Collections and Generics

CSIS 3540

Client Server Systems

Class 03

Topics

- Collections
  - ArrayList
  - HashTable
  - Stack
  - Queue
- Boxing/Unboxing
- Generic Collections

Useful Collections

The System.Collections Namespace

When the .NET platform was first released, programmers frequently used the nongeneric collection classes found within the `System.Collections` namespace, which contains a set of classes used to manage and organize large amounts of in-memory data. Table 9-1 documents some of the more commonly used collection classes of this namespace and the core interfaces they implement.

Table 9-1. *Useful Types of System.Collections*

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`

Collections interfaces

Table 9-2. *Key Interfaces Supported by Classes of System.Collections*

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

Using ICollection interface

- Note that any non-generic ICollection also implements IDictionary
- See FunWithCollections

```
/// <summary>
/// Display an arbitrary collection on the console
/// </summary>
/// <param name="collection">Collection</param>
/// <param name="header">String to display on first line</param>
static void Display(ICollection collection, string header = null)
{
    if (header != null)
        Console.WriteLine("\n" + header + "\n");
    foreach (DictionaryEntry d in collection)
    {
        Console.WriteLine($"{d.Key} {d.Value}");
    }
}
```

ArrayList

- Basically, a linked list that can be used like an array for referencing elements.
- You can add any object with the Add() method, then retrieve it using an index.
  - AddRange() allows the addition anything that implements ICollection (Stack, Queue, …)
- Can also iterate over an ArrayList using foreach
  - Display() method from prior slide will work
- Note need to cast

```
// ArrayList example

ArrayList greetingsArrayList = new ArrayList();

greetingsArrayList.AddRange(entries);

Display(greetingsArrayList, "Display() using ArrayList, foreach");

Console.WriteLine("Inline using ArrayList, normal for statement, and DictionaryEntry
cast");

for (int i = 0; i < greetingsArrayList.Count; i++)
{
    DictionaryEntry d = (DictionaryEntry)greetingsArrayList[i];
    Console.WriteLine($"{d.Key} {d.Value}");
}
```

Hashtable

- uses Add(object key, object value)
- allows fast lookup using Indexer
  - put the key as an array index and the value will be returned
  - see hash[“2”] below
- Can use foreach to iterate, but notice need to use DictionaryEntry using Display() method from prior slide
- keys must be unique or an exception will be thrown

```
// Hashtable example with inline initialization

Hashtable greetingsHashtable = new Hashtable
{
    { "1", "Hello" },
    { "2", "Goodbye" },
    { "3", "Maybe" },
}; // also use SortedList() as an alternative

// another example for add elements using Add()

//Hashtable greetingsHashtable = new Hashtable(); // also use SortedList() as an alternative

//greetingsHashtable.Add("1", "Hello");
//greetingsHashtable.Add("2", "Goodbye");
//greetingsHashtable.Add("3", "Maybe");

Display(greetingsHashtable, "Display using HashTable");

// do a hash lookup

Console.WriteLine("In greetingsHashtable, the value of {0} is {1}",
    "2",
    greetingsHashtable["2"]);
```

Queue

- FIFO (first in first out)
- Enqueue(object o) puts the object in the queue
- Dequeue() returns the object and removes it from the queue

```
Queue queue = new Queue();

queue.Enqueue(entry[0]);
queue.Enqueue(entry[1]);
queue.Enqueue(entry[2]);

Console.WriteLine("queue count {0}", queue.Count);
while (queue.Count > 0)
{
    DictionaryEntry d = (DictionaryEntry)queue.Dequeue();
    Console.WriteLine("dequeue {0} {1}", d.Key, d.Value);
}
```

Stack

- LIFO (last in, first out)
- Push(object o) puts the object on the stack
- Pop() returns the last object pushed on the stack and removes it

```
Stack stack = new Stack();

stack.Push(entry[0]);
stack.Push(entry[1]);
stack.Push(entry[2]);

Console.WriteLine("stack count {0}", stack.Count);
while (stack.Count > 0)
{
    DictionaryEntry d = (DictionaryEntry)stack.Pop();
    Console.WriteLine("pop stack {0} {1}", d.Key, d.Value);
}
```

