Collection Classes
and
Generics

AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

Agenda

- Collections
- Generic collections

Collections

- The .NET Framework provides specialized classes for data storage and retrieval. Eg.:
  - ArrayList
  - Queue
  - Stack
  - StringCollection
  - Hashtable
- An ArrayList is a sophisticated version of an array.
  The ArrayList class provides some features that are offered in most System.Collections classes but are not in the Array class. For example:
  - The capacity of an Array is fixed, whereas the capacity of an ArrayList is automatically expanded as required. If the value of the Capacity property is changed, the memory reallocation and copying of elements are automatically done.
  - ArrayList provide methods that add, insert, or remove a range of elements. In Array, you can get or set the value of only one element at a time.

Prefer the generic collection classes!

Use of Non-Generic Collections

- You can add any type you want (and you may mix them).
- But you must typecast back from System.Object when you want to access the stored object!

Prefer the generic collection classes!

```
public class Member …

public class MyClass
{
   ArrayList members = new ArrayList();
   …
   while ((str = s.ReadLine()) != null)
      members.Add(new Member(str));
   ..
   Member mem = (Member)members[i];
}
```

GENERIC COLLECTIONS

Generic Collections

- .NET now (since version 2.0) supports ***generics*** (much like C++ templates)
  - generics offer you efficient, type-safe collections
  - see the `System.Collections.Generic` namespace
- Example:
  - instead of an *ArrayList*, use a generic *List<>*

```
public class Member …

public class MyClass
{
   List<Member> members = new List<Member>();
   …
   while ((str = s.ReadLine()) != null)
      members.Add(new Member(str));
   ..
   Member mem = members[i];
}
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

When to Use Generic Collections

- Using generic collections is generally recommended, because you gain the immediate benefit of type safety without having to derive from a base collection type and implement type-specific members.

- In addition, generic collection types generally perform better than the corresponding nongeneric collection types when the collection elements are value types, because with generics there is no need to box the elements.

- Microsoft's recommendation:
  - Applications should use the generic collection classes in the System.Collections.Generic namespace, which provide greater type-safety and efficiency than their non-generic counterparts.

Some Generic Collection Classes

List<T>	Represents a strongly typed list of objects that can be accessed by index.
LinkedList<T>	Represents a doubly linked list that provides O(1) insertion and removal operations.
SortedList<TKey, TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
Queue<T>	Represents a first-in, first-out collection of objects.
Stack<T>	Represents a variable size last-in-first-out (LIFO) collection of instances of the same arbitrary type.
HashSet<T>	Represents a set of values.
Dictionary<TKey, TValue>	Represents a collection of keys and values.
SortedDictionary<TKey, TValue>	Represents a collection of key/value pairs that are sorted on the key. With O(log n) insertion and retrieval operations, making it a useful alternative to SortedList.
KeyedCollection<TKey, TItem>	Provides the abstract base class for a collection whose keys are embedded in the values.

Why all these collection types?

- Collections can vary, depending on how the elements are:
  - stored,
  - how they are sorted,
  - how searches are performed, and
  - how comparisons are made.
- The Queue class provide first-in-first-out lists.
- The Stack class provide last-in-first-out lists.
- The SortedList class provide sorted versions of the Hashtable class and the Dictionary generic class.
- The elements of a Dictionary are accessible only by the key of the element, but the elements of a SortedList or a KeyedCollection are accessible either by the key or by the index of the element.
- The indexes in all collections are zero-based, except Array, which allows arrays that are not zero-based.

Overriding Equals and GetHashCode

- Classes typically override ***Equals*** to work with .NET framework collection classes
  - overriding Equals requires overriding of *GetHashCode*

```
public class FamilyName
{
   .
   .
   .

   // Two FamilyNames are equal if names are equal:
   public override bool Equals(object obj)
   {
      if (obj != null && obj.GetType().Equals(this.GetType()))
        return this.Name.Equals(((FamilyName)obj).Name);
      else
        return false;
   }

   public override int GetHashCode()
   {
      // RULE: if 2 objects are Equals, must return same hash code:
      return this.Name.GetHashCode();
   }
```

```
// let collection do linear search for us:
index = namesCollection.IndexOf(new FamilyName(username, 0.0, 0));
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING