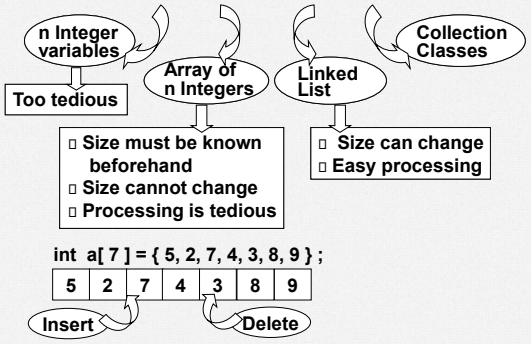
Collections and Generics

C# Programming

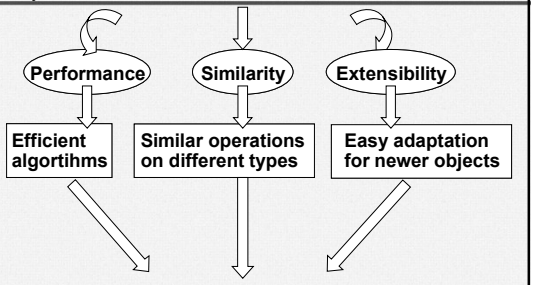
Collections

- The .NET Framework provides specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables.
- Most collection classes implement the same interfaces, and these interfaces may be inherited to create new collection classes that fit more specialized data storage needs.
- Generic collection classes provide increased type-safety and in some cases can provide better performance, especially when they store value types.

Store n Integers



Why Collections?



System.Collections

Collections

- Collection is a set of classes that include data structures that store number of objects and provide operations to work with them as one entity. The set of classes in this framework use efficient algorithms to manage the collection that results in better performances.
- Collection Classes are present in one of the following namespaces:
  - **System.Collections**
    - A collection of container classes that allows storage of any type of .NET objects
  - **System.Collections.Generic**
    - A collection of container classes that allows only specified type of objects.
    - In some cases these classes perform better especially when storing value types.

Non-generic Interfaces

- Most of the collection classes share common interfaces.
- Collection Interfaces
  - **ICollection**
  - **IList**
  - **IDictionary**
- Interface iterators/enumerators
  - **IEnumerator** and **IEnumerable**
  - **IDictionaryEnumerator**
  - Collections supports **foreach** statement for iteration. They are generally used instead of the above two since they are very convenient.

Presented by

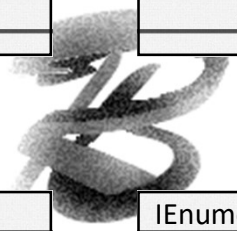
Ranjan Bhatnagar

ICollection

- **ICollection** interface is parent for all the other non-generic interfaces – **IList and IDictionary**
- It inherits from **IEnumerable** .
- Important members and their description in API:
  - **Count**
  - **void CopyTo(Array array, int index)**
  - **IEnumerator GetEnumerator()**

IList

- Classes implementing **IList** is an interface that allows a list of any objects that can be individually accessed using index.
- Implements **ICollection, IEnumerable**
- Members and their description in API:
  - **IsFixedSize**
  - **IsReadOnly**
  - **Item**
  - **int Add (Object value)**
  - **void Clear**
  - **bool Contains (Object value).**
  - **int IndexOf (Object value)**
  - **void Insert (int index, Object value)**
  - **void Remove (Object value).**
  - **void RemoveAt (int index)**



IDictionary

- Classes implementing **IDictionary** allow non-generic collection of key/value pairs.
- Implements **ICollection, IEnumerable**
- Members and their description in API:
  - **IsFixedSize**
  - **IsReadOnly**
  - **Item**
  - **Keys**
  - **Values**
  - **void Add**
  - **void Clear ()**
  - **bool Contains (Object key)**
  - **IDictionaryEnumerator GetEnumerator ()**
  - **void Remove (Object key)**

IEnumerable and IEnumerator

- Both the interfaces are defined in **System.Collections** namespace.
- Used to make iteration through an array or collection simpler.
- **IEnumerable** allows use of **foreach** statements to iterate through an array or collection simpler.
- **IEnumerble** interface has a method
  IEnumerator GetEnumerator();
- **IEnumerator** interface has following methods
  - **bool MoveNext()**
  - **Current**
  - **Reset()**

