

# Collections, Generic



# Lesson Objectives

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

## Section 1

# COLLECTIONS

- A collection is a set of related data that may not necessarily belong to the same data type that can be set or modified dynamically at run-time.
- Accessing collections is similar to accessing arrays, where elements are accessed by their index numbers. However, there are differences between arrays and collections in C#.
- The following table lists the differences between arrays and collections:

Array	Collection
Cannot be resized at run-time.	Can be resized at run-time.
The individual elements are of the same data type.	The individual elements can be of different data types.
Do not contain any methods for operations on elements.	Contain methods for operations on elements.

- The `System.Collections` namespace in C# allows you to construct and manipulate a collection of objects that includes elements of different data types. The `System.Collections` namespace defines various collections such as dynamic arrays, lists, and dictionaries.
- The `System.Collections` namespace consists of classes and interfaces that define the different collections.
- The following table lists the commonly used classes and interfaces in the `System.Collections` namespace:

Class/Interface	Description
<code>ArrayList</code> Class	Provides a collection that is similar to an array except that the items can be dynamically added and retrieved from the list and it can contain values of different types
<code>Stack</code> Class	Provides a collection that follows the Last-In-First-Out (LIFO) principle, which means the last item inserted in the collection, will be removed first
<code>Hashtable</code> Class	Provides a collection of key and value pairs that are arranged, based on the hash code of the key
<code>SortedList</code> Class	Provides a collection of key and value pairs where the items are sorted, based on the keys
<code>IDictionary</code> Interface	Represents a collection consisting of key/value pairs
<code>IDictionaryEnumerator</code> Interface	Lists the dictionary elements
<code>IEnumerable</code> Interface	Defines an enumerator to perform iteration over a collection
<code>ICollection</code> Interface	Specifies the size and synchronization methods for all collections
<code>IEnumerator</code> Interface	Supports iteration over the elements of the collection
<code> IList</code> Interface	Represents a collection of items that can be accessed by their index number

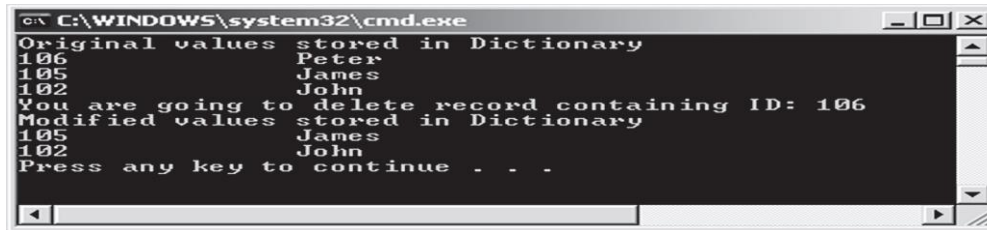
- The following code demonstrates the use of the commonly used classes and interfaces of the System.Collections namespace:

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

class Employee : DictionaryBase
{
    public void Add(int id, string name)
    {
        Dictionary.Add(id, name);
    }
    public void OnRemove(int id)
    {
        Console.WriteLine("You are going to delete record
        containing ID: " + id);
        Dictionary.Remove(id);
    }
    public void GetDetails()
    {
        IDictionaryEnumerator objEnumerate =
        Dictionary.GetEnumerator();
        while (objEnumerate.MoveNext())
        {
            Console.WriteLine(objEnumerate.Key.ToString() +
            "\t\t" +
            objEnumerate.Value);
        }
    }
}
```

```
static void Main(string[] args)
{
    Employee objEmployee = new Employee();
    objEmployee.Add(102, "John");
    objEmployee.Add(105, "James");
    objEmployee.Add(106, "Peter");
    Console.WriteLine("Original values stored in
    Dictionary");
    objEmployee.GetDetails();
    objEmployee.OnRemove(106);
    Console.WriteLine("Modified values stored in
    Dictionary");
    objEmployee.GetDetails();
}
}
```

- ◆ In the code:
  - ◆ The class **Employee** is inherited from the `DictionaryBase` class.
  - ◆ The `DictionaryBase` class is an abstract class. The details of the employees are inserted using the methods present in the `DictionaryBase` class.
  - ◆ The user-defined `Add()` method takes two parameters, namely **id** and **name**. These parameters are passed onto the `Add()` method of the `Dictionary` class.
  - ◆ The `Dictionary` class stores these values as a key/value pair.
  - ◆ The `OnRemove()` method of `DictionaryBase` is overridden. It takes a parameter specifying the key whose key/value pair is to be removed from the `Dictionary` class.
  - ◆ This method then prints a warning statement on the console before deleting the record from the `Dictionary` class.
  - ◆ The `Dictionary.Remove()` method is used to remove the key/value pair.
  - ◆ The `GetEnumerator()` method returns an `IDictionaryEnumerator`, which is used to traverse through the list.
- ◆ The following figure displays the output of the example:



```
C:\WINDOWS\system32\cmd.exe
Original values stored in Dictionary
106 Peter
105 James
102 John
You are going to delete record containing ID: 106
Modified values stored in Dictionary
105 James
102 John
Press any key to continue . . .
```



- Following are the features of ArrayList class:

The `ArrayList` class is a variable-length array, that can dynamically increase or decrease in size. Unlike the `Array` class, this class can store elements of different data types.

The `ArrayList` class allows you to specify the size of the collection, during program execution and also allows you to define the capacity that specifies the number of elements an array list can contain.

However, the default capacity of an `ArrayList` class is 16. If the number of elements in the list reaches the specified capacity, the capacity of the list gets doubled automatically. It can accept null values and can also include duplicate elements.

The `ArrayList` class allows you to add, modify, and delete any type of element in the list even at run-time.

The elements in the `ArrayList` can be accessed by using the index position. While working with the `ArrayList` class, you need not bother about freeing up the memory.

The `ArrayList` class consists of different methods and properties that are used to add and manipulate the elements of the list.

# ArrayList Class 2-3

## ■ Methods

- ✓ The methods of the `ArrayList` class allow you to perform actions such as adding, removing, and copying elements in the list.
- ✓ The following table displays the commonly used methods of the `ArrayList` class:

Method	Description
<b>Add</b>	Adds an element at the end of the list
<b>Remove</b>	Removes the specified element that has occurred for the first time in the list
<b>RemoveAt</b>	Removes the element present at the specified index position in the list
<b>Insert</b>	Inserts an element into the list at the specified index position
<b>Contains</b>	Determines the existence of a particular element in the list
<b>IndexOf</b>	Returns the index position of an element occurring for the first time in the list
<b>Reverse</b>	Reverses the values stored in the <code>ArrayList</code>
<b>Sort</b>	Rearranges the elements in an ascending order

## ■ Properties

- ✓ The properties of the `ArrayList` class allow you to count or retrieve the elements in the list. The following table displays the commonly used properties of the `ArrayList` class:

Property	Description
<code>Capacity</code>	Specifies the number of elements the list can contain
<code>Count</code>	Determines the number of elements present in the list
<code>Item</code>	Retrieves or sets value at the specified position

# ArrayList Class 3-3

- The following code demonstrates the use of the methods and properties of the ArrayList class:

## Snippet

```
using System;
using System.Collections;

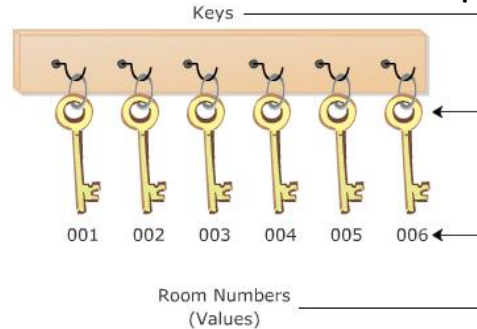
class ArrayCollection
{
    static void Main(string[] args)
    {
        ArrayList objArray = new ArrayList();
        objArray.Add("John");
        objArray.Add("James");
        objArray.Add("Peter");
        objArray.RemoveAt(2);
        objArray.Insert(2, "Williams");
        Console.WriteLine("Capacity: " + objArray.Capacity);
        Console.WriteLine("Count: " + objArray.Count);
        Console.WriteLine();
        Console.WriteLine("Elements of the ArrayList");
        foreach (string str in objArray)
        {
            Console.WriteLine(str);
        }
    }
}
```

## Output

