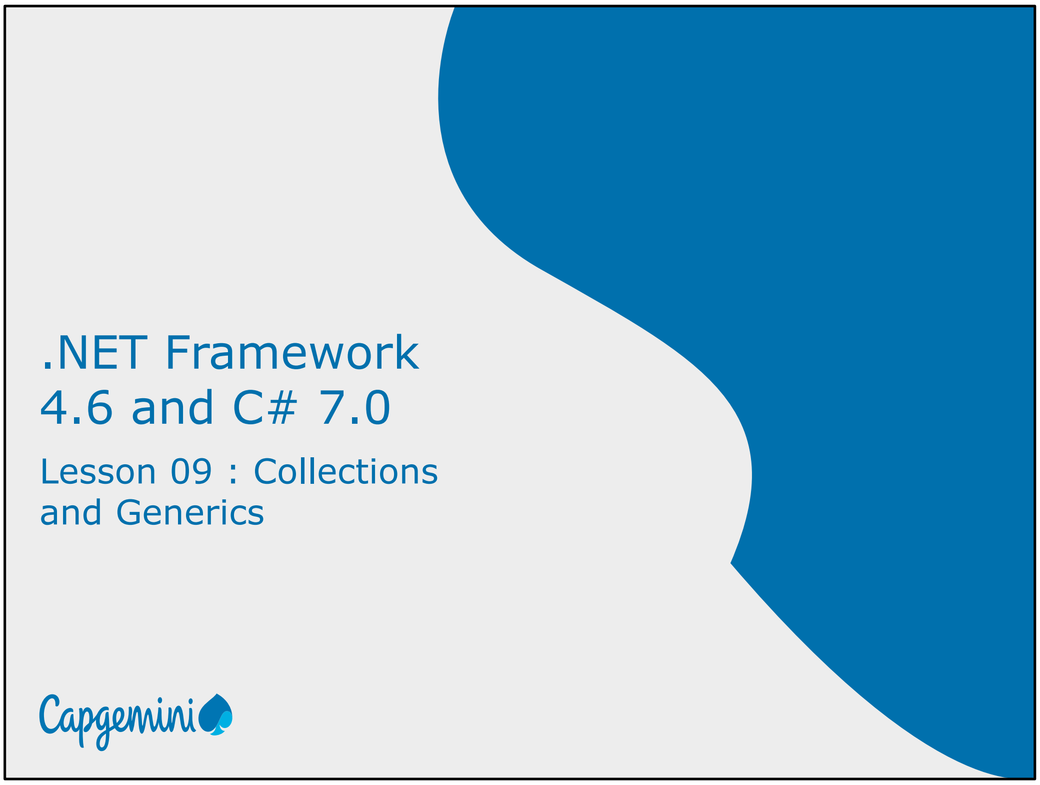
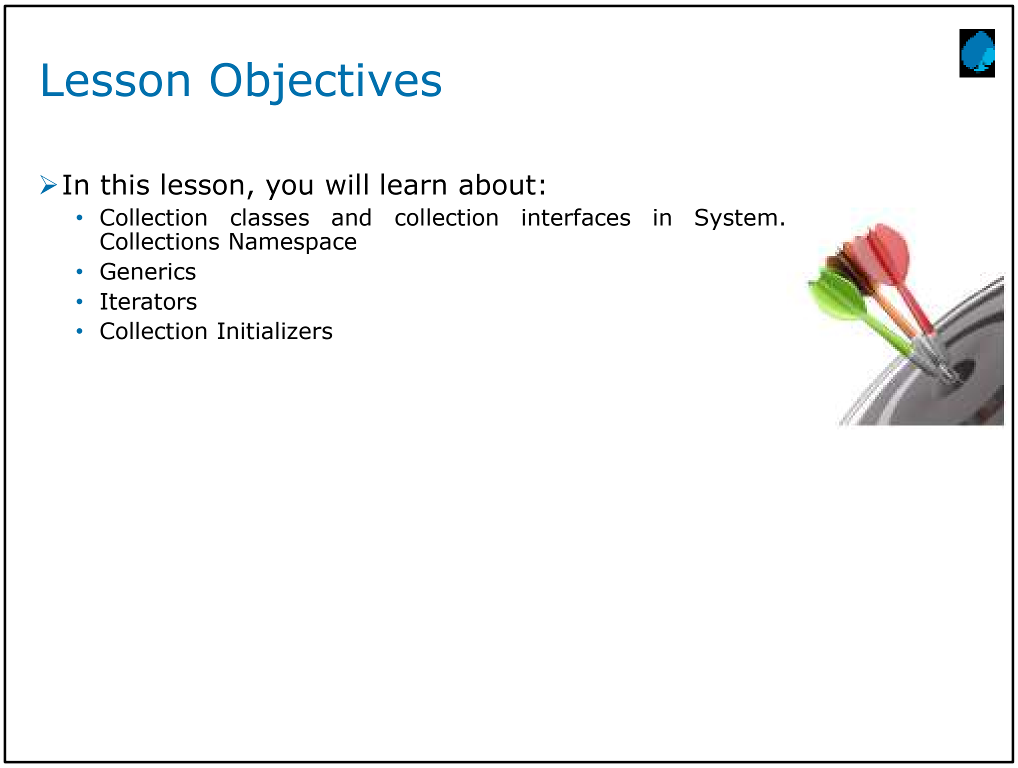


.NET Framework 4.6 and C# 7.0

Lesson 09 : Collections and Generics



Lesson Objectives

- In this lesson, you will learn about:
  - Collection classes and collection interfaces in System. Collections Namespace
  - Generics
  - Iterators
  - Collection Initializers

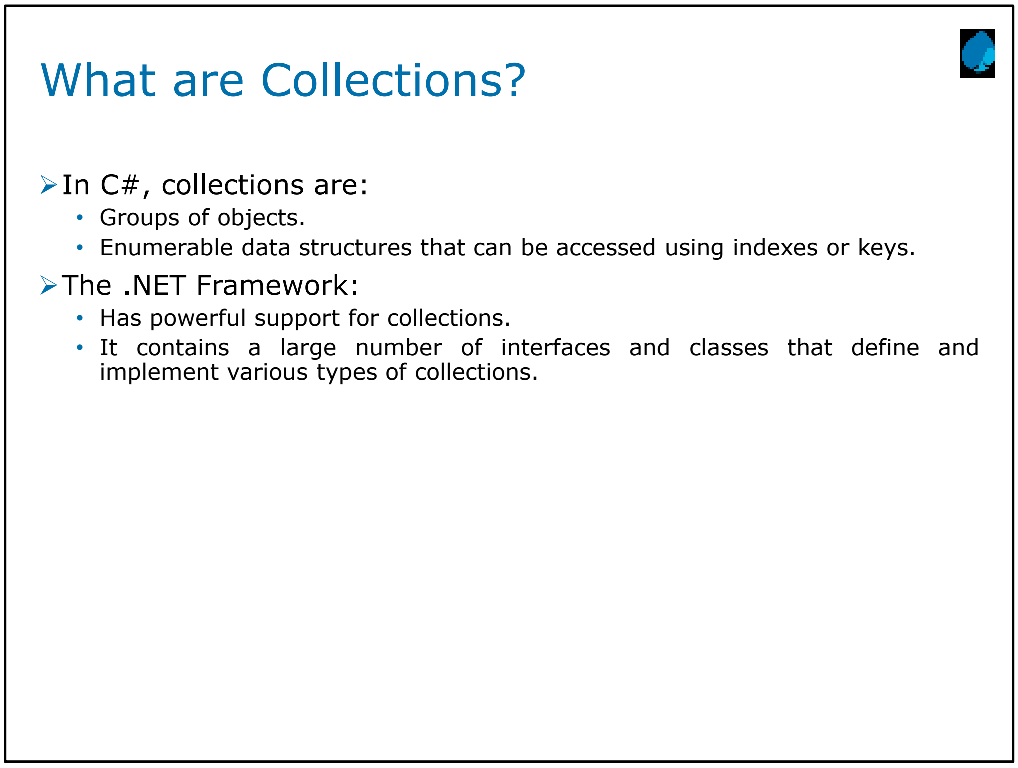
The .NET Framework supports four general types of collections: non-generic, specialized, bit based, and generic. The first three have been part of C# since version 1.0. The generic collections were added by C# 2.0. The non-generic collections implement several fundamental data structures, including a dynamic array, stack, and queue. They also include *dictionaries*, in which you can store key/ value pairs.

Need for collections

- You are developing an Student-tracking application.
- You implement a data structure named Student to store student information.
- However, you do not know the number of records that you need to maintain.
- You can store the data structure in an array, but then you would need to write code to add each new employee.
- To add a new item to an array, you first have to create a new array that has room for an additional element.

Need for collections

- Then, you need to copy the elements from the original array into the new array and add the new element.
- To simplify this process, the .NET Framework provides classes that are collectively known as Collections.
- By using collections, you can store several items within one object.
- Collections have methods that you can use to add and remove items.
- These methods automatically resize the corresponding data structures without requiring additional code.