IDictionaryEnumerator

- This interface is used to enumerate non-generic dictionary objects (objects of type **IDictionary**). Dictionary objects are associative arrays.
- Properties and their description in API:
  - **Entry:** Gets both the key and the value of the current dictionary entry.
  - **Key:** Gets the key of the current dictionary entry.
  - **Value:** Gets the value of the current dictionary entry.
  - **Current**: Gets the current element in the collection. (Inherited from **IEnumerator**.)
  - **bool MoveNext()** : Advances the enumerator to the next element of the collection. (Inherited from **IEnumerator**.)
  - **void Reset()** : Sets the enumerator to its initial position, which is before the first element in the collection.

Cautions about Enumerators

- Enumerators can be used to read the data in the collection, but they cannot be used to modify the underlying collection.
- Also if the underlying collection is changed while enumerating (by another thread) , the enumerator is invalidated and the next call to **MoveNext** or **Reset** throws an **InvalidOperationException**.
- In other words, enumerating through a collection is not thread-safe.
- Even when a collection is synchronized, other threads can still modify the collection, which causes the enumerator to throw an exception.
- Therefore it is recommended that a lock is obtained for the collection, while enumerating to make it thread-safe.

```
lock(collection.SyncRoot){
  foreach(var item in collection){
  // do stuff   }}
```

Presented by

Ranjan Bhatnagar

Non-Generic Collections

- **Classes directly inheriting from ICollection**
  - BitArray
  - Stack
  - Queue
- **Classes directly inheriting from IList**
  - ArrayList
- **Classes directly inheriting from IDictionary**
  - Hashtable
  - SortedList

BitArray Class

- This class holds an array of bit values which are represented as **System.Boolean.**
- The **true** indicates bit value 1 and false indicates the bit value 0.
- Implements **ICollection, IEnumerable, ICloneable**
- The size of a **BitArray** is specified and index beyond the size will throw an **ArgumentException**.
- The instance methods in this class are not thread safe.

Example - BitArray

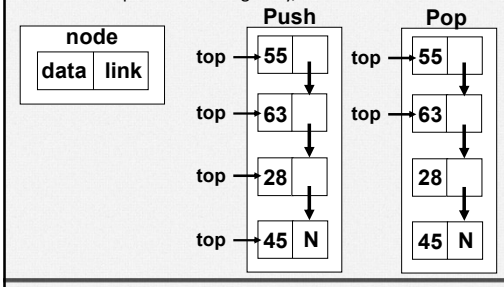
```
using System;
using System.Collections;
namespace Sample
{
    class Class1
    {
        static void Main(string[] args)
        {
            int[] a = { 1, 2, 3 };
            BitArray b1 = new BitArray(a);
            BitArray b2 = new BitArray(5);
            BitArray b3 = new BitArray(5, true);
            for (int i = 0; i < b1.Count; i++)
                Console.WriteLine("{0} ", b1[i]);
            Console.WriteLine("More on BitArray");
            int[] a1 = { -10, -10 };
            int[] a2 = { 10, 10 };
            BitArray b4 = new BitArray(a1);
            BitArray b5 = new BitArray(a2);
```

Example - BitArray

```
            b4.And(b5);
            b4.Or(b5);
            b4.Xor(b5);
            b4.Set(1, true);
            b4.SetAll(true);
            for (int i = 0; i < b4.Count; i++)
                Console.WriteLine("{0} ", b4[i]);
            for (int i = 0; i < b4.Count; i++)
                Console.WriteLine("{0} ", b5[i]);
        }
    }
}
```

Stacks

- Last In First Out (LIFO) list
- Usage - Function calls, Expression evaluation, ISRs
- Can be implemented using array, linked list



stack Class

- Represents a simple last-in-first-out (LIFO) non-generic collection of objects
- Implements **ICollection, IEnumerable, ICloneable**
- The capacity of a Stack is the number of elements the Stack can hold.
- It is a dynamic collection in the sense that when elements are added to it, its capacity is automatically increased if required through reallocation.
- It is implemented as a circular buffer.
- Stack accepts nothing (**null**) as a valid value