Boxing/unboxing

- Automatic conversion (boxing) of any type to object
- Must cast to unbox
- See **IssuesWithNonGenericCollections**

```
// Make a int value type.
int myInt = 25;

// Box the int into an object reference.
object boxedInt = myInt;

// Unbox
int unboxedInt = (int)boxedInt;
```

Using Collections

- See **FunWithCollections**
- Biggest issue is need to cast
- For example, let's use the DictionaryEntry class, which allows us to create key,value pairs (properties of object type)
- **Note need to cast in the for loop** – better to use generics

```
DictionaryEntry[] entry = new DictionaryEntry[3];

entry[0].Key = "1";
entry[0].Value = "Hello";
entry[1].Key = "2";
entry[1].Value = "Goodbye";
entry[2].Key = "3";
entry[2].Value = "Maybe";
// ArrayList example

ArrayList greetingsArrayList = new ArrayList();

greetingsArrayList.AddRange(entry);

Display(greetingsArrayList, "Display using ArrayList, foreach");

Console.WriteLine("Inline using ArrayList, normal for statement, and DictionaryEntry cast");

for (int i = 0; i < greetingsArrayList.Count; i++)
{
    DictionaryEntry d = (DictionaryEntry)greetingsArrayList[i];
    Console.WriteLine($"{d.Key} {d.Value}");
}
```

Generics

- Classes can be generic
- Can hold arbitrary types of objects.
- Form: ClassName<T,U,V,…> where <T,U,V,…> are object types
- Very powerful!
- See **GenericPoint**

```
public class Point<T>
{
    // Generic properties
    public T X { get; set; }
    public T Y { get; set; }

    // Generic constructor.
    public Point(T xVal, T yVal)
    {
        X = xVal;
        Y = yVal;
    }

    public override string ToString()
    {
        // Format plays nice if args are null

        return $"Point coordinate [{X}, {Y}]";
    }

    /// <summary>
    /// Reset fields to default values. If ref, sets to null
    /// </summary>
    public void ResetPoint()
    {
        X = default(T);
        Y = default(T);
    }
}
```