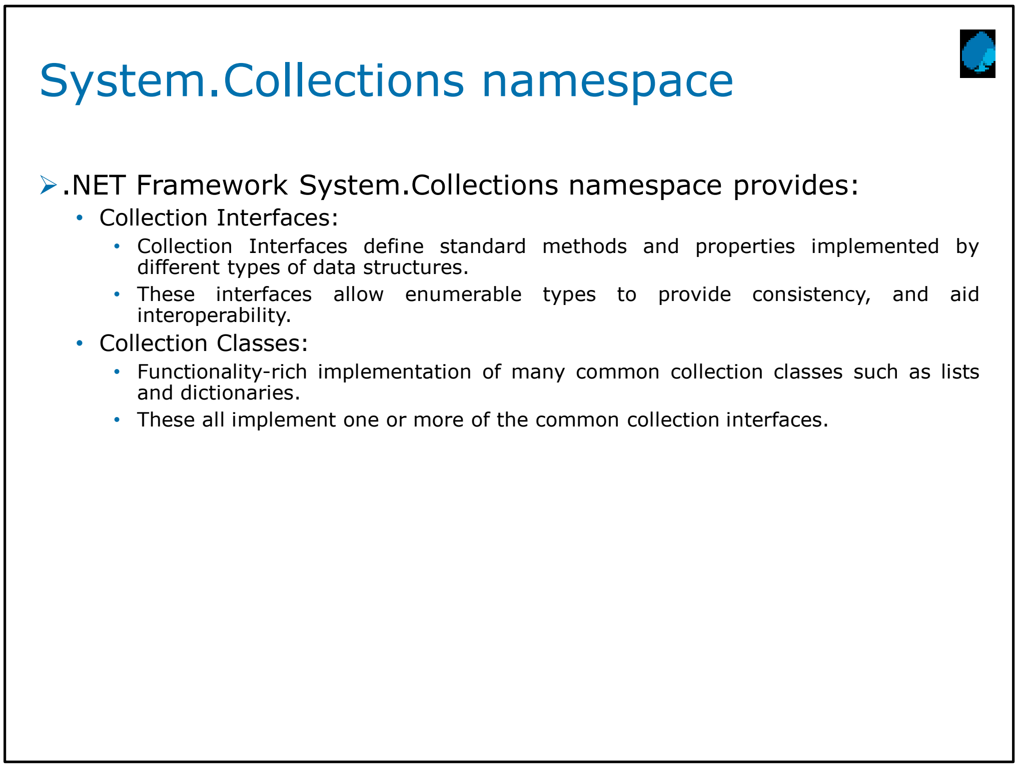
What are Collections?

- In C#, collections are:
  - Groups of objects.
  - Enumerable data structures that can be accessed using indexes or keys.
- The .NET Framework:
  - Has powerful support for collections.
  - It contains a large number of interfaces and classes that define and implement various types of collections.

What are Collections?

Collections simplify many programming tasks because they offer off-the-shelf solutions to several common, but sometimes tedious-to-develop, data structures. For example, there are built-in collections that support dynamic arrays, linked lists, stacks, queues, and hash tables.
The principal benefit of collections is that they standardize the way groups of objects are handled by your programs. All collections are designed around a set of clearly defined interfaces. Several built-in implementations of these interfaces, such as ArrayList, Hashtable, Stack, and Queue, are provided, which you can use as-is.

The non-generic collections operate on data of type **object**. Thus, they can be used to store any type of data, and different types of data can be mixed within the same collection. Of course, because they store **object** references, they are not type-safe. The non-generic collection classes and interfaces are in **System.Collections**.



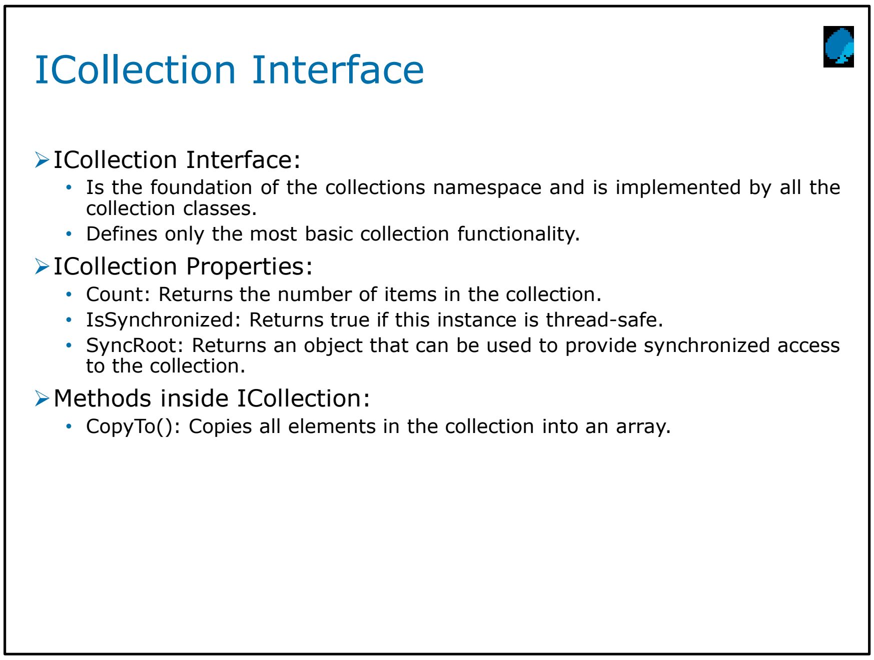
System.Collections namespace

- .NET Framework System.Collections namespace provides:
  - Collection Interfaces:
    - Collection Interfaces define standard methods and properties implemented by different types of data structures.
    - These interfaces allow enumerable types to provide consistency, and aid interoperability.
  - Collection Classes:
    - Functionality-rich implementation of many common collection classes such as lists and dictionaries.
    - These all implement one or more of the common collection interfaces.

System.Collection namespace:

The release of C# 2.0 caused a major change to collections because it added generic collections. This resulted in the addition of many new generic classes and interfaces, and two completely new namespaces. The inclusion of the generic collections essentially doubled the number of collection classes and interfaces. Thus, the Collections API is now quite large.

System.Collections defines a number of non-generic interfaces. It is necessary to begin with the collection interfaces because they determine the functionality common to all of the non-generic collection classes. ICollection: Defines the elements that all non-generic collections must have.



ICollection Interface

- ICollection Interface:
  - Is the foundation of the collections namespace and is implemented by all the collection classes.
  - Defines only the most basic collection functionality.
- ICollection Properties:
  - Count: Returns the number of items in the collection.
  - IsSynchronized: Returns true if this instance is thread-safe.
  - SyncRoot: Returns an object that can be used to provide synchronized access to the collection.
- Methods inside ICollection:
  - CopyTo(): Copies all elements in the collection into an array.

The ICollection Interface:

The ICollection interface is the foundation upon which all non-generic collections are built. It declares the core methods and properties that all non-generic collections will have. It also inherits the IEnumerable interface.
Properties inside ICollection:
Count is the most often used property because it contains the number of elements currently held in a collection. If Count is zero, then the collection is empty.

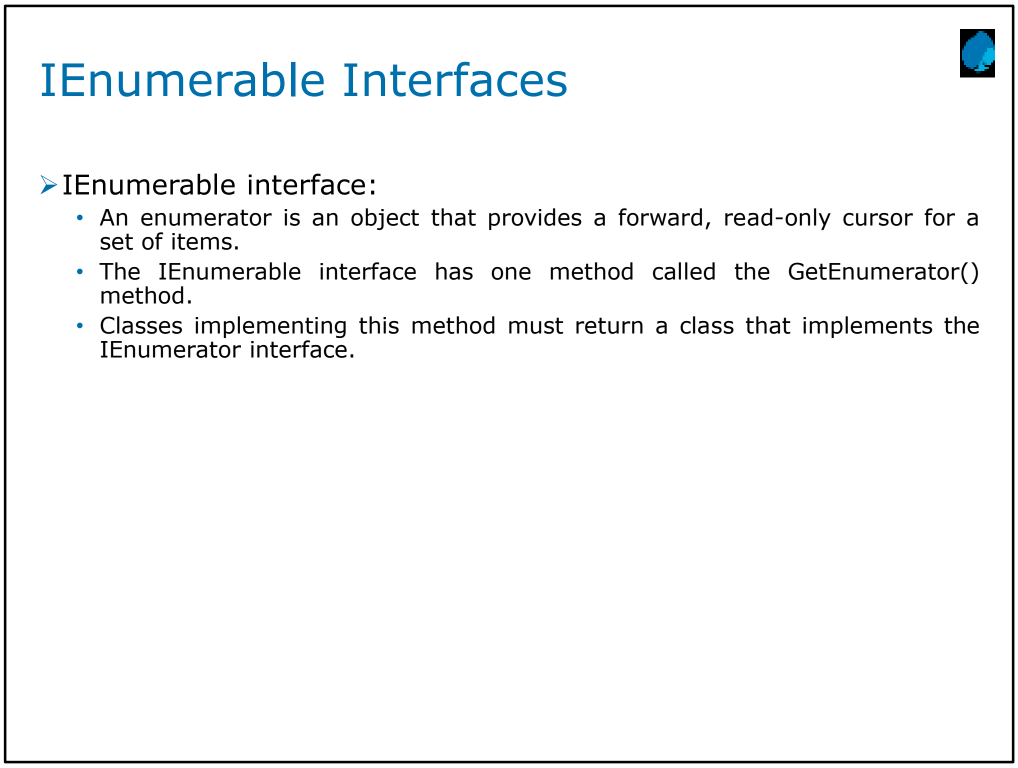
Methods inside ICollection:
- ICollection defines the following method:
- void CopyTo(Array target, int startIdx)
- CopyTo( ) copies the contents of a collection to the array specified by target, beginning at the index specified by startIdx. Thus, CopyTo( ) provides a pathway from a collection to a standard C# array.

