

Module 11

"Collections and Generics"



TEKNOLOGISK
INSTITUT

Agenda

- ▶ **Introducing Generics**
- ▶ Generic Collections
- ▶ Creating Generic Methods and Types
- ▶ Lab 11
- ▶ Discussion and Review

Classes of the `System.Collections` Namespace

- ▶ The classes in **`System.Collections`** namespace all operate on **object**

Class	Meaning
<code>ArrayList</code>	Dynamically sized array of objects
<code>HashTable</code>	Objects indexed by an object key
<code>SortedList</code>	Dictionary sorted by index keys of type object
<code>Queue</code>	First-in, first-out queue of objects
<code>Stack</code>	Last-in, first-out queue of objects

Stack

- ▶ **Stack** is a container ensuring last-in, first-out behavior

Member of Stack	Meaning
Push()	Adds an object to the top of the stack
Pop()	Removes the object at the top of the stack
Peek()	Returns the object at the top of the stack without removing it

```
Stack stack = new Stack();  
stack.Push( new Car( "Fred", 90 ) );  
stack.Push( new Car( "Mary", 100 ) );  
Car top = stack.Peek() as Car;  
Car removed = stack.Pop() as Car;  
foreach( Car c in stack )  
{  
    Console.WriteLine( c.PetName );  
}
```



The System.Collections. Specialized Namespace

- ▶ Most of the classes are rarely used

Specialized Class	Meaning
BitVector32	Manipulations of 32-bits and integers
ListDictionary	IDictionary as a singly-linked list
HybridDictionary	Best of ListDictionary and HashTable
NameValueCollection	Sorted map of strings to strings
StringCollection	A collection of strings
StringDictionary	Strongly typed HashTable with string keys
CollectionsUtil	Helpers for case-insensitive string collections
StringEnumerator	For iteration over a StringCollection object



Problems...!

- ▶ You can insert anything into a **Stack**!

```
Stack stack = new Stack();  
stack.Push( new Car( "Fred", 90 ) );  
stack.Push( new Car( "Mary", 100 ) );  
stack.Push( "Hello, World" );  
stack.Push( 87 );
```



```
Car top = stack.Peek() as Car;  
Car removed = stack.Pop() as Car;
```



```
foreach( Car c in stack )  
{  
    Console.WriteLine( c.PetName );  
}
```



- ▶ The problem is that type-safety is missing



We Could Create a CarStack!

- ▶ Yahooo! Oh joy...! Hooray! Or...?

```
class CarStack
{
    private Stack m_Stack;
    public CarStack() { m_Stack = new Stack(); }
    public Car Peek() { return m_Stack.Peek() as Car; }
    public void Push( Car c ) { m_Stack.Push( c ); }
    public Car Pop() { return m_Stack.Pop() as Car; }
}
```

- ▶ There is a **CollectionBase** class supplied to inherit from for type-safe collection
- ▶ What about a **PersonStack**? A **PointStack**? ...
- ▶ What about an **IntStack**?
 - Boxing and unboxing



Wouldn't It Be Nice If...

- ▶ ... we only needed to construct each type once?
- ▶ ... and it had no (un)boxing performance hit?

```
class Stack<T>
{
    public Stack { ... }
    public T Peek() { ... }
    public void Push( T t ) { ... }
    public T Pop() { ... }
    ...
}
```

- ▶ I.e. "generic" types!



The Interfaces of **System.Collections.Generic**

- ▶ The collection interfaces introduced earlier have generic counterparts
- ▶ `IEnumerable<T>`
- ▶ `IComparable<T>`
- ▶ `ICollection<T>`
- ▶ ...
- ▶ `IList<T>`
- ▶ `IDictionary<K,V>`
- ▶ `ISet<T>`
- ▶ `IEnumerator<T>`
- ▶ `IComparer<T>`
- ▶ ...



Agenda

- ▶ Introducing Generics
- ▶ **Generic Collections**
- ▶ Creating Generic Methods and Types
- ▶ Lab 11
- ▶ Discussion and Review

The Classes of the **System.Collections.Generic** Namespace

- ▶ Type-safe, reusable, and efficient collection classes

Class	Meaning
List<T>	Dynamically sized list of elements of type T
Dictionary<K,V>	Values of type V indexed by an element key of type K
SortedDictionary<K,V>	Values of type V indexed and sorted by keys of type K
Queue<T>	First-in, first-out queue of elements of type T
Stack<T>	Last-in, first-out queue of elements of type T
HashSet<T>	Set of elements of type T
SortedSet<T>	Sorted set of elements of type T

- ▶ These implement the generic interfaces on the previous slide
- ▶ Never use the non-generic collections!