Presented by

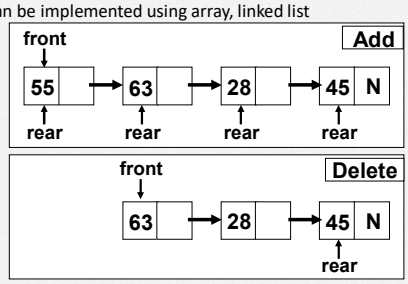
Ranjan Bhatnagar

Stack Class Example

```
using System;
using System.Collections;
class Class1
{
    static void Main(string[] args)
    {
        Stack st = new Stack();
        st.Push(10);
        st.Push(11);
        st.Push(12);
        st.Push(13);
        foreach (var i in st)
            Console.WriteLine(i);
        Console.WriteLine("Element popped: {0}", st.Pop());
        IEnumerator e = st.GetEnumerator();
        while (e.MoveNext())
            Console.WriteLine(e.Current);
    }
}
```

Queue

- First In First Out (FIFO) list
- Usage - Function calls, Expression evaluation
- Can be implemented using array, linked list



Queue Class

- Represents a first-in, first-out collection of objects.
- Implements ICollection, IEnumerable, ICloneable
- Objects stored in a Queue are inserted at rear and removed from the front.
- Queue accepts Nothing (null) as a valid value. The constructors of Queue are similar to one in Stack.
- Members (apart from the interface methods)
  - public virtual Object Dequeue () Removes and returns the object at the beginning of the Queue.
  - public virtual void Enqueue (Object obj): Adds an object to the end of the Queue
  - public virtual Object Peek (): Returns the object at the beginning of the Queue without removing it.

Queue Example

```
using System;
using System.Collections;
class Class1
{
    static void Main(string[] args)
    {
        Queue q = new Queue();
        q.Enqueue("Message1");
        q.Enqueue("Message2");
        q.Enqueue("Message3");
        q.Enqueue("Message4");
        Console.WriteLine("First message: {0}", q.Dequeue());
        Console.WriteLine("The Front element is {0}", q.Peek());
        IEnumerator e = q.GetEnumerator();
        while (e.MoveNext())
            Console.WriteLine(e.Current);
        bool b = q.Contains("Message3");
        Console.WriteLine(b);
    }
}
```

ArrayList

- **ArrayList** is an array whose size is dynamically increased as required.
- Implements the **IList** , **ICollection**, **IEnumerable, ICloneable**
- This class provides several methods like sort and search.
- The **ArrayList** by itself is not sorted. Hence before calling binary search methods it must be sorted.
- The capacity can be decreased by calling **TrimToSize** or by setting the **Capacity** property explicitly.

Example ArrayList

```
using System;
using System.Collections;
namespace Sample
{
    class Class1
    {
        static void Main(string[] args)
        {
            ArrayList arr = new ArrayList();
            arr.Add('a');
            arr.Add(43);
            arr.Add(6.7);
            arr.Add("Rahul");
            arr.Insert(1, "Deepti");
            for (int i = 0; i < arr.Count; i++)
                Console.WriteLine(arr[i]);
        }
    }
}
```

```
a
Deepti
43
6.7
Rahul
```

Presented by

Ranjan Bhatnagar

Sort() methods

- void Sort: Sorts the elements in the entire ArrayList.
- void Sort(IComparer) Sorts the elements in the entire ArrayList using the specified comparer.
- void Sort(Int32, Int32, IComparer): Sorts the elements in a range of elements in ArrayList using the specified comparer.

