

Advanced C# Constructs

Hans-Wolfgang Loidl
<H.W.Loidl@hw.ac.uk>

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh



Semester 1 2021/22



Advanced C# Features

We will cover the following *advanced* C# features:

- Collections
- Indexers
- Generics
- Exceptions
- Delegates



Collections

- **Collections** provide a general framework for putting objects of the same type together.
- Examples are arrays, or pre-defined classes Stack, List, Queue, Dictionary.
- Constructs are available to iterate over all elements of a collection.
- A user-defined class can be made a collection by implementing certain interfaces such as `IEnumerable` or `ICollection`.

⁰<http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/container.cs>



Indexers

- Indexers make it possible to *treat a class as if it were an array*.
- An indexer is a special kind of property.
- It defines get and set methods, which are parametrised by an index argument.
- Read and write uses of the class in array notation are then translated into calls to these get and set methods.



Indexer Example

```
1 public class ListBox {
2     private string[] strings;
3     private int ctr = 0;
4
5     public ListBox (params string[] initStrs) {
6         strings = new String[256];
7         foreach (string s in initStrs) {
8             strings[ctr++] = s;
9         }
10    }
11    public void Add (string s) {
12        if (ctr >= strings.Length) {
13            // ToDo: handle overflow
14        } else {
15            strings[ctr++] = s;
16        } }
}
```



Indexer Example (cont'd)

```
1 // indexer
2 public string this[int index] {
3     get {
4         if (index < 0 || index >= strings.Length) {
5             // handle error case
6         } else {
7             return strings[index];
8         }
9     }
10    set {
11        if (index >= ctr) {
12            // handle error case
13        } else {
14            strings[index] = value;
15        }
16    } }
17 public int GetNumEntries() { return ctr; } }
```



Using the Indexer

We can now treat the ListBox class like an array of strings, eg.

```
1 for (int i = 0; i < lbt.GetNumEntries(); i++) {
2     Console.WriteLine("lbt [{0}]: {1}", i, lbt[i]);
3 }
```

Object `lbt` is treated like an array



Generics

- So far we always had to specify the concrete element type of a collection.
- **Generics** offer the possibility to *leave the type of an element undefined*.
- To this end a **type-variable** is specified.
- An example is the pre-defined List class:
`public class List<T> { ... }`
- T is a type-variable, which stands for the element type of the list.
- The methods in the class work over any basis type T, i.e. they are **polymorphic**.
- When using the list you specify the element type, eg.
`List<int> myList = new List<int>();`



Generic Classes

- Other pre-defined generic classes are:

- ▶ `List<T>`
- ▶ `Stack<T>`
- ▶ `Queue<T>`
- ▶ `Dictionary<K,V>`

- It is possible to restrict the type variable:

```
1 public class Node<T> where T: IComparable
```

- It can only be instantiated for a type that implements the `IComparable` interface.



Generic Interfaces

- Several generic interfaces can be implemented to make iteration over collections simpler.
- With an implementation of the `IEnumerable<T>` interface it is possible to use a `foreach` loop on the collection.



Generic Interface Example

```
1 public class ListBox : IEnumerable<String> {
2     private string[] strings;
3     private int ctr = 0;
4
5     // enumerator
6     public IEnumerator<string> GetEnumerator() {
7         foreach (string s in strings) {
8             yield return s;
9         }
10    // required to fulfill IEnumerable
11    System.Collections.IEnumerator System.Collections.
12        IEnumerable.GetEnumerator() {
13        throw new NotImplementedException();
14    }
15 }
```



Using the Enumerator

Now we can use a `foreach` loop on a `ListBox lbt`:

Object `lbt` is used as a container

```
1 foreach (string s in lbt) {
2     Console.WriteLine("Value: {0}", s);
3 }
```



Exceptions

- **Exceptions** provide language constructs to deal with foreseen error cases in the code.
- For example when accessing an array an exception should be thrown if the index is out of range.
- An exception is an object that contains information about the error.
- An exception handler then deals with the error case.
- The handler can be defined in the method itself, or in any of the calling methods.
- **No exception should be unhandled.**



Exceptions Example

Checking for array bounds in `ListBox`:

```
1 public string this[int index] {
2     get {
3         if (index < 0 || index >= strings.Length) {
4             throw new OutOfBoundsException();
5         } else {
6             return strings[index];
7         }
8     }
}
```



Exceptions Example

A concrete exception class must inherit from the `Exception` class:

```
1 public class OutOfBoundsException : System.Exception {
2     public OutOfBoundsException(string msg) {
3         base(msg);
4     }
5 }
```

An exception is caught by attaching an exception handler to the code, eg.

```
1 try {
2     x = lbt[i]; // dangerous code
3 } catch (OutOfBoundsException e) {
4     Console.WriteLine("Index out of bounds; msg: {0}",
5         e.Message);
6 }
```



Delegates

- **Delegates** are the objected-oriented technique for defining *higher-order functions*, i.e. *functions that can take other functions as arguments*.
- A delegate refers to a *method*.
- To declare a delegate the type of a method is specified, e.g.

```
1 public delegate int FindResult(object o1, object
    o2);
```

- A concrete method can be instantiated for the delegate if it matches its result and parameter types.
- Anonymous methods or lambda abstractions can also be instantiated for a delegate.



