

Session: **13**

Generics and Iterators

- ◆ Define and describe generics
- ◆ Explain creating and using generics
- ◆ Explain iterators



For Aptech Centre Use Only

- ◆ Generics are a kind of parameterized data structures that can work with value types as well as reference types.
- ◆ You can define a class, interface, structure, method, or a delegate as a generic type in C#.

Example

- ◆ Consider a C# program that uses an array variable of type Object to store a collection of student names.
- ◆ The names are read from the console as value types and are boxed to enable storing each of them as type Object.
- ◆ In this case, the compiler cannot verify the data stored against its data type as it allows you to cast any value to and from Object.
- ◆ If you enter numeric data, it will be accepted without any verification.
- ◆ To ensure type-safety, C# introduces generics, which has a number of features including the ability to allow you to define generalized type templates based on which the type can be constructed later.

Namespaces, Classes, and Interfaces for Generics 1-3

- ◆ There are several namespaces in the .NET Framework that facilitate creation and use of generics which are as follows:

`System.Collections.ObjectModel`

- This allows you to create dynamic and read-only generic collections.

`System.Collections.Generic`

- The namespace consists of classes and interfaces that allow you to define customized generic collections.

◆ **Classes:**

- ❖ The `System.Collections.Generic` namespace consists of classes that allow you to create type-safe collections.

Namespaces, Classes, and Interfaces for Generics 2-3

- ❖ The following table lists some of the widely used classes of the `System.Collections.Generic` namespace:

Class	Descriptions
<code>Comparer</code>	Is an abstract class that allows you to create a generic collection by implementing the functionalities of the <code>IComparer</code> interface
<code>Dictionary.KeyCollection</code>	Consists of keys present in the instance of the <code>Dictionary</code> class
<code>Dictionary.ValueCollection</code>	Consists of values present in the instance of the <code>Dictionary</code> class
<code>EqualityComparer</code>	Is an abstract class that allows you to create a generic collection by implementing the functionalities of the <code>IEqualityComparer</code> interface

Namespaces, Classes, and Interfaces for Generics 3-3

◆ Interfaces

- ◆ The `System.Collections.Generic` namespace consists of interfaces that allow you to create type-safe collections.
- ◆ The following table lists some of the widely used interfaces of the `System.Collections.Generic` namespace:

Interface	Descriptions
<code>IComparer</code>	Defines a generic method <code>Compare()</code> that compares values within a collection
<code>IEnumerable</code>	Defines a generic method <code>GetEnumerator()</code> that iterates over a collection
<code>IEqualityComparer</code>	Consists of methods which check for the equality between two objects

System.Collections.ObjectModel 1-5

- ◆ The System.Collections.ObjectModel namespace consists of classes that can be used to create customized generic collections.
- ◆ The following table shows the classes contained in the System.Collections.ObjectModel namespace:

Classes	Descriptions
Collection<>	Provides the base class for generic collections
KeyedCollection<>	Provides an abstract class for a collection whose keys are associated with values
ReadOnlyCollection<>	Is a read-only generic base class that prevents modification of collection

System.Collections.ObjectModel 2-5

- ◆ The following code demonstrates the use of the `ReadOnlyCollection<>` class:

Snippet

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
class ReadOnly
{
    static void Main(string[] args)
    {
        List<string> objList = new List<string>();
        objList.Add("Francis");
        objList.Add("James");
        objList.Add("Baptista");
        objList.Add("Micheal");
        ReadOnlyCollection<string> objReadOnly = new
        ReadOnlyCollection
        <string>(objList);
        Console.WriteLine("Values stored in the read only
        collection");
        foreach (string str in objReadOnly)
        {
            Console.WriteLine(str);
        }
        Console.WriteLine();
    }
}
```


System.Collections.ObjectModel 3-5