```
Capacity: 4
Count: 3
Elements of the ArrayList
John
James
Williams
```

# Hashtable Class 1-5

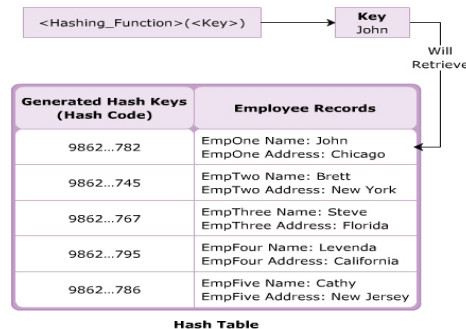
- ◆ Consider the reception area of a hotel where you find the keyholder storing a bunch of keys.
- ◆ Each key in the keyholder uniquely identifies a room and thus, each room is uniquely identified by its key.
- ◆ The following figure demonstrates a real-world example of unique keys:



- ◆ Similar to the keyholder, the `Hashtable` class in C# allows you to create collections in the form of keys and values.
- ◆ It generates a `hashtable` which associates keys with their corresponding values.
- ◆ The `Hashtable` class uses the `hashtable` to retrieve values associated with their unique key.

# Hashtable Class 2-5

- ◆ The `Hashtable` generated by the `Hashtable` class uses the hashing technique to retrieve the corresponding value of a key.
- ◆ Hashing is a process of generating the hash code for the key and the code is used to identify the corresponding value of the key.
- ◆ The `Hashtable` object takes the key to search the value, performs a hashing function and generates a hash code for that key.
- ◆ When you search for a particular value using the key, the hash code is used as an index to locate the desired record.
- ◆ For example, a student name can be used as a key to retrieve the student id and the corresponding residential address. The following figure represents the `Hashtable`:



- ◆ The `Hashtable` class consists of different methods and properties that are used to add and manipulate the data within the `hashtable`.
- ◆ The methods of the `Hashtable` class allow you to perform certain actions on the data in the `hashtable`.
- ◆ The following table displays the commonly used methods of the `Hashtable` class:

Method	Description
Add	Adds an element with the specified key and value
Remove	Removes the element having the specified key
CopyTo	Copies elements of the hashtable to an array at the specified index
ContainsKey	Checks whether the hashtable contains the specified key
ContainsValue	Checks whether the hashtable contains the specified value
GetEnumerator	Returns an <code>IDictionaryEnumerator</code> that traverses through the <code>Hashtable</code>

## ■ Properties

- ✓ The properties of the `Hashtable` class allow you to access and modify the data in the hashtable.
- ✓ The following figure displays the commonly used properties of the `Hashtable` class:

Property	Description
Count	Specifies the number of key and value pairs in the hashtable
Item	Specifies the value, adds a new value or modifies the existing value for the specified key
Keys	Provides an <code>ICollection</code> consisting of keys in the hashtable
Values	Provides an <code>ICollection</code> consisting of values in the hashtable
IsReadOnly	Checks whether the <code>Hashtable</code> is read-only

- The following code demonstrates the use of the methods and properties of the `Hashtable` class:

## Snippet

```
using System;
using System.Collections;