Delegates Example

We design a class for storing and playing media, eg.

```
1 public class MediaStorage {
2     public delegate int PlayMedia();
3     public void ReportResult(PlayMedia playerDelegate) {
4         if (playerDelegate() == 0) {
5             Console.WriteLine("Media played successfully");
6         } else {
7             Console.WriteLine("Error in playing media.");
8         }
9     }
```



Delegates Discussion

- In the `ReportResult` method the `playerDelegate` is called, which refers to a concrete method without fixing it in the code.
- At compile time only the type of the delegate needs to be known.
- At run-time the delegate must be instantiated with one concrete method.
- This is the same abstraction step as it is done for data when using an (abstract) class as base type, and instantiating it with a sub-class at run-time.



Delegates Example (cont'd)

Now the `ReportResult` method can be applied for different kinds of players, eg.

```
1 public class AudioPlayer {
2     private int audioPlayerStatus;
3     public int PlayAudioFile() {
4         Console.WriteLine("Playing audio file");
5         audioPlayerStatus = 0;
6         return audioPlayerStatus;
7     }
8 }
```



Using Delegates

To use the delegate we instantiate it to a concrete player.

```
1 MediaStorage ms = new MediaStorage();
2 AudioPlayer aPlayer = new AudioPlayer();
3 VideoPlayer vPlayer = new VideoPlayer();
4 // instantiate the delegate
5 MediaStorage.PlayMedia aDelegate =
6     new MediaStorage.PlayMedia(aPlayer.PlayAudioFile);
7 MediaStorage.PlayMedia vDelegate =
8     new MediaStorage.PlayMedia(vPlayer.PlayVideoFile);
9 // provide instances to the method using the delegate
10 ms.ReportResult(aDelegate);
11 ms.ReportResult(vDelegate);
```

⁰<http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/delegates1.cs>



Delegates and GUIs

- One frequent application of delegates is in GUI programming, when handling *events*.
- An event is for example a mouse click.
- In the GUI code a delegate is used to refer to the method that will handle the mouse click.
- In the application code an instance for the delegate is provided to perform the actual work.
- This achieves a separation of concerns between the GUI and the application.



Another Delegate Example

We want to implement a way to apply a function twice.

```
1 class TestClass {
2     public static int Double(int val) {
3         return val*2;
4     }
5
6     public static void Main(string []args) {
7         ...
8         Console.WriteLine("Applying double once on {0}
9             gives {1}",
10            x, TestClass.Double(x));
11        Console.WriteLine("Applying double twice on {0}
12            gives {1}",
13            x, Twice.twice(Double, x));
14    }
15 }
```



How can we implement a class `Twice` with a method `twice`?

```
1 // simple higher-order example, using delegates
2 // this class takes an int -> int function and applies
3 // it twice
4 public class Twice {
5     // delegate, specifying the type of the function
6     // argument
7     public delegate int Worker(int i);
8
9     // the higher-order function twice applies the
10    // worker function twice
11    public static int twice(Worker worker, int x) {
12        return worker(worker(x));
13    }
14 }
```



Anonymous Methods

- When instantiating a delegate with a very short method it is cumbersome to define a method only to provide an instance to the delegate.
- In these cases *anonymous methods* can be used, e.g. for increasing its argument:
`delegate(ref int counter) { counter++; }`
- This form can be used instead of the name of a concrete method.



Lambda Expressions

- *Lambda expressions* are a generalisation of anonymous methods.
- They behave like (unnamed) functions in a functional language, e.g. double a value: `(int i) => { 2*i };`
- or just: `i => 2*i`
- Whereas anonymous methods can only be used in the context of delegates, lambda expressions can be used wherever a method is expected.
- This is used for example in the Language Integrated Query (LINQ) engine of C# for accessing databases.



Summary

- These advanced features provide powerful tools of abstraction, to generate re-usable code.
- They enable structured control over *collections*, adapting language features such as foreach loops to user-defined classes.
- They enable the *abstraction over types, through generics*.
- They enable the *abstraction over methods, through delegates*, in a way similar to abstracting data through class hierarchies.
- Be aware of these language concepts when you design your application: their use can save a lot of code and programming effort.



Exercises

- Modify the *binary search tree* example, using generics over the element type. Implement an indexer, for direct access to the i-th element, and an enumerator, to enable foreach loops.
- Use *delegates* to define a method that applies a method to every element of a tree.