```
Console.WriteLine("Total number of elements in the read  
only collection: " + objReadOnly.Count);  
if (objList.Contains("Francis"))  
{  
    objList.Insert(2, "Peter");  
}  
Console.WriteLine("\nValues stored in the list after  
modification");  
foreach (string str in objReadOnly)  
{  
    Console.WriteLine(str);  
}  
string[] array = new string[objReadOnly.Count * 2];  
objReadOnly.CopyTo(array, 5);  
Console.WriteLine("\nTotal number of values that can be  
stored in the new array: " + array.Length);  
Console.WriteLine("Values in the new array");  
foreach (string str in array)  
{  
    if (str == null)  
    {  
        Console.WriteLine("null");  
    }  
    else  
    {  
        Console.WriteLine(str);  
    }  
}  
}
```

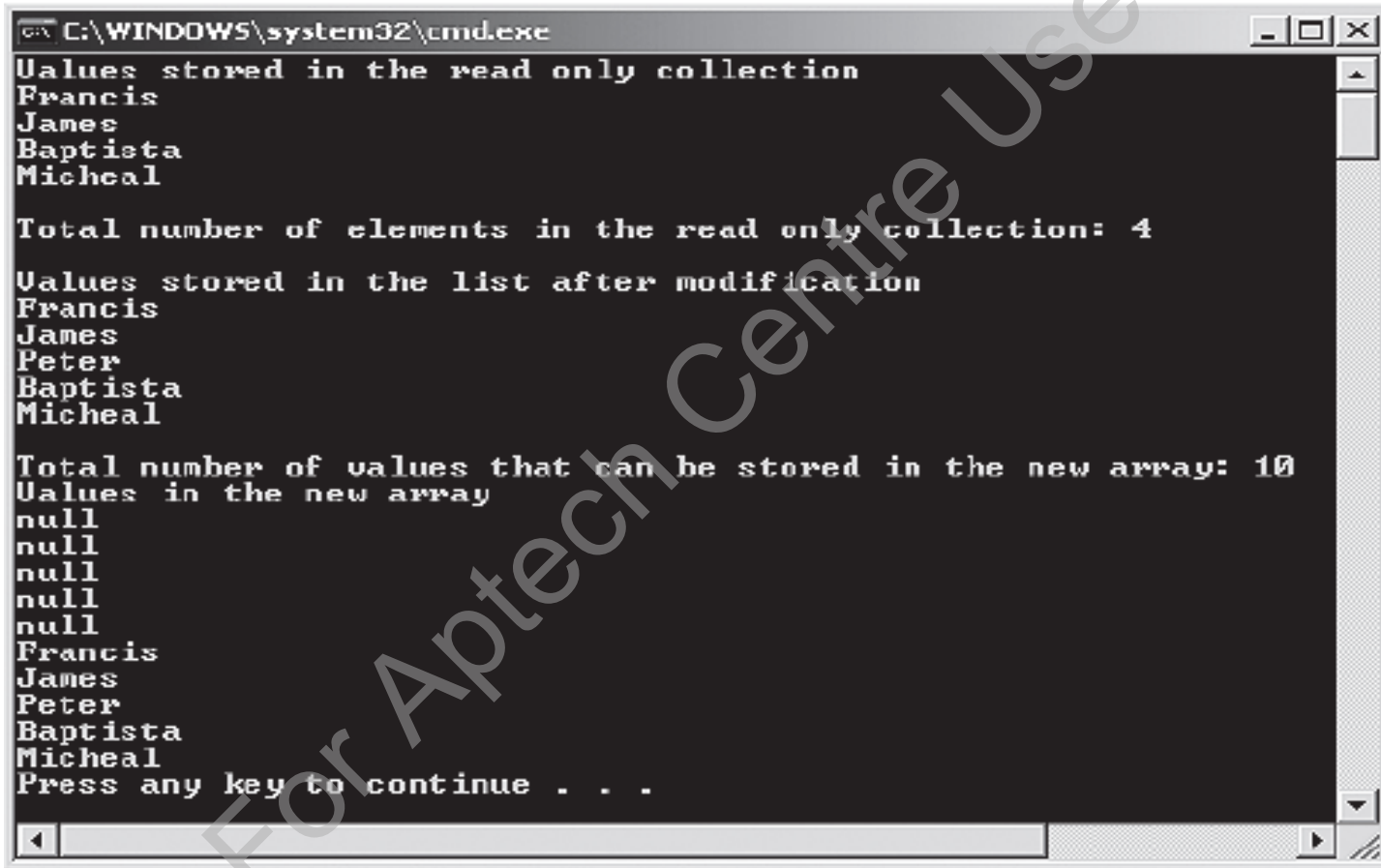
◆ In the code:

- ◆ The `Main()` method of the **ReadOnly** class creates an instance of the `List` class.
- ◆ The `Add()` method inserts elements in the instance of the `List` class.
- ◆ An instance of the `ReadOnlyCollection` class of type `String` is created and the elements stored in the instance of the `List` class are copied to the instance of the `ReadOnlyCollection` class.
- ◆ The `Contains()` method checks whether the `List` class contains the specified element.
- ◆ If the `List` class contains the specified element, **Francis**, then the new element, **Peter**, is inserted at the specified index position, 2.
- ◆ The code creates an array variable that is twice the size of the `ReadOnlyCollection` class.
- ◆ The `CopyTo()` method copies the elements from the `ReadOnlyCollection` class to the array variable from the fifth position onwards.

System.Collections.ObjectModel 5-5

- ◆ The following figure displays the output of the code:

Output



```
C:\WINDOWS\system32\cmd.exe
Values stored in the read only collection
Francis
James
Baptista
Micheal

Total number of elements in the read only collection: 4

Values stored in the list after modification
Francis
James
Peter
Baptista
Micheal

Total number of values that can be stored in the new array: 10
Values in the new array
null
null
null
null
null
Francis
James
Peter
Baptista
Micheal
Press any key to continue . . .
```

Creating Generic Types

- ◆ Following are the features of a generic declaration:

A generic declaration always accepts a **type parameter**, which is a placeholder for the required data type.

The type is specified only when a generic type is referred to or constructed as a type within a program.

The process of creating a generic type begins with a generic type definition containing type parameters that acts like a blueprint.

Later, a generic type is constructed from the definition by specifying actual types as the generic type arguments, which will substitute for the type parameters or the placeholders.

- ◆ Generics ensure type-safety at compile-time.
- ◆ Generics allow you to reuse the code in a safe manner without casting or boxing.
- ◆ A generic type definition is reusable with different types but can accept values of a single type at a time.
- ◆ Apart from reusability, there are several other benefits of using generics. These are as follows:
 - ◆ Improved performance because of low memory usage as no casting or boxing operation is required to create a generic
 - ◆ Ensured strongly-typed programming model
 - ◆ Reduced run-time errors that may occur due to casting or boxing



- ◆ Generic classes define functionalities that can be used for any data type and are declared with a class declaration followed by a **type parameter** enclosed within angular brackets.
- ◆ While declaring a generic class, you can apply some restrictions or constraints to the type parameters by using the `where` keyword. However, applying constraints to the type parameters is optional.
- ◆ Thus, while creating a generic class, you must generalize the data types into the type parameter and optionally decide the constraints to be applied on the type parameter.
- ◆ The following syntax is used for creating a generic class:

Syntax

```
<access_modifier> class <ClassName><<type parameter list>> [where <type parameter constraint clause>]
```

- ◆ `where`,
 - ❖ `access_modifier`: Specifies the scope of the generic class. It is optional.
 - ❖ `ClassName`: Is the name of the new generic class to be created.
 - ❖ `<type parameter list>`: Is used as a placeholder for the actual data type.
 - ❖ `type parameter constraint clause`: Is an optional class or an interface applied to the type parameter with the `where` keyword.

- ◆ The following code creates a generic class that can be used for any specified data type:

Snippet