class HashCollection
{
    static void Main(string[] args)
    {
        Hashtable objTable = new Hashtable();
        objTable.Add(001, "John");
        objTable.Add(002, "Peter");
        objTable.Add(003, "James");
        objTable.Add(004, "Joe");
        Console.WriteLine("Number of elements in the hash table: " +
            objTable.Count);
        ICollection objCollection = objTable.Keys;
        Console.WriteLine("Original values stored in hashtable are:
            ");
        foreach (int i in objCollection)
        {
            Console.WriteLine (i + " : " + objTable[i]);
        }
        if (objTable.ContainsKey(002))
        {
            objTable[002] = "Patrick";
        }
        Console.WriteLine("Values stored in the hashtable after
            removing values");
        foreach (int i in objCollection)
        {
            Console.WriteLine(i + " : " + objTable[i]);
        }
    }
}
```



# SortedList Class 1-4

- The `SortedList` class represents a collection of key and value pairs where elements are sorted according to the key.
- By default, the `SortedList` class sorts the elements in ascending order, however, this can be changed if an `Comparable` object is passed to the constructor of the `SortedList` class.
- These elements are either accessed using the corresponding keys or the index numbers.
- If you access elements using their keys, the `SortedList` class behaves like a hashtable, whereas if you access elements based on their index number, it behaves like an array.
- The `SortedList` class consists of different methods and properties that are used to add and manipulate the data in the sorted list.

## ■ Methods

- ✓ The methods of the `SortedList` class allow you to perform certain actions on the data in the sorted list.
- The following table displays the commonly used methods of the `SortedList` class:

Method	Description
Add	Adds an element to the sorted list with the specified key and value
Remove	Removes the element having the specified key from the sorted list
GetKey	Returns the key at the specified index position
GetByIndex	Returns the value at the specified index position
ContainsKey	Checks whether the instance of the <code>SortedList</code> class contains the specified key
ContainsValue	Checks whether the instance of the <code>SortedList</code> class contains the specified value
RemoveAt	Deletes the element at the specified index

## ■ Properties

- ✓ The properties of the SortedList class allow you to access and modify the data in the sorted list.

Property	Description
Capacity	Specifies the number of elements the sorted list can contain
Count	Specifies the number of elements in the sorted list
Item	Returns the value, adds a new value or modifies the existing value for the specified key
Keys	Returns the keys in the sorted list
Values	Returns the values in the sorted list

- The following code demonstrates the use of methods and properties of the SortedList class:

## Snippet

```
using System;
using System.Collections;

class SortedCollection
{
    static void Main(string[] args)
    {
        SortedList objSortList = new SortedList();
        objSortList.Add("John", "Administration");
        objSortList.Add("Jack", "Human Resources");
        objSortList.Add("Peter", "Finance");
        objSortList.Add("Joel", "Marketing");
        Console.WriteLine("Original values stored in the
sorted list");
        Console.WriteLine("Key \t\t Values");
        for (int i=0; i<objSortList.Count; i++)
        {
            Console.WriteLine(objSortList.GetKey(i) + " \t\t " +
objSortList.GetByIndex(i));
        }
        if (!objSortList.ContainsKey("Jerry"))
        {
            objSortList.Add("Jerry", "Construction");
        }
        objSortList["Peter"] = "Engineering";
        objSortList["Jerry"] = "Information Technology";
        Console.WriteLine();
        Console.WriteLine("Updated values stored in hashtable");
        Console.WriteLine("Key \t\t Values");
        for (int i = 0; i < objSortList.Count; i++)
        {
            Console.WriteLine(objSortList.GetKey(i) + " \t\t " +
objSortList.GetByIndex(i));
        }
    }
}
```

# Dictionary Generic Class 1-5

- The `System.Collections.Generic` namespace contains a vast number of generic collections.
- One of the most commonly used among these is the `Dictionary` generic class that consists of a generic collection of elements organized in key and value pairs and maps the keys to their corresponding values.
- Unlike other collections in the `System.Collections` namespace, it is used to create a collection of a single data type at a time.
- Every element that you add to the dictionary consists of a value, which is associated with its key and can retrieve a value from the dictionary by using its key.
- The following syntax declares a `Dictionary` generic class:

## Syntax

```
Dictionary<TKey, TValue>
```

- Where,
  - ✓ `TKey`: Is the type parameter of the keys to be stored in the instance of the `Dictionary` class.
  - ✓ `TValue`: Is the type parameter of the values to be stored in the instance of the `Dictionary` class.

- The `Dictionary` generic class consists of different methods and properties that are used to add and manipulate elements in a collection.

- ✓ **Methods**

- The methods of the `Dictionary` generic class allow you to perform certain actions on the data in the collection.
- The following table displays the commonly used methods of the `Dictionary` generic class:

Method	Description
Add	Adds the specified key and value in the collection
Remove	Removes the value associated with the specified key
ContainsKey	Checks whether the collection contains the specified key
ContainsValue	Checks whether the collection contains the specified value
GetEnumerator	Returns an enumerator that traverses through the Dictionary
GetType	Retrieves the Type of the current instance

## ■ Properties

- ✓ The properties of the `Dictionary` generic class allow you to modify the data in the collection.
- ✓ The following table displays the commonly used properties of the `Dictionary` generic class:

Property	Description
<code>Count</code>	Determines the number of key and value pairs in the collection
<code>Item</code>	Returns the value, adds a new value or modifies the existing value for the specified key
<code>Keys</code>	Returns the collection containing the keys
<code>Values</code>	Returns the collection containing the values

# Dictionary Generic Class 4-5

- The following code demonstrates the use of the methods and properties of the Dictionary class:

## Snippet

