ICT365

Software Development Frameworks

Dr Afaq Shah

Murdoch UNIVERSITY

Collections and Generics

Murdoch
UNIVERSITY

In this Topic

Collections

Generics

Array+

To help overcome the limitations of a simple array, the .NET base class libraries ship with a number of namespaces containing collection classes .

Unlike a simple C# array, collection classes are built to dynamically resize themselves on the fly as you insert or remove items.

When the .NET platform was first released, programmers frequently used the classes of the System.Collections namespace to store and interact with bits of data used within an application.

In .NET 2.0, the C# programming language was enhanced to support a feature termed generics ; and with this change, a brand new namespace was introduced in the base class libraries: System.Collections.Generic .

Nongeneric collections are typically designed to operate on System.Object types and are, therefore, loosely typed containers (however, some nongeneric collections do operate only on a specific type of data, such as string objects).

In contrast, generic collections are much more type safe, given that you must specify the “type of type” they contain upon creation.

Generalized Collection Classes

- Array-like data structures

  ArrayList

  Queue

  Stack

  Hashtable

  SortedList

- Offer programming convenience for specific access needs.

- Store *objects*

  Add anything.

  Typecast on removal.

System.Collections Class	Meaning in Life	Key Implemented Interfaces
`ArrayList`	Represents a dynamically sized collection of objects listed in sequential order	`IList`, `ICollection`, `IEnumerable`, and `ICloneable`
`BitArray`	Manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0)	`ICollection`, `IEnumerable`, and `ICloneable`
`Hashtable`	Represents a collection of key-value pairs that are organized based on the hash code of the key	`IDictionary`, `ICollection`, `IEnumerable`, and `ICloneable`
`Queue`	Represents a standard first-in, first-out (FIFO) collection of objects	`ICollection`, `IEnumerable`, and `ICloneable`
`SortedList`	Represents a collection of key-value pairs that are sorted by the keys and are accessible by key and by index	`IDictionary`, `ICollection`, `IEnumerable`, and `ICloneable`
`Stack`	A last-in, first-out (LIFO) stack providing push and pop (and peek) functionality	`ICollection`, `IEnumerable`, and `ICloneable`

System.Collections Interface	Meaning in Life
`ICollection`	Defines general characteristics (e.g., size, enumeration, and thread safety) for all nongeneric collection types
`ICloneable`	Allows the implementing object to return a copy of itself to the caller
`IDictionary`	Allows a nongeneric collection object to represent its contents using key-value pairs
`IEnumerable`	Returns an object implementing the `IEnumerator` interface (see next table entry)
`IEnumerator`	Enables `foreach` style iteration of collection items
`IList`	Provides behavior to add, remove, and index items in a sequential list of objects

The Problems of Nongeneric Collections

- Successful .NET applications have been built over the years using these nongeneric collection classes (and interfaces), but issues..
- Using the System.Collections and System.Collections.Specialized classes can result in some poorly performing code, especially when manipulating numerical data (e.g., value types).
- The CLR must perform a number of memory transfer operations when you store structures in any nongeneric collection class prototyped to operate on System.Objects, slowing runtime execution speed.
- Most of the nongeneric collection classes are not type safe because (again) they were developed to operate on System.Objects, and they could therefore contain anything at all.
- If a .NET developer needed to create a highly type-safe collection (e.g., a container that can hold objects implementing only a certain interface), the only real choice was to create a new collection class by hand.

Generics

New in .NET 2.0 (Visual Studio 2005)

- Generics provide better performance because they do not result in boxing or unboxing penalties when storing value types.
- Generics are type safe because they can contain only the type of type you specify. Generics greatly reduce the need to build custom collection types because you specify the "type of type" when creating the generic container.
- Generics are generally the preferred alternative.

(Object collection classes offer advantages in certain unusual cases.)

Collection Classes

- To use these classes we must write

using System.Collections.Generic; for generics

Included in the default C# template

using System.Collections;

for object collections

Not included in the default C# template

Generics

- C# generics are like C++ *templates*.

- Classes written with blanks to be filled in by the user (parameters)

  Cannot be instantiated directly.

  We create a specialized version for a specific type by supplying the name of the type when the class is used.

  Not a macro!

- Supplying the parameter effectively creates a new class, which can be instantiated.

The Generic List Class

- List<T> is the generic List class

T represents a class name parameter to be supplied in declarations.