BinarySearch() Methods

- Members apart from the methods of interfaces
  - **int BinarySearch(Object)** : Locates the object in the sorted **ArrayList** using the default comparer and returns the zero-based index of the element.
  - The objects in the **ArrayList** must be of **IComparable** type
  - **BinarySearch(Object, IComparer):** Locates the object in the **ArrayList** using the specified comparer and returns the zero-based index of the element.
  - **BinarySearch(Int32, Int32, Object, IComparer) :** Locates the object in the sorted **ArrayList** using the specified comparer and returns the zero-based index of the element

IComparable and IComparer

- The sort and the binary search methods relies on how the elements will be ordered in the **ArrayList**.
- The order can be specified in 2 ways:
  1. Using **IComparable** :
     - **int CompareTo(object obj);**
     - A class implements this interface and provides implementation for **CompareTo method** which becomes the "default comparer"
  2. Using **IComparer** :
     - **int Compare (Object x, Object y);**
     - If the "default comparer" of a class is not suited then, another class is created that provides implementation for Compare method which becomes the "specified comparer ".

Example: sort using default comparer

```
using System;
using System.Collections;
class Flower : IComparable
{
    int petals;
    string name;
    public Flower(int p, string c)
    {
        petals = p;
        name = c;
    }
    public int CompareTo(object obj)
    {
        return petals - ((Flower)obj).petals;
    }
    public override string ToString()
    {
        return name + " with " + petals + " petals";
    }
}
```

Example: sort using default comparer

```
class ArrayListSort
{
    public static void Main()
    {
        ArrayList flowers = new ArrayList();
        flowers.Add(new Flower(5, "Lily"));
        flowers.Add(new Flower(10, "Rose"));
        flowers.Add(new Flower(5, "Hibiscus"));
        flowers.Add(new Flower(6, "Tulips"));
        flowers.Sort();
        foreach (Flower f in flowers)
            Console.WriteLine(f);
        Console.WriteLine(
              flowers.BinarySearch(new Flower(6, "Tulips")));
    }
}
```

Hashtable

- This class is like a map that has a collection of key-value pairs.
- The collection is organized based on the hash code of the key.
- Hashtable uses the hash value of a key object to put the key-value into the bucket. The search happens by first getting the hash code of the key and then looking for the key-value pair in the appropriate bucket.
- One object per bucket will lead almost to a linear search. Hence it is very important that hash code generating method is written efficiently.
- The size of the Hashtable dynamically increases as elements are added to it.

Presented by

Ranjan Bhatnagar

Key

- The key cannot be Nothing (**null**). They must be immutable.
- They cannot be duplicate. If duplicate is entered, it is overwritten.
- It is mandated that the object used as key must have one of the following implementations
  1. override the **GetHashCode** method and the **Equals** method
  2. Implement **IEqualityComparer**
  3. Implement the **IHashCodeProvider** interface

  or the **IComparer** interface (depreciated, *so we will not look at this*)
- It is important that the hash code methods are implemented properly for the **Hashtable** to work efficiently.

Key Cont…

- If both set of methods in 1, 2 or 3 (Prev. slide) must have similar behaviour. For instance, if **GetHashCode()** is case sensitive then **equals** must also be case sensitive.
- These method must return same values when called with the same parameters while the key is in the **Hashtable**.
- Also if two things are equal then they *must* return the same value for hash code.
- If the hash codes are equal for 2 objects, it is *not* necessary for them to be the same; this is a collision, and Equals
- Iteration order of **Hashtable** in C# cannot be determined.
- The equality is determined by **Equals** method (of the key class or **IEqualityComparer** class**)** or **CompareTo()**method of **IComparer** depending on the context in which it is used.

Example - HashTable

```
using System;
using System.Collections;
class Class1
{
    static void Main(string[] args)
    {
        Hashtable h = new Hashtable();
        h.Add("mo", "Monday");
        h.Add("tu", "Tuesday");
        h.Add("we", "Wednesday");
        h.Add("th", "Thursday");
        h.Add("fr", "Friday");
        h.Add("sa", "Saturday");
        h.Add("su", "Sunday");
        IDictionaryEnumerator e = h.GetEnumerator();
        while (e.MoveNext())
            Console.WriteLine(e.Key + "\t" + e.Value);
```

Example - HashTable

```
        Console.WriteLine("The Keys are : ");
        ICollection k = h.Keys;
        foreach (object i in k)
            Console.WriteLine(i);
        Console.WriteLine("The Values are : ");
        ICollection v = h.Values;
        foreach (object i in v)
            Console.WriteLine(i);
        h["su"] = "Sun";
        h["mo"] = "Mon";
        Console.WriteLine(h["fr"]);
    }
}
```