Using Generic Types

- ▶ Substitute T with a concrete type whenever it is used

```
List<int> list = new List<int>();  
list.Add( 42 );  
list.Add( 87 );  
list.Add( 112 );  
  
foreach( int i in list )  
{  
    Console.WriteLine( i );  
}
```

```
List<string> list = new  
List<string>();  
list.Add( "Hello" );  
list.Add( "World" );  
  
foreach( string s in list )  
{  
    Console.WriteLine( s );  
}
```



Queue<T>

- ▶ **Queue<T>** is a type-safe container ensuring first-in, first-out behavior

Member of Queue<T>	Meaning
Dequeue()	Removes and returns the element at beginning of queue
Enqueue()	Adds an element to the end of queue
Peek()	Returns the element at the beginning

```
Queue<Car> queue = new Queue<Car>();
queue.Enqueue( new Car( "Fred", 90 ) );
queue.Enqueue( new Car( "Mary", 100 ) );
Car first = queue.Peek();
Car removed = queue.Dequeue();
foreach( Car c in queue )
{
    Console.WriteLine( c.PetName );
}
```



Dictionary<K,V>

- ▶ **Dictionary<K,V>** is a container of values of type V indexed by an element key of type K

Member of Dictionary<K,V>	Meaning
Add()	Adds an key-value pair to the dictionary
Remove()	Removes the element with the specified key

- ▶ Iterate dictionaries by using **KeyValuePair<K,V>**

```
Dictionary<int, string> dict = new Dictionary<int, string>();  
dict.Add( 19, "Kim Aabech" );  
dict.Add( 7, "Stephan Petersen" );  
Console.WriteLine( "Number 11 is {0}", dict[ 11 ] );  
  
foreach( KeyValuePair<int, string> kv in dict )  
{  
    Console.WriteLine( "Player {0} is {1}", kv.Key, kv.Value );  
}
```

HashSet<T>

- ▶ **HashSet<T>** is a set of values of type **T**

Member of HashSet<T>	Meaning
Add()	Adds an element to the set
Remove()	Removes the specified element in the set

- ▶ There is also a **SortedSet<T>**

- Needs **IComparer<T>**
- See Lab 11.3

```
HashSet<int> set = new HashSet<int>();  
set.Add( 42 );  
set.Add( 87 );  
set.Add( 42 );  
set.Remove( 42 );  
  
foreach( int i in set )  
{  
    Console.WriteLine( i );  
}
```



Collection Initializers

- ▶ Collections can be conveniently initialized via *collection initializer syntax*

```
List<int> list = new List<int> { 42, 87, 112 };
```

```
List<string> list = new List<string> { "Hello", "World" };
```

```
SortedSet<int> set = new SortedSet<int> { 87, 42, 112, 176 };
```

- ▶ Note: Only works for those collection classes with an **Add()** method, i.e. not
 - Stack<T>
 - Queue<T>
 - LinkedList<T>
 - ...



Index Initializers

- ▶ New in C# 6.0: Index initializers are now provided for collection initializers

```
var lineUp = new Dictionary<int, string>
{
    [19] = "Kim Aabech",
    [11] = "Jesper Lange",
    [7]  = "Stephan Petersen" }
};
```



Agenda

- ▶ Introducing Generics
- ▶ Generic Collections
- ▶ **Creating Generic Methods and Types**
- ▶ Lab 11
- ▶ Discussion and Review

Defining Generic Methods

- ▶ You can define methods operating on generic types

```
void Swap<T>( ref T a, ref T b )  
{  
    T temp = a;  
    a = b;  
    b = temp;  
}  
  
int i = 42;  
int j = 87;  
Swap<int>( ref i, ref j );  
  
string s = "Hello";  
string t = "World";  
Swap<string>( ref s, ref t );
```

- ▶ Such methods cannot be defined inside generic classes or structs!
- ▶ T is "free" to match any type
 - Use **typeof(T)** to retrieve instantiated type