```
using System;
using System.Collections;
class DictionaryCollection{
    static void Main(string[] args) {
        Dictionary<int, string> objDictionary = new Dictionary<int,
        string>();
        objDictionary.Add(25, "Hard Disk");
        objDictionary.Add(30, "Processor");
        objDictionary.Add(15, "MotherBoard");
        objDictionary.Add(65, "Memory");
        ICollection objCollect = objDictionary.Keys;
        Console.WriteLine("Original values stored in the collection");
        Console.WriteLine("Keys \t Values");
        Console.WriteLine("-----");
        foreach (int i in objCollect){
            Console.WriteLine(i + " \t " + objDictionary[i]);
        }
        objDictionary.Remove(65);
        Console.WriteLine();
        if (objDictionary.ContainsValue("Memory")) {
            Console.WriteLine("Value Memory could not be deleted");
        }
        else {
            Console.WriteLine("Value Memory deleted successfully");
        }
        Console.WriteLine();
        Console.WriteLine("Values stored after removing element");
        Console.WriteLine("Keys \t Values");
        Console.WriteLine("-----");
        foreach (int i in objCollect) {
            Console.WriteLine(i + " \t " + objDictionary[i]);
        }
    }
}
```





# Collection Initializers 1-2

- Collection initializers allow adding elements to the collection classes of the `System.Collections` and `System.Collections.Generic` namespaces that implements the `IEnumerable` interface using element initializers.
- The element initializers can be a simple value, an expression, or an object initializer.
- When a programmer uses collection initializer, the programmer is not required to provide multiple `Add()` methods to add elements to the collection, making the code concise. It is the responsibility of the compiler to provide the `Add()` methods when the program is compiled.
- The following code uses a collection initializer to initialize an `ArrayList` with integers:

## Snippet

```
using System;
using System.Collections;

class Car
{
    static void Main (string [] args)
    {
        ArrayList nums=new ArrayList{1,2,3*6,4,5};
        foreach (int num in nums)
        {
            Console.WriteLine("{0}", num);
        }
    }
}
```

# Collection Initializers 2-2

- In the code:
  - ✓ The `Main()` method uses collection initializer to create an `ArrayList` object initialized with integer values and expressions that evaluates to integer values.
  - ✓ As collection often contains objects, collection initializers accept object initializers to initialize a collection.
  - ✓ The following code shows a collection initializer that initializes a generic `Dictionary` object with an integer keys and `Employee` objects:

## Snippet

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

class Employee{
    public String Name { get; set; }
    public String Designation { get; set; }
}

class CollectionInitializerDemo{
    static void Main(string[] args)
    {
        Dictionary<int, Employee> dict = new Dictionary<int,
Employee>() {
            { 1, new Employee {Name="Andy Parker",
Designation="Sales Person"}}, { 2, new Employee {Name="Patrick Elvis",
Designation="Marketing Manager"}}
        };
    }
}
```

# Collections - Summary

- The `System.Collections.Generic` namespace consists of generic collections that allow reusability of code and provide better type-safety.
- The `ArrayList` class allows you to increase or decrease the size of the collection during program execution. `Hashtable` class.
- The `Hashtable` class stores elements as key and value pairs where the data is organized based on the hash code. Each value in the `hashtable` is uniquely identified by its key.
- The `SortedList` class allows you to store elements as key and value pairs where the data is sorted based on the key.
- The `Dictionary` generic class represents a collection of elements organized in key and value pairs.

## Section 2

# GENERICICS

- 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.

- 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.

- 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



- **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

- 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.

