = 3;
arr[2] = 5;

List<int> list = new List<int>(arr); // Copy to List.
Console.WriteLine(list.Count);
```

Search an element in the List

```
//Searching element in List
List<int> primes = new List<int>(new int[] { 19, 23, 29 });
int index = primes.IndexOf(23); // Exists.
Console.WriteLine(index);

index = primes.IndexOf(10); // Does not exist.
Console.WriteLine(index);
```

Generic Collection

- **Contains:** This method scans a List. It searches for a specific element to see if that element occurs at least once in the collection.
- **Exists** returns whether a List element is present. This is an instance method that returns true or false depending on whether any element matches the Predicate parameter. We invoke this method with a lambda expression.

```
//Using Exist method
bool exists = primes.Exists(element => element > 10);
Console.WriteLine(exists);
```

- **Find** is used to search an element in the List it is more efficient than using a for loop. We invoke this method with a lambda expression.

```
// Finds first element greater than 20
int result = primes.Find(item => item > 20);

Console.WriteLine(result);
```

Generic Collection

- **Join string list.** We use **string.Join** on a List of strings. This is helpful when we need to turn several strings into one comma-delimited string. It requires the **ToArray** instance method on List. This ToArray is not an extension method.

```
//Join String using string.Join
List<string> cities = new List<string>();
cities.Add("New York");
cities.Add("Mumbai");
cities.Add("Berlin");
cities.Add("Istanbul");

// Join strings into one CSV line.
string line = string.Join(",", cities.ToArray());
Console.WriteLine(line);
```

Generic Collection

Convert List to single string using StringBuilder

```
List<string> cats = new List<string>(); // Create new list of strings
cats.Add("Devon Rex"); // Add string 1
cats.Add("Manx"); // 2
cats.Add("Munchkin"); // 3
cats.Add("American Curl"); // 4
cats.Add("German Rex"); // 5

StringBuilder builder = new StringBuilder();
foreach (string cat in cats) // Loop through all strings
{
    builder.Append(cat).Append("|"); // Append string to StringBuilder

    string result = builder.ToString(); // Get string from StringBuilder
    Console.WriteLine(result);
}
```

Generic Collection

We can Remove element based on the value with Remove() or based on the index with RemoveAt().

```
List<string> dogs = new List<string>()
{ "maltese", "otterhound",
    "rottweiler","bulldog","whippet" };
dogs.Remove("bulldog"); // Remove bulldog
foreach (string dog in dogs)
{
    Console.WriteLine(dog);
}
// Contains: maltese, otterhound, rottweiler, whippet

dogs.RemoveAt(1); // Remove second dog

foreach (string dog in dogs)
{
    Console.WriteLine(dog);
}
// Contains: maltese, rottweiler, whippet
```

Generic Collection

CopyTo () copies the list elements to an array, before coping we must make sure the array is properly allocated.

```
var list = new List<int>() { 5, 6, 7 };

// Create an array with length of three.
int[] array = new int[list.Count];

// Copy the list to the array.
list.CopyTo(array);

// Display.
Console.WriteLine(array[0]);
Console.WriteLine(array[1]);
Console.WriteLine(array[2]);
```

Generic Collection

Converting a List to an array

```
        List<string> list1 = new List<string>();
        list1.Add("one");
        list1.Add("two");
        list1.Add("three");
        list1.Add("four");
        list1.Add("five");

