Advanced Programming Methods
Lecture 7

Content

Generics and Collections Part 2

- Java
- C#

IMPORTANT: The deadline of LAB67 has been extended to Laboratory 9. In Laboratory 8 you will get a new assignment!!!!

C# GENERICS

C#’s genericity mechanism, available since C# 2.0

Most common use:

- Use (and implement) generic yet type-safe containers

```
List<String> safeBox = new List<String>();
```

- Compile-time type-checking is enforced
- Custom generic classes (and methods)
- Syntax: different position of the generic parameter w.r.t. Java
  - Java: `public <T> T foo (T x);`
  - C#: `public T foo <T> (T x);`
- Bounded genericity

```
public T test <T> (T x) where T:Interface1, Interface2
```

C# vs Java

- Unlike Java, genericity is supported natively by .NET bytecode

- Hence, basically all limitations of Java generics disappear:

  - Can instantiate generic parameter with value types
  - At runtime you can tell the difference between **List<Integer>** and **List<String>**
  - Exception classes can be generic classes
  - Can instantiate a generic type parameter provided a clause **where G : new()** constrains the parameter to have a default constructor

C# vs. Java

- Can get the default value of a generic type parameter `T t = default (T);`
- Arrays can have elements of a generic type parameter
- A static member can reference a generic type parameter

Another consequence is that raw types (unchecked generic types without any type argument) don’t exist in C#

Generics and Inheritance

- Let S be a subtype of T

- There is no inheritance relation between:
  `SomeGenericClass<S>` and **`SomeGenericClass<T>`**

  In particular: the former is not a subtype of the latter

- However, let AClass be a non-generic type:

 `S<AClass>` is a subtype of **`T<AClass>`**

Replacing Wildcards in C#

- There's no C# equivalent of Java's wildcards

- But most of Java's wildcard code can be ported to C# (not necessarily resulting in cleaner code)

Consider the following hierarchy of classes:

class Circle:Shapes{...}

class Rectangle:Shapes{...}

What should be the signature of a method **drawShapes** that takes a list of **Shape** objects and draws all of them?

DrawShapes(List<Shape> shapes)

- **this doesn't work on a List<Circle>, which is not a subtype of List<Shape>**

Replacing Wildcards in C#

Solution: use a helper class with bounded genericity

```
class DrawHelper <T> where T: Shape {
public static void DrawShapes( List<T> shapes);
}
```

The use of the method:

```
 DrawHelper<Shape>.DrawShapes(listOfShapes);
 DrawHelper<Circle>.DrawShapes(listOfCircles);
```

Generics can be used with:

- Types
    - Struct
    - Interface
    - Class
    - Delegate
- Methods
- Generic Constraints

Some examples on the next slides

Generics can be used:

- to easily create non-generic derived types:

```
public class IntStack : Stack<int> {...}
```

- in internal fields, properties and methods of a class:

```
public struct Customer<T>{
    private static List<T> customerList;
    private T customerInfo;
    public T CustomerInfo { get; set; }
    public int CompareCustomers( T customerInfo );}
```

- A base class or interface can be used as a constraint.
- For instance

```
public interface IDrawable { public void Draw(); }
```

- Need a constraint that our type T implements the IDrawable interface.

```
public class SceneGraph<T> where T : IDrawable {
public void Render() { … T node; …
        node.Draw();
}
}
```

- No need to cast
  - Compiler uses type information to decide

- Can also specify a class constraint.
- That is, require a reference type:

```
public class CarFactory<T> where T : class {
private T currentCar = null;
```

- Forbids CarFactory<int> and other value types.
- Useful since I can not set an int to null.

- Alternatively, require a value (struct) type.

```
public struct Nullable<T> where T : struct {
private T value;
```

- The ***default*** keyword

```
public class GraphNode<T> {
    private T nodeLabel;
    private void ClearLabel() { nodeLabel = default(T); }
```

- If T is a reference type default(T) will be null.
- For value types all bits are set to zero.

- Special constraint using the *new* keyword:

```
public class Stack<T> where T : new() {
    public T PopEmpty() {
        return new T();
    }
}
```

- Parameter-less ***constructor constraint***
- Type T must provide a public parameter-less constructor
- No support for other constructors or other method syntaxes.
- The new() constraint must be the last constraint.

