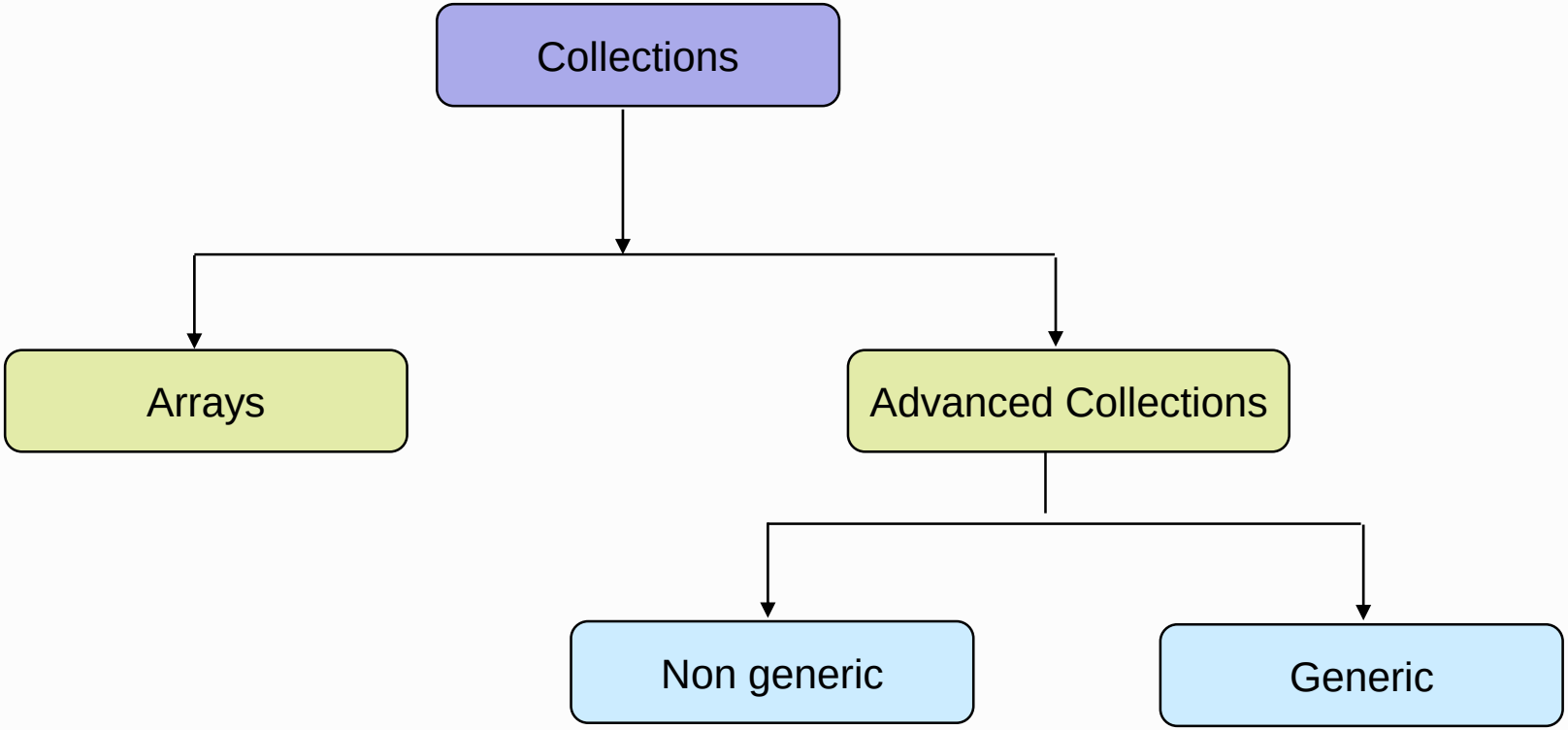
Collection Framework

Objectives

- On completion of this session you will be able to
  - Define collections and list different collection classes in .NET framework
  - List the collection interfaces
  - Implement `IEnumerable, ICollection, IComparable` interfaces in user defined classes.
  - Use iterator method to iterate over the collections
  - Use the classes defined in `System.Collections` namespace
  - List the advantages of generic collections over their non - generic counterparts
  - Use the classes defined in `System.Collections.Generic` namespace
  - Write you own generic methods and classes.

Collections

- A collection is a set of similarly typed objects that are grouped together.