        // Convert to string array.
        string[] array1 = list1.ToArray();
        Test(array1);
    }

    static void Test(string[] array)
    {
        Console.WriteLine("Array received: " + array.Length);
    }
```

Generic Collection

Dictionary:

- A Dictionary stores Key-Value pairs where the key must be unique
- When defining the Dictionary we have provide the type of the key and the type of the value
- Before adding a KeyValuePair into a dictionary, check that the key does not exist using the ContainsKey() method
- Dictionary cannot include duplicate or null keys, where as values can be duplicated or set as null. Keys must be unique otherwise it will throw a runtime exception.
- Use a foreach or for loop to iterate a dictionary
- Use dictionary indexer to access individual item

Generic Collection

Important Methods of Dictionary

Method	Description
Add	Adds an item to the Dictionary collection.
Add	Add key-value pairs in Dictionary<TKey, TValue> collection.
Remove	Removes the first occurance of specified item from the Dictionary<TKey, TValue>.
Remove	Removes the element with the specified key.
ContainsKey	Checks whether the specified key exists in Dictionary<TKey, TValue>.
ContainsValue	Checks whether the specified key exists in Dictionary<TKey, TValue>.
Clear	Removes all the elements from Dictionary<TKey, TValue>.
TryGetValue	Returns true and assigns the value with specified key, if key does not exists then return false.

Generic Collection

Adding elements into Dictionary

```
static void Main(string[] args)
{
    //Defining and Adding elements into Dictionary
    Dictionary<string, long> phonebook = new Dictionary<string, long>();
    phonebook.Add("Alex", 4154346543);
    phonebook["Jessica"] = 4159484588;
}
```

```
IDictionary<string, long> phonebook1= new Dictionary<string, long>();

phonebook1.Add(new KeyValuePair<string, long>("John", 4017653929));
phonebook1.Add(new KeyValuePair<String, long>("Jamie", 6012347898));

//The following is also valid
phonebook1.Add("Alvin", 6053892345);
```

Generic Collection

Searching elements in the Dictionary

ContainsKey: It searches the given Key is present in the Dictionary if it's there it returns true else returns false.

```
Dictionary<string, int> dictionary = new Dictionary<string, int>();

dictionary.Add("apple", 1);
dictionary.Add("windows", 5);

// See whether Dictionary contains this string.
if (dictionary.ContainsKey("apple"))
{
    int value = dictionary["apple"];
    Console.WriteLine(value);
}

// See whether it contains this string.
if (!dictionary.ContainsKey("acorn"))
{
    Console.WriteLine(false);
}
```

Generic Collection-Dictionary

Accessing elements in Dictionary using foreach loop will evaluate each element individually, and an index is not needed.

- Foreach loop returns an Enumeration in the result set.

- A foreach-loop on KeyValuePairs is faster than the looping over Keys and accessing values in the loop body.

```
//Accessing using foreach
Dictionary<string, int> d = new Dictionary<string, int>()
{
        {"cat", 2},
        {"dog", 1},
        {"llama", 0},
        {"iguana", -1}
};
// Loop over pairs with foreach.
foreach (KeyValuePair<string, int> pair in d)
{
    Console.WriteLine("{0}, {1}", pair.Key, pair.Value);
}
// Use var keyword to enumerate dictionary.
foreach (var pair in d)
{
    Console.WriteLine("{0}, {1}", pair.Key, pair.Value);
}
```

Generic Collection

We cannot perform sorting method directly on Dictionary rather we can sort the keys in a separate List collection and loop it.

```
//Sorting Dictionary

// Acquire keys and sort them.
var list = d.Keys.ToList();
list.Sort();

// Loop through keys.
foreach (var key in list)
{
    Console.WriteLine("{0}: {1}", key, d[key]);
}
```

Delegates

- Delegate is a type which safely encapsulates a method
- A Delegate is a type that references a method
- Once a delegate is assigned to a method, it behaves exactly like that method
- The Delegate method can be used like any other method with parameter and return value
- The type of the delegate is defined by name of the delegate
- A delegate does not care about class of the object that it references, only matter is that the method signature should be the same as of delegate

Delegates have following properties,

- Delegates are similar to C++ function pointer but it is type safe in nature.
- Delegate allows method to pass as an argument.
- Delegate can be chained together.
- Multiple methods can be called on a single event.

Delegates

Syntax of a delegate

Step 1 - Declaration

Delegate is getting declared here.