```
using System;
using System.Collections.Generic;

class General<T>
{
    T[] values;
    int _counter = 0;
    public General(int max)
    {
        values = new T[max];
    }
    public void Add(T val)
    {
        if (_counter < values.Length)
        {
            values[_counter] = val;
            _counter++;
        }
    }
}

public void Display()
```



```
{
    Console.WriteLine("Constructed Class is of type: " +
        typeof(T));
    Console.WriteLine("Values stored in the object of
        constructed class are: ");
    for (int i = 0; i < values.Length; i++)
    {
        Console.WriteLine(values[i]);
    }
}
}
class Students
{
    static void Main(string[] args)
    {
        General<string> objGeneral = new General<string>(3);
        objGeneral.Add("John");
        objGeneral.Add("Patrick");
        objGeneral.Display();
        General<int> objGeneral2 = new General<int>(2);
        objGeneral2.Add(200);
        objGeneral2.Add(35);
        objGeneral2.Display();
    }
}
```

- ◆ In the code:
 - ◆ A generic class definition for **General** is created that takes a type parameter `T`.
 - ◆ The generic class declares a parameterized constructor with an `int` value.
 - ◆ The **Add**() method takes a parameter of the same type as the generic class.
 - ◆ The method **Display**() displays the value type specified by the type parameter and the values supplied by the user through the object.
 - ◆ The **Main**() method of the class `Students` creates an instance of the generic class `General` by providing the type parameter value as `string` and `total` value to be stored as 3.
 - ◆ This instance invokes the **Add**() method which takes student names as values.
 - ◆ These student names are displayed by invoking the **Display**() method.
 - ◆ Later, another object is created of a different data type, `int`, based on the same class definition.
 - ◆ The class definition is generic, we need not change the code now, but can reuse the same code for an `int` data type as well.
 - ◆ Thus, using the same generic class definition, we can create two different lists of data.

Output

Constructed Class is of type: System.String

Values stored in the object of constructed class are:

John

Patrick

Constructed Class is of type: System.Int32

Values stored in the object of constructed class are:

200

35

Constraints on Type Parameters 1-4

- ◆ You can apply constraints on the type parameter while declaring a generic type.
- ◆ A constraint is a restriction imposed on the data type of the type parameter and are specified using the `where` keyword.
- ◆ They are used when the programmer wants to limit the data types of the type parameter to ensure consistency and reliability of data in a collection.
- ◆ The following table lists the types of constraints that can be applied to the type parameter:

Constraints	Descriptions
<code>T : struct</code>	Specifies that the type parameter must be of a value type only except the null value
<code>T : class</code>	Specifies that the type parameter must be of a reference type such as a class, interface, or a delegate
<code>T : new()</code>	Specifies that the type parameter must consist of a constructor without any parameter which can be invoked publicly
<code>T : <base class name></code>	Specifies that the type parameter must be the parent class or should inherit from a parent class
<code>T : <interface name></code>	Specifies that the type parameter must be an interface or should inherit an interface

Constraints on Type Parameters 2-4

The following code creates a generic class that uses a class constraint:

Snippet

```
using System;
using System.Collections.Generic;

class Employee
{
    string _empName;
    int _empID;
    public Employee(string name, int num)
    {
        _empName = name;
        _empID = num;
    }
    public string Name
    {
        get
        {
            return _empName;
        }
    }
    public int ID
    {
        get
        {
            return _empID;
        }
    }
}
```

Constraints on Type Parameters 3-4

```
class GenericList<T> where T : Employee
{
    T[] _name = new T[3];
    int _counter = 0;
    public void Add(T val)
    {
        _name[_counter] = val;
        _counter++;
    }
    public void Display()
    {
        for (int i = 0; i < _counter; i++)
        {
            Console.WriteLine(_name[i].Name + ", " + _name[i].ID);
        }
    }
}
class ClassConstraintDemo
{
    static void Main(string[] args)
    {
        GenericList<Employee> objList = new
        GenericList<Employee>();
        objList.Add(new Employee("John", 100));
        objList.Add(new Employee("James", 200));
        objList.Add(new Employee("Patrich", 300));
        objList.Display();
    }
}
```

Constraints on Type Parameters 4-4

In the code:

- ◆ The class **GenericList** is created that takes a type parameter **T**.
- ◆ This type parameter is applied a class constraint, which means the type parameter can only include details of the **Employee** type.
- ◆ The generic class creates an array variable with the type parameter **T**, which means it can include values of type **Employee**.
- ◆ The `Add()` method consists of a parameter **val**, which will contain the values set in the `Main()` method.
- ◆ Since, the type parameter should be of the **Employee** type, the constructor is called while setting the values in the `Main()` method.

Output

```
John, 100  
James, 200  
Patrich, 300
```


Inheriting Generic Classes 1-2

- ◆ A generic class can be inherited same as any other non-generic class in C# and can act both as a base class or a derived class.
- ◆ While inheriting a generic class in another generic class, you can use the generic type parameter of the base class instead of passing the data type of the parameter.
- ◆ However, while inheriting a generic class in a non-generic class, you must provide the data type of the parameter instead of the base class generic type parameter.
- ◆ The constraints imposed at the base class level must be included in the derived generic class.
- ◆ The following figure displays a generic class as base class:

```
Generic -> Generic

public class Student<T>
{
}

public class Marks<T>: Student<T>
{
}

Generic -> Non-Generic

public class Student<T>
{
}

public class Marks: Student<int>
{
}
```

Inheriting Generic Classes 2-2

- ◆ The following syntax is used to inherit a generic class from an existing generic class:

Syntax

```
<access_modifier> class <BaseClass><<generic type parameter>>{}  
<access_modifier> class <DerivedClass> : <BaseClass><<generic type  
parameter>>{}
```

where,

- ◆ `access_modifier`: Specifies the scope of the generic class.
 - ◆ `BaseClass`: Is the generic base class.
 - ◆ `<generic type parameter>`: Is a placeholder for the specified data type.
 - ◆ `DerivedClass`: Is the generic derived class.
- ◆ The following syntax is used to inherit a non-generic class from a generic class:

Syntax