IEqualityComparer, DictionaryEntry

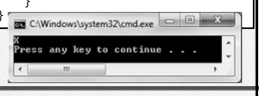
- IEqualityComparer

  Methods in **IEqualityComparer**
  - **bool Equals(Object x,Object y):** Determines whether the specified objects are equal.
  - **int GetHashCode(Object obj):** Returns a hash code for the specified object.
- DictionaryEntry
- The key/value pair stored in a **DictionaryEntry** object.
- The **foreach** statement on **Hashtable** returns the **DictionaryEntry**.
- Properties
  - **Key**
  - **Value**

Implementing Hash Code

```
using System;
using System.Collections;
class Name
{
    public String name;
    public Name(string nm)
    {
        this.name = nm;
    }
    public override int GetHashCode()
    {
        return (name.ToCharArray())[0];
    }
    public override bool
                Equals(object obj)
    {
        Name s = (Name)obj;
        return name.Equals(s.name);
    }
    public override string ToString()
    {
        return name;
    }
}
```

```
class HashtableEx
{
    public static void Main()
    {
        Hashtable nlist =
                    new Hashtable();
        nlist.Add(
        new Name("Trisha"), "V");
        nlist.Add(
        new Name("Nisha"), "VI");
        nlist.Add(
        new Name("Nimisha"), "X");
        nlist.Add(
        new Name("Varsha"), "IV");
        nlist.Add(
        new Name("Treena"), "II");
        Console.WriteLine(
        nlist[new Name("Nimisha")]);
    }
}
```



Presented by

Ranjan Bhatnagar

SortedList class

- Represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.
- Implements **IDictionary, ICollection, IEnumerable**
- This class also relies on **IComparable** and **IComparer** for comparison of keys during the sorts.
- That is the elements of a **SortedList** are sorted by the keys either according to a specific **IComparer** or according to the **IComparable** implementation provided.
- A **SortedList** does not allow duplicate keys.

Example - SortedList

```
using System;
using System.Collections;
class Class1
{
    static void Main(string[] args)
    {
        SortedList s = new SortedList();
        s.Add("Maharastra    ", "Mumbai");
        s.Add("Karnataka     ", "Bangalore");
        s.Add("Andhra Pradesh", "Hyderabad");
        s.Add("Tamilnadu     ", "chennai");
        s.Add("Bihar         ", "Patna");
        s.Add("Rajastan      ", "Jaipur");
        s.Add("Orissa        ", "Bhubaneshwar");
        Console.WriteLine("Elements in the SotredList: ");
        IDictionaryEnumerator e = s.GetEnumerator();
        while (e.MoveNext())
            Console.WriteLine(e.Key + "\t" + e.Value);
```

Example - SortedList

```
        s.Remove("TamilNadu");
        Console.WriteLine("The Keys are : ");
        ICollection k = s.Keys;
        foreach (object i in k)
            Console.WriteLine(i + " ");
        Console.WriteLine("The Values are : ");
        ICollection v = s.Values;
        foreach (object i in v)
            Console.WriteLine(i + " ");
        Console.WriteLine("Value at 3rd Index 3: "
                          + s.GetByIndex(3));
        Console.WriteLine("Key at 3rd Index:  "
                          + s.GetKey(3));
        Console.WriteLine("The Index of key Bihar"
                          + s.IndexOfKey("Bihar"));
        Console.WriteLine("The Index of value Jaipur"
                          + s.IndexOfValue("Jaipur"));
    }
}
```

Downside of non-generic collection

- Note that with non-generic types like ArrayList, there is a convenience that any data can be stored in it.
- But there is a performance cost to pay.
- Any item added to the non-generic collection must be upcasted to object type.
- For value types, they must be boxed when they are added to the list, and unboxed when they are retrieved. Both the casting and the boxing and unboxing operations decrease performance.
- Other important limitation is the lack of compile-time type checking.

Generics

- Generics are a way to create type-safe classes without compromising on performance.
- It allows creating methods (algorithms) in a generic way and insists specification of type on usage.
- Also since we don't have to write multiple implementation of same code for different types, there is a gain in productivity.
- Generic can be used for both class and struct.

