Advanced C# Constructs

Hans-Wolfgang Loidl

<hwloidl@macs.hw.ac.uk>

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

HERIOT WATT UNIVERSITY

Semester 1 2017/18

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

```
public class ListBox {
  private string[] strings;
  private int ctr = 0;

  public ListBox (params string[] initStrs) {
    strings = new String[256];
    foreach (string s in initStrs) {
      strings[ctr++] = s;
    }
  }
  public void Add (string s) {
   if (ctr >= strings.Length) {
        // ToDo: handle overflow
   } else {
        strings[ctr++] = s;
  }  }
```

Indexer Example (cont'd)

```
// indexer
public string this[int index] {
  get {
    if (index<0 || index>=strings.Length) {
      // handle error case
    } else {
      return strings[index];
    }
  }
  set {
    if (index >= ctr) {
      // handle error case
    }  else {
      strings[index] = value;
    }
  } }
public int GetNumEntries() { return ctr; }  }
```

Using the Indexer

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

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

[0] http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/indexers1.cs

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:

```
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

```
public class ListBox : IEnumerable<String> {
  private string[] strings;
  private int ctr = 0;

  // enumerator
  public IEnumerator<string> GetEnumerator() {
   foreach (string s in strings) {
     yield return s;
  }
  // required to fulfill IEnumerable
  System.Collections.IEnumerator System.Collections.
      IEnumerable.GetEnumerator(){
    throw new NotImplementedException();
  }
```

Using the Enumerator

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

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

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`:

```
public string this[int index] {
  get {
    if (index <0 || index>=strings.Length)   {
      throw new OutOfBoundsException();
    } else {
      return strings[index];
    }
  }
```

Exceptions Example

A concrete exception class must inherit from the Exception class:

```
public class OutOfBoundsException : System.Exception {
    public OutOfBoundsException(string msg) {
      base(msg);
    }
}
```

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

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

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.**

```
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.

```
public class MediaStorage {
  public delegate int PlayMedia();
  public void ReportResult(PlayMedia playerDelegate) {
    if (playerDelegate() == 0) {
      Console.WriteLine("Media played successfully");
    } else {
      Console.WriteLine("Error in playing media.");
    }
}
```

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.

```
public class AudioPlayer {
   private int audioPlayerStatus;
   public int PlayAudioFile() {
      Console.WriteLine("Playing audio file");
      audioPlayerStatus = 0;
      return audioPlayerStatus;
   }
}
```

Using Delegates

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

```
MediaStorage ms = new MediaStorage();
AudioPlayer aPlayer = new AudioPlayer();
VideoPlayer vPlayer = new VideoPlayer();
// instantiate the delegate
MediaStorage.PlayMedia aDelegate =
  new MediaStorage.PlayMedia(aPlayer.PlayAudioFile);
MediaStorage.PlayMedia vDelegate =
  new MediaStorage.PlayMedia(vPlayer.PlayVideoFile);
// provide instances to the method using the delegate
ms.ReportResult(aDelegate);
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.

```
class TestClass {
  public static int Double(int val) {
    return val*2;
  }

  public static void Main(string []args) {
    ...
    Console.WriteLine("Applying double once on {0} gives {1}",
      x, TestClass.Double(x));
    Console.WriteLine("Applying double twice on {0} gives {1}",
      x, Twice.twice(Double, x));
   }
  }
}
```

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

```
// simple higher-order example, using delegates
// this class takes an int -> int function and applies
    it twice
public class Twice {
  // delegate, specifying the type of the function
     argument
  public delegate int Worker(int i);

  // the higher-order function twice applies the
  // worker function twice
  public static int twice(Worker worker, int x) {
    return worker(worker(x));
  }
}
```

[0] http://www.macs.hw.ac.uk/~hwloidl/Courses/F21SC/Samples/delegates2.cs

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.**