```
<access_modifier> class <BaseClass><<generic type parameter>>{}  
<access_modifier> class <DerivedClass> : <BaseClass><<type  
parameter value>>{}
```

where,

- ◆ `<type parameter value>`: Can be a data type such as `int`, `string`, or `float`.

- ◆ Generic methods process values whose data types are known only when accessing the variables that store these values.
- ◆ A generic method is declared with the generic type parameter list enclosed within angular brackets.
- ◆ Defining methods with type parameters allow you to call the method with a different type everytime.
- ◆ You can declare a generic method within generic or non-generic class declarations.
- ◆ When you declare a generic method within a generic class declaration, the body of the method refers to the type parameters of both, the method and class declaration.
- ◆ Generic methods can be declared with the following keywords:
 - ◆ **Virtual:** The generic methods declared with the `virtual` keyword can be overridden in the derived class.
 - ◆ **Override:** The generic method declared with the `override` keyword overrides the base class method. However, while overriding, the method does not specify the type parameter constraints since the constraints are overridden from the overridden method.
 - ◆ **Abstract:** The generic method declared with the `abstract` keyword contains only the declaration of the method. Such methods are typically implemented in a derived class.

- ◆ The following syntax is used for declaring a generic method:

Syntax

```
<access_modifier><return_type><MethodName><<type parameter list>>
```

- ◆ where,
 - ◆ `access_modifier`: Specifies the scope of the method.
 - ◆ `return_type`: Determines the type of value the generic method will return.
 - ◆ `MethodName`: Is the name of the generic method.
 - ◆ `<type parameter list>`: Is used as a placeholder for the actual data type.

- ◆ The following code creates a generic method within a non-generic class:

Snippet

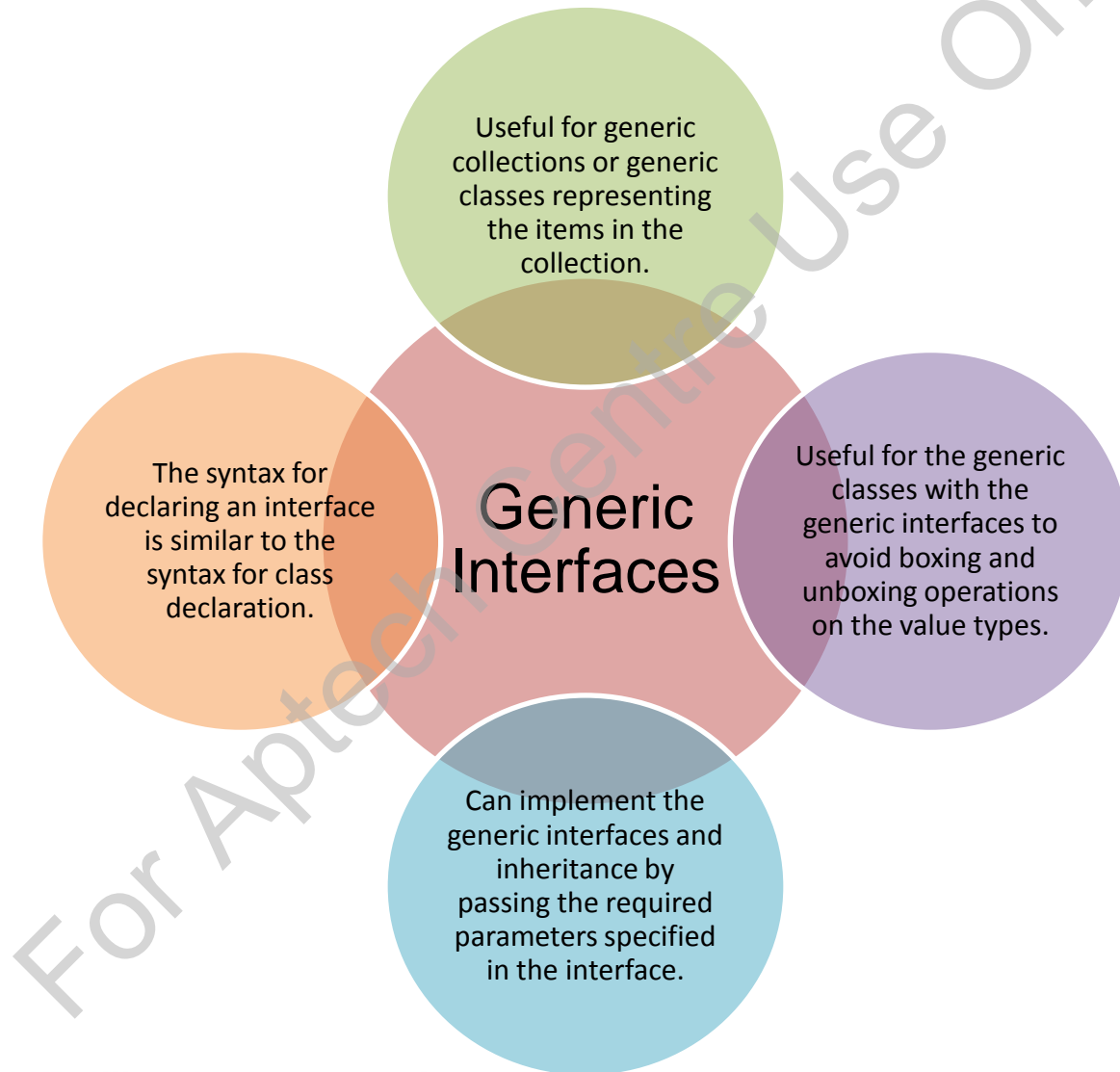
```
using System;
using System.Collections.Generic;
class SwapNumbers{
    static void Swap<T>(ref T valOne, ref T valTwo) {
        T temp = valOne;
        valOne = valTwo;
        valTwo = temp;
    }
    static void Main(string[] args) {
        int numOne = 23;
        int numTwo = 45;
        Console.WriteLine("Values before swapping: " + numOne + " & " + numTwo);
        Swap<int>(ref numOne, ref numTwo);
        Console.WriteLine("Values after swapping: " + numOne + " & " + numTwo);
    }
}
```

- ◆ In the code:
 - ◆ The class **SwapNumbers** consists of a generic method **Swap()** that takes a type parameter **T** within angular brackets and two parameters within parenthesis of type **T**.
 - ◆ The **Swap()** method creates a variable **temp** of type **T** that is assigned the value within the variable **valOne**.
 - ◆ The **Main()** method displays the values initialized within variables and calls the **Swap()** method by providing the type **int** within angular brackets.
 - ◆ This will substitute for the type parameter in the generic method definition and will display the swapped values within the variables.

Output