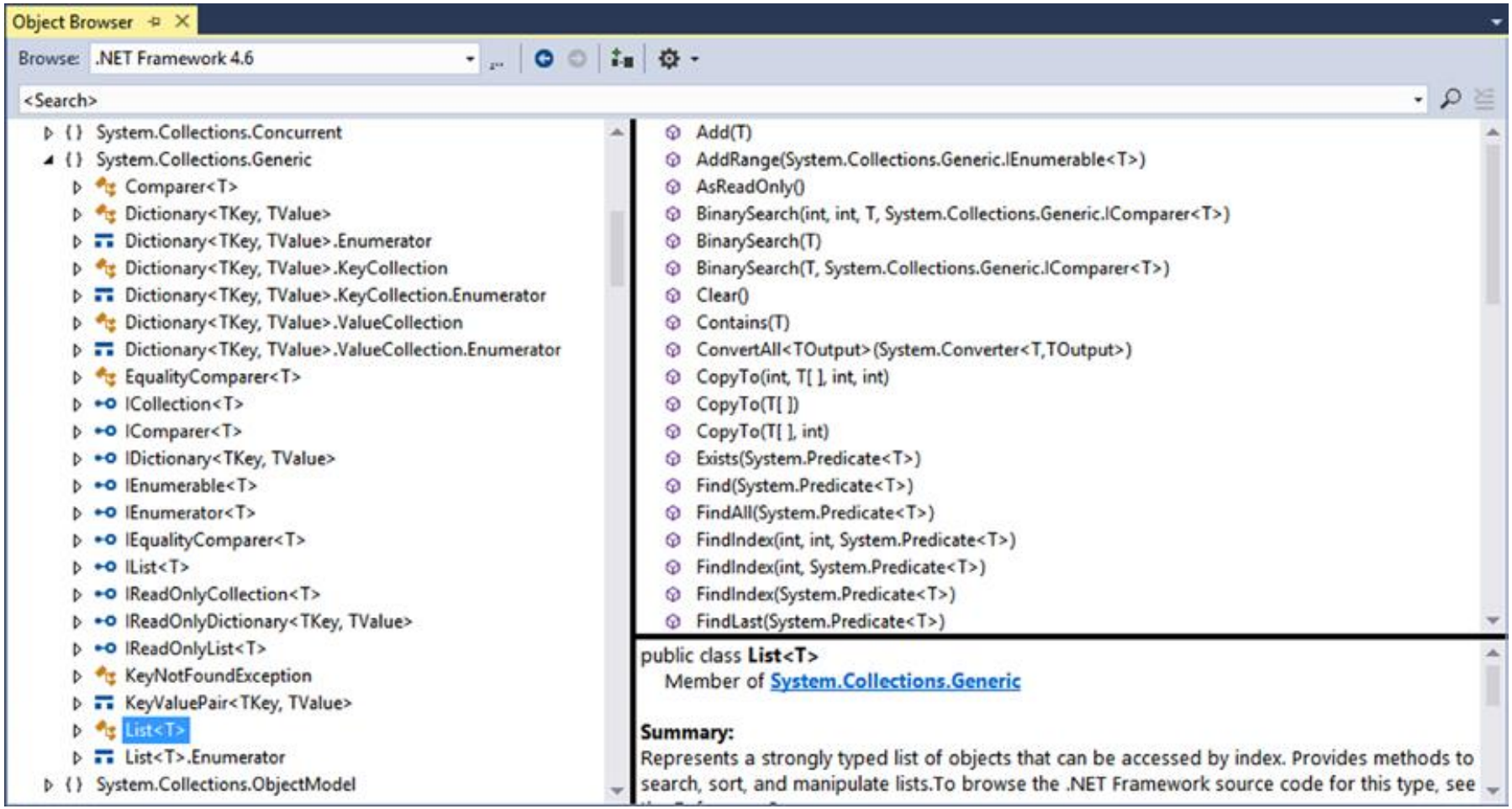
Generic Collections

- Form of CollectionName<T,…,T> where T is type
  - Can list multiple types
- You will likely never have to create a particular generic Collection, but it is the **best** way to utilize Collections
  - No need to cast!!
  - No need to box/unbox!
- See List<T> below
  - Add(), Sort(), and other useful methods
- More than just Collections
  - Only classes, structures, interfaces, and delegates can be written generically; enum types cannot
- See **IssuesWithNonGenericCollections**

```
            List<int> moreInts = new List<int>();
            moreInts.Add(10);
            moreInts.Add(2);
            moreInts.Sort();
            int sum = moreInts[0] + moreInts[1];

            Console.WriteLine("Array of Ints {0} {1} sum = {2}", moreInts[0], moreInts[1],
sum);
```

List<T> members (mscorlib)



System Generic Collections (mscorlib)

No ArrayList!!

Table 9-5. *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.
`LinkedList<T>`	`ICollection<T>`, `IEnumerable<T>`	This represents a doubly linked list.
`List<T>`	`ICollection<T>`, `IEnumerable<T>`,`IList<T>`	This is a dynamically resizable sequential list of items.
`Queue<T>`	`ICollection` (Not a typo! This is the nongeneric 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 nongeneric collection interface), `IEnumerable<T>`	This is a generic implementation of a last-in, first-out (LIFO) list.

Interfaces used by Generic Collections

Table 9-4. *Key Interfaces Supported by Classes of System.Collections.Generic*

System.Collections.Generic Interface	Meaning in Life
`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

