Session 11: Generics and Iterators

Objectives

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

Generics

- 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

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

Class	Description
`Comparer`	Is an abstract class that allows you to create a generic collection by implementing the functionalities of the `IComparer` interface
`Dictionary.KeyCollection`	Consists of keys present in the instance of the `Dictionary` class
`Dictionary.ValueCollection`	Consists of values present in the instance of the Dictionary class
`EqualityComparer`	Is an abstract class that allows you to create a generic collection by implementing the functionalities of the `IEqualityComparer` 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.
- Following table lists some widely used interfaces of `System.Collections.Generic` namespace:

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

System.Collections.ObjectModel

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

Class	Description
`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

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.

Benefits

- 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.
- Following are the benefits:
  - 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

- 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.
- 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.

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.

- Following table lists the types of constraints that can be applied to the type parameter:

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

Constraints on Type Parameters 2-4

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.
- The constraints imposed at the base class level must be included in the derived generic class.
- 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

- 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.

- 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 1-3

- 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 every time.

- 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.

Generic Methods 2-3

- 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.

Generic Methods 3-3

- 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);
    }
}
```

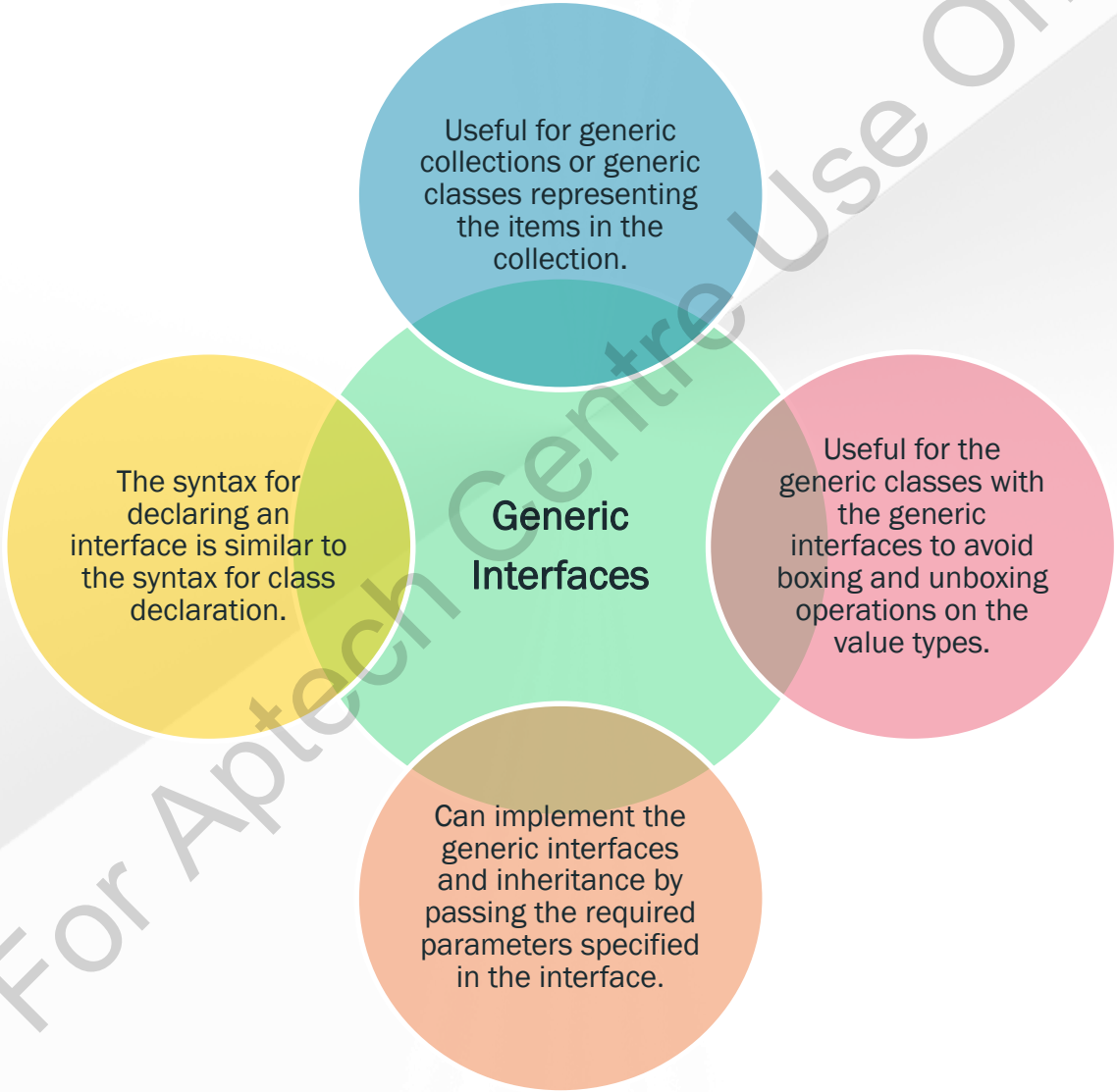
- In the code:
  - The class **SwapNumbers** consists of a generic method **Swap()** that takes a type parameter T within angular brackets and two parameters within parenthesis of type T.
  - The **Swap()** method creates a variable temp of type T that is assigned the value within the variable **valOne**.

Output

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

Generic Interfaces 1-4

- Following are the features of generic interfaces:



Generic Interfaces 2-4

- 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.

Generic Interfaces 3-4

- 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.Write("Addition of two integer values is: ");
     Console.WriteLine(objInterface.Addition(numOne, numTwo));
     Console.Write("Subtraction of two integer values is: ");
      Console.WriteLine(objInterface.Subtraction(numOne, numTwo));
    }
}
```