Collection with generics

- All the collection interfaces have that 'generic' counter-part also.
- They are defined in **System.Collections.Generic** namespace
- The generic collection classes are **type-safe**.
- Also it helps us avoiding multiple implementations of the same code for different type.
- They became part of .NET framework from 2.0 onwards.

System.Collections.Generic Namespace

- Key Interfaces Supported by Classes of System.Collections.Generic
  - ICollection<T> Defines general characteristics (e.g., size, enumeration, and thread safety) for all generic collection types.
  - IComparer<T> Defines a way to compare to objects.
  - IDictionary<TKey, TValue> Allows a generic collection object to represent its contents using key/value pairs.
  - IEnumerable<T> Returns the IEnumerator<T> interface for a given object.
  - IEnumerator<T> Enables foreach-style iteration over a generic collection.
  - IList<T> Provides behavior to add, remove, and index items in a sequential list of objects.
  - ISet<T> Provides the base interface for the abstraction of sets.

Stack of Integer

```
using System;
public class Stack
{
    int[] data;
    int top = -1;
    public Stack(int size)
    {
        data = new int[size];
    }
    public void Push(int value)
    {
        top++;
        data[top] = value;
    }
    public int Pop()
    {
        int value = data[top];
        top--;
        return value;
    }
    public int GetTopElement()
    {   return data[top];    }
    public void Print()
    {
      for (int i = 0; i <= top; i++)
          Console.WriteLine(data[i]);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Stack s = new Stack(5);
        s.Push(5);
        s.Push(12);
        s.Print();
        int n = s.Pop();
        Console.WriteLine(n);
        s.Print();
    }
}
```

Stack for Any Data Type

```
using System;
public class Stack
{
   object[] data;
   int top = -1;
   public Stack(int size)
   {
      data = new object[size];
   }
   public void Push(object value)
   {
       top++;
       data[top] = value;
   }
   public object Pop()
   {
       object value = data[top];
       top--;
       return value;
   }
   public object GetTopElement()
   {  return data[top]; }
   public void Print()
   {
    for (int i = 0; i <= top; i++)
          Console.WriteLine(data[i]);
   }
}
class Program
{
   static void Main(string[] args)
   {
        Stack s=new Stack(5);
        s.Push (5);
        s.Push ("Demo");
        s.Push (new Computer());
        s.Push (new Account());
        s.Push (12);
        s.Print();
        Console.WriteLine(s.Pop());
        s.Print();
   }
}
```

Generic Stack Class

```
using System;
public class Stack<T>
{
    T[] data;
    int top = -1;
    public Stack(int size)
    {
        data = new T[size];
    }
    public void Push(T value)
    {
        top++;
        data[top] = value;
    }
    public T Pop()
    {
        T value = data[top];
        top--;
        return value;
    }
    public T GetTopElement()
    {
        return data[top];
    }
   public void Print()
   {
     for (int i = 0; i <= top; i++)
        Console.WriteLine(
                data[i].ToString());
   }
}
class Program
{
    static void Main(string[] args)
    {
        Stack<int> s = new
                    Stack<int>(5);
        s.Push(2);
        s.Push(5);
        s.Push(22);
        s.Push("Demo"); // Error.
        s.Print();
        Stack<string> ss = new
              Stack<string>(10);
        ss.Push("Demo");
        ss.Push("Test");
    }
}
```

Generic classes

- Some other generic classes are
  - **Stack<T> :** generic equivalent of the **Stack**
  - **Queue<T> :** generic equivalent of the **Queue**
  - **List<T>** : generic equivalent of the **ArrayList**
  - **Dictionary <TKey,KValue>:** generic equivalent of the **Hashtable** (**Dictionary** will throw an exception if you try to reference a key that doesn't exist. **Hashtable** will just return **null**.

    bool TryGetValue(TKey key, out TValue value): returns true if the key is present and value is populated. This method can be used to avoid runtime exceptions)
  - **SortedList<TKey,KValue>:** generic equivalent of the **SortedList**

