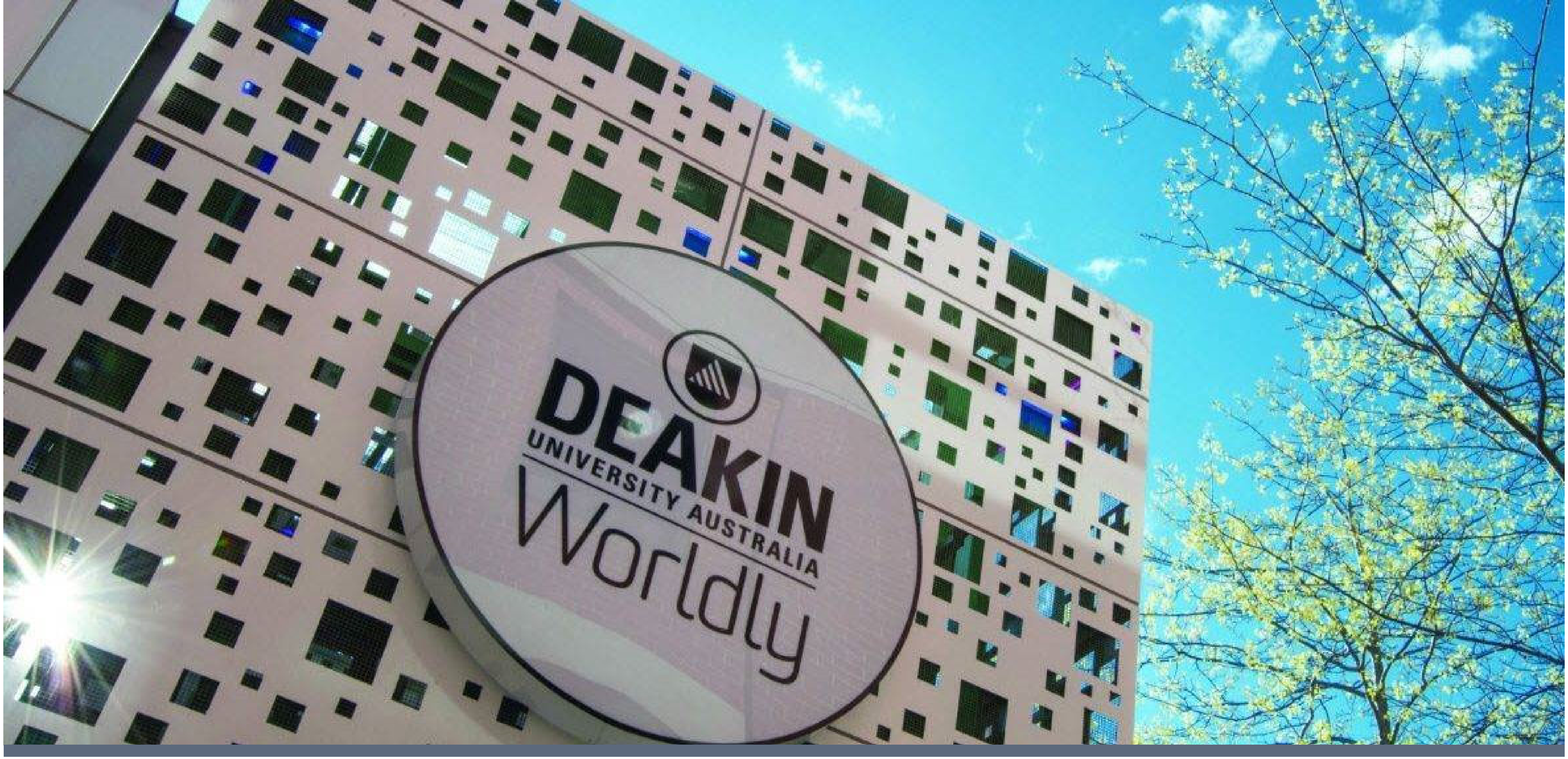

Lecture 09. Extension Methods. Anonymous Types and Methods. Lambda Expressions

SIT232 Object-Oriented Development

Extension Methods: The Idea

- Once a type is defined and compiled into an assembly its definition is, more or less, final.
  - → The only way to update, remove or add new members is to recode and recompile the code.
- **Extension methods** allow existing compiled types to gain new functionality
  - – without recompilation, and
  - – without touching the original assembly
- Extension methods allow existing classes to be extended without relying on inheritance or having to change the class's source code.

Defining Extension Methods

- An extension method must be defined in a top-level static class.
- An extension method **must be defined as static**.
- Use **this** keyword before its first argument to specify the class to be extended.
- Extension methods are "attached" to the extended class.

  → Can also be called statically through the defining static class.
- An extension method with the same name and signature as an instance method will not be called.
- Extension methods cannot be used to override existing methods.
- The concept of extension methods cannot be applied to fields and properties.
- Overuse of extension methods is not a good style of programming.

Extension Methods: Example

```
public static class Extensions
{
   public static int CharacterCount(this string str)
   {
       char[] delimiters = new char[] { ' ', '.', '?' };
       return str.Split(delimiters).Length;
   }
}


static void Main()
{
   string s = "Hello Extension Methods";
   int i = s.CharacterCount();
   Console.WriteLine(i);
}
```

Extension Methods: Example

```
public static class Extensions
{
   public static void IncreaseWith(this List<int> list, int amount)
   {
       for (int i = 0; i < list.Count; i++) list[i] += amount;
   }
}

static void Main()
{
   List<int> ints = new List<int> { 1, 2, 3, 4, 5 };
   ints.IncreaseWith(5);  // 6, 7, 8, 9, 10
}
```

Anonymous Types

- Encapsulate a set of read-only properties and their value into a single object.
- There is no need to explicitly define a type first.
- To define an anonymous type, use of the new var keyword in conjunction with the object initialization syntax.
- Anonymous types are reference types directly derived from System.Object.