- A generic type parameter, like a regular type, can have zero or more interface constraints

```
public class GraphNode<T> {
    where T : ICloneable, IComparable
        ...}
```

- A type parameter can only have one where clause, so all constraints must be specified within a single where clause.
- You can also have one type parameter be dependent on another.

```
public class SubSet<U,V> where U : V
public class Group<U,V>
    where V : IEnumerable<U>  { … }
```

- C# also allow you to parameterize a method with generic types:

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

- The method does not need to be static.

```
public class Report<T> : where T Iformatter{...}
public class Insurance {
   public Report<T> ProduceReport<T>() where T : Iformatter  {...}
}
```

- In C#, generic types can be compiled into a class library or dll and used by many applications.

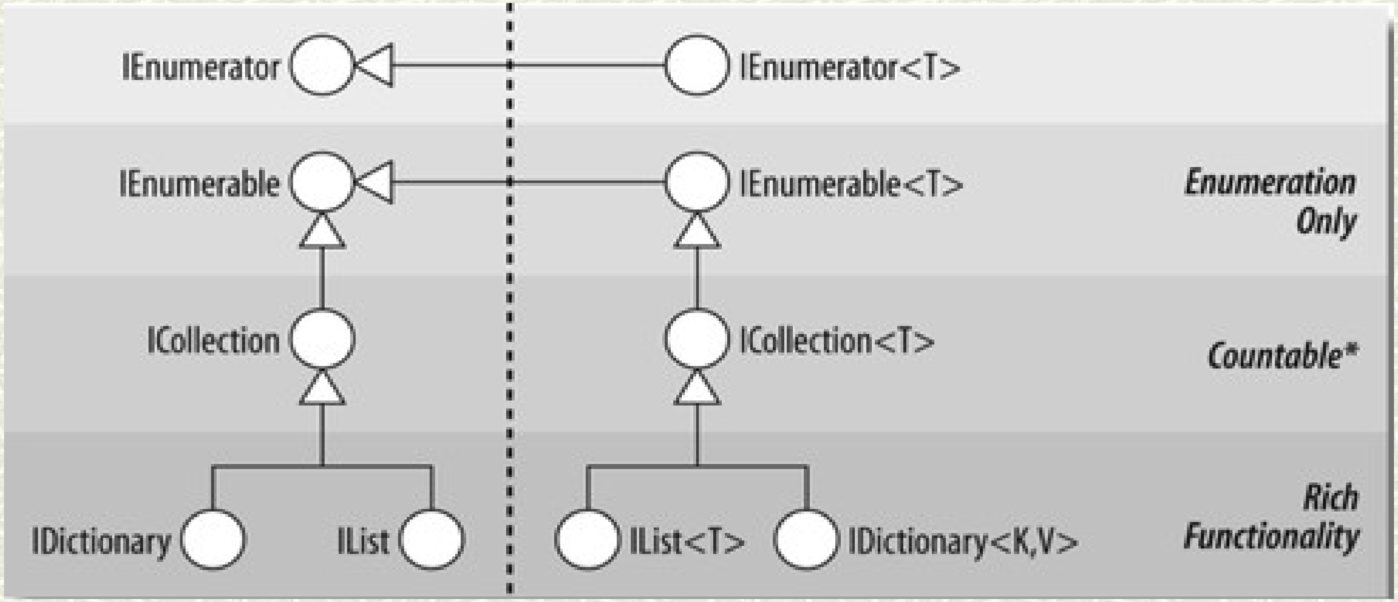
- Differs from C++ templates, which use the source code to create a new type at compile time.

- Hence, when compiling a generic type, the compiler needs to ensure that the code will work for any type.

C# COLLECTIONS

Data Structures in C#

- `System.Collections` for nongeneric collection classes and interfaces.
- `System.Collections.Specialized` for strongly typed nongeneric collection classes.
- `System.Collections.Generic` for generic collection classes and interfaces.
- `System.Collections.ObjectModel` contains proxies and bases for custom collections.



IEnumerator - IEnumerable

```
//System.Collections

public interface IEnumerator {
  bool MoveNext( );
  object Current { get; }
  void Reset( );
}
```

```
//System.Collections

public interface IEnumerable{
  IEnumerator GetEnumerator( );
}
```

```
//System.Collections.Generic
public interface IEnumerator<T> :
 IEnumerator,IDisposable{
  T Current { get; }
}
```

```
//System.Collections.Generic
public interface IEnumerable<T> :
          IEnumerable{
  IEnumerator<T> GetEnumerator( );
}
```

If a class implements the `IEnumerable` interface, then the `foreach` statement can be used on that class.