`System.Array` class

- The base class of all array types.

```
 int[] IntArray = new int[5] { 22,
            11 33, 55, 44 };
foreach ( int i in myArr )
{
   Console.Write( "\t{0}", i );
}
Array.Sort(IntArray);
Array.Reverse(IntArray);
```

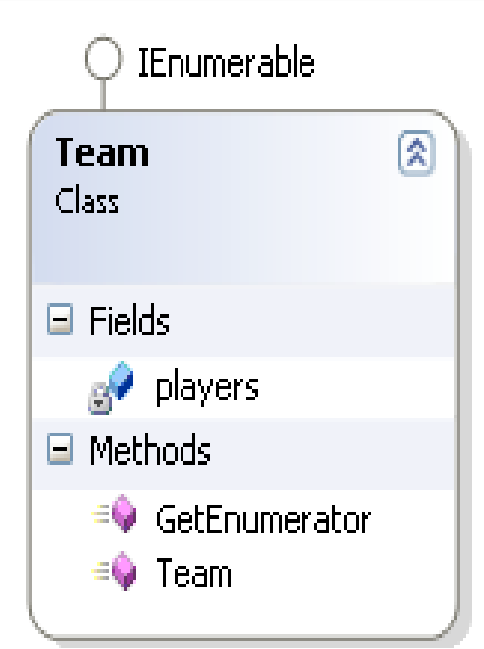
Collection Interfaces

- Allow collections to support a different behavior

Interface	Description
`IEnumerator`	Supports a simple iteration over collection
`IEnumerable`	Supports foreach semantics
`ICollection`	Defines size, enumerators and synchronization methods for all collections.
`IList`	represents a collection of objects that could be accessed by index.
`IComparable`	Defines a generalized comparison method to create a type-specific comparison
`IComparer`	Exposes a method that compares two objects.
`IDictionary`	Represents a collection of key-and-value pairs

Implementing `IEnumerable` Interface

```
public class Team: IEnumerable
{
  private Player[] players;
  public Team()
   {
    players =  new Player[3];
    players[0] = new Player("Yuraj", 28);
    players[1] = new Player("Rahul", 34);
    players[2] = new Player("Sourav", 34);
   }
  public IEnumerator GetEnumerator()
  {
   return players.GetEnumerator();}
  }
}
```

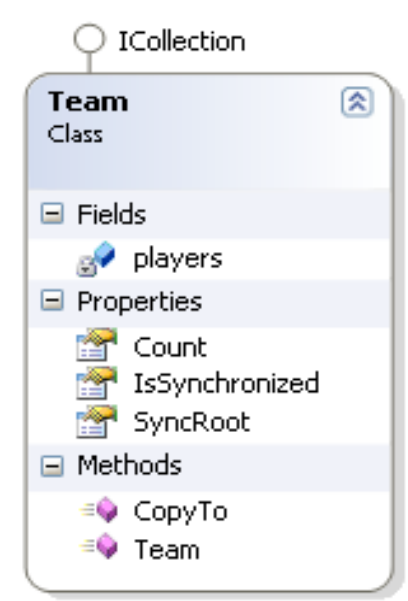



```
Team india= new Team();
foreach (Player c in india)
{Console.WriteLine(c.Name, c.runs);}
```

Implementing `ICollection` Interface

- To determine No. of elements in container.
- Ability to copy elements into `System.Array` Type.

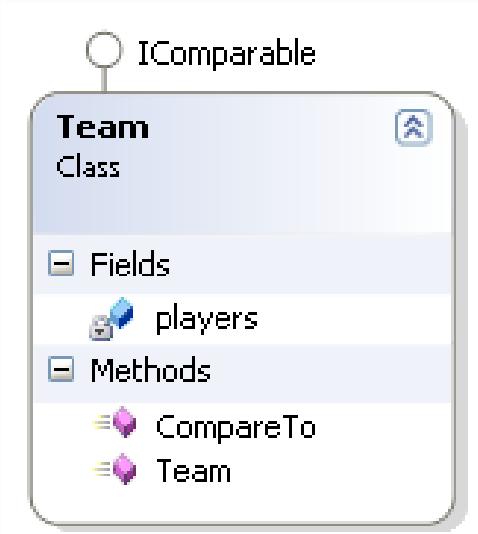
```
public class Team: ICollection
  { private Player[] players;
    public Team(){…}
     public int Count
     {
      get{
           return Player.Count;
         }
      }
. . .
  }
```




```
Team india= new Team();
foreach (Player c in india)
{Console.WriteLine(c.Name, c.runs);}
```