Generic interfaces

- There is a generic counter-part for all the non-generic interface.
  - ICollection<T>
  - IEnumerable<T>
  - IList<T>
  - IDictionary<TKey, TValue>
- The compare interfaces also have generic counter parts
  - Comparer<T>
  - IComparable<T>

Presented by

Ranjan Bhatnagar

Default Values in Generic

- default(T) is used to return the default value of the type specified at the time of generic instantiation. That is for int it would be 0 and for object it would be null.

```
public T Pop()
{
    if (--top >= 0)
    {
        return data[top];
    }
    else
    {
        top = -1;
        return default(T);
    }
}
```

Generic with Multiple Types

```
using System;
using System.Collections.Generic;
class KeyValuePair<K, T>
{
    public K Key;
    public T Value;
    public KeyValuePair()
    {
        Key = default(K);
        Value = default(T);
    }
    public KeyValuePair(K key, T item)
    {
        Key = key;
        Value = item;
    }
}
```

Generic with Multiple Types Cont.

```
public class Map<K, T>
{
    int Size;
    int currIndex = -1;
    KeyValuePair<K, T>[] Pair;
    public Map(): this(10)
    {
    }
    public Map(int size)
    {
        this.Size = size;
        Pair = new KeyValuePair<K, T>[Size];
    }
    public void Put(K key, T item)
    {
        KeyValuePair<K, T> newKeyValuePair
                    = new KeyValuePair<K, T>(key, item);
        if (currIndex++ > Size)
            throw new OverflowException();
        else
            Pair[currIndex] = newKeyValuePair;
    }
```

Generic with Multiple Types Cont.

```
    public T Get(K key)
    {
        foreach (KeyValuePair<K, T> k in Pair)
        {
            if (k.Key.Equals(key))
              return k.Value;
        }
       throw new KeyNotFoundException();
    }
}
class Test
{
    public static void Main()
    {
        Map<int, string> m = new Map<int, string>();
        m.Put(1, "abc");
        m.Put(2, "def");
        m.Put(3, "ghi");
        Console.WriteLine(m.Get(2));
    }
}
```

Classes of System.Collections.Generic

Generic Class	Supported Key Interfaces	Meaning in Life
Dictionary<TKey, TValue>	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	This represents a generic collection of keys and values.
List<T>	ICollection<T>, IEnumerable<T>, IList<T>	This is a dynamically resizable sequential list of items.
LinkedList<T>	ICollection<T>, IEnumerable<T>	This represents a doubly linked list.
Queue<T>	ICollection (not a typo! This is the non-generic collection interface), IEnumerable<T>	This is a generic implementation of a first-in, first-out (FIFO) list.
SortedDictionary <TKey, TValue>	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	This is a generic implementation of a sorted set of key/value pairs.
SortedSet<T>	ICollection<T>, IEnumerable<T>, ISet<T>	This represents a collection of objects that is maintained in sorted order with no duplication.
Stack<T>	ICollection (not a typo! This is the non-generic collection interface), IEnumerable<T>	This is a generic implementation of a last-in, first-out (LIFO) list.

Generic Constraints

- Compiler converts the generic code to IL independent of the type the program using generic(client) uses.
- This may mean that client can pass the types that may be incompatible with respect to generic code. That means that it may hit type safety.
- So compiler takes lot of precaution before compiling the generic code.
- Generic constraints are ways to specify the rules that the client must follow to use the generic code.

Presented by

Ranjan Bhatnagar

Types of Generic Constraints

- Derivation constraint
  - Generic type parameter derives from a base type such an interface or a class
- Default constructor constraint
  - Generic type parameter exposes a default public constructor
- Reference/value type constraint
  - Constrains the generic type parameter to be a reference or a value type

IComparable

- .NET provided System.IComparable interface that can be used to make sure that objects have implemented comparisons methods in the desired manner.

```
public interface IComparable<T>
{
    int CompareTo(T other);
    bool Equals(T other);
}
```

Derivation Constraint

```
using System.IO;
using System;
using System.Collections.Generic;
class KeyValuePair<K, T> where K : IComparable<K>
{
    public K Key;
    public T Value;
    public KeyValuePair()
    {
        Key = default(K);
        Value = default(T);
    }
    public KeyValuePair(K key, T item)
    {
        Key = key;
        Value = item;
    }
}
```