```
Modifer  delegate  return_type  delegate_name ( [Parameter....])
```

Step 2 - Instantiation

Object of delegate is getting created as passing method as argument

```
Delegate_name delg_object_name = new Delegate_name( method_name);
```

Step 3 - Invocation

Delegate is getting called here.

```
Delg_object_name([parameter....]);
```

Delegates

```
public delegate int AddDelegate(int num1, int num2);

static void Main(string[] args)
{
    // Creating method of delegate, and passing Function Add as argument.
    AddDelegate funct1 = new AddDelegate(Add);
    // Invoking Delegate.....
    int k = funct1(7, 2);
    Console.WriteLine(" Sumation = {0}", k);
    Console.Read();

}
```

Anonymous Method

- Anonymous method is a method without any name
- Anonymous method can be defined using the delegate keyword
- Anonymous method must be assigned to a delegate
- Anonymous method can be passed as a parameter
- Anonymous method can be used as event handlers
- Anonymous method can access outer variables or functions

Limitations of Anonymous Method:

- It cannot contain jump statement like goto, break or continue.
- It cannot access ref or out parameter of an outer method
- It cannot be used on the left side of the is operator

Anonymous Method

- Inline Anonymous Method

```
static void Main(string[] args)
{
    SqaureRoot doSquare;
    //Inline Anonymous method
    doSquare = x => Math.Sqrt(x);
    Console.WriteLine(doSquare(122));
```

Anonymous Method

- Let us see an example for Inline and multi-line anonymous methods

```
public delegate double SqaureRoot(int value1);
public delegate int Calculate(int value1, int value2, int value3);
class Demo1
{
    static void Main(string[] args)
    {
        SqaureRoot doSquare;
        doSquare = x => Math.Sqrt(x);
        Console.WriteLine(doSquare(122));
        Calculate cal;
        cal = (x, y, z) =>
        {
            Console.WriteLine("Multiple");
            var result = x * y * z;
            return result;
        };
```

Lambda Expression

- A lambda expression is an unnamed methods written in the place of a delegate instance.
- The lambda expression is a shorter way of representing anonymous method using some special syntax

The following anonymous method checks if student is teenager or not.

```
delegate(Student s) { return s.Age > 12 && s.Age < 20; };
```

The above anonymous method can be represented using a Lambda Expression as below

```
s => s.Age > 12 && s.Age < 20
```

Extension Method

- Extension methods enable you to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.
- Extension methods can be added to your own custom class, .NET framework classes, or third party classes or interfaces
- An extension method is a special kind of static method, but they are called as if they were instance methods on the extended type.
- An extension method is a static method of a static class, where the "this" modifier is applied to the first parameter. The type of the first parameter will be the type that is extended.
- Extension methods are only in scope when you explicitly import the namespace into your source code with a using directive.

Extension Method

Important points for the use of Extension Methods:

- An extension method must be defined in a top-level static class.
- An extension method with the same name and signature as an instance method will not be called.
- Extension methods cannot be used to override existing methods.
- The concept of extension methods cannot be applied to fields, properties or events.
- Overuse of extension methods is not a good style of programming.

Extension Method

Let us see an example for extension method IsGreaterThan() is an extension method for int type, which returns true if the value of the int variable is greater than the supplied integer parameter.

Example: Extension Method

```
int i = 10;

bool result = i.IsGreaterThan(100); //returns false
```

The IsGreaterThan() method is not a method of int data type (Int32 struct). It is an extension method written by the programmer for the int data type.

The IsGreaterThan() extension method will be available throughout the application by including the namespace in which it has been defined.

Extension Method

- To define an extension method, first of all, define a static class

we have created an IntExtensions class under the ExtensionMethods namespace in the following example. The IntExtensions class will contain all the extension methods applicable to int data type.

Example: Create a Class for Extension Methods

```
namespace ExtensionMethods
{
    public static class IntExtensions
    {

    }
}
```

The first parameter of the extension method specifies the type on which the extension method is applicable. Since we are going to use this extension method on int type the first parameter must be int type preceded with this modifier.

Extension Method

Example: Define an Extension Method