Implementing `IComparable` Interface

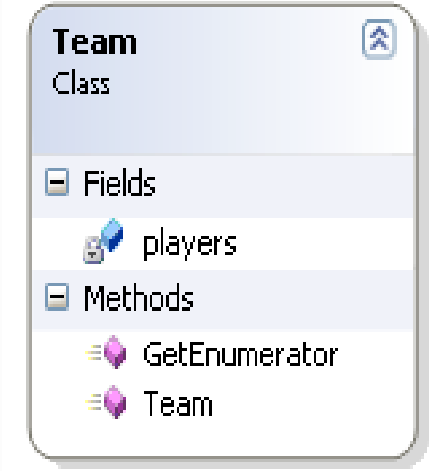
```
public class Player: IComparable
  {
    int IComparable.CompareTo(object obj)
    {
     Player temp = (Player)obj;
    if (this.runs > temp.runs)
       return 1;
    if (this.runs < temp.runs)
       return -1;
    else
       return 0;
    }
  }
```




```
Team india= new Team();
// add five players with score
Array.Sort(india);
//display sorted array
```

Using Iterator Method

```
public class Team
  { private Player[] players= new Player[3];
     public Team()
    {players[0] = new Player("Yuraj", 28);
     players [1] = new Player("Rahul", 34);
     players[2] = new Player("Sourav", 34);
    }
    public IEnumerator GetEnumerator()
    {
     foreach(Player p in players)
     {
       yield return p;
   }
}
```




```
Team india= new Team();
foreach (Player c in india)
{Console.WriteLine(c.Name, c.runs);}
```

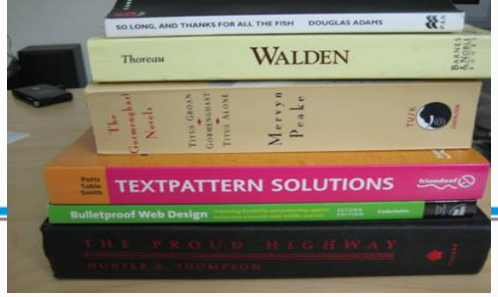
ArrayList class

Represents list which is similar to a single dimensional array that can be resized dynamically.

```
ArrayList countries = new ArrayList();
countries.Add("India");
countries.Add ("Pakistan");
countries.Add("Spain");
Console.WriteLine( "Count:{0}",countries.Count );

foreach ( Object obj in countries )
  Console.WriteLine( " {0}", obj );
```

Stack class



- Represents a simple Last-In-First-out (LIFO) non-generic collection of objects.

```
Stack numStack = new Stack();
numStack.Push(10);
numStack.Push(20);
numStack.Push (30);
Console.WriteLine("Element removed:",numStack.Pop();
```

Queue class

- Represents a first-in, first-out (FIFO) collection of non- generic collection of objects.
- Used for sequential processing.

```
Queue myQueue = new Queue();
myQueue.Enqueue("Clark");
myQueue.Enqueue("Ann");
myQueue.Enqueue("Joe");
Console.WriteLine("\tCapacity:{0}",
                  myQueue.Capacity );
Console.WriteLine("(Dequeue)\t{0}",
                        myQueue.Dequeue () );
```

Removes and returns the object at the beginning of the Queue

`HashTable` class

- Represents a collection of key/value pairs that are organized based on the hash code of the key.
- Each element is a key/value pair stored in a `DictionaryEntry` object.

```
Hashtable h = new Hashtable( ) ;
h.Add ( "mo", "Monday" ) ;
h.Add ( "tu", "Tuesday" ) ;
h.Add ( "we", "Wednesday" ) ;
IDictionaryEnumerator e = h.GetEnumerator( ) ;
while ( e.MoveNext( ) )
      Console.WriteLine ( e.Key + "\t" + e.Value ) ;
```

Why Generics?

```
class Hashtable{
public Hashtable();
public Object Get(Object);
public void Add(Object,Object);
}
```

```
Hashtable addrBook;
..
addrBook.Add("John Kennedy", 44812);
..
int extension = (int) addrBook.Get("John
  Kennedy");
..
```

Slow with implicit boxing

Slow with type tests and unboxing of primitive types

Generics

- Are classes, structures, interfaces, and methods that have placeholders (type parameters) for one or more of the types they store or use.

```
class Hashtable <K,V> {
public Hashtable();
public Object Get(K)
public void Add(K,V)
}
```

Instantiate with actual parameters

Parameterized on types

```
Hashtable<string,int> addrBook;
..
addrBook.Add("John Kennedy", 44812);
..
int extension = addrBook.Get("Don Syme");
..
```