# Constraints on Type Parameters

- 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 : &lt;base class name&gt;</code>	Specifies that the type parameter must be the parent class or should inherit from a parent class
<code>T : &lt;interface name&gt;</code>	Specifies that the type parameter must be an interface or should inherit an interface

- 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:

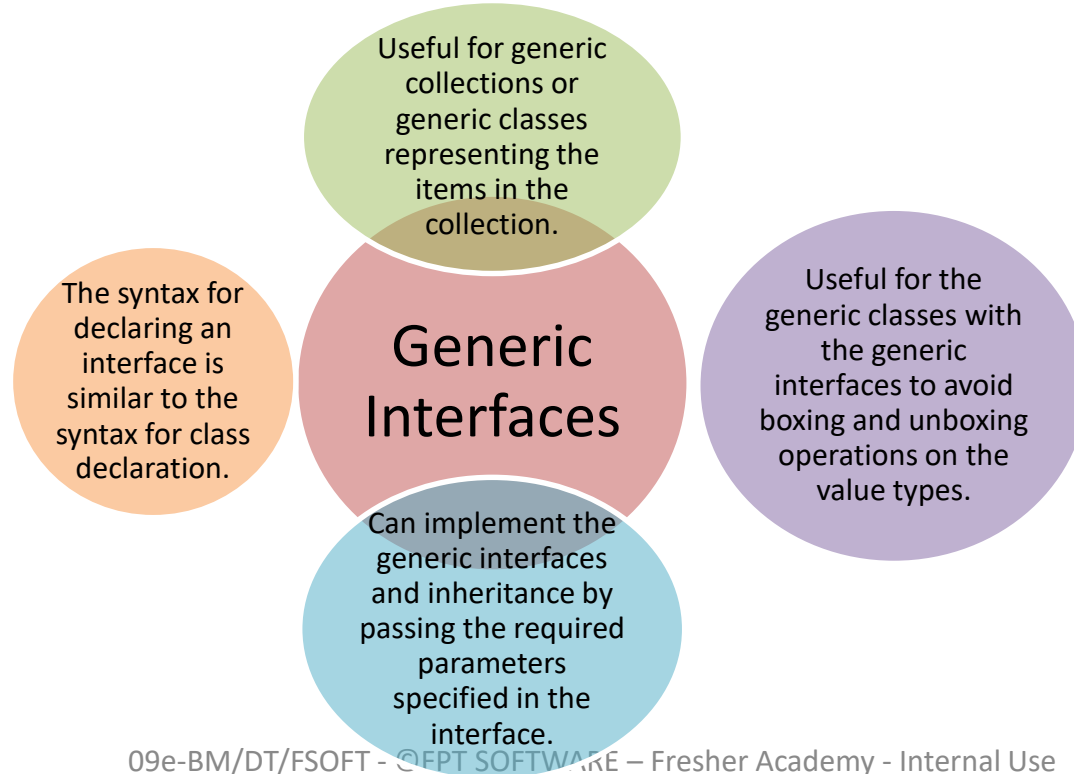
```
Generic -> Generic

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

Generic -> Non-Generic

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

- Following are the features of generic interfaces:



# Generic - 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.



## Section 3

# 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.
- 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.

- 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 provide a simplified and faster way of iterating through the values of 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 1-2

- 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 2-2

- 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);
        }
    }
}
```

# 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

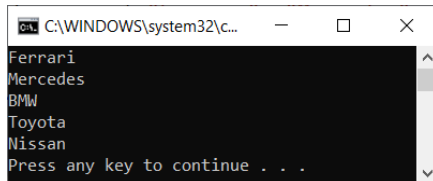
- The following code demonstrates how to create a named iterator:

## Snippet

```
using System;
using System.Collections;

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



```
C:\WINDOWS\system32\c...
Ferrari
Mercedes
BMW
Toyota
Nissan
Press any key to continue . . .
```

# Iterators - Summary

- 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.



# Lesson Summary

- The `System.Collections.Generic` namespace consists of generic collections that allow reusability of code and provide better type-safety.
- The `ArrayList` class allows you to increase or decrease the size of the collection during program execution. The `Hashtable` class.
- The `Hashtable` class stores elements as key and value pairs where the data is organized based on the hash code. Each value in the `hashtable` is uniquely identified by its key.
- The `SortedList` class allows you to store elements as key and value pairs where the data is sorted based on the key.
- The `Dictionary` generic class represents a collection of elements organized in key and value pairs.
- 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.
- 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.

# Thank you