Example:

```
var point = new { X = 3, Y = 5 };
```

Anonymous Types: Example

```
// Use an anonymous type representing a car
var myCar = new { Color = "Red", Brand = "BMW", Speed = 180 };
Console.WriteLine("My car is a {0} {1}.", myCar.Color, myCar.Brand);
```

- At compile time, the C# compiler will auto-generate an uniquely named class.
- The class name is not visible from C#

  → using implicit typing (var keyword) is mandatory

Arrays of Anonymous Types

You can define and use arrays of anonymous types through the following syntax:

```
var arr = new[]
{
   new { X = 3, Y = 5 },
   new { X = 1, Y = 2 },
   new { X = 0, Y = 7 }
};

foreach (var item in arr)
{
   Console.WriteLine("({0}, {1})", item.X, item.Y);
}
```

Anonymous Methods

- Anonymous methods are **methods without name**
- They can take parameters and return values
- They are declared through the delegate keyword

```
// declare delegate data type
public delegate void AnonymousDelegate(string str);

// create a delegate object of type AnonymousDelegate
AnonymousDelegate anonymous = delegate(string str)
{
   // define the implementation
   Console.WriteLine(str);
};

// invoke the delegate
anonymous(“Hello Anonymous Types”);
```

Predefined Delegates

- A generic predefined **void** (procedure) delegate with parameters of types T1, T2 and T3

  Action < T1, T2, T3 >

- A generic predefined delegate with **return value** of type TResult

  Func < T1, T2, Tresult >

- Both have quite a lot of overloads

Examples:

```
Func<string, int> predefinedIntParse = int.Parse;
int number = predefinedIntParse("50");

Action<object> predefinedAction = Console.WriteLine;
predefinedAction(1000);
```

Lambda Expressions

- A lambda expression is an anonymous function containing expressions and statements
- Lambda expressions
  - Use of “maps to“ lambda operator ‘=>’
  - The left side specifies the input parameters
  - The right side holds the expression or statement
- Lambda expressions are a clearer way to achieve the same thing as an anonymous delegate. Its form

  ```
  (type1 arg1, type2 arg2, ...)  =>  expression
  ```

  is equivalent to

  ```
  delegate (type1 arg1, type2 arg2, ...) {   return expression;   }
  ```

Lambda Expressions: The Art

Write a function that will filter an integer value out of a list.

```
public static List<int> FilterOut(List<int> list, int target)
{
   List<int> myList = new List<int>();
   foreach(int myValue in list) {
       if( myValue != target ) {
          myList.Add(myValue);
       }
   }
   return myList;
}

public void SomeMethod(List<int> values) {
   // gives us a list without the sixes.
    List<int> myNewList = FilterOut(values, 6);
}
```

It’s great, but the customer wants to be able to filter out 3, … many values.

What would you do?

Lambda Expressions: The Art

Now, you do not have to change your filter code at all, no matter what the customer asks for.

```
public static List<int> FilterOut (
   List<int> list, Func<int, bool> filterCriteria )
{
   List<int> myList = new List<int>();
   foreach (int myValue in list)
   {
       if ( filterCriteria(myValue) ) continue;
       myList.Add(myValue);
   }
   return myList;
}

public void SomeMethod(List<int> values) {
   // gives us a list without the sixes.
   List<int> myNewList = FilterOut(values, x => x == 6);
   // gives us a list without the sixes and sevens.
   List<int> myNewList = FilterOut(values, x => x == 6 || x == 7 );
}
```

Delegates Holding Lambda Functions

- Lambda functions can be stored in variables of type **delegate**
- Delegates are typed references to functions
- Standard function delegates in .NET:

  Func<TResult>, Func<T,TResult>, Func<T1,T2,TResult>, …

Example:

```
Func<int, bool> intFunc = (x) => x < 10;
if ( intFunc(5) )  Console.WriteLine("5 < 10");
```

Lambda Functions: Application

Lambda functions are usually used with collection extension methods like

FindAll() and RemoveAll()

```
List<int> list = new List<int>() { 1, 2, 3, 4 };
List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);
foreach (var num in evenNumbers)
{
   Console.Write("{0} ", num);
}
Console.WriteLine(); // 2 4

list.RemoveAll(x => x > 3); // 1 2 3
```

Lambda Functions: Application

Sorting with a Lambda Expression

```
var pets = new Pet[]
{
   new Pet { Name="Sharo", Age=8 },
   new Pet { Name="Rex", Age=4 },
   new Pet { Name="Strela", Age=1 },
   new Pet { Name="Bora", Age=3 }
};

var sortedPets = pets.OrderBy(pet => pet.Age);
foreach (Pet pet in sortedPets)
{
   Console.WriteLine("{0} -> {1}", pet.Name, pet.Age);
}
```

Lambda Code Expressions

```
List<int> list = new List<int>() { 20, 1, 4, 8, 9, 44 };

// Process each argument with code statements
List<int> evenNumbers = list.FindAll((x) =>
{
   Console.WriteLine("value of x is: {0}", x);
   return (x % 2) == 0;
});

Console.WriteLine("Here are your even numbers:");
foreach (int even in evenNumbers) Console.Write("{0}\t", even);
```