Example Cont.

```
public class Map<K, T> where K : IComparable<K>
{
    int Size;
    int currIndex = -1;
    KeyValuePair<K, T>[] Pair;
    public Map() : this(10) { }
    public Map(int size)
    {
        this.Size = size;
        Pair = new KeyValuePair<K, T>[Size];
    }
    public void Put(K key, T item)
    {
        foreach (KeyValuePair<K, T> k in Pair)
            if (k != null)
                if (k.Key.CompareTo(key) == 0)
                    throw new ArgumentException("duplicate key!");
        KeyValuePair<K, T> newKeyValuePair
            = new KeyValuePair<K, T>(key, item);
        if (currIndex++ > Size) throw new OverflowException();
        else Pair[currIndex] = newKeyValuePair;
    }
```

Example Cont.

```
    public T Get(K key)
    {
        foreach (KeyValuePair<K, T> k in Pair)
        {
            if (k.Key.CompareTo(key) == 0) return k.Value;
        }
        throw new KeyNotFoundException();
    }
}
class TTest
{
    public static void Main()
    {
        Map<int, string> m = new Map<int, string>();
        m.Put(1, "abc");
        m.Put(2, "def");
        m.Put(2, "def");
        Console.Write(m.Get(1));
    }
}
```

Multiple Derivation constraint and visibility

- **public class Map<K, T> where K : IComparable<K>, ICloneable {…}**

Note that if class is used then class name must appear first

- **public class Map<K, T> where K : IComparable<K> where T : IComparable<K> {…}**
- **public class Map<T,U> where T : U {…}**
- The visibility of the class or the interface used with derivation constraint must be same or more than the generic class itself. For instance, if generic class is **public**, then class or interface used with derivation constraint cannot be **internal**.

Presented by

Ranjan Bhatnagar

Constructor Constraint

- An issue with the Map code is it allows null as Key in the Map?

```
Map<String, String> m1 = new Map<String, String>();
String s1=null;
m1.put(s1,"abc");
```

- There are two things that could be done,
  - On null key we could throw an Exception
  - Or call default constructor
- The second approach requires the Key object to expose default constructor so that generic code could create Key object when a null is passed.
- This calls for a Constructor Constraint which is specified by new keyword with where keyword.

Constructor Constraint in action

```
using System;
using System.Collections.Generic;
class KeyValuePair<K, T> where K : IComparable<K>, new()
{
    public K Key;
    public T Value;
    public KeyValuePair()
    {
        Key = new K();
        Value = default(T);
    }
    public KeyValuePair(K key, T item)
    {
        K def = default(K);
        if (Object.Equals(key, def))
            Key = new K();
        else
            Key = key;
        Value = item;
    }
}
```

Constructor Constraint cont.

```
public class Map<K, T> where K : IComparable<K>, new()
{
    int Size;
    int currIndex = -1;
    KeyValuePair<K, T>[] Pair;
    public Map() : this(10) { }
    public Map(int size)
    {
        this.Size = size;
        Pair = new KeyValuePair<K, T>[Size];
    }

    public void Put(K key, T item)
    {
        foreach (KeyValuePair<K, T> k in Pair)
            if (k != null)
                if (k.Key.CompareTo(key) == 0)
                    throw new ArgumentException("duplicate key!");
        KeyValuePair<K, T> newKeyValuePair =
                    new KeyValuePair<K, T>(key, item);
        if (currIndex++ > Size) throw new OverflowException();
        else
            Pair[currIndex] = newKeyValuePair;
    }
```

Constructor Constraint cont.

```
public T Get(K key)
    {
        foreach (KeyValuePair<K, T> k in Pair)
        {
            if (k.Key.CompareTo(key) == 0) return k.Value;
        }
        throw new KeyNotFoundException();
    }
}
class TTest
{
    public static void Main()
    {
        Map<int, string> m = new Map<int, string>();
        m.Put(1, "abc");
        m.Put(2, "def")
        m.Put(2, "def")
        Console.Write(m
    }
}
```

Thanks

Presented by

Ranjan Bhatnagar