Since ICollection inherits IEnumerable, it also includes the sole method defined by IEnumerable, that is, GetEnumerator( ), which is shown here:

- IEnumerator GetEnumerator( ): It returns the enumerator for the collection.

IEnumerable: Defines the **GetEnumerator()** method, which supplies the enumerator for a collection class.

A feature related to **IEnumerator** and **IEnumerable** is the *iterator.* Iterators were added by C# 2.0. They simplify the process of creating classes, such as custom collections, that can be cycled through by a **foreach** loop.



IEnumerable Interfaces

- IEnumerable interface:
  - An enumerator is an object that provides a forward, read-only cursor for a set of items.
  - The IEnumerable interface has one method called the GetEnumerator() method.
  - Classes implementing this method must return a class that implements the IEnumerator interface.

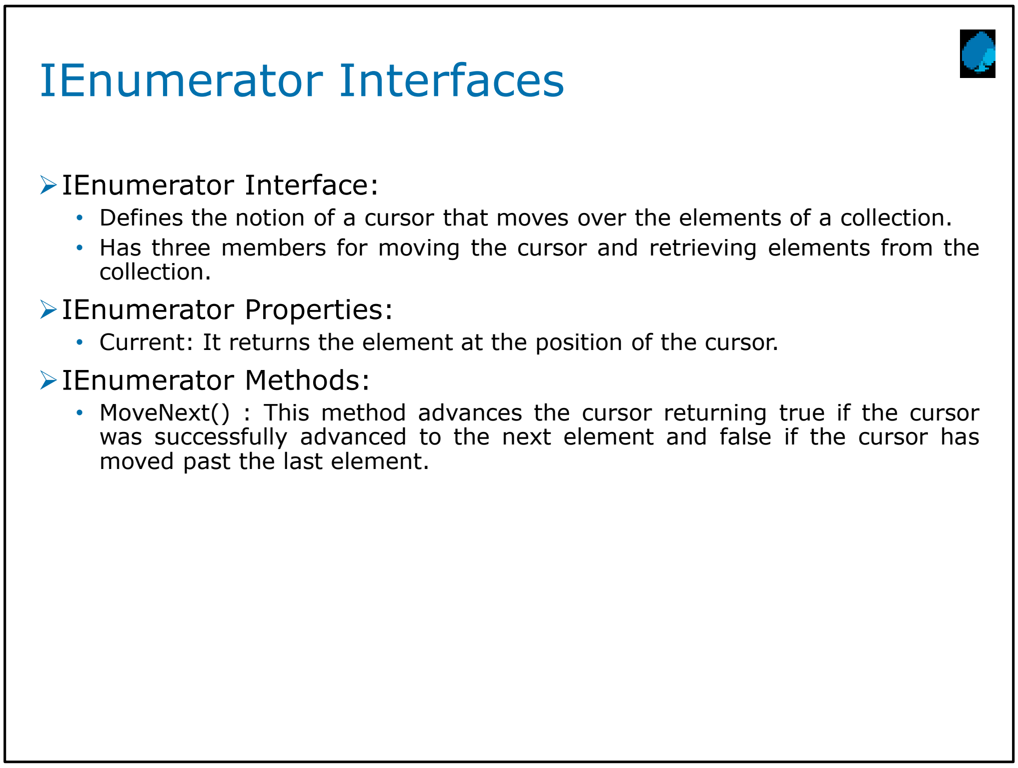
The IEnumerable Interface:
Fundamental to all collections is the concept of an enumerator, which is supported by the IEnumerator and IEnumerable interfaces. An enumerator provides a standardized way of accessing the elements within a collection, one at a time. Thus, it enumerates the contents of a collection. Because each collection must implement IEnumerable, the elements of any collection class can be accessed through the methods defined by IEnumerator. Therefore, with only small changes, the code that cycles through one type of collection can be used to cycle through another. As a point of interest, the foreach loop uses the enumerator to cycle through the contents of a collection.

IEnumerable is the non-generic interface that a class must implement if it is to support enumerators. As explained, all of the non-generic collection classes implement IEnumerable because it is inherited by ICollection. The sole method defined by IEnumerable is GetEnumerator( ), which is shown here:

```
IEnumerator GetEnumerator( )
```

It returns the enumerator for the collection. Also, implementing IEnumerable allows the contents of a collection to be obtained by a foreach loop.

IEnumerator Provides methods that enable the contents of a collection to be obtained one at a time.



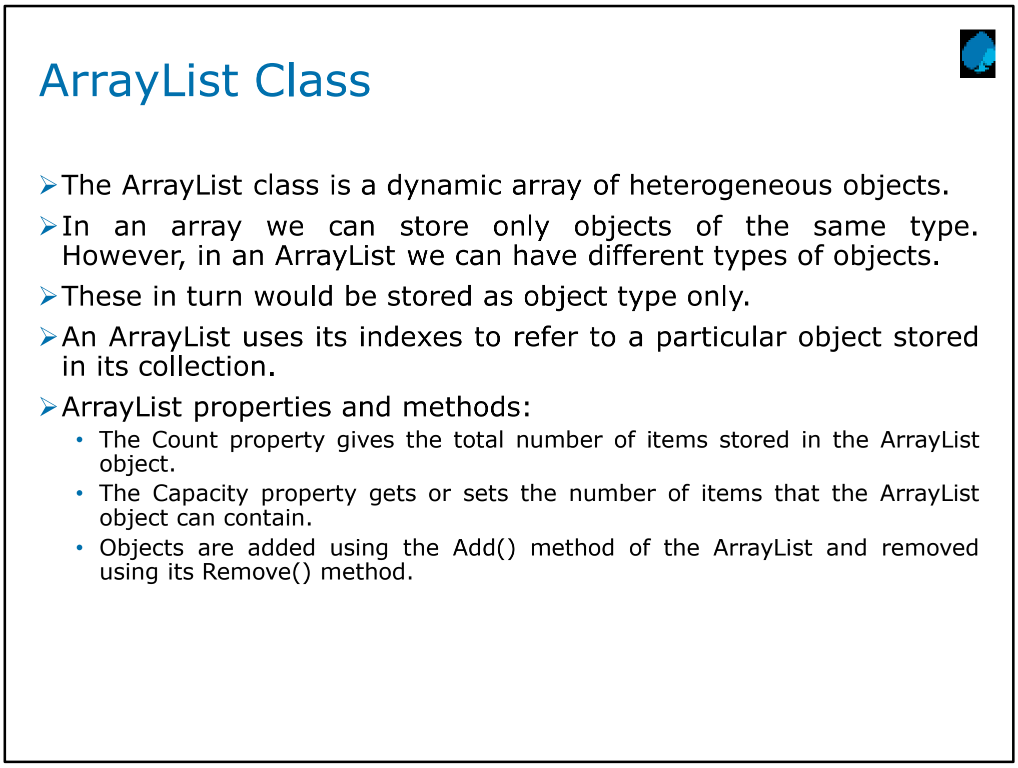
IEnumerator Interfaces

- IEnumerator Interface:
  - Defines the notion of a cursor that moves over the elements of a collection.
  - Has three members for moving the cursor and retrieving elements from the collection.
- IEnumerator Properties:
  - Current: It returns the element at the position of the cursor.
- IEnumerator Methods:
  - MoveNext() : This method advances the cursor returning true if the cursor was successfully advanced to the next element and false if the cursor has moved past the last element.

The IEnumerator Interface:
IEnumerator is the interface that defines the functionality of an enumerator.

Using its methods, you can cycle through the contents of a collection. For collections that store key/ value pairs (dictionaries), GetEnumerator( ) returns an object of type IDictionaryEnumerator, rather than IEnumerator. DictionaryEnumerator inherits IEnumerator and adds functionality to facilitate the enumeration of dictionaries.

ArrayList: A dynamic array. This is an array that can grow as needed.



ArrayList Class

- The ArrayList class is a dynamic array of heterogeneous objects.
- In an array we can store only objects of the same type. However, in an ArrayList we can have different types of objects.
- These in turn would be stored as object type only.
- An ArrayList uses its indexes to refer to a particular object stored in its collection.
- ArrayList properties and methods:
  - The Count property gives the total number of items stored in the ArrayList object.
  - The Capacity property gets or sets the number of items that the ArrayList object can contain.
  - Objects are added using the Add() method of the ArrayList and removed using its Remove() method.

Collection Classes:
The ArrayList class is a dynamic array of heterogeneous objects. Note that in an array we can store only objects of the same type. In an ArrayList, however, we can have different type of objects; these in turn would be stored as object type only. We can have an ArrayList object that stores integer, float, string, etc., but all these objects would only be stored as object type. An ArrayList uses its indexes to refer to a particular object stored in its collection. The Count property gives the total number of items stored in the ArrayList object. The Capacity property gets or sets the number of items that the ArrayList object can contain. Objects are added using the Add() method of the ArrayList and removed using its Remove() method.

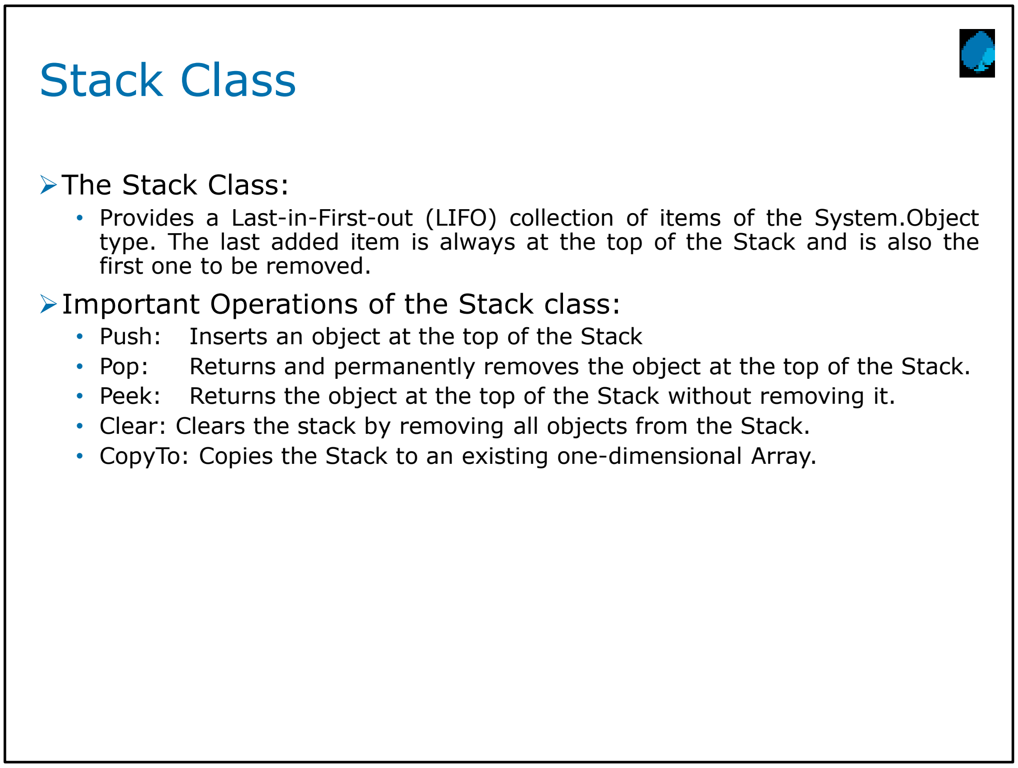
ArrayList: Example

- Example:

```
class Test
{
    static void Main()
    {
        int intValue = 100;
        double doubleValue = 20.5;
        ArrayList arrayList = new ArrayList();
            arrayList.Add("John");
            arrayList.Add(intValue);
            arrayList.Add(doubleValue);

            for (int index = 0; index <arrayList.Count; index++)
                    Console.WriteLine(arrayList[index]);
    }
}
```

Stack: A last-in, first-out list.



Stack Class

- The Stack Class:
  - Provides a Last-in-First-out (LIFO) collection of items of the System.Object type. The last added item is always at the top of the Stack and is also the first one to be removed.
- Important Operations of the Stack class:
  - Push: Inserts an object at the top of the Stack
  - Pop: Returns and permanently removes the object at the top of the Stack.
  - Peek: Returns the object at the top of the Stack without removing it.
  - Clear: Clears the stack by removing all objects from the Stack.
  - CopyTo: Copies the Stack to an existing one-dimensional Array.

The Stack Class:

A Stack can be visualized as a Stack of books arranged one on top of the other. You can pick or access the top-most book in the stack because it was the last book added. The Stack Class works on the same principle.

The stack is one of the most important data structures in computing. The collection class that supports a stack is called Stack
It implements the ICollection, IEnumerable, and ICloneable interfaces. Stack is a dynamic collection that grows as needed to accommodate the elements it must store.

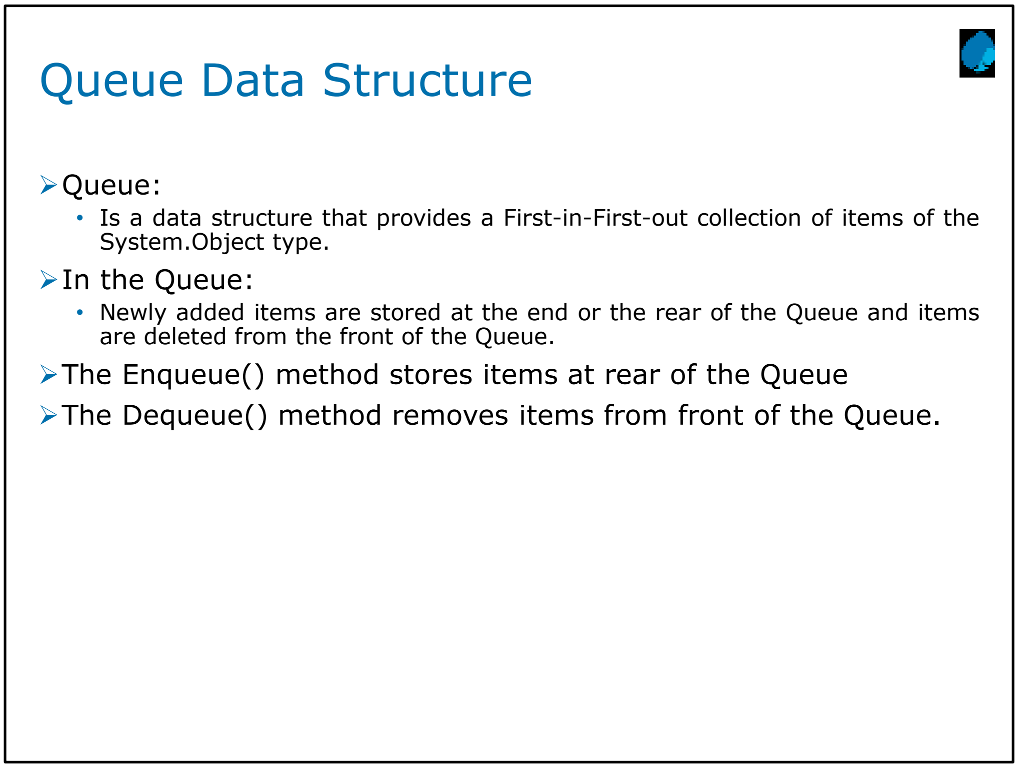
Stack Class: Example

```
class Test
{
    static void Main()
    {
        Stack stackObject = new Stack();
        stackObject.Push("Joydip");
        stackObject.Push("Steve");
        stackObject.Push("Jini");

        while (stackObject.Count > 0)
            Console.WriteLine(stackObject.Pop());
        Console.ReadLine();
    }
}
```

You use the Stack class when you need to retrieve objects in the reverse order in which you added them. In other words, the last object added to a stack is the first to be retrieved. A stack follows the last-in, first out (LIFO) principle, where elements are added to and removed from a collection at the same end of the stack.

Queue: A first-in, first-out list.



Queue Data Structure

- Queue:
  - Is a data structure that provides a First-in-First-out collection of items of the System.Object type.
- In the Queue:
  - Newly added items are stored at the end or the rear of the Queue and items are deleted from the front of the Queue.
- The Enqueue() method stores items at rear of the Queue
- The Dequeue() method removes items from front of the Queue.

The Queue:

A Queue can be visualized as a line in a grocery shop, where customers enter the queue from the rear and exit the queue from the front. Therefore the first customer to enter the queue is the first to exit the queue. The Queue class follows the same principle.

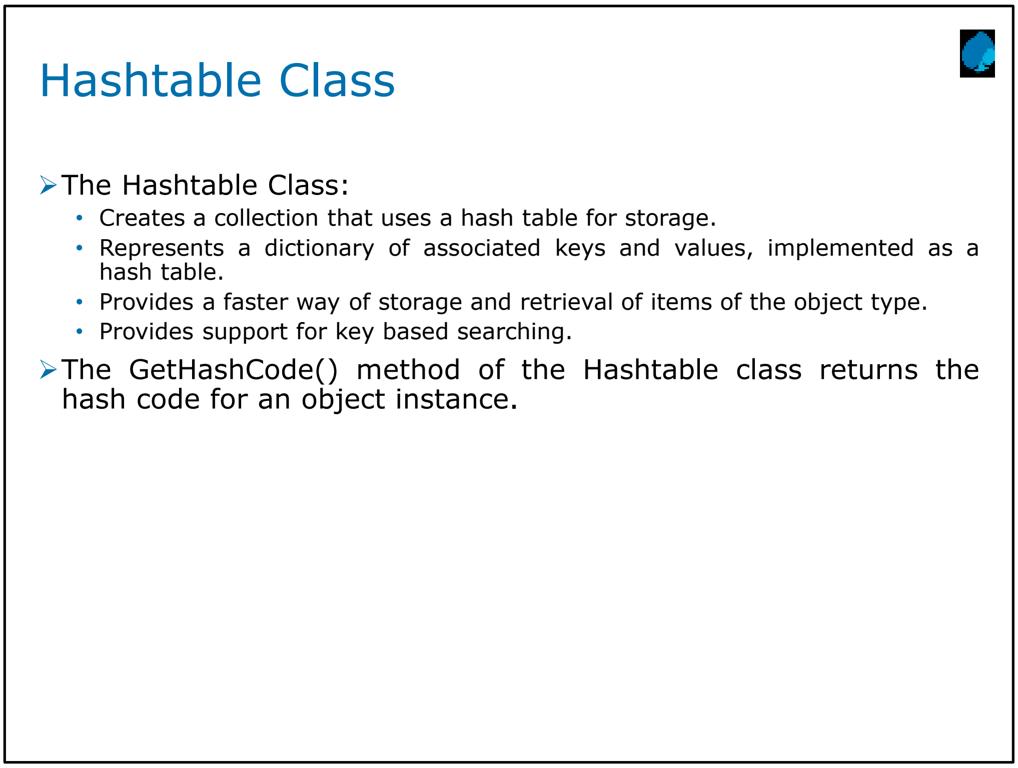
The collection class that supports a queue is called Queue. It implements the ICollection, IEnumerable, and ICloneable interfaces. Queue is a dynamic collection that grows as needed to accommodate the elements it must store. When more room is needed, the size of the queue is increased by a growth factor, which by default is 2.0.

Queue: Example

```
class Test
{
    static void Main()
    {
        Queue queueObject = new Queue();
        queueObject.Enqueue("Joydip");
        queueObject.Enqueue("Steve");
        queueObject.Enqueue("Jini");

        while (queueObject.Count > 0)
            Console.WriteLine(queueObject.Dequeue());
        Console.ReadLine();
    }
 }
```

Hashtable: A hash table for key/value pairs.



Hashtable Class

- The Hashtable Class:
  - Creates a collection that uses a hash table for storage.
  - Represents a dictionary of associated keys and values, implemented as a hash table.
  - Provides a faster way of storage and retrieval of items of the object type.
  - Provides support for key based searching.
- The GetHashCode() method of the Hashtable class returns the hash code for an object instance.

The Hashtable Class:

The Hashtable class represents a collection of name/value pairs that are organized on the basis of the hash code of the key being specified.

A hash table stores information using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored in the table. The transformation of the key into its hash code is performed automatically—you never see the hash code, itself. The advantage of hashing is that it allows the execution time of lookup, retrieve, and set operations to remain constant, even for large sets. Hashtable implements the IDictionary, ICollection, IEnumerable, ISerializable, IDeserializationCallback, and ICloneable interfaces.

Hashtable Class: Example

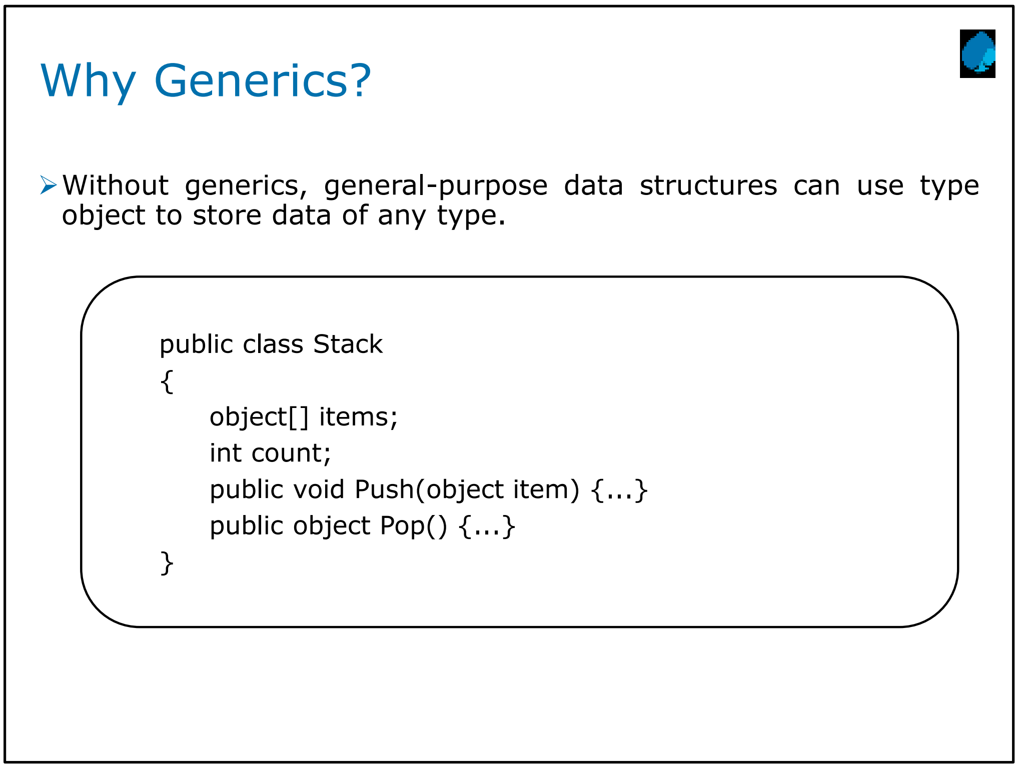
```
class Test
{
        static void Main()
    {
            Hashtable hashTable = new Hashtable();
            hashTable.Add(1, "Joydip");
            hashTable.Add(2, "Manashi");
            hashTable.Add(3, "Jini");
            hashTable.Add(4, "Piku");
            Console.WriteLine("The keys and values are:");
            foreach (int k in hashTable.Keys)
            {
                    Console.WriteLine(k);
                    Console.WriteLine(hashTable[k].ToString());
            }
    }
}
```

Additional notes for instructor

Demo

- Demo on Collection Classes

Add the notes here.



Why Generics?

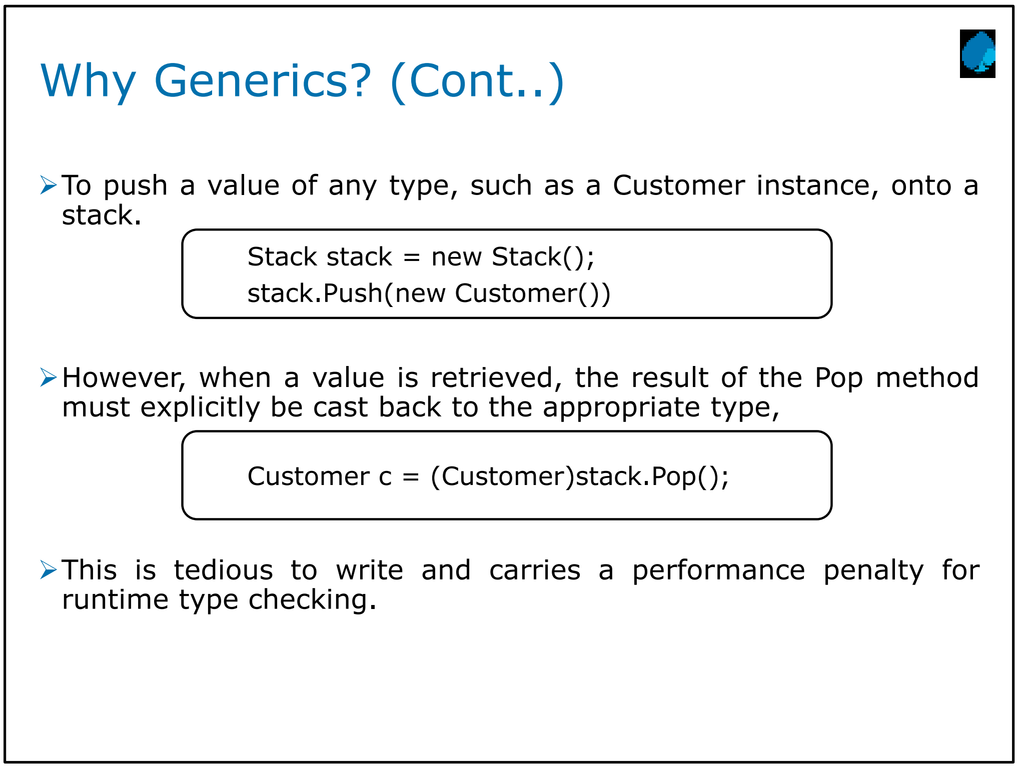
- Without generics, general-purpose data structures can use type object to store data of any type.

```
public class Stack
{
      object[] items;
      int count;
      public void Push(object item) {...}
      public object Pop() {...}
}
```

Why Generics?

Without generics, general-purpose data structures can use type object to store data of any type.
For example, the following simple Stack class stores its data in an object array, and its two methods, Push and Pop, use object to accept and return data, respectively.

```
public class Stack
{
      object[] items;
      int count;
      public void Push(object item) {...}
      public object Pop() {...}
}
```



Why Generics? (Cont..)

- To push a value of any type, such as a Customer instance, onto a stack.

```
Stack stack = new Stack();
stack.Push(new Customer())
```

- However, when a value is retrieved, the result of the Pop method must explicitly be cast back to the appropriate type,

```
Customer c = (Customer)stack.Pop();
```

- This is tedious to write and carries a performance penalty for runtime type checking.

Why Generics? (Cont..)
Although using type object makes the Stack class flexible, it is not without drawbacks.
For example, it is possible to push a value of any type, such as a Customer instance, onto a stack. However, when a value is retrieved, the result of the Pop method must explicitly be cast back to the appropriate type, which is tedious to write and carries a performance penalty for runtime type checking.

```
Stack stack = new Stack();
stack.Push(new Customer());
Customer c = (Customer)stack.Pop();
```

Why Generics? (Cont..)

- Similarly, if a value of a value type, such as an int, is passed to the Push method, it is automatically boxed.
- When the int is later retrieved, it must be unboxed with an explicit type cast.

```
Stack stack = new Stack();
stack.Push(3);
int i = (int)stack.Pop();
```

- Such boxing and unboxing operations add performance overhead because they involve dynamic memory allocations and runtime type checks.

Why Generics? (Cont..)
If a value of a value type, such as an int, is passed to the Push method, it is automatically boxed. When the Int is later retrieved, it must be unboxed with an explicit type cast.

```
Stack stack = new Stack();
stack.Push(3);
int i = (int)stack.Pop();
```

Such boxing and unboxing operations add performance overhead because they involve dynamic memory allocations and runtime type checks.
A further issue with the Stack class is that it is not possible to enforce the kind of data placed on a stack. Indeed, a Customer instance can be pushed on a stack and then accidentally cast to the wrong type after it is retrieved.

```
Stack stack = new Stack();
stack.Push(new Customer());
string s = (string) stack.Pop();
```

Although the previous code is an improper use of the Stack class, the code is technically speaking correct and a compile-time error is not reported. The problem does not become apparent until the code is executed, at which point an InvalidCastException is thrown. The Stack class would clearly benefit from the ability to specify its element type. With generics, that becomes possible.

What is Generics?

- Generics provide a facility for creating types that have type parameters.
- Following example declares a generic Stack class with a type parameter T:

```
public class Stack<T>
{
     T[ ] items;
     int count;
     public void Push(T item) {...}
     public T Pop() {...}
}
```

What is Generics?
Generics permit classes, structs, interfaces, delegates, and methods to be parameterized by the types of data they store and manipulate.
C# generics will be immediately familiar to users of generics in Eiffel or Ada or to users of C++ templates; however, they do not suffer many of the complications of the latter. Generics provide a facility for creating types that have type parameters.
The following example declares a generic Stack class with a type parameter T.
The type parameter is specified in < and > delimiters after the class name.
Rather than forcing conversions to and From object, instances of Stack<T> accept the type for which they are created and store data of that type without conversion.
The type parameter T acts as a placeholder until an actual type is specified at use. Note that T is used as the element type for the internal items array, the type for the parameter to the Push method, and the return type for the Pop method.

```
public class Stack<T>
{
          T[] items;
          int count;
          public void Push(T item) {...}
          public T Pop() {...}
}
```

What is Generics? (cont..)

- The type parameter is specified in < and > delimiters after the class name.
- The type parameter T acts as a placeholder until an actual type is specified at use.
- In the following example, int is given as the type argument for T:

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

What is Generics? (Cont..)
When the generic class Stack<T> is used, the actual type to substitute for T is specified. In the following example, int is given as the type argument for T.

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

What is Generics? (cont..)

- Similarly we can have:

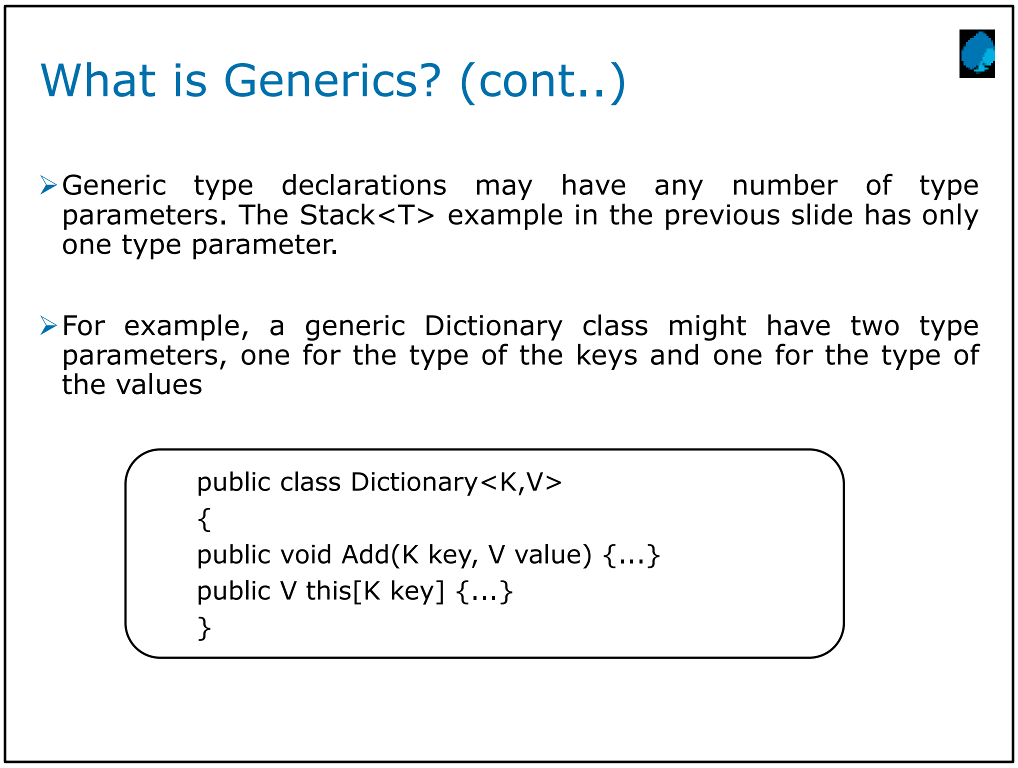
```
Stack<Customer> objStack = new
Stack<Customer>();
objStack.Push(new Customer());
Customer objCust = objStack.Pop();
```

What is Generics? (Cont..)

The Stack<int> type is called a constructed type. In the Stack<int>type, every occurrence of T is replaced with the type argument int. When an instance of Stack<int> is created, the native storage of the items array is an int[] rather than object[] , providing substantial storage efficiency compared to the nongeneric Stack . Likewise, the Push and Pop methods of a Stack<int> operate on int values, making it a compile-time error to push values of other types onto the stack and eliminating the need to explicitly cast values back to their original type when they are retrieved. Generics provide strong typing, meaning for example that it is an error to push an int onto a stack of Customer objects. Just as a Stack<int> is restricted to operate only on int values, so is Stack<Customer> restricted to Customer objects, and the compiler will report errors on the last two lines of the following example.

```
Stack<Customer> objStack = new Stack<Customer>();
objStack.Push(new Customer());
Customer c = objStack.Pop();
objStack.Push(3); // Type mismatch error
```

```
int x = objStack.Pop(); // Type mismatch error
```



What is Generics? (cont..)

- Generic type declarations may have any number of type parameters. The Stack<T> example in the previous slide has only one type parameter.
- For example, a generic Dictionary class might have two type parameters, one for the type of the keys and one for the type of the values

```
public class Dictionary<K,V>
{
public void Add(K key, V value) {...}
public V this[K key] {...}
}
```

What is Generics? (Contd..)

Generic type declarations may have any number of type parameters. The previous Stack<T> example has only one type parameter, but a generic Dictionary class might have two type parameters, one for the type of the keys and one for the type of the values.

```
public class Dictionary<K,V>
{
public void Add(K key, V value) {...}
public V this[K key] {...}
}
```

When Dictionary<K,V> is used, two type arguments would have to be supplied.

```
Dictionary<string,Customer> objDict = new
```

```
Dictionary<string,Customer>();

objDict.Add("Peter", new Customer());
Customer objCust = objDict ["Peter"];
```

What is Generics? (cont..)

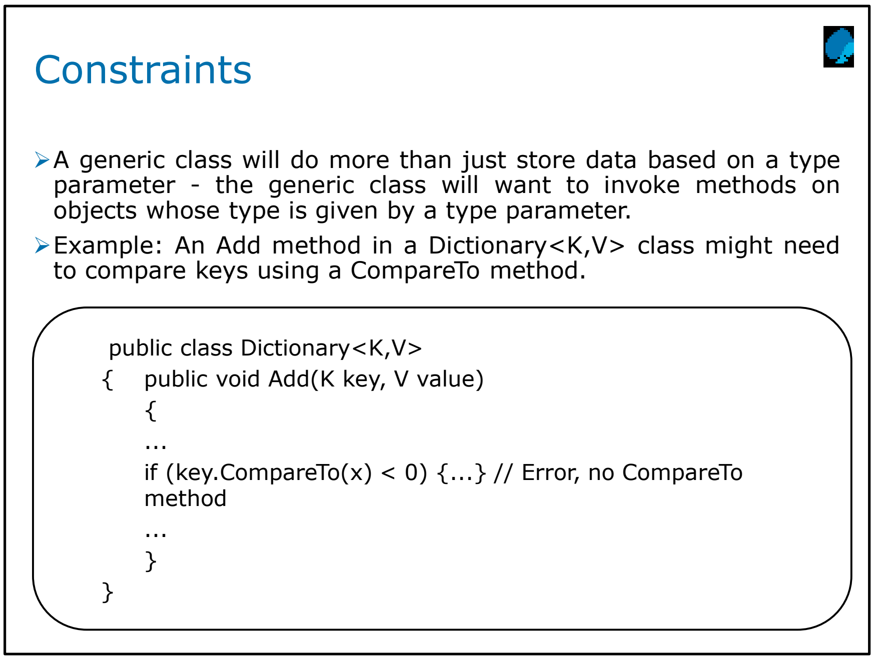
- When Dictionary<K,V> is used, two type arguments would have to be supplied:

```
Dictionary<string,Customer> objDict = new Dictionary<string,Customer>();
objDict.Add("Peter", new Customer());
Customer objCust = objDict["Peter"];
```

Demo

- Demo on Generics

Add the notes here.