List<T> class

- Represents a strongly typed list of objects that can be accessed by index.
- Generic equivalent of `ArrayList` class.

```
List<string> months = new
List<string>();
months.Add("January");
months.Add("February");
months.Add("April");
foreach(string mon in months)
   Console.WriteLine(mon);

months.Insert(2,"March");
```

or

```
int i;
for(i=0;i<months.Count;i++)

Console.WriteLine(months[i]);
```

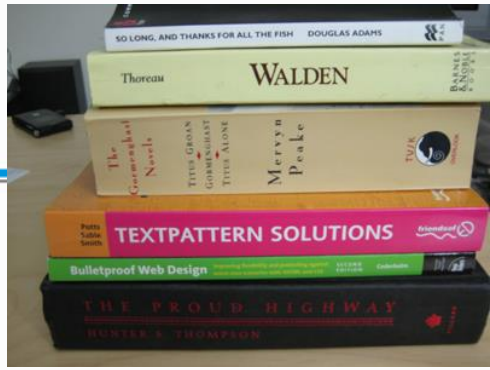
List of UserDefined Objects

```
class EmpComparer:IComparer<Employee>{
public int  Compare(Employee e1, Employee e2){
int ret = e1.Name.Length.CompareTo(e2.Name.Length);
    return ret; }
}
```

```
class Employee
{
   int eid
   string ename;
//appropriate
   constructor and
   properties present
}
```

```
List<Employee> list1 = new List<Emplo
list1.Add(new Employee(1,"Andrew"));
list1.Add(new Employee(2,"Joe"));
list1.Add(new Employee(3, "Neil"));
list1.Add(new Employee(4, "Timmy"));
 EmpComparer ec = new EmpComparer();
 list1.Sort(ec);
 foreach (Employee e in list1)
       Console.WriteLine(e.Id + "---" + e.Name);
```

Stack<T> class



Stack of integers

```
Stack<int> numStack = new Stack();
numStack.Push(10);
numStack.Push(20);
numStack.Push (30);
Console.WriteLine("Element removed:",numStack.Pop();
```

Queue<T> class


```
Queue<string> q = new Queue<string>();
q.Enqueue("Message1");
q.Enqueue("Message2");
q.Enqueue("Message3");
Console.WriteLine("First message: {0}",
                              q.Dequeue());
Console.WriteLine("The element at the head is
          {0}", q.Peek());
IEnumerator<string> e = q.GetEnumerator();
while (e.MoveNext())
      Console.WriteLine(e.Current);
```

First message out of the queue

LinkedList<T> class

- Represents a doubly linked list.
- Each node is of the type **LinkedListNode**.

```
 LinkedList<string> l1 = new LinkedList<string>();
l1.AddFirst(new LinkedListNode<string>("Apple"));
l1.AddLast(new LinkedListNode<string>("Papaya"));
l1.AddFirst(new LinkedListNode<string>("Orange"));

LinkedListNode<string> node = l1.First;
Console.WriteLine(node.Value);
Console.WriteLine(node.Next.Value);
```

Dictionary<K,V> class

- Represents a collection of keys and values.
- Keys cannot be duplicate.

```
Dictionary<int,string>phone = new Dictionary<int, string>();
phone.Add(1, "James");
phone.Add(12, "Jimmy");
phone.Add(3, "James");
phone[12] = "Kim";
Console.WriteLine("Name is {0}", phone[12]);
 if (!phone.ContainsKey(4))
        phone.Add(4, "Tim");
Console.WriteLine("Name is {0}", phone[4]);
```

Value is duplicated and not the key

Custom Generic Class

```
class GenericStackClass<T> {
int top, mSize;
T[] mArr;
public GenericStackClass(int size)
{. . .
    mArr = new T[mSize];
}
public void Push(T element)
{ //code here
}
public T Pop()
{ //code here
}
}
```

```
GenericStackClass<int>iStack = new
          GenericStackClass<int>(3);
iStack.Push(10);
GenericStackClass<Complex> cObj = new
      GenericStackClass<Complex>(3);
cObj.Push(new Complex());
```

Custom Generic Method

- Generic methods are used if same algorithm works well for various datatypes e.g. swapping, sorting, searching algorithm, etc.

```
static void Swap<T>(ref T a, ref T b)
{
   T temp;
   temp = a;
   a = b;
   b = temp;
}
```

placeholder

```
static int Main()
{
int x = 10, y = 20;
Swap<int>(ref x, ref y);
//or
Swap(ref x, ref y)
}
```

Quick Recap…

- Non - generic collection classes are available in System.Collections namespace.
- Generic collection classes are available in System.Collections.Generic namespace.
- Compiler generates a type specific implementation in case of generic types.
- Generic classes or interfaces are type safe than non generic counterparts.
- Generic classes show better performance than non generic classes as there is no type casting required.