IEnumerable -foreach

```
class Set : ISet, IEnumerable{
   object[] elems;
   public IEnumerator GetEnumerator(){ …}
   //…
}
…
   Set s=new Set();
   s.add(“ana”);
   s.add(“are”);
   s.add(“mere”);
   foreach(Object o in s){
    Console.WriteLine(“{0} “,o);
   }
```

ICollection/ ICollection<T>

```
public interface ICollection : IEnumerable{
   void CopyTo (Array array, int index);
   int Count {get;}
   bool IsSynchronized {get;}
   object SyncRoot {get;}
}

public interface ICollection<T> : IEnumerable<T>, IEnumerable{
  void Add(T item);
  void Clear(  );
  bool Contains (T item);
  void CopyTo (T[] array, int arrayIndex);
  int Count { get; }
  bool IsReadOnly { get; }
  bool Remove (T item);
}
```

IList / IList<T>

```
public interface IList : ICollection, IEnumerable{
  object this [int index] { get; set }
  bool IsFixedSize { get; }
  bool IsReadOnly  { get; }
  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);
}
public interface IList<T> : ICollection<T>, IEnumerable<T>,
                 IEnumerable{
  T this [int index] { get; set; }
  int IndexOf (T item);
  void Insert (int index, T item);
 void RemoveAt (int index);
}
```

IDictionary

```
public interface IDictionary : ICollection, IEnumerable{
   IDictionaryEnumerator GetEnumerator( );
   bool Contains (object key);
   void Add      (object key, object value);
   void Remove   (object key);
   void Clear( );
   object this [object key] { get; set; }
   bool IsFixedSize         { get; }
   bool IsReadOnly          { get; }
   ICollection Keys         { get; }
   ICollection Values       { get; }
}

public interface IDictionaryEnumerator : IEnumerator
{
   DictionaryEntry Entry { get; }
   object Key { get; }
   object Value { get; }
}
```

The IComparable Interface

This require one method CompareTo which returns

-1 if the first value is less than the second

0 if the values are equal

1 if the first value is greater than the second

This is a member of the class and compares `this` to an instance passed as a parameter

```
public interface IComparable {
 int CompareTo(object obj)
}
```

The IComparer Interface

This is similar to IComparable but is designed to be implemented in a class outside the class whose instances are being compared

Compare() works just like CompareTo()

```
public interface IComparer {
 int Compare(object o1, object o2);
}
```