Constraints

- A generic class will do more than just store data based on a type parameter - the generic class will want to invoke methods on objects whose type is given by a type parameter.
- Example: An Add method in a Dictionary<K,V> class might need to compare keys using a CompareTo method.

```
 public class Dictionary<K,V>
{    public void Add(K key, V value)
     {
     …
     if (key.CompareTo(x) < 0) {…} // Error, no CompareTo
     method
     …
     }
}
```

Generics: Constraints
Commonly, a generic class will do more than just store data based on a type parameter. Often, the generic class will want to invoke methods on objects whose type is given by a type parameter. For example, an Add method in a Dictionary<K,V> class might need to compare keys using a CompareTo method.

```
public class Dictionary<K,V>
    {
        public void Add(K key, V value)
                    {
                    ...
                    if (key.CompareTo(x) < 0) {...} // Error, no CompareTo method
                    ...
                    }
                    }
```

Because the type argument specified for K could be any type, the only members that can be assumed to exist on the key parameter are those declared by type object, such as Equals, GetHashCode, and ToString; a compile-time error therefore occurs in the previous example. It is of course possible to cast the key parameter to a type that contains a CompareTo method.

Constraints (Cont..)

- To provide stronger compile-time type checking and reduce type casts, C# permits an optional list of. constraints to be supplied for each type parameter.
- A type parameter constraint specifies a requirement that a type must fulfill in order to be used as an argument for that type parameter.
- Constraints are declared using the word where, followed by the name of a type parameter, followed by a list of class or interface types and optionally the constructor constraint new().

Generics: Constraints (Cont..)
For example, the key parameter could be cast to IComparable.public class Dictionary<K,V>

```
{
        public void Add(K key, V value)
        {
        ...
        if (((IComparable)key).CompareTo(x) < 0) {...}
        ...
        }
}
```

Although this solution works, it requires a dynamic type check at runtime, which adds overhead. It furthermore defers error reporting to runtime, throwing an InvalidCastException if a key does not implement IComparable.

Constraints (Cont..)

- For the Dictionary<K,V> class to ensure that keys always implement IComparable, the class declaration can specify a constraint for the type parameter K.

```
public class Dictionary<K,V> where K: IComparable
{
    public void Add(K key, V value)
    {
            ...
            if (key.CompareTo(x) < 0) {...}
            ...
    }
}
```

Constraints (Cont..)

- Given the above declaration, the compiler will ensure that any type argument supplied for K is a type that implements IComparable
- For a given type parameter, it is possible to specify any number of interfaces as constraints, but no more than one class

```
public class EntityTable<K,E>
where K: IComparable<K>, IPersistable
where E: Entity, new()
{
      public void Add(K key, E entity)
      {           ...
                  if (key.CompareTo(x) < 0) {...}
                  ...
      }
}
```

Demo

- Demo on constraints

Add the notes here.

Generic Methods

- A type parameter may not be needed for an entire class but is needed only inside a particular method

```
void PushMultiple(Stack<int> stack, params int[] values)
{
     foreach (int value in values)
               stack.Push(value);
}
```

- The above method can be used to push multiple int values

```
Stack<int> stack = new Stack<int>();
PushMultiple( stack, 1, 2, 3, 4);
```

Generic Methods (cont..)

- The previous method works with the particular constructed type Stack<int> only.
- To have it work with any Stack<T>, the method must be written as a generic method.
- A generic method has one or more type parameters specified in < and > delimiters after the method name.
- A generic PushMultiple method:

```
void PushMultiple<T>(Stack<T> stack, params T[] values)
{
    foreach (T value in values)
            stack.Push(value);
}
```

Generic Methods (cont..)

- When calling a generic method, type arguments are given in angle brackets in the method invocation
- Example:

```
Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

Demo

- Demo on Generic Method

Add the notes here.

Generic Interfaces

- Often used to define interfaces either for generic Collection classes, or for the generic classes that represent items in the Collection
- Preferable to use generic interfaces, such as IComparable<T> rather than IComparable, in order to avoid boxing and unboxing operations on value types.
- Generic Interfaces inside System.Collections.Generic:
  - ICollection<T>
  - IComparable<T>
  - IEnumerable<T>
  - IEnumerator<T>
  - IComparer<T>

Generic Interfaces:
It is often useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection. With generic classes it is preferable to use generic interfaces, such as IComparable<T> rather than IComparable, in order to avoid boxing and unboxing operations on value types. The .NET Framework 2.0 class library defines several new generic interfaces for use with the new collection classes in the System.Collections.Generic namespace.

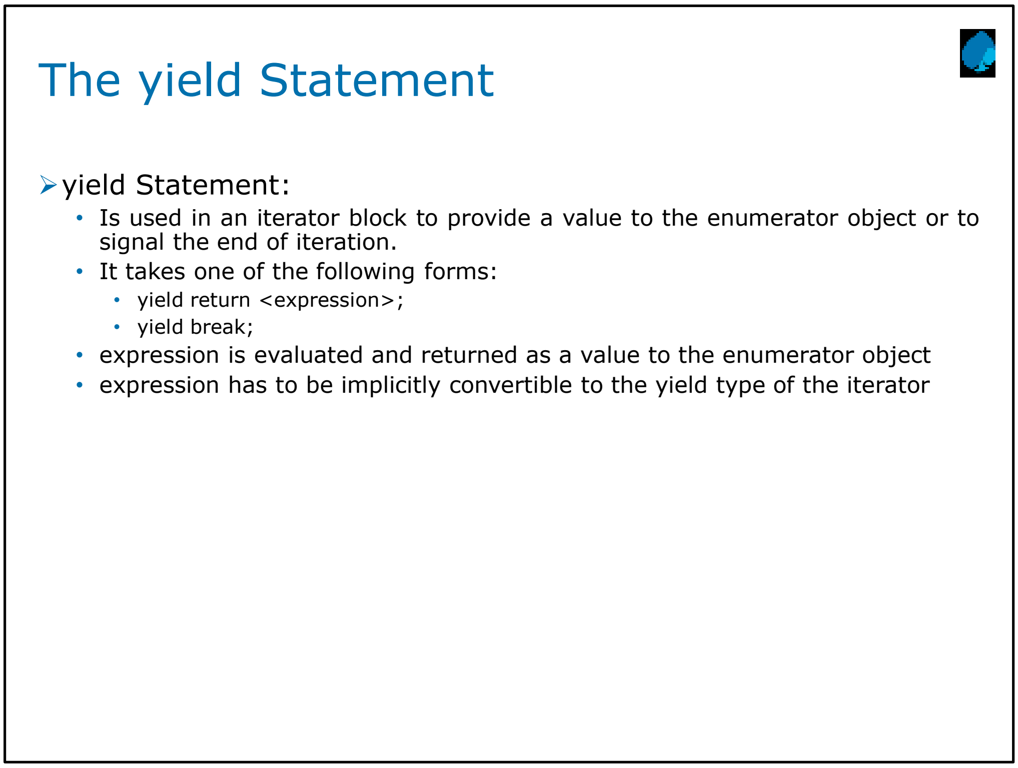
An iterator is a method, operator, or accessor that returns the members of a set of objects, one member at a time, from start to finish. For example, assuming some array that has five elements, then an iterator for that array will return those five elements, one at a time. By implementing an iterator, you make it possible for an object of a class to be used in a **foreach** loop.

What are Iterators?

- An iterator is a method, get accessor or operator that enables you to support foreach iteration in a class or struct without having to implement the entire IEnumerable interface.
- An iterator is a section of code that returns an ordered sequence of values of the same type.
- The iterator code uses the yield return statement to return each element in turn, yield break ends the iteration.
- When the compiler detects an iterator, it automatically generates the Current(), MoveNext() and Dispose() methods of the IEnumerable or IEnumerable<T> interface.
- The return type of an iterator must be IEnumerable, IEnumerator, IEnumerable<T>, or IEnumerator<T>
- Iterators are especially useful with collection classes.

What are Iterators?
Iterators are a new feature in C# 2.0. An iterator is a method, get accessor or operator that enables you to support for each iteration in a class or struct without having to implement the entire IEnumerable interface. Instead, you provide just an iterator, which simply traverses the data structures in your class. When the compiler detects your iterator, it will automatically generate the Current, MoveNext and Dispose methods of the IEnumerable or IEnumerable<T> interface.
An iterator is a section of code that returns an ordered sequence of values of the same type. An iterator can be used as the body of a method, an operator, or a get accessor.
The iterator code uses the yield return statement to return each element in turn. yield break ends the iteration.
Multiple iterators can be implemented on a class. Each iterator must have a unique name just like any class member, and can be invoked by client code in a foreach statement as follows: foreach(int x in SampleClass.Iterator2){}

The return type of an iterator must be IEnumerable, IEnumerator, IEnumerable<T>, or IEnumerator<T>.
The yield keyword is used to specify the value, or values, returned. When the yield return statement is reached, the current location is stored. Execution is restarted from this location the next time the iterator is called.
Iterators are especially useful with collection classes, providing an easy way to iterate non-trivial data structures such as binary trees.



The yield Statement

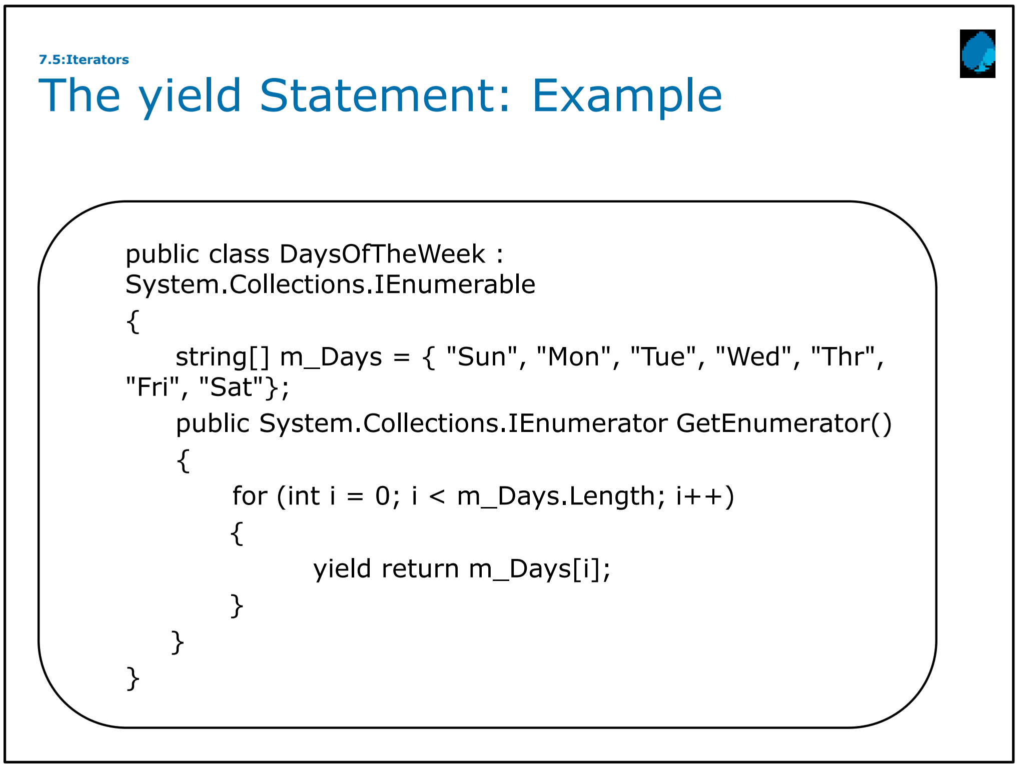
- yield Statement:
  - Is used in an iterator block to provide a value to the enumerator object or to signal the end of iteration.
  - It takes one of the following forms:
    - yield return <expression>;
    - yield break;
  - expression is evaluated and returned as a value to the enumerator object
  - expression has to be implicitly convertible to the yield type of the iterator

The yield Statement:
In the following example, the yield statement is used inside an iterator block, which is the method Power(int number, int power). When the Power method is invoked, it returns an enumerable object that contains the powers of a number. Notice that the return type of the Power method is IEnumerable, an iterator interface type

