SIT232 - OBJECT ORIENTED DEVELOPMENT

Session 4. Relationships

Outline

- Session 04. Relationships
  - Objectives
  - Object and class relationships
  - Relationships in C#
  - Delegates
  - Indexers
  - Namespaces
  - Report Writing

SESSION 4. RELATIONSHIPS

Objectives

- At the end of this session you should:
  - Be familiar **object and class relationships** and be able to exploit them to develop object-oriented applications;
  - Understand how both **value type and reference type objects** are created and destroyed and how the *garbage collector manages memory* on your behalf;
  - Recognise the **different types of constructors** and be able to exploit them to *initialise objects correctly*;
  - Be able to **implement copying of objects and correctly** *solve the shallow copy problem*;
  - Understand the **role and problems with destructors**, the *solution Dispose methods provide*, and when to use them;
  - Be able to apply **indexers and ToString methods** in your classes; and
  - Be able to prepare **properly formatted reports**.

Object and class relationships

- *Object relationships and class relationships are different but closely related*
  - *Object relationships are instances of class relationships*
- One object relationship
  - **Link**
- Four class relationships
  - **Association**
  - **Aggregation**
  - **Composition**
  - **Inheritance**
    - Covered in Session 5, does not result in object relationship

Object and class relationships

- Object relationship:
  - *Link is usually uni-directional*, i.e., one *object can invoke the services/methods of another object, but not vice-versa*
    - *Does not prevent data travelling in both directions*, e.g., through output parameters and return values
  - *Direction of the link* is often referred to navigability

Object and class relationships

- Class relationships:
  - **Association**
    - A general relationship between two classes
    - Usually bi-directional, i.e., peer-to-peer
  - **Aggregation – tighter form of association**
    - Whole/part hierarchy, where *one object is "part of" another*
      - Part object can be part of more than one whole
    - Navigability usually limited where the *whole object can navigate to the part*
    - Example: A Book is part of a Order
    - *Loosely coupled lifetimes*
  - **Composition – tighter form of aggregation**
    - Part object can only ever be *part of one whole*
    - *Tightly coupled lifetimes (usually linked)*

Relationships in C#

- Consider two objects
  - **Client** object invokes methods of **supplier** object
- To implement a ***relationship there are two tasks***:
  - Declare a variable in client to reference the supplier

```
private Student _SingleStudent;
private Student [] _StudentArray = new Student[size];
private List<Student> _Enrolment = new List<Student>();
```

  - Store ***memory address of supplier in client's reference***
    - Application *specific but one object must obtain or be provided with the address of the other object*

```
public void EnrolStudent(Student student)
{
    _Enrolment.Add(student);
}
```

Relationships in C#

- The above information *works for association and aggregation*
  - Only *issue is whether the relationship needs to be uni-directional or bi-directional*
- **Composition** however is more difficult
  - Recall that lifetimes are tightly coupled
  - This is only possible in C# for value types, which are usually *treated as attributes not relationships*
  - Can be *approximated by never revealing an objects memory address* but relies on all future programmers maintaining this **encapsulation** (not reliable)

Delegates

- **Delegates** are similar to *object references*, except they reference a method instead
  - Most commonly used for *event handlers for Windows GUI components*
- **Elements** to be considered:
  - How to *create a delegate data type*
  - How to *create a delegate variable*
  - How to *assign the method reference* to the **delegate variable**
  - How to *invoke a method* using the **delegate variable**

Delegates

- How to **create a delegate data type**

[access_modifier2]`delegate` *return_type delegate_type_name*`(`*parameter_list*`);`

  - Must match the methods to be referenced

    - *One delegate type per method signature and return type*

- How to **create a delegate variable**

[access_modifier]delegate_type_name variable_name;

Delegates

- How to **assign the method reference to the delegate variable**

```
variable_name = method_name;
```

- How to **invoke a method using the delegate variable**
  - *No different to a method call, just use the delegate variable name*

Delegates

```
/*************************************************************
** File: DelegateDemo.cs
** Author/s: Justin Rough
** Description:
**     A simple program demonstrating how to create a delegate
** type, create a delegate variable, assign a method reference
** to the delegate variable and invoke the method referenced
** by the delegate.
*************************************************************/
using System;

namespace DelegateDemo
{
    class DelegateDemo
    {
        private delegate string ValueTester(int value);

        static string IsZero(int value)
        {
            if (value == 0)
                return string.Format("{0} is zero", value);
            else
                return string.Format("{0} is NOT zero", value);
        }

        static void Main(string[] args)
        {
            ValueTester delegateVariable = IsZero;

            Console.Write("Enter an integer: ");
            int number = Convert.ToInt32(Console.ReadLine());

            Console.WriteLine(delegateVariable(number));
        }
    }
}
```

File: DelegateDemo.cs

```
static string GetOtherInput(int data)
{
    do{ Console.Write("Enter an integer: ");
    }while(!  int.TryParse(Console.ReadLine(), out data));
    return data.ToString();
}
static void Main(string[] args)
{
    ValueTester delegateVariable = IsZero;
    Console.Write("Enter an integer: ");
    int number = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine(delegateVariable(number));
    delegateVariable = GetOtherInput;
    Console.WriteLine(delegateVariable(number));
    Console.WriteLine("\nAfter calling GetOtherInput() number = {0}\n\n", number);
}
```

```
static string Day(int d)
{
    string result = "";
    switch(d)
    {
        case 1: case 21: case 31: result = d.ToString()+"st"; break;