- Provides traditional list operations

Insert

Delete

- Also provides array operations

Access by position, using index notation

```
// A partial listing of the List<T> class.
namespace System.Collections.Generic
{
  public class List<T> :
    IList<T>, ICollection<T>, IEnumerable<T>, IReadOnlyList<T>
    IList, ICollection, IEnumerable
  {
...
    public void Add(T item);
    public ReadOnlyCollection<T> AsReadOnly();
    public int BinarySearch(T item);
    public bool Contains(T item);
    public void CopyTo(T[] array);
    public int FindIndex(System.Predicate<T> match);
    public T FindLast(System.Predicate<T> match);
    public bool Remove(T item);
    public int RemoveAll(System.Predicate<T> match);
    public T[] ToArray();
    public bool TrueForAll(System.Predicate<T> match);
    public T this[int index] { get; set; }
  }
}
```

List<T> Methods

- Add (T item)

  Add item at end of list

- Insert (int index, T item)

  Insert item at a specific position

- Remove (T item)

  Remove first occurance of item

- RemoveAt (int index)

  Remove item at specified position

List<T> Methods

- Clear()

  Removes all items from the list

- bool Contains(T item)

  Determines if item is in the list

- int IndexOf(T item)

  Returns index of item, or -1 if not in list.

- Sort()

... more

List<T> Indexer

- Array index notation can be used to get or set a specified item.

```
int_list[5]  =  17;

int temp = int_list[5];
```

- Throws an exception if int_list[5] does not exist.

List<T> Properties

Count	Number of items in the list

List<int> Example

```
static void Main(string[] args)
{
    List<int> ints = new List<int>();

    ints.Add(1);
    ints.Add(2);
    ints.Add(3);

    foreach (int i in ints)
    {
        Console.WriteLine(i.ToString() );
    }
    Console.ReadLine();
}
```

List<Circle> Example

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main(string[] args)
    {
        List<Circle> circles = new List<Circle>();

        circles.Add(new Circle("C1", 1.0));
        circles.Add(new Circle("C2", 2.0));
        circles.Add(new Circle("c3", 3.0));

        foreach (Circle c in circles)
        {
            Console.WriteLine(c.Name());
        }
        Console.ReadLine();
    }
}
```

Accessing List by Position

```
static void Main(string[] args)
{
    List<Circle> circles = new List<Circle>();

    circles.Add(new Circle("C1", 1.0));
    circles.Add(new Circle("C2", 2.0));
    circles.Add(new Circle("c3", 3.0));

    for (int i = 0; i < circles.Count; ++i)
    {
        Console.WriteLine(circles[i].Name());
    }
    Console.ReadLine();
}
```

Same result

Inserting

```
static void Main(string[] args)
{
    List<Circle> circles = new List<Circle>();

    circles.Add(new Circle("C1", 1.0));
    circles.Add(new Circle("C2", 2.0));
    circles.Add(new Circle("c3", 3.0));

    Circle c4 = new Circle("C4", 4.0);
    circles.Insert(2, c4);

    for (int i = 0; i < circles.Count; ++i)
    {
        Console.WriteLine(circles[i].Name());
    }
    Console.ReadLine();
}
```

Inserting and Deleting

```
static void Main(string[] args)
{
    List<Circle> circles = new List<Circle>();
    circles.Add(new Circle("C1", 1.0));
    circles.Add(new Circle("C2", 2.0));
    circles.Add(new Circle("c3", 3.0));
    Circle c4 = new Circle("C4", 4.0);
    circles.Insert(2, c4);
    circles.RemoveAt(1);
    for (int i = 0; i < circles.Count; ++i)
    {
        Console.WriteLine(circles[i].Name());
    }
    Console.ReadLine();
}
```

Sorting

- List<T> has a Sort method.
- Parameter class must implement the IComparable<T> interface.

- Example:

```
class Schedule_Record : IComparable<Schedule_Record>
{
    private String college;
    ...
```

IComparable interface

- Class must implement CompareTo() method, taking an object of the same type as its parameter.

Return negative int if this object < other

Return 0 if this object == other

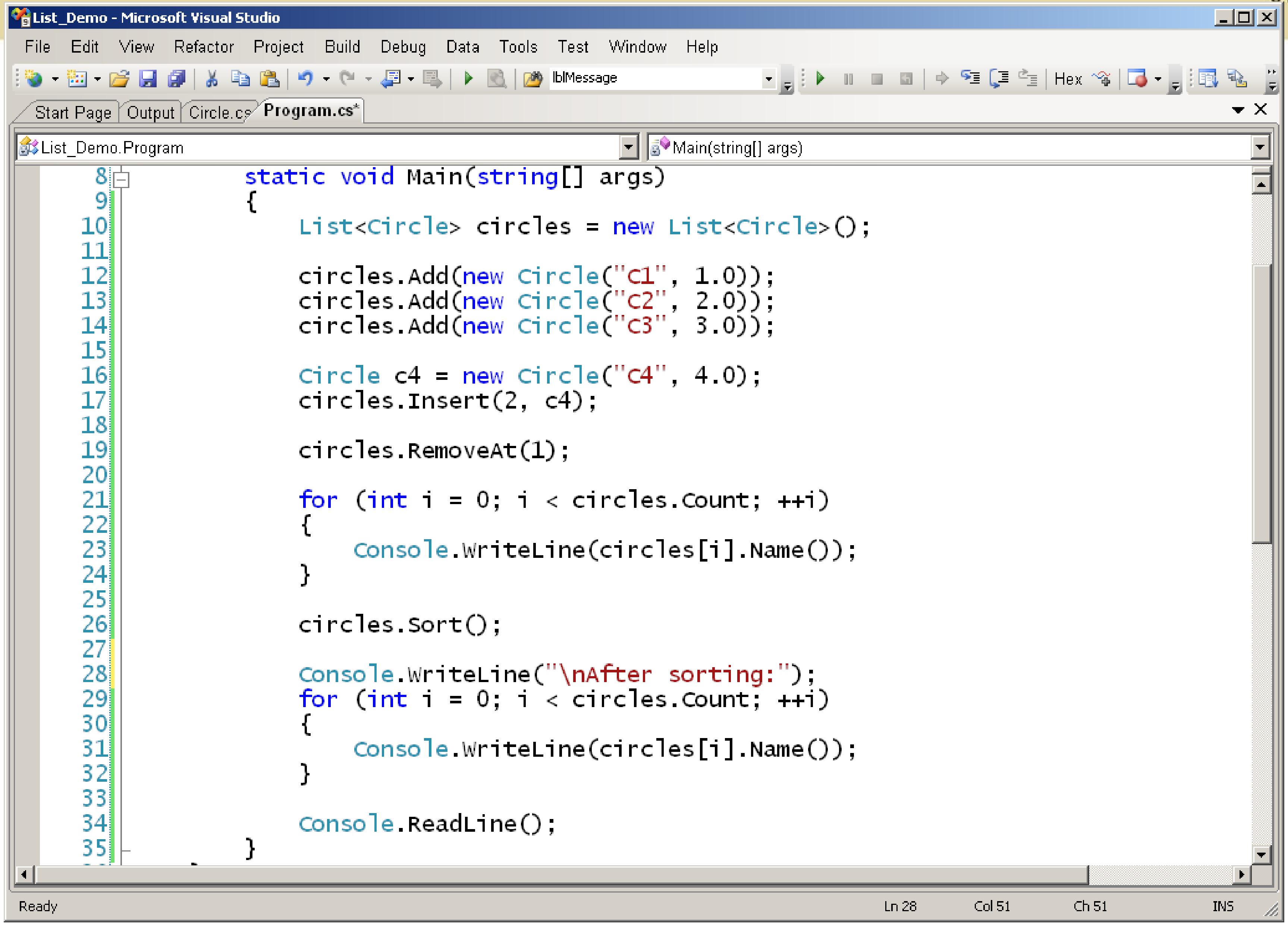
return positive int if this object > other

Same as strcmp() in C