Generic Interfaces 4-4

- 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

- 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);
     }
}
```

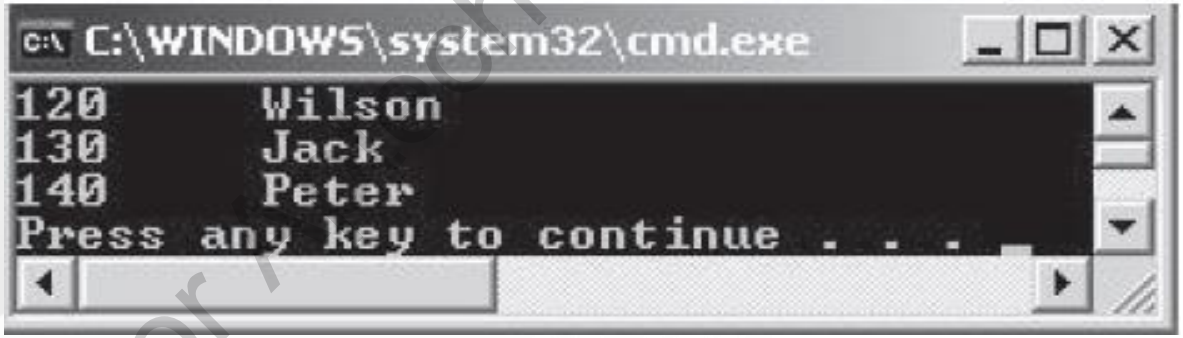
Generic Interface Constraints 3-4

```
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`**.

- Following figure shows the output of the code to create a generic interface:



Generic Delegates

- 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.

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

- 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

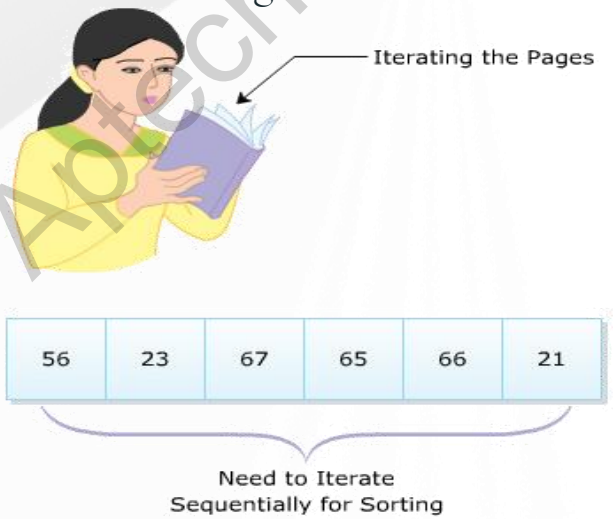
- In the code:
  - The **General** class has two overloaded methods with different type parameters.
  - The overloaded methods are invoked by specifying appropriate values.
  - The methods store these values in the respective variables defined in the **General** class.
  - These values indicate the ID and name of the employee.
  - 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.

- Following figure shows the output of the code to overload methods using type parameters:

Iterators

- 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.
- 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 must be sorted.
- An iterator is not a data member but is a way of accessing the member.
- Iterators specify the way, values are generated, when the foreach statement accesses the elements within a collection.
- Consider an array variable consisting of 6 elements, where the iterator can return all the elements within an array one by one.
- Following figure illustrates these analogies:

Example



Benefits

- 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 required.
  - Iterators can return values without consuming memory by referring each value in the list.

Implementation

- 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.

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.

- Following code demonstrates how to create a generic iterator to iterate through values of any type:

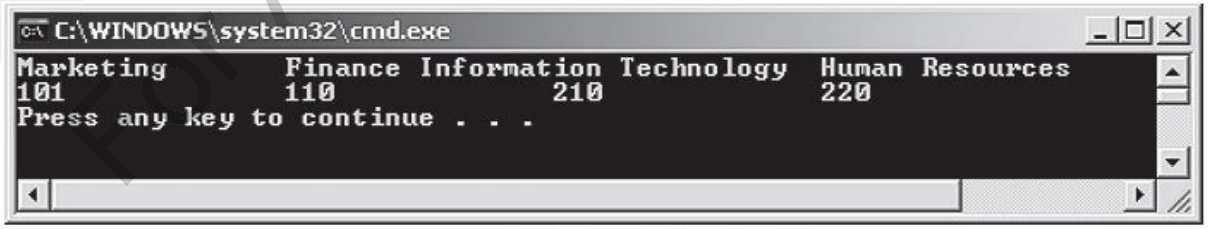
Snippet

```
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;
         }
      }
}
```

Generic Iterators 2-3

```
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();
      }
}
```

- Following figure shows the output of the code to create a generic iterator:



Generic Iterators 3-3

- 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.

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.
- 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

- 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.

Summary

- 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.