TEKNOLOGISK
INSTITUT

Module 11

Collections and Generics

TEKNOLOGISK INSTITUT

Classes of the `System.Collections` Namespace

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

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

TEKNOLOGISK INSTITUT

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("Ford"));
stack.Push(new Car("Fiat"));
Car top = stack.Peek() as Car;
Car removed = stack.Pop() as Car;
foreach (Car c in stack)
{
    Console.WriteLine(c.Model);
}
```

The System.Collections.Specialized Namespace

TEKNOLOGISK INSTITUT

- 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

TEKNOLOGISK
INSTITUT

Annoying Problems

- You can insert anything into a `Stack`!

```
Stack stack = new Stack();
stack.Push(new Car("Ford"));
stack.Push(new Car("Fiat"));
stack.Push(1);
stack.Push("name");
```

- The problem is that type-safety is missing

TEKNOLOGISK INSTITUT

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

TEKNOLOGISK
INSTITUT

Wouldn't It Be Nice If…

- … we only needed to construct once class to hold many types?

```
class MyStack<T>
{
    private ArrayList lst = new ArrayList();
    public void Push(T item) {
        lst.Add(item);
    }
    public T Pop() {
        if (lst.Count == 0)
            throw new ApplicationException("Error...");
        T item = (T)lst[lst.Count-1];
        lst.Remove(item);
        return item;
    }
}
```

- I.e. "generic" types!

TEKNOLOGISK
INSTITUT

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>`
- `...`

TEKNOLOGISK INSTITUT

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

TEKNOLOGISK
INSTITUT

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>

TEKNOLOGISK INSTITUT

- 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("Ford"));
queue.Enqueue(new Car("Fiat"));
Car first = queue.Peek();
Car removed = queue.Dequeue();
foreach (Car c in queue)
{
    Console.WriteLine(c.Model);
}
```

Dictionary<K,V>

TEKNOLOGISK INSTITUT

- 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, "Michell Cronberg");
dict.Add(7, "Mathias Cronberg");
Console.WriteLine("Number 7 is {0}", dict[7]);

foreach (KeyValuePair<int, string> kv in dict)
{
    Console.WriteLine("Player {0} is {1}", kv.Key, kv.Value);
}
```

TEKNOLOGISK INSTITUT

HashSet<T>

- HashSet<T> is a set of values of type T with unique values

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);
}
```

TEKNOLOGISK INSTITUT

Collection Initializers

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

```
List<int> list1 = new List<int> { 42, 87, 112 };
List<string> list2 = 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>`
  - …

TEKNOLOGISK
INSTITUT

Index Initializers

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

```
var lineUp = new Dictionary<int, string>
{
    [19] = "Michell Cronberg",
    [11] = "Mikkel Cronberg",
    [7] = "Mathias Cronberg"
};
```

TEKNOLOGISK INSTITUT

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

TEKNOLOGISK INSTITUT

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);
```

TEKNOLOGISK INSTITUT

The `default` Keyword for Generics

- The default value for the instantiated type can be retrieved via `default( T )`
- This is the “usual” default zero whitewash value
  - `null` for reference 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; } }

    public void Reset()
    {
        x = default(T);
        y = default(T);
    }
}
```

TEKNOLOGISK INSTITUT

Constraining Generic Types with the `where` Keyword

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

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

TEKNOLOGISK
INSTITUT

Examples of Constraining

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

```
public struct Point<T> where T : struct
```

```
public struct Point<T> where T : new()
```

```
public struct Point<T> where T : IComparable<T>
```

TEKNOLOGISK
INSTITUT

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