Session: 13

Generics and Iterators

Objectives

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

Example

Consider a program that uses an array variable of type **Object** to store a collection of student names.

The names are read from the console and stored as type Object.

If you enter numeric data, it will be accepted without any verification because it allows to cast any value to and from Object !

- 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.

- *Generics are a kind of **parameterized** data structures that can work with value types as well as reference types.*

Benefits of Generics

- ensure type-safety at compile-time:
  - ensure strongly-typed programming model
- allow to reuse the code in a safe manner without casting or boxing:
  - reduce run-time errors
  - Improve performance because of low memory usage as no casting or boxing operation is required.
- can be reusable with different types but can accept values of a single type at a time.

Namespaces, Classes, and Interfaces for Generics

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

`System.Collections.ObjectModel`

- consists of classes and interfaces that allow you to create type-safe collections.

`System.Collections.Generic`

- consists of classes and interfaces that can be used as collections in the object model of a reusable library

Generic Classes

- are declared with **type parameters** enclosed within angular brackets.
- can apply some restrictions or constraints to the type parameters by using the **`where`** keyword.

Syntax

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

- where,
  - **`type-parameter list:`** Is used as a placeholder for the actual data type.
  - **`type-parameter constraint clause:`** Is an option applied to the type parameter with the **`where`** keyword.

- Example:
  - `public class Student<T> { … }`

Constraints on Type Parameters

- A constraint is a restriction imposed on the data type of the type parameter and are specified using the **`where`** keyword.
- Types of constraints that can be applied to the type parameter:

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

Inheriting Generic Classes

- inherit a generic class from an existing generic class:

Syntax

```
<acc_modifier> class <BaseClass><<type-parameter>>{ }
<acc_modifier> class <DerivedClass>:<BaseClass><<type-parameter>>{ }
```

where,

- <**type-parameter**>: Is a placeholder for the specified data type.
- **DerivedClass**: Is the generic derived class.

- inherit a non-generic class from a generic class:

Syntax

```
<acc_modifier> class <BaseClass><<type-parameter>>{ }
<acc_modifier> class <DerivedClass>:<BaseClass><<type-par-value>>{ }
```

where,

- <**type-par-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.
- declared with the type-parameter list enclosed within angular brackets.
- allow to call the method with a different type.
- can be declared within generic or non-generic class.
- When declared within a generic class, the body of the method refers to the type parameters of both, the method and class declaration.
- can be declared with the following keywords:
  - **Virtual**.
  - **Override**: while overriding, the method does not specify the type parameter constraints since the constraints are overridden from the overridden method.
  - **Abstract**

Syntax

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

where,

- **Type-parameter constraint clause**: Is an optional class or an interface applied to the type parameter with the **where** keyword.

Generic Delegates

- Delegates are reference types that encapsulate a reference to a method that has the same signature and return type.
- Features of a generic delegate:
  - 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.

```
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.

Overloading Methods Using Type Parameters

- can overload methods of a generic class that take generic 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.

```
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);
    }
}
```

Overriding Virtual Methods in Generic Class

- Can override methods in generic classes by using keywords **virtual** and **override**.

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;

     }
}
```

```
class Student<T>:GeneralList<T> {
   public T Value;

   public Student(T val1, T val2):
                  base(val1) {
     Value = val2;
   }

   public override T GetValue() {
     Console.Write(base.GetValue());
     return Value;

   }
}
```

Iterators

- 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.
- Benefits of iterator:
  - provide a simplified and faster way of iterating through the values of a collection.
  - reduce the complexity of providing an enumerator for a collection.
  - can return large number of values.
  - can be used to evaluate and return only those values that are needed.
  - can return values without consuming memory by referring each value in the list.

Implementation Iterators

- Iterators can be created by implementing interface **IEnumerable**:

  override method **IEnumerator GetEnumerator()**

- The iterator block uses :
  - **yield return** to provide values to the instance of the enumerator
  - **yield break** to terminate the iteration process.

```
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);
        }
    }
}
```

Generic Iterators

- Generic iterators are created by method **GetEnumerator()** returning an object of the generic **IEnumerator<T>** or **IEnumerable<T>** interface.
- They are used to iterate through values of any value type.
- For example

```
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();
      }
}
```

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.