Implementing IComparable

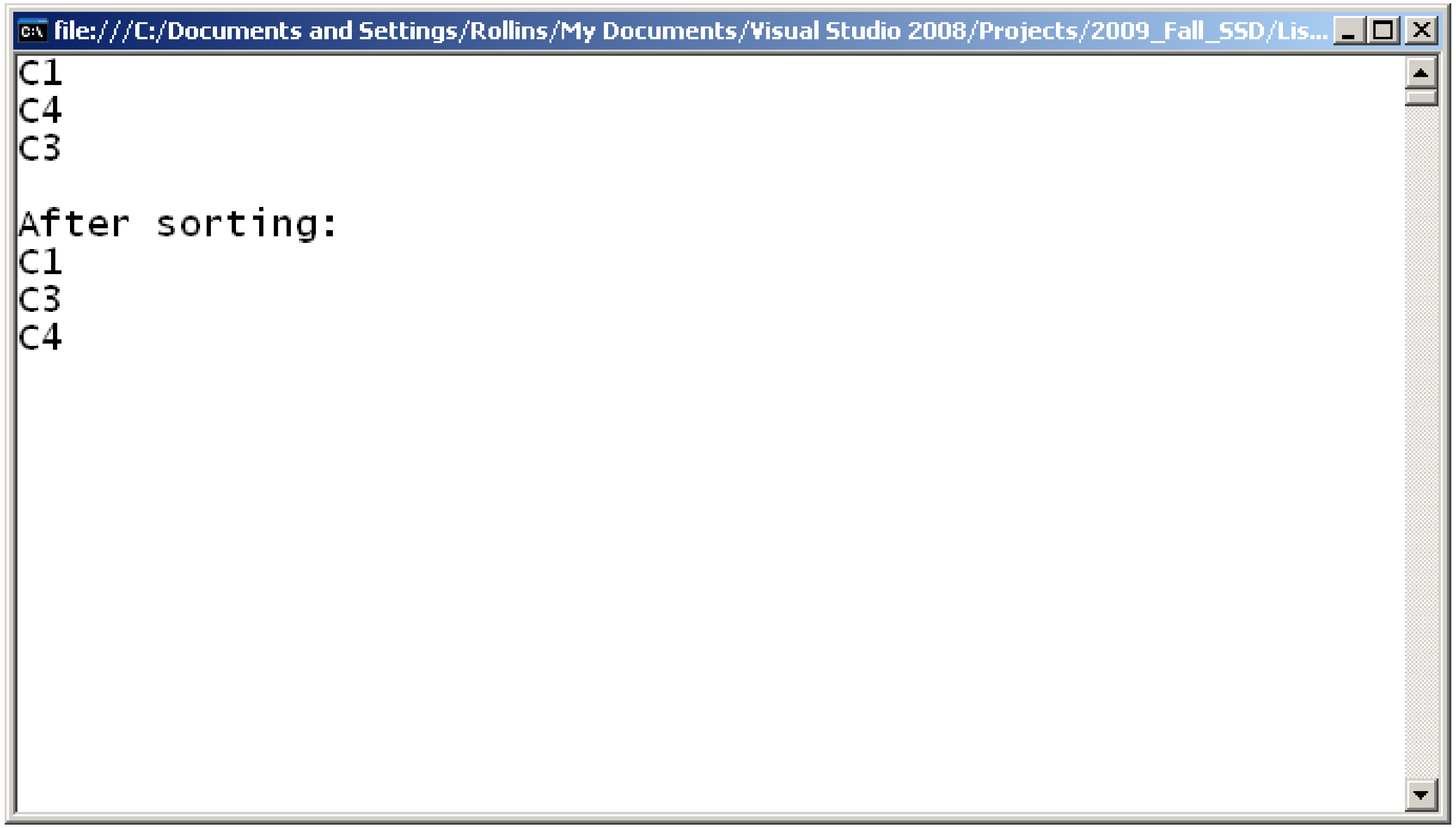
```
class Circle : IComparable<Circle>
{
    private String name;
    private double radius = 0.0;
...
    public int CompareTo(Circle other)
    {
        if (this.radius < other.radius)
            return -1;
        else if (this.radius > other.radius)
            return 1;
        else
            return 0;
    }
```

Murdoch UNIVERSITY