```
Values before swapping: 23 & 45
Values after swapping: 45 & 23
```

- ◆ Following are the features of generic interfaces:



- ◆ The following syntax is used for creating a generic interface:

Syntax

```
<access_modifier> interface <InterfaceName><<type  
parameter list>> [where <type parameter constraint  
clause>]
```

where,

- ◆ `access_modifier`: Specifies the scope of the generic interface.
- ◆ `InterfaceName`: Is the name of the new generic interface.
- ◆ `<type parameter list>`: Is used as a placeholder for the actual data type.
- ◆ `type parameter constraint clause`: Is an optional class or an interface applied to the type parameter with the `where` keyword.

- ◆ The following code creates a generic interface that is implemented by the non-generic class:

Snippet

```
using System;
using System.Collections.Generic;
interface IMaths<T>{
    T Addition(T valOne, T valTwo);
    T Subtraction(T valOne, T valTwo);
}
class Numbers : IMaths<int>{
    public int Addition(int valOne, int valTwo)
    {
        return valOne + valTwo;
    }
    public int Subtraction(int valOne, int valTwo){
        if (valOne > valTwo){
            return (valOne - valTwo);
        }
        else{
            return (valTwo - valOne);
        }
    }
}
static void Main(string[] args){
    int numOne = 23;
    int numTwo = 45;
    Numbers objInterface = new Numbers();
    Console.WriteLine("Addition of two integer values is: ");
    Console.WriteLine(objInterface.Addition(numOne, numTwo));
    Console.WriteLine("Subtraction of two integer values is: ");
    Console.WriteLine(objInterface.Subtraction(numOne, numTwo));
}
```

◆ In the code:

- ◆ The generic interface **IMaths** takes a type parameter `T` and declares two methods of type `T`.
- ◆ The class **Numbers** implements the interface **IMaths** by providing the type `int` within angular brackets and implements the two methods declared in the generic interface.
- ◆ The `Main()` method creates an instance of the class **Numbers** and displays the addition and subtraction of two numbers.

Output

Addition of two integer values is: 68

Subtraction of two integer values is: 22

Generic Interface Constraints 1-4

- ◆ You can specify an interface as a constraint on a type parameter to enable the members of the interface within, to use the generic class.
- ◆ In addition, it ensures that only the types that implement the interface are used and also specify multiple interfaces as constraints on a single type parameter.



Generic Interface Constraints 2-4

- ◆ The following code creates a generic interface that is used as a constraint on a generic class:

Snippet

```
using System;
using System.Collections.Generic;
interface IDetails
{
    void GetDetails();
}
class Student : IDetails
{
    string _studName;
    int _studID;
    public Student(string name, int num)
    {
        _studName = name;
        _studID = num;
    }
    public void GetDetails()
    {
        Console.WriteLine(_studID + "\t" + _studName);
    }
}
```

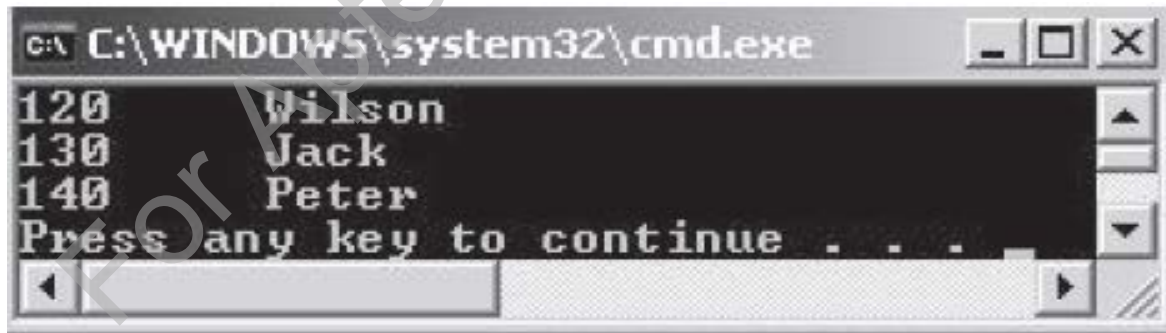
Generic Interface Constraints 3-4

```
class GenericList<T> where T : IDetails
{
    T[] _values = new T [3];
    int _counter = 0;
    public void Add(T val)
    {
        _values[_counter] = val;
        _counter++;
    }
    public void Display()
    {
        for (int i = 0; i < 3; i++)
        {
            _values[i].GetDetails();
        }
    }
}

class InterfaceConstraintDemo
{
    static void Main(string[] args)
    {
        GenericList<Student> objList = new GenericList<Student>();
        objList.Add(new Student("Wilson", 120));
        objList.Add(new Student("Jack", 130));
        objList.Add(new Student("Peter", 140));
        objList.Display();
    }
}
```

Generic Interface Constraints 4-4

- ◆ In the code:
 - ◆ An interface **IDetails** declares a method **GetDetails()**.
 - ◆ The class **Student** implements the interface **IDetails**.
 - ◆ The class **GenericList** is created that takes a type parameter **T**.
 - ◆ This type parameter is applied an interface constraint, which means the type parameter can only include details of the **IDetails** type.
 - ◆ The **Main()** method creates an instance of the class **GenericList** by passing the type parameter value as **Student**, since the class **Student** implements the interface **IDetails**.
- ◆ The following figure shows the output of the code to create a generic interface:



```
C:\WINDOWS\system32\cmd.exe
120 Wilson
130 Jack
140 Peter
Press any key to continue . . .
```

Generic Delegates 1-4

- ◆ Delegates are reference types that encapsulate a reference to a method that has a signature and a return type.
- ◆ Following are the features of a generic delegate:
 - ◆ Delegates can also be declared as generic.
 - ◆ It can be used to refer to multiple methods in a class with different types of parameters.
 - ◆ The number of parameters of the delegate and the referenced methods must be the same.
 - ◆ The type parameter list is specified after the delegate's name in the syntax.

- ◆ The following syntax is used to declare a generic delegate:

Syntax