Inference of Method Type Parameters

- ▶ The C# compiler will try to infer the types when omitted
- ▶ In the case of **Swap<T>** it is successful

```
static void DisplayBaseClass<T>()  
{  
    Console.WriteLine( "Base class of {0} is {1}",  
        typeof( T ),  
        typeof( T ).BaseType );  
}
```

```
DisplayBaseClass<string>(); ✓  
DisplayBaseClass<int>(); ✓  
DisplayBaseClass(); ✗
```

- ▶ Occasionally, the type must be explicit supplied



Creating Generic Structures and Classes

- ▶ You can easily create your own generic types

```
public struct Point<T>
{
    private T x;
    private T y;
    public Point( T x, T y )
    {
        this.x = x;
        this.y = y;
    }
    public T X { get { return x; } }
    public T Y { get { return y; } }
}
```

```
Point<int> pt1 =
    new Point<int>( 42, 87 );
Console.WriteLine( pt1 );

Point<double> pt2 =
    new Point<double>( 11.2, 8.7 );
Console.WriteLine( pt2 );
```



The default Keyword for Generics

- ▶ The default value for the instantiated type can be retrieved via `default(T)`

```
public struct Point<T>
{
    ...
    public void Reset()
    {
        x = default( T );
        y = default( T );
    }
}
```

```
Point<int> pt1 =
    new Point<int>( 42, 87 );
pt1.Reset();
Console.WriteLine( pt1 );

Point<bool> pt2 =
    new Point<bool>( true, false );
pt2.Reset();
Console.WriteLine( pt2 );

Point<string> pt3 =
    new Point<string>( "Hello","World" );
pt3.Reset();
Console.WriteLine( pt3 );
```



Constraining Generic Types with the `where` Keyword

Generic Constraint	Meaning
<code>where T : struct</code>	T must ultimately derive from <code>System.ValueType</code>
<code>where T : class</code>	T must be a reference type
<code>where T : new()</code>	T must have a default constructor
<code>where T : BaseClass</code>	T must derive from the class specified by <i>BaseClass</i>
<code>where T : Interface</code>	T must implement the interface specified by <i>Interface</i>

- ▶ Multiple constraints can be separated by commas
- ▶ There can be only one *BaseClass*, but many *Interfaces*
- ▶ `new()` must be last in constraint sequence

Examples of Constraining

- ▶ Constraints can be applied to both generic classes and generic methods

```
void Swap<T>( ref T a, ref T b ) where T : struct  
{  
    ...  
}
```

```
static void SetNew<T>( ref T a ) where T : new()  
{  
    a = new T();  
}
```

```
class LinkedList<K, V> where K : struct, IComparable<K>  
                     where V: Car, new()  
{  
    ...  
}
```



Generic Types as Base Classes

- ▶ Deriving from instantiated generic classes is exactly as usual

```
class Garage : List<Car>
{
    // ...
}
```

- ▶ When deriving from “pure” generic classes, all constraints must be met

```
public class MyOtherList<T> : MyList<T> where T : new()
{
    public override void PrintList( T data ) { ... }
}
```

```
public abstract class MyList<T> where T : new()
{
    private List<T> list = new List<T>();
    public abstract void PrintList( T data )
}
```

Quiz: Generics – Right or Wrong?

```
List<int> list = new List { 42, 87, 112 };
```



```
Queue<bool> list = new Queue<bool> { false, true, true };
```



```
List<int> list = new List<int> { 42, 87, 112 };  
list.Add( 176 );
```



```
class MyClass<T> where T : class  
{  
    ...  
}
```



```
MyClass<int> mci =  
    new MyClass<int>();
```



```
MyClass<string> mcs =  
    new MyClass<string>();
```

```
class MyClass2<T>  
    where T : new()  
{  
    ...  
}
```



```
MyClass2<int> mci =  
    new MyClass2<int>();
```



```
MyClass2<string> mcs =  
    new MyClass2<string>();
```



Lab 11: Using Collections



Discussion and Review

- ▶ Introducing Generics
- ▶ Generic Collections
- ▶ Creating Generic Methods and Types



WINCUBATE

Jesper Gulmann Henriksen

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : jgh@wincubate.net

WWW : <http://www.wincubate.net>

Hasselvangel 243

8355 Solbjerg

Denmark