```csharp
        static void Main(string[] args)
        {
            List<Circle> circles = new List<Circle>();

            circles.Add(new Circle("C1", 1.0));
            circles.Add(new Circle("C2", 2.0));
            circles.Add(new Circle("C3", 3.0));

            Circle c4 = new Circle("C4", 4.0);
            circles.Insert(2, c4);

            circles.RemoveAt(1);

            for (int i = 0; i < circles.Count; ++i)
            {
                Console.WriteLine(circles[i].Name());
            }

            circles.Sort();

            Console.WriteLine("\nAfter sorting:");
            for (int i = 0; i < circles.Count; ++i)
            {
                Console.WriteLine(circles[i].Name());
            }

            Console.ReadLine();
        }
```

Program Running


file:///C:/Documents and Settings/Rollins/My Documents/Visual Studio 2008/Projects/2009_Fall_SSD/Lis...

```
C1
C4
C3

After sorting:
C1
C3
C4
```

The Generic Queue

- Queue<T>

  http://msdn.microsoft.com/en-us/library/7977ey2c.aspx
- Methods

  Enqueue (T item)

  Add an item to the end of the queue

  T Dequeue()

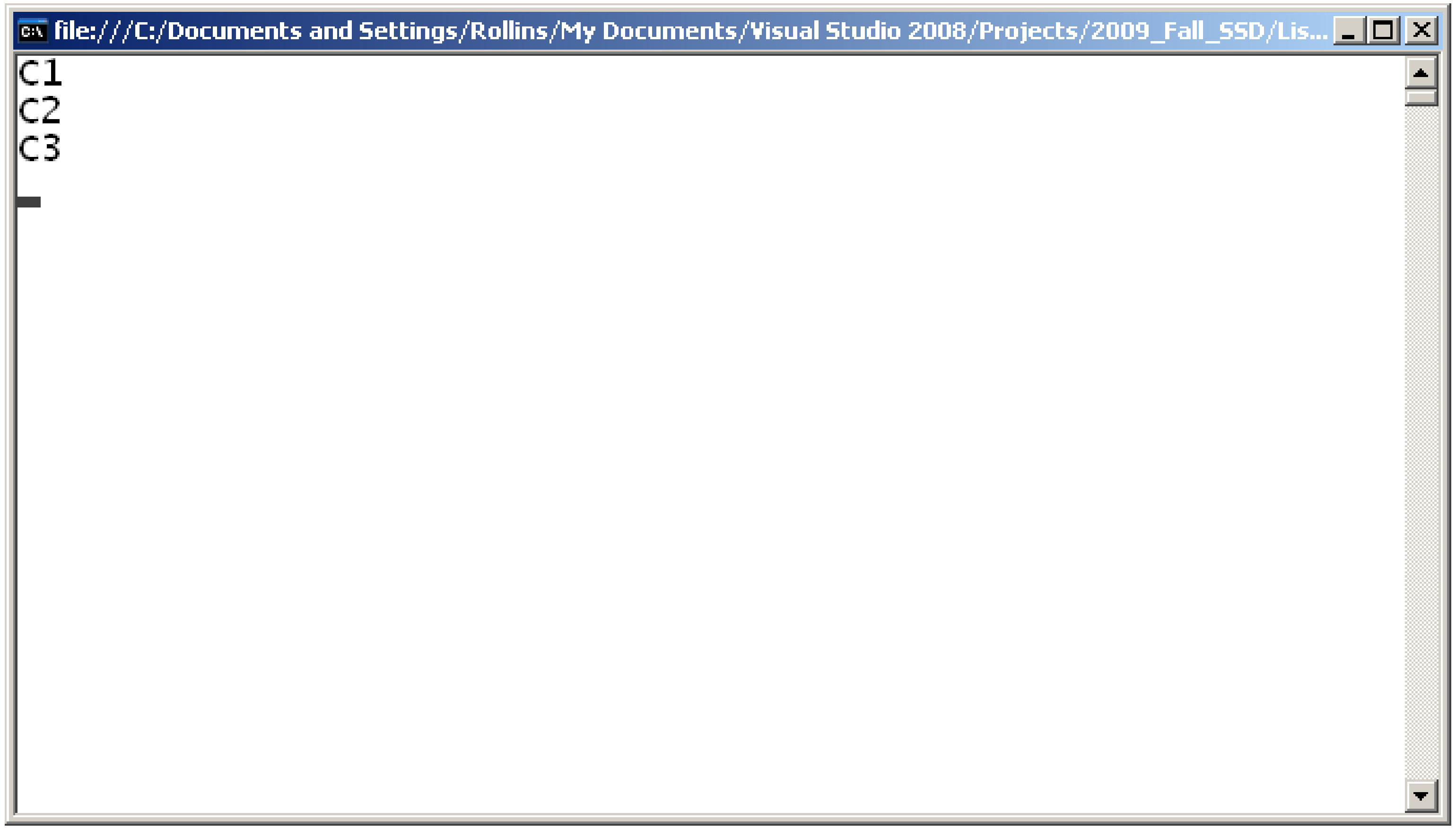
  Removes and returns object at the head of the queue

  Clear(), Contains(), Peek(), ... many more

Queue<Circle> Example

```
static void Main(string[] args)
{
    Queue<Circle> circles = new Queue<Circle>();
    circles.Enqueue(new Circle("C1", 1.0));
    circles.Enqueue(new Circle("C2", 2.0));
    circles.Enqueue(new Circle("c3", 3.0));

    while (circles.Count > 0)
    {
        Circle c = circles.Dequeue();
        Console.WriteLine(c.Name());
    }
    Console.ReadLine();
}
```

Queue<Circle> Example Running



file:///C:/Documents and Settings/Rollins/My Documents/Visual Studio 2008/Projects/2009_Fall_SSD/Lis...