```
namespace ExtensionMethods
{
    public static class IntExtensions
     {
        public static bool IsGreaterThan(this int i, int value)
        {
            return i > value;
        }
    }
}
```

Now, you can include the ExtensionMethods namespace wherever you want to use this extension method.

Extension Method

```
using ExtensionMethods;

class Program
{
    static void Main(string[] args)
    {
        int i = 10;

        bool result = i.IsGreaterThan(100);

        Console.WriteLine(result);
    }
}
```

Extension Method

- Extension methods are additional custom methods which were originally not included with the class.
- Extension methods can be added to custom, .NET Framework or third party classes, structs or interfaces.
- The first parameter of the extension method must be of the type for which the extension method is applicable, preceded by the **this** keyword.
- Extension methods can be used anywhere in the application by including the namespace of the extension method.

Partial Class and Partial Methods

Class in C# resides in a separate physical file with a .cs extension. C# provides the ability to have a single class implementation in multiple .cs files using the ***partial modifier*** keyword.

The *partial* modifier can be applied to a class, method, interface or structure.

Example : MyPartialClass splits into two files, PartialClassFile1.cs and PartialClassFile2.cs

Example: PartialClassFile1.cs

```
public partial class MyPartialClass
{
    public MyPartialClass()
    {
    }

    public void Method1(int val)
    {
        Console.WriteLine(val);
    }
}
```

Example: PartialClassFile2.cs

```
public partial class MyPartialClass
{
    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}
```

Partial Class and Partial Methods

- MyPartialClass in PartialClassFile1.cs defines the constructor and one public method, Method1, whereas PartialClassFile2 has only one public method, Method2.
- The compiler combines these two partial classes into one class as below:

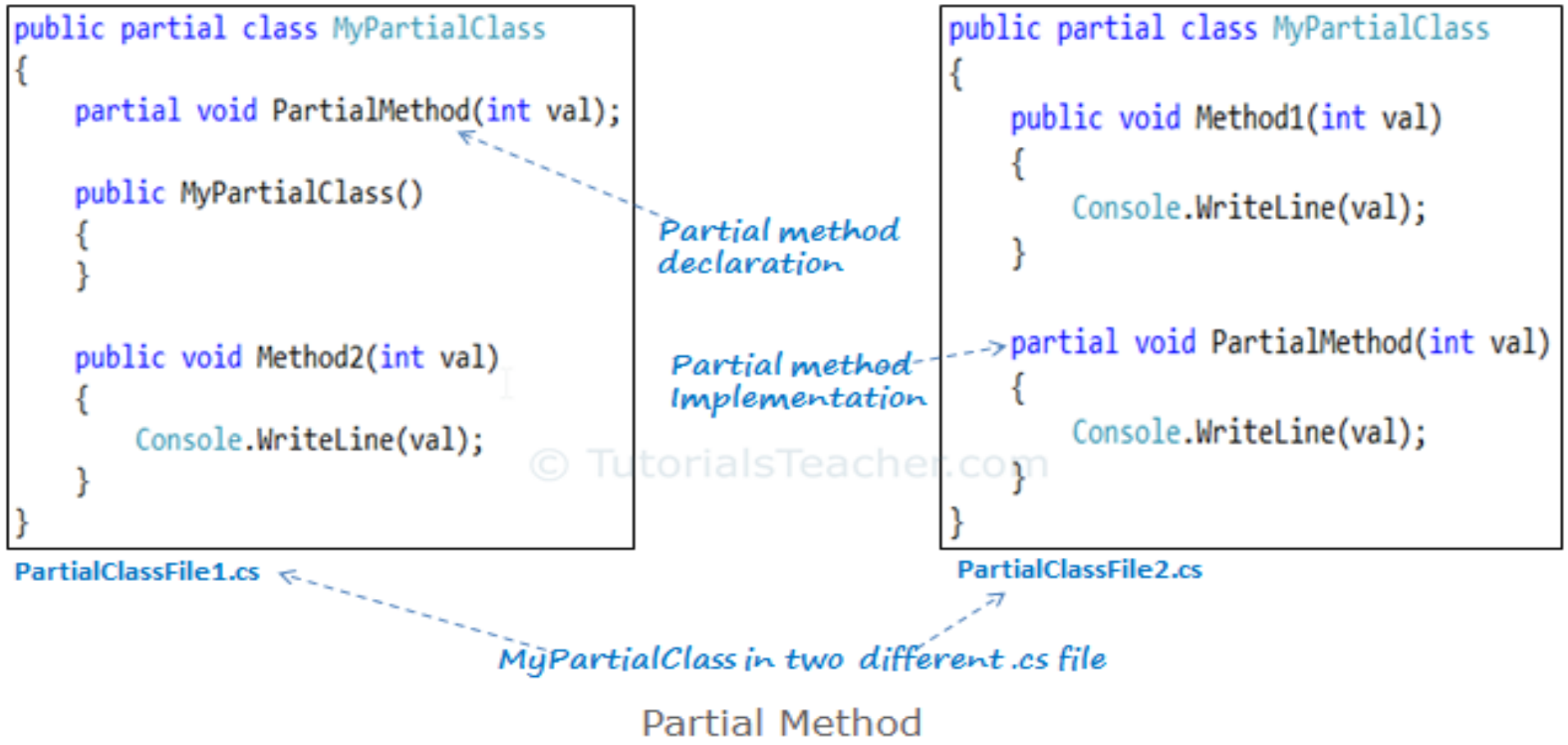
```
public class MyPartialClass
{
    public MyPartialClass()
    {
    }

    public void Method1(int val)
    {
        Console.WriteLine(val);
    }

    public void Method2(int val)
    {
        Console.WriteLine(val);
    }
}
```

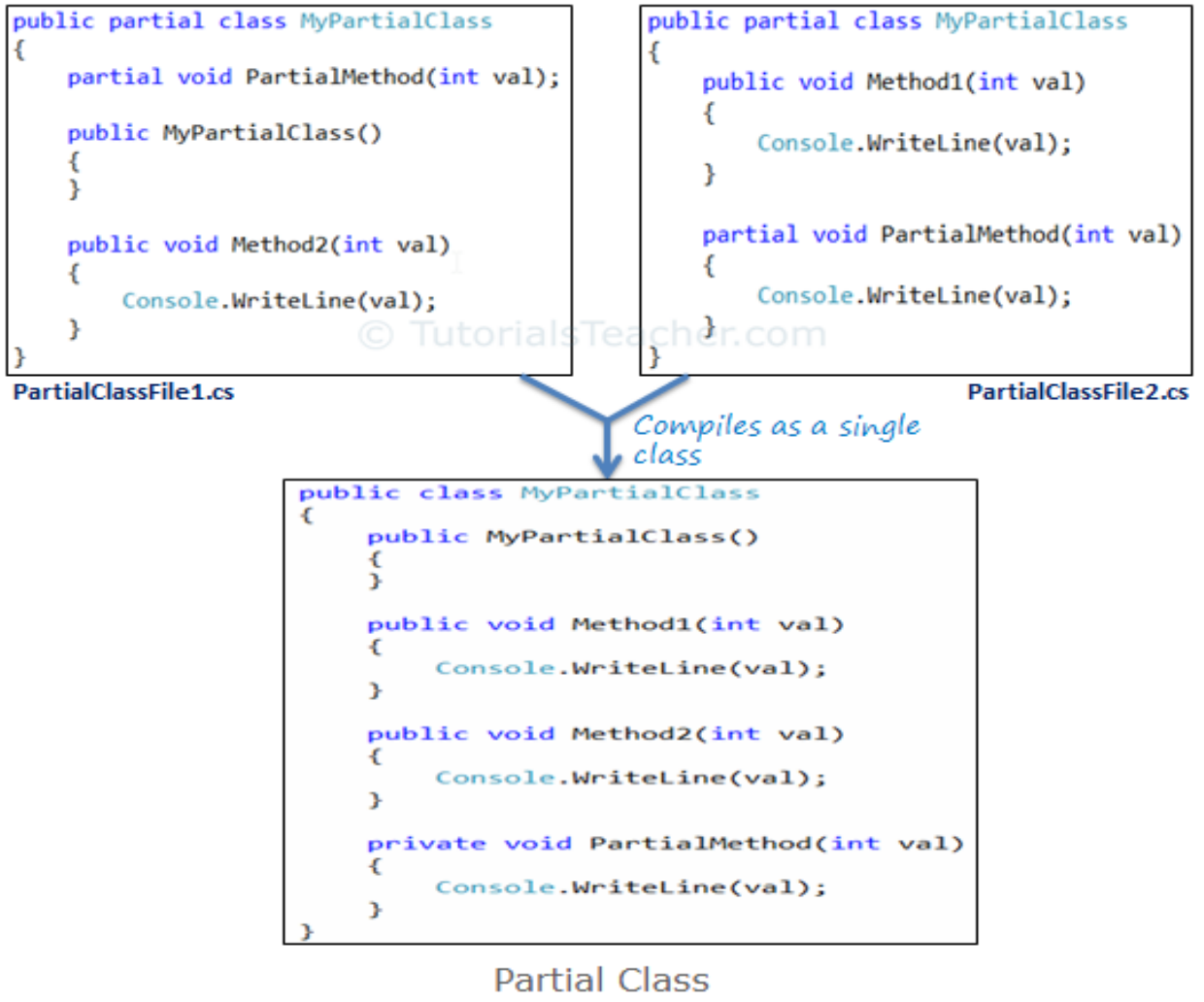
Partial Class and Partial Methods

The following image illustrates partial class and partial methods:



Partial Class and Partial Methods

The compiler combines the two partial classes into a single final class:



Partial Class and Partial Methods

Advantages of Partial Class:

- Multiple developers can work simultaneously with a single class in separate files.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. For example, Visual Studio separates HTML code for the UI and server side code into two separate files: .aspx and .cs files.

Partial Methods

- A partial method must be declared in one of the partial classes.
- A partial method may or may not have an implementation.
- If the partial method doesn't have an implementation in any part then the compiler will not generate that method in the final class.

Requirement of partial method:

- The partial method declaration must began with the partial modifier.
- The partial method can have a ref but not an out parameter.
- Partial methods are implicitly private methods.
- Partial methods can be static methods.
- Partial methods can be generic.

Sealed Class

- Sealed class in C# with the sealed keyword cannot be inherited. In the same way, the sealed keyword can be added to the method.
- When you use sealed modifiers in C# on a method, then the method loses its capabilities of overriding. The sealed method should be part of a derived class and the method must be an overridden method.

Let us see an example of sealed class:

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

namespace Demo {
    class Program {
        static void Main(string[] args) {
            Result ob = new Result();
            string str = ob.Display();

            Console.WriteLine(str);
            Console.ReadLine();
        }
    }

    public sealed class Result {
        public string Display() {
            return "Passed";
        }
    }
}
```

Static Class

Static Class:

- The static modifier makes an item non-instantiable, it means the static item cannot be instantiated. If the static modifier is applied to a class then that class cannot be instantiated using the new keyword.
- If the static modifier is applied to a variable, method or property of class then they can be accessed without creating an object of the class, just use className.propertyName, className.methodName.

Static Constructor:

A static or non-static class can have a static constructor without any access modifiers like public, private, protected, etc.

- A static constructor in a non-static class runs only once when the class is instantiated for the first time.
- A static constructor in a static class runs only once when any of its static members accessed for the first time.