Sorting an ArrayList

To use CompareTo() of IComparable

```
ArrayList.Sort()
```

To use a custom comparer object

```
ArrayList.Sort(IComparer cmp)
```

To sort a range

```
ArrayList.Sort(int start, int len,
IComparer cmp)
```

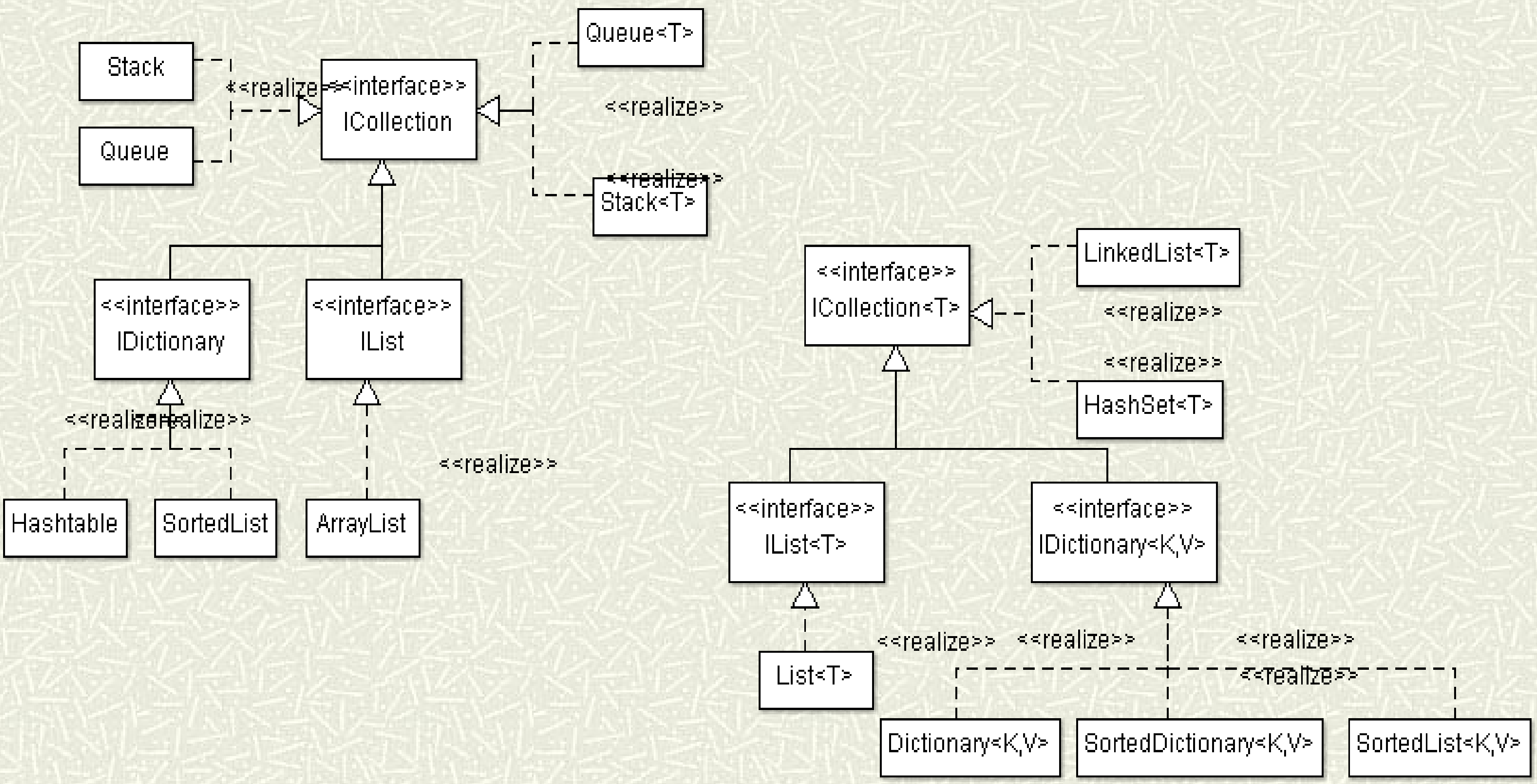
ICloneable Interface

This guarantees that a class can be cloned

The Clone method can be implemented to make a shallow or deep clone

```
public interface ICloneable {
 object Clone();
}
```

Data structures hierarchy



Generic Collections

Generic Class	Description
Dictionary<K, V>	Generic unordered dictionary
LinkedList<E>	Generic doubly linked list
List<E>	Generic ArrayList
Queue<E>	Generic queue
SortedDictionary<K, V>	Generic dictionary implemented as a tree so that elements are stored in order of the keys
SortedList<K, E>	Generic binary tree implementation of a list. Can have any type of subscript. More efficient than SortedDictionary in some cases.
Stack<E>	Generic stack

Array Class

The `Array` class is the implicit base class for all single and multidimensional arrays.

It provides a common set of methods to all arrays, regardless of their declaration or underlying element type.

Length and Rank:

```
public int  GetLength      (int dimension);
public int  Length       { get; }
public int Rank { get; }    // Returns number of dimensions in array
```

Searching in a one-dimensional array: `BinarySearch`, `IndexOf`, `LastIndexOf`, etc (static methods, overloaded).

Sorting: `Sort` (overloaded static method).

Reversing elements: `Reverse` (overloaded static method).

Copying: `Copy` (overloaded static method).

IDisposable

Some objects require explicit code to release resources such as open files, locks, operating system handles, and unmanaged objects. This is called disposal, and it is supported through the `IDisposable` interface.

```
public interface IDisposable{
    void Dispose(  );
}
```

IDisposable

- C#'s `using` statement provides a syntactic shortcut for calling `Dispose` on objects that implement `IDisposable`, using a `try / finally` block.

```
using (Font font = new Font("Courier", 12.0f)) {
   //code that uses the object font
}
```

The compiler converts this to:

```
Font font = new Font("Courier", 12.0f);
try{
  // ... Use font
}
finally{
  if (font != null) font.Dispose();
}
```

The `finally` block ensures that the `Dispose` method is called even when an exception is thrown, or the code exits the block early.

IDisposable

Multiple objects can be used with a `using` statement, but they must be declared inside the `using` statement, or nested:

```
using (Font f3 = new Font("Arial", 10.0f), f4 = new Font("Arial", 9.0f)){
    // Use fonts f3 and f4.
}

using (Font f3 = new Font("Arial", 10.0f))
 using (Font f4 = new Font("Arial", 9.0f)){
    // Use fonts f3 and f4.
}
```

IDisposable

The Framework follows a set of rules in its disposal logic.

These rules are not hard-wired to the framework or C# language.

The rules purpose is to define a consistent protocol to users.

Once disposed, an object cannot be reactivated, and calling its methods or properties may throw exceptions or give incorrect results.

Calling an object's `Dispose` method repeatedly causes no error.

If disposable object `x` contains disposable object `y`, the `Dispose` method of `x` automatically calls the `Dispose` method of `y` - unless instructed otherwise.