```
C1
C2
C3
```

Dictionary<K,V>

Murdoch UNIVERSITY

- http://msdn.microsoft.com/en-us/library/xfhwa508(VS.80).aspx

- Stores (Key, Value) pairs

  Class **`KeyValuePair<K,V>`**

- Template parameters

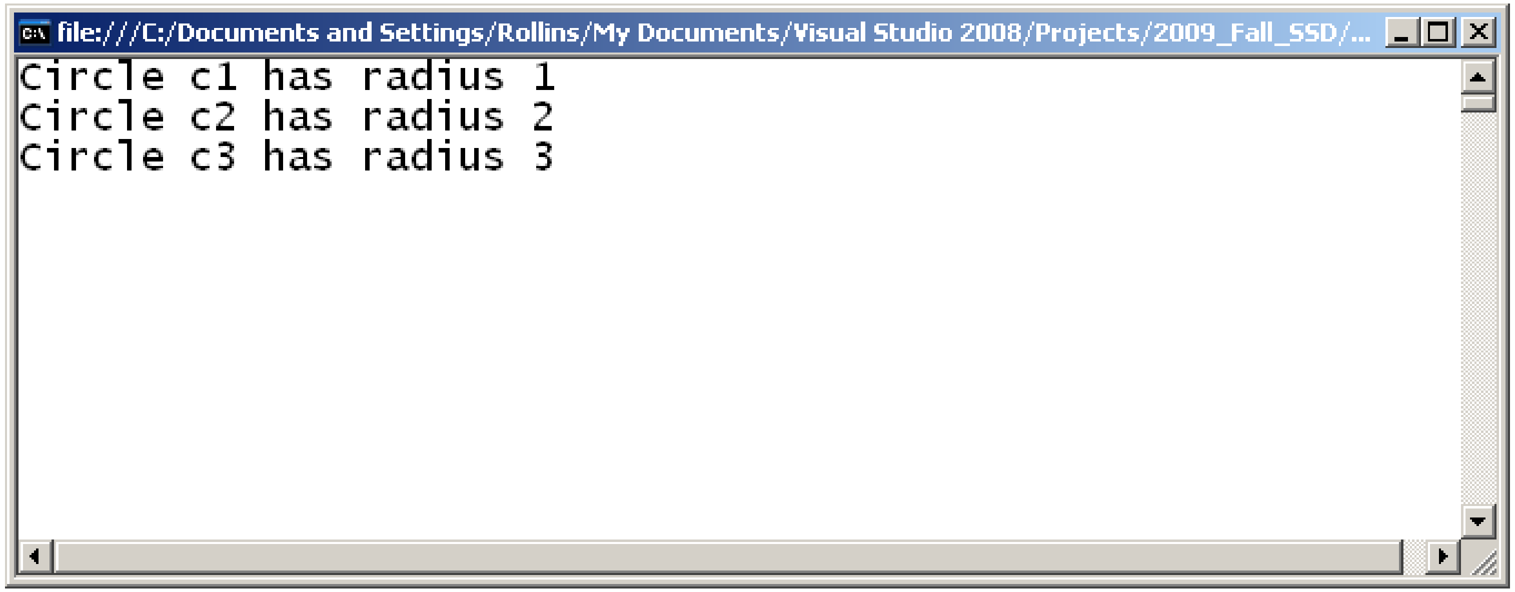
  K is type of Key

  V is type of Value

Dictionary<K,V>

```
static void Main(string[] args)
{
    Dictionary<String, Circle> circles =
        new Dictionary<String, Circle>();
    circles.Add("c1", new Circle("C1", 1.0));
    circles.Add("c2", new Circle("C2", 2.0));
    circles.Add("c3", new Circle("c3", 3.0));
    foreach (KeyValuePair<String, Circle> kvp in circles)
    {
        String k = kvp.Key;
        Circle c = kvp.Value;
        Console.WriteLine("Circle {0} has radius {1}",
                   k, c.Radius());
    }
    Console.ReadLine();
}
```

Dictionary<K,V> Example


file:///C:/Documents and Settings/Rollins/My Documents/Visual Studio 2008/Projects/2009_Fall_SSD/...

```
Circle c1 has radius 1
Circle c2 has radius 2
Circle c3 has radius 3
```

Using Key as Indexer

```
static void Main(string[] args)
{
    Dictionary<String, Circle> circles =
        new Dictionary<String, Circle>();
    circles.Add("c1", new Circle("C1", 1.0));
    circles.Add("c2", new Circle("C2", 2.0));
    circles.Add("c3", new Circle("c3", 3.0));
    Circle c = circles["c2"];
       Console.WriteLine("Circle {0} has radius {1}",
        c.Name(), c.Radius());

    Console.ReadLine();
}
```

About Dictionary<K,V>

- Key class must have a compare for equality operation.

- Keys must be unique.

Attempting to Add an item with a key already in the Dictionary will result in an exception.

Can *set* entry with an existing key, using indexer notation.

Adding with Existing Key

```
static void Main(string[] args)
{
    Dictionary<String, Circle> circles =
        new Dictionary<String, Circle>();
    circles.Add("c1", new Circle("C1", 1.0));
    circles.Add("c2", new Circle("C2", 2.0));
    circles.Add("c3", new Circle("c3", 3.0));
    Circle c = circles["c2"];
       Console.WriteLine("Circle {0} has radius {1}",
        c.Name(), c.Radius());
    circles["c2"] = new Circle("New C2", 200.0);
    c = circles["c2"];
    Console.WriteLine("Circle {0} has radius {1}",
        c.Name(), c.Radius());
    Console.ReadLine();
}
```

Hashtable versus Dictionary