```
using System;
using System.Collections;
public class List
{
            public static IEnumerable Power(int number, int exponent)
            {
                        int counter = 0;
                        int result = 1;
                        while (counter++ < exponent)
                        {
                         result = result * number; yield return result;
                        }
            }
            static void Main()
            {
                        foreach (int i in Power(2, 8))
                        {
                        Console.Write("{0} ", i);
                        }
            }
}
```

Output

```
2 4 8 16 32 64 128 256
```



The yield Statement: Example

```
public class DaysOfTheWeek :
System.Collections.IEnumerable
{
    string[] m_Days = { "Sun", "Mon", "Tue", "Wed", "Thr",
"Fri", "Sat"};
    public System.Collections.IEnumerator GetEnumerator()
    {
        for (int i = 0; i < m_Days.Length; i++)
        {
            yield return m_Days[i];
        }
    }
}
```

Iterators: An Example

In this example, the class DaysOfTheWeek is a simple collection class that stores the days of the week as strings. After each iteration of a foreach loop, the next string in the collection is returned.

```
public class DaysOfTheWeek : System.Collections.IEnumerable
{
        string[] m_Days = { "Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat" };
        public System.Collections.IEnumerator GetEnumerator()
        {
              for (int i = 0; i < m_Days.Length; i++)
              {
                        yield return m_Days[i];
              }
        }
}

class TestDaysOfTheWeek
{
          static void Main()
          {
                 DaysOfTheWeek week = new DaysOfTheWeek();
                 foreach (string day in week)
                {
                       System.Console.Write(day + " ");
                }
          }
}
```

Output

```
Sun Mon Tue Wed Thr Fri Sat
```

Iterators: Example

```
class TestDaysOfTheWeek
{
        static void Main()
        {
              DaysOfTheWeek week = new DaysOfTheWeek();
              foreach (string day in week)
              {
                       System.Console.Write(day + " ");
              }
        }
}
```

Demo

- Demo on Iterators

Add the notes here.

What are Collection Initializers?

- Any object that implements IEnumerable<T> and has a public Add method can have its values initialized with a collection initializer
- A collection initializer consists of a sequence of element initializers, enclosed by { and } tokens and separated by commas.
- Example:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 6, 8, 9 };
```

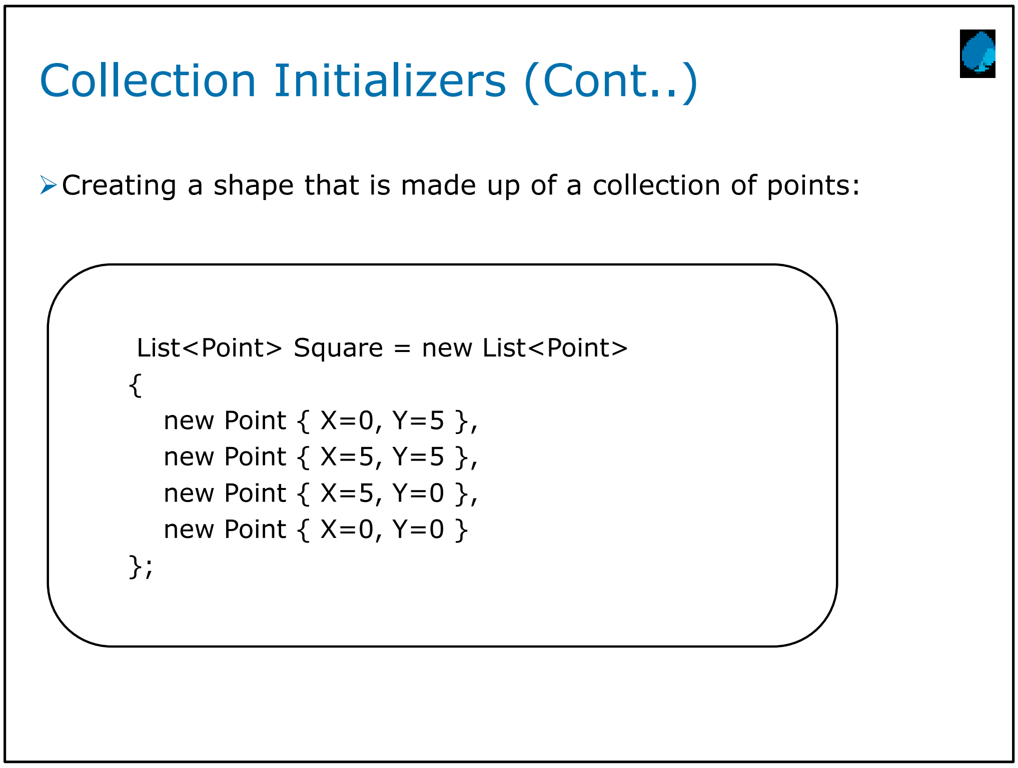
What are Collection Initializers?

A collection initializer consists of a sequence of element initializers, enclosed by { and } tokens and separated by commas. Each element initializer specifies an element to be added to the collection object being initialized. To avoid ambiguity with member initializers, element initializers cannot be assignment expressions.

Following is an example of an object creation expression that includes a collection initializer:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 6, 8, 9 };
```

The collection object to which a collection initializer is applied must be of a type that implements System.Collections.Generic.ICollection<T> for exactly one T. Furthermore, an implicit conversion must exist from the type of each element initializer to T. A compile-time error occurs if these requirements are not satisfied. A collection initializer invokes the ICollection<T>.Add(T) method for each specified element in order.



Collection Initializers (Cont..)

- Creating a shape that is made up of a collection of points:

```
List<Point> Square = new List<Point>
{
    new Point { X=0, Y=5 },
    new Point { X=5, Y=5 },
    new Point { X=5, Y=0 },
    new Point { X=0, Y=0 }
};
```

What are Collection Initializers? (Cont..)
Using the Point class created above, let's create a shape that is made up of a collection of points.

```
List<Point> Square = new List<Point>
{
   new Point { X=0, Y=5 },
   new Point { X=5, Y=5 },
   new Point { X=5, Y=0 },
   new Point { X=0, Y=0 }
};
```

The following class represents a contact with a name and a list of phone numbers:

```
Public class Contact
{
            string name;
            List<string> phoneNumbers = new List<string>();
            public string Name
            {
                        get { return name; }
                        set { name = value; } }
            public List<string> PhoneNumbers
            {
                        get { return phoneNumbers; }}

}
```

A List<Contact> can be created and initialized as follows:

```
var contacts = new List<Contact>
{
            new Contact{
            Name = "Chris Smith",
            PhoneNumbers = { "206-555-0101", "425-882-8080“}},

            new Contact {
            Name = "Bob Harris",
            PhoneNumbers = { "650-555-0199" }}

};
```

Summary

- In this module, we explored System.Collections namespace and the collection interfaces and classes present in it.
- These collection Interfaces and classes are:

Collection Interfaces	**Collection Classes**
ICollection	ArrayList
IEnumerable	Stack
IEnumerator	Queue
	BitArray
	HashTable

Review Question

- What are the advantages of an ArrayList? How is it different from an Array?
- What is the use of the IEnumerable interface?
- What are the different operations possible with a BitArray Class?
- What is the difference between pop and peek method of a Stack class?
- What is the need for Generics?
- Can Delegates also be made Generic?