List<T>, Stack<T>, Queue<T>, SortedSet<T>

- See FunWithGenericCollections

Generic Interfaces

- IComparer<T> interface

- Comparison used by collections
  - List<T>.Sort(new CompareObjects<T>())
  - new SortedList()
- See
  - FunWithGenericCollections

```
    class SortPeopleByAge : IComparer<Person>
    {
        public int Compare(Person firstPerson, Person secondPerson)
        {
            if (firstPerson.Age > secondPerson.Age)
                return 1;
            if (firstPerson.Age < secondPerson.Age)
                return -1;
            else
                return 0;
        }
    }
    class SortPeopleByFirstName : IComparer<Person>
    {
        public int Compare(Person a, Person b)
        {
            return a.FirstName.CompareTo(b.FirstName);

        }
    }
```

```
    List<Person> people = new List<Person>()
    {
         new Person {FirstName= "Homer", LastName="Simpson", Age=47},
         new Person {FirstName= "Marge", LastName="Simpson", Age=45},
         new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
         new Person {FirstName= "Bart", LastName="Simpson", Age=8}
     };
     people.Sort(new SortPeopleByFirstName()); // sort list
```

```
SortedSet<Person> setOfPeople = new SortedSet<Person>(new
SortPeopleByAge())
{
    new Person {FirstName= "Homer", LastName="Simpson", Age=47},
    new Person {FirstName= "Marge", LastName="Simpson", Age=45},
    new Person {FirstName= "Lisa", LastName="Simpson", Age=9},
    new Person {FirstName= "Bart", LastName="Simpson", Age=8}
};
```

Generic methods

Whenever you have a group of overloaded methods that differ only by incoming arguments, this is your clue that generics could make your life easier. Consider the following generic Swap<T> method that can swap any two Ts:

```
// This method will swap any two items.
// as specified by the type parameter <T>.
static void Swap<T>(ref T a, ref T b)
{
  Console.WriteLine("You sent the Swap() method a {0}",
    typeof(T));
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

Custom Generic Methods

- Can be used with objects as well!
- See **CustomGenericMethods**

```
public static class MyGenericMethods
{
    public static void Swap<T>(ref T a, ref T b)
    {
        Console.WriteLine("You sent the Swap() method a {0}",
            typeof(T));
        T temp;
        temp = a;
        a = b;
        b = temp;
    }

    public static void DisplayBaseClass<T>()
    {
        Console.WriteLine("Base class of {0} is: {1}.",
            typeof(T), typeof(T).BaseType);
    }
}
-----------
            Person p1 = new Person() { FirstName = "John", LastName = "Doe", Age = 21 };
            Person p2 = new Person() { FirstName = "Sven", LastName = "Klepsch", Age = 15 };

            MyGenericMethods.DisplayBaseClass<Person>();
            Console.WriteLine("Before swap: {0}, {1}", p1, p2);
            MyGenericMethods.Swap(ref p1, ref p2);
            Console.WriteLine("After swap: {0}, {1}", p1, p2);
```

Constrained generics

Using this keyword, you can add a set of constraints to a given type parameter, which the C# compiler will check at compile time. Specifically, you can constrain a type parameter as described in Table 9-8.

Table 9-8. *Possible Constraints for Generic Type Parameters*

Generic Constraint	Meaning in Life
`where T : struct`	The type parameter <T> must have `System.ValueType` in its chain of inheritance (i.e., <T> must be a structure).
`where T : class`	The type parameter <T> must not have `System.ValueType` in its chain of inheritance (i.e., <T> must be a reference type).
`where T : new()`	The type parameter <T> must have a default constructor. This is helpful if your generic type must create an instance of the type parameter because you cannot assume you know the format of custom constructors. Note that this constraint must be listed last on a multiconstrained type.
`where T : NameOfBaseClass`	The type parameter <T> must be derived from the class specified by `NameOfBaseClass`.
`where T : NameOfInterface`	The type parameter <T> must implement the interface specified by `NameOfInterface`. You can separate multiple interfaces as a comma-delimited list.

Unless you need to build some extremely type-safe custom collections, you might never need to use the `where` keyword in your C# projects. Regardless, the following handful of (partial) code examples illustrate how to work with the `where` keyword.

You will rarely encounter cases where you need to build a complete custom generic collection class; however, you can use the `where` keyword on generic methods, as well. For example, if you want to specify that your generic `Swap<T>()` method can operate only on structures, you would update the method like this:

```
// This method will swap any structure, but not classes.
static void Swap<T>(ref T a, ref T b) where T : struct
{
  ...
}
```

Note that if you were to constrain the `Swap()` method in this manner, you would no longer be able to swap `string` objects (as is shown in the sample code) because `string` is a reference type.