```
Hashtable numbers = new Hashtable();

          numbers.Add(1, "one");

          numbers.Add(2, "two");

          numbers.Add(3, "three");

          numbers.Add(4, "four");

          numbers.Add(5, "five");

foreach (DictionaryEntry num in numbers)

        {

             MessageBox.Show(num.Key + "   -   "
+ num.Value);

          }
```

```
Dictionary<int, string> dictionary = new
Dictionary<int, string >();
          dictionary.Add(1,"one");
          dictionary.Add(2,"two");
          dictionary.Add(3,"three");
          dictionary.Add(4,"four");
          dictionary.Add(5, "five");
          foreach
(KeyValuePair<int,string> pair in
dictionary)
          {
             MessageBox.Show(pair.Key +
"   -   " + pair.Value);
```

Summary

- The .NET Generics make life easier.

- Use List<T> like an array

  without worrying about size.

  Plus additional features!

- Use Dictionary<K,V> to store and retrieve objects by Key value.

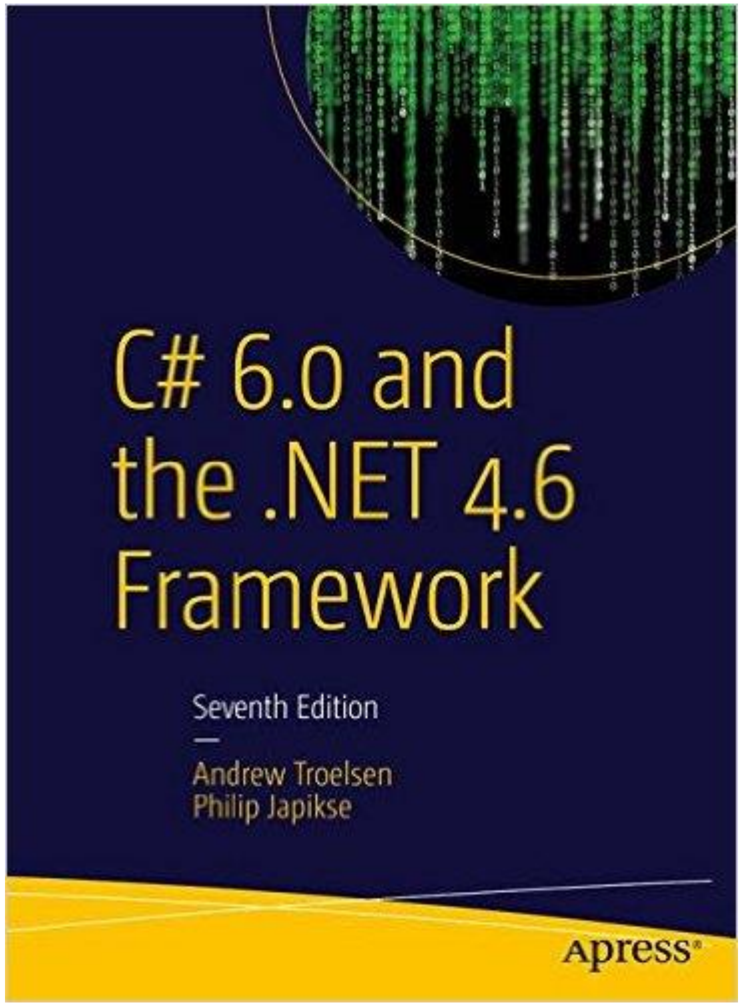
- Use Stack and Queue when you need those concepts.

Reading/ reference

http://prospero.murdoch.edu.au/record=b2962780~S1

Chapter: Collections and Generics

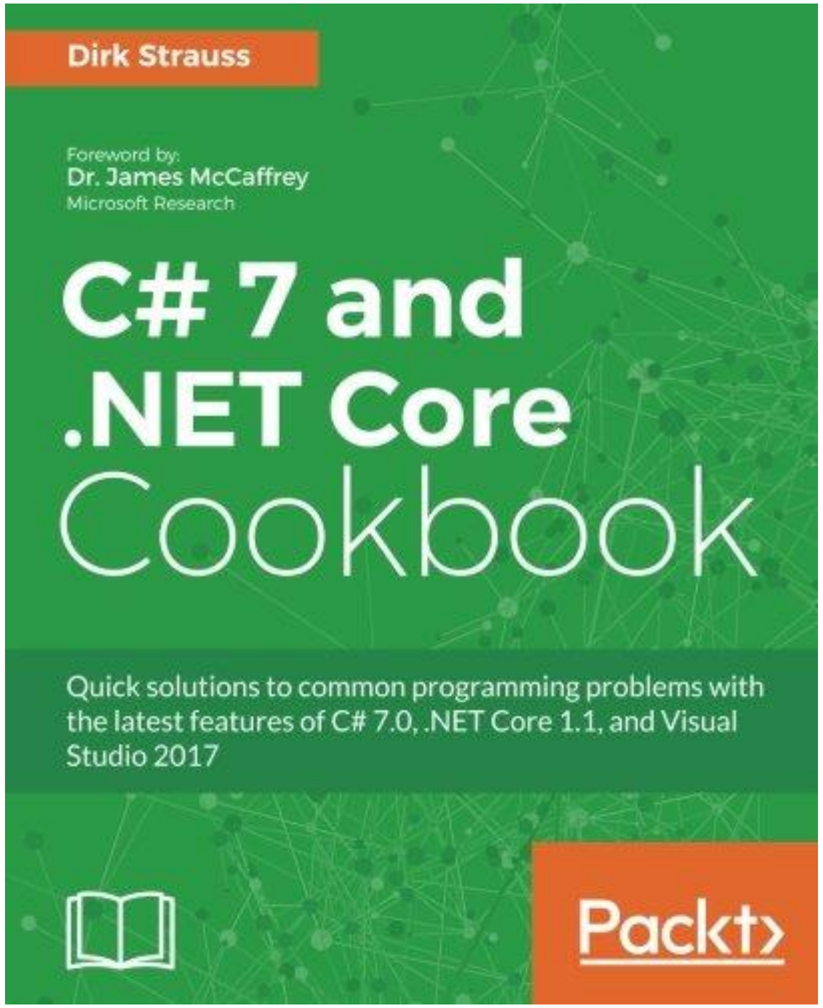
Chapter: Delegates, Events, and Lambda Expressions



Reading/ reference

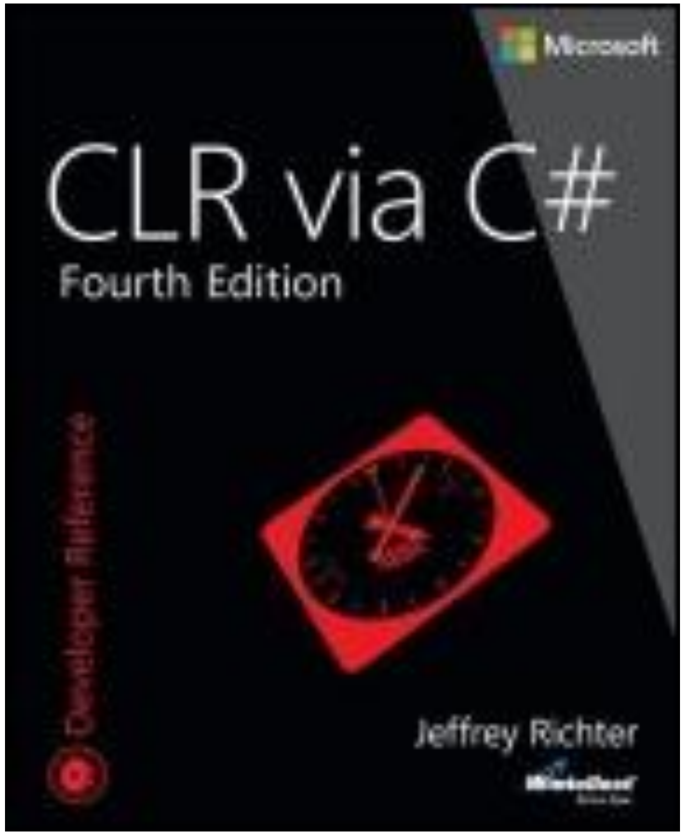
http://prospero.murdoch.edu.au/record=b2962781~S1

Chapter: REGULAR EXPRESSIONS


Dirk Strauss
Foreword by:
Dr. James McCaffrey
Microsoft Research
C# 7 and .NET Core Cookbook
Quick solutions to common programming problems with the latest features of C# 7.0, .NET Core 1.1, and Visual Studio 2017
Packt

Reading/ reference

Chapter 12. Generics



- **System.Collections Namespaces**
- https://msdn.microsoft.com/en-us/library/mt481475(v=vs.110).aspx

- **System.Collections.Generic Namespace**
- https://msdn.microsoft.com/en-us/library/system.collections.generic(v=vs.110).aspx