```
delegate <return_type><DelegateName><type  
parameter list>(<argument_list>);
```

- ◆ where,
 - ◆ `return_type`: Determines the type of value the delegate will return.
 - ◆ `DelegateName`: Is the name of the generic delegate.
 - ◆ `type parameter list`: Is used as a placeholder for the actual data type.
 - ◆ `argument_list`: Specifies the parameter within the delegate.

- ◆ The following code declares a generic delegate:

Snippet

```
using System;
delegate T DelMath<T>(T val);
class Numbers
{
    static int NumberType(int num)
    {
        if(num % 2 == 0)
            return num;
        else
            return (0);
    }
    static float NumberType(float num)
    {
        return num % 2.5F;
    }
    public static void Main(string[] args)
    {
        DelMath<int> objDel = NumberType;
        DelMath<float> objDel2 = NumberType;
        Console.WriteLine(objDel(10));
        Console.WriteLine(objDel2(108.756F));
    }
}
```

- ◆ In the code:
 - ◆ A generic delegate is declared in the **Numbers** class.
 - ◆ In the `Main()` method of the class, an object of the delegate is created, which is referring to the **NumberType()** method and takes the parameter of `int` type.
 - ◆ An `integer` value is passed to the method, which displays the value only if it is an even number.
 - ◆ Another object of the delegate is created in the `Main()` method, which is referring to the **NumberType()** method and takes the parameter of `float` type.
 - ◆ A `float` value is passed to the method, which displays the remainder of the division operation. Therefore, generic delegates can be used for overloaded methods.
- ◆ The following figure shows the output of the code to declare a generic delegate:

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The command prompt has a black background with white text. The output of the program is displayed as three lines: '10', '1.255997', and 'Press any key to continue . . .'. The window has standard Windows window controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

Overloading Methods Using Type Parameters 1-3

- ◆ Methods of a generic class that take generic type parameters can be overloaded.
- ◆ The programmer can overload the methods that use type parameters by changing the type or the number of parameters.
- ◆ However, the type difference is not based on the generic type parameter, but is based on the data type of the parameter passed.



Overloading Methods Using Type Parameters 2-3

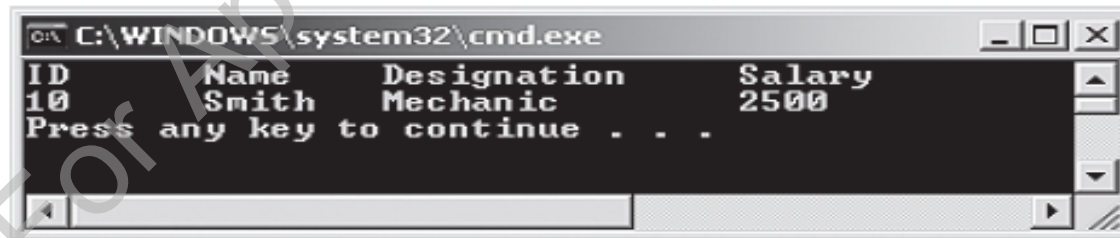
- ◆ The following code demonstrates how to overload methods that use type parameters:

Snippet

```
using System;
using System.Collections;
using System.Collections.Generic;
class General<T, U>{
    T _valOne;
    U _valTwo;
    public void AcceptValues(T item) {
        _valOne = item;
    }
    public void AcceptValues(U item) {
        _valTwo = item;
    }
    public void Display() {
        Console.Write(_valOne + "\t" + _valTwo);
    }
}
class MethodOverload{
    static void Main(string[] args) {
        General<int, string> objGenOne = new General<int, string>();
        objGenOne.AcceptValues(10);
        objGenOne.AcceptValues("Smith");
        Console.WriteLine("ID\tName\tDesignation\tSalary");
        objGenOne.Display();
        General<string, float> objGenTwo = new General<string, float>();
        objGenTwo.AcceptValues("Mechanic");
        objGenTwo.AcceptValues(2500);
        Console.Write("\t");
        objGenTwo.Display();
        Console.WriteLine();
    }
}
```

Overloading Methods Using Type Parameters 3-3

- ◆ In the code:
 - ◆ The **General** class has two overloaded methods with different type parameters.
 - ◆ In the `Main()` method, the instance of the **General** class is created. The class is initialized by specifying the data type for the generic parameters `T` and `U` as `string` and `int` respectively.
 - ◆ The overloaded methods are invoked by specifying appropriate values.
 - ◆ The methods store these values in the respective variables defined in the **General** class.
 - ◆ These values indicate the ID and name of the employee.
 - ◆ Another instance of the **General** class is created specifying the type of data the class can contain as `string` and `float`.
 - ◆ The overloaded methods are invoked by specifying appropriate values.
 - ◆ The methods store these values in the respective variables defined in the **General** class.
 - ◆ These values indicate the designation and salary of the employee.
- ◆ The following figure shows the output of the code to overload methods using type parameters:



```
C:\WINDOWS\system32\cmd.exe
ID      Name      Designation      Salary
10      Smith      Mechanic         2500
Press any key to continue . . .
```

Overriding Virtual Methods in Generic Class 1-3

- ◆ Methods in generic classes can be overridden same as the method in any non-generic class.
- ◆ To override a method in the generic class, the method in the base class must be declared as `virtual` and this method can be overridden in the derived class, using the `override` keyword as shown in the following code:

Snippet

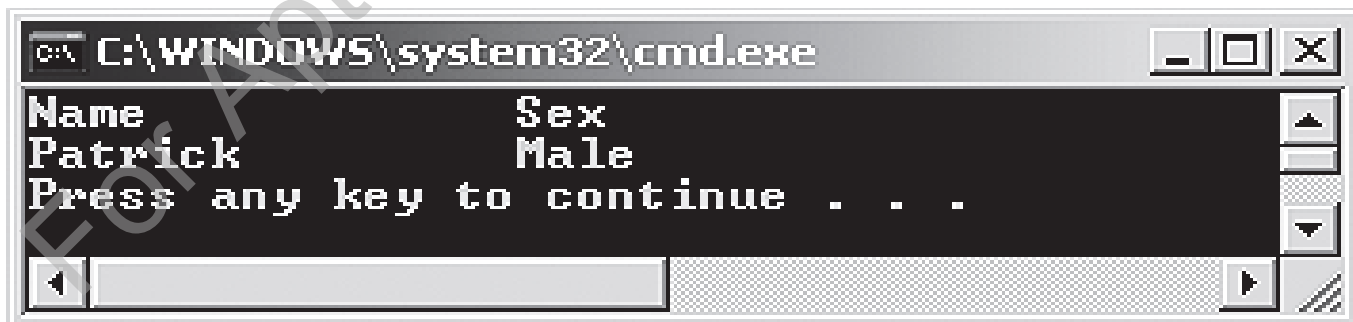
```
using System;
using System.Collections;
using System.Collections.Generic;
class GeneralList<T>
{
    protected T ItemOne;
    public GeneralList(T valOne)
    {
        ItemOne = valOne;
    }
    public virtual T GetValue()
    {
        return ItemOne;
    }
}
```


Overriding Virtual Methods in Generic Class 2-3

```
    }  
    }  
class Student<T> : GeneralList<T>  
{  
    public T Value;  
    public Student(T valOne, T valTwo) : base (valOne)  
    {  
        Value = valTwo;  
    }  
    public override T GetValue()  
    {  
        Console.Write (base.GetValue() + "\t\t");  
        return Value;  
    }  
}  
class StudentList  
{  
    public static void Main()  
    {  
        Student<string> objStudent = new Student<string>("Patrick",  
            "Male");  
        Console.WriteLine("Name\t\tSex");  
        Console.WriteLine(objStudent.GetValue());  
    }  
}
```

Overriding Virtual Methods in Generic Class 3-3

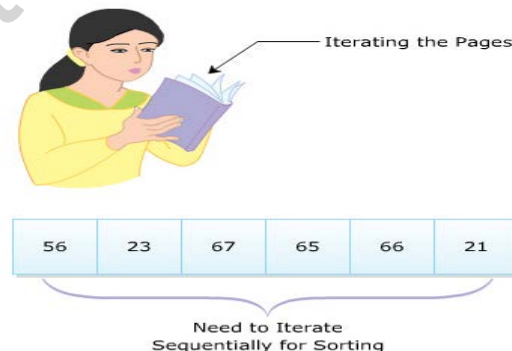
- ◆ In the code:
 - ◆ The **GeneralList** class consists of a constructor that assigns the name of the student.
 - ◆ The **GetValue()** method of the **GeneralList** class is overridden in the **Student** class.
 - ◆ The constructor of the **Student** class invokes the base class constructor by using the base keyword and assigns the gender of the specified student.
 - ◆ The **GetValue()** method of the derived class returns the sex of the student.
 - ◆ The name of the student is invoked by using the base keyword to call the **GetValue()** method of the base class.
 - ◆ The **StudentList** class creates an instance of the **Student** class. This instance invokes the **GetValue()** method of the derived class, which in turn invokes the **GetValue()** method of the base class by using the base keyword.
- ◆ The following figure shows the output of the code to override virtual methods for generic class:



```
C:\WINDOWS\system32\cmd.exe
Name      Sex
Patrick   Male
Press any key to continue . . .
```

- ◆ Consider a scenario where a person is trying to memorize a book of 100 pages.
- ◆ To finish the task, the person has to iterate through each of these 100 pages.
- ◆ Similar to this person who iterates through the pages, an iterator in C# is used to traverse through a list of values or a collection.
- ◆ It is a block of code that uses the `foreach` loop to refer to a collection of values in a sequential manner.
- ◆ For example, consider a collection of values that needs to be sorted.
- ◆ To implement the logic manually, a programmer can iterate through each value sequentially using iterators to compare the values.
- ◆ An iterator is not a data member but is a way of accessing the member.
- ◆ It can be a method, a `get` accessor, or an operator that allows you to navigate through the values in a collection.
- ◆ Iterators specify the way, values are generated, when the `foreach` statement accesses the elements within a collection.
- ◆ They keep track of the elements in the collection, so that you can retrieve these values if required.
- ◆ Consider an array variable consisting of 6 elements, where the iterator can return all the elements within an array one by one.
- ◆ The following figure illustrates these analogies:

Example



- ◆ For a class that behaves like a collection, it is preferable to use iterators to iterate through the values of the collection with the `foreach` statement.
- ◆ By doing this, one can get the following benefits:
 - ◆ Iterators provide a simplified and faster way of iterating through the values of a collection.
 - ◆ Iterators reduce the complexity of providing an enumerator for a collection.
 - ◆ Iterators can return large number of values.
 - ◆ Iterators can be used to evaluate and return only those values that are needed.
 - ◆ Iterators can return values without consuming memory by referring each value in the list.

Implementation 1-3

- ◆ Iterators can be created by implementing the `GetEnumerator()` method that returns a reference of the `IEnumerator` interface.
- ◆ The iterator block uses the `yield` keyword to provide values to the instance of the enumerator or to terminate the iteration.
- ◆ The `yield return` statement returns the values, while the `yield break` statement ends the iteration process.
- ◆ When the program control reaches the `yield return` statement, the current location is stored, and the next time the iterator is called, the execution is started from the stored location.



Implementation 2-3

- ◆ The following code demonstrates the use of iterators to iterate through the values of a collection:

Snippet

```
using System;
using System;
using System.Collections;
class Department : IEnumerable
{
    string[] departmentNames = {"Marketing", "Finance",
    "Information Technology", "Human Resources"};
    public IEnumerator GetEnumerator()
    {
        for (int i = 0; i < departmentNames.Length; i++)
        {
            yield return departmentNames[i];
        }
    }
    static void Main (string [] args)
    {
        Department objDepartment = new Department();
        Console.WriteLine("Department Names");
        Console.WriteLine();
        foreach(string str in objDepartment)
        {
            Console.WriteLine(str);
        }
    }
}
```

- ◆ In the code:
 - ◆ The class **Department** implements the interface **IEnumerable**.
 - ◆ The class **Department** consists of an array variable that stores the department names and a `GetEnumerator()` method, that contains the `for` loop.
 - ◆ The `for` loop returns the department names at each index position within the array variable.
 - ◆ This block of code within the `GetEnumerator()` method comprises the iterator in this example.
 - ◆ The `Main()` method creates an instance of the class **Department** and contains a `foreach` loop that displays the department names.
- ◆ The following code displays the use of iterators:



```
C:\WINDOWS\system32\cmd.exe
Department Names
Marketing
Finance
Information Technology
Human Resources
Press any key to continue . . .
```


Generic Iterators 1-3

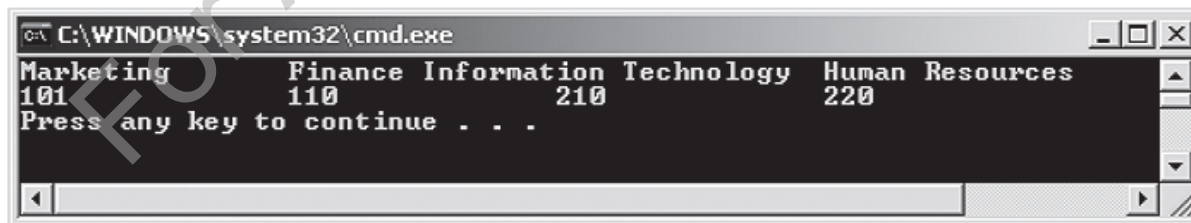
- ◆ C# allows programmers to create generic iterators.
- ◆ Generic iterators are created by returning an object of the generic `Ienumerator <T>` or `Ienumerable <T>` interface.
- ◆ They are used to iterate through values of any value type.
- ◆ The following code demonstrates how to create a generic iterator to iterate through values of any type:

Snippet

```
using System;
using System.Collections.Generic;
class GenericDepartment<T>
{
    T[] item;
    public GenericDepartment(T[] val)
    {
        item = val;
    }
    public IEnumerator<T> GetEnumerator()
    {
        foreach (T value in item)
        {
            yield return value;
        }
    }
}
```

```
class GenericIterator
{
    static void Main(string[] args)
    {
        string[] departmentNames = { "Marketing", "Finance",
        "Information Technology", "Human Resources" };
        GenericDepartment<string> objGeneralName = new
        GenericDepartment<string>(departmentNames);
        foreach (string val in objGeneralName)
        {
            Console.Write(val + "\t");
        }
        int[] departmentID = { 101, 110, 210, 220 };
        GenericDepartment<int> objGeneralID = new
        GenericDepartment<int>(departmentID);
        Console.WriteLine();
        foreach (int val in objGeneralID)
        {
            Console.Write(val + "\t\t");
        }
        Console.WriteLine();
    }
}
```

- ◆ The following figure shows the output of the code to create a generic iterator:



C:\WINDOWS\system32\cmd.exe

Marketing	Finance	Information Technology	Human Resources
101	110	210	220

Press any key to continue . . .

- ◆ In the code:
 - ◆ The generic class, **GenericDepartment**, is created with the generic type parameter T.
 - ◆ The class declares an array variable and consists of a parameterized constructor that assigns values to this array variable.
 - ◆ In the generic class, **GenericDepartment**, the `GetEnumerator()` method returns a generic type of the `IEnumerator` interface.
 - ◆ This method returns elements stored in the array variable, using the `yield` statement.
 - ◆ In the **GenericIterator** class, an instance of the **GenericDepartment** class is created that refers to the different department names within the array.
 - ◆ Another object of the **GenericDepartment** class is created, that refers to the different department IDs within the array.

Implementing Named Iterators 1-2

- ◆ Another way of creating iterators is by creating a method, whose return type is the `IEnumerable` interface.
- ◆ This is called a **named iterator**. Named iterators can accept parameters that can be used to manage the starting and end points of a `foreach` loop.
- ◆ This flexible technique allows you to fetch the required values from the collection.
- ◆ The following syntax creates a named iterator:

Syntax

```
<access_modifier> IEnumerable <IteratorName>  
    (<parameter list>){}
```

- ◆ where,
 - ◆ `access_modifier`: Specifies the scope of the named iterator.
 - ◆ `IteratorName`: Is the name of the iterator method.
 - ◆ `parameter list`: Defines zero or more parameters to be passed to the iterator method.

Implementing Named Iterators 2-2

- ◆ The following code demonstrates how to create a named iterator:

Snippet

```
using System;
class NamedIterators{
    string[] cars = { "Ferrari", "Mercedes", "BMW", "Toyota", "Nissan"};
    public IEnumerable GetCarNames() {
        for (int i = 0; i < cars.Length; i++) {
            yield return cars[i];
        }
    }
    static void Main(string[] args) {
        NamedIterators objIterator = new NamedIterators();
        foreach (string str in objIterator.GetCarNames()) {
            Console.WriteLine(str);
        }
    }
}
```

Output

Ferrari
Mercedes
BMW
Toyota
Nissan

- ◆ In the code:
 - ◆ The **NamedIterators** class consists of an array variable and a method **GetCarNames()**, whose return type is **IEnumerable**.
 - ◆ The **for** loop iterates through the values within the array variable.
 - ◆ The **Main()** method creates an instance of the class **NamedIterators** and this instance is used in the **foreach** loop to display the names of the cars from the array variable.

- ◆ Generics are data structures that allow you to reuse a code for different types such as classes or interfaces.
- ◆ Generics provide several benefits such as type-safety and better performance.
- ◆ Generic types can be declared by using the type parameter, which is a placeholder for a particular type.
- ◆ Generic classes can be created by the class declaration followed by the type parameter list enclosed in the angular brackets and application of constraints (optional) on the type parameters.
- ◆ An iterator is a block of code that returns sequentially ordered values of the same type.
- ◆ One of the ways to create iterators is by using the GetEnumerator() method of the IEnumerable or IEnumerator interface.
- ◆ The yield keyword provides values to the enumerator object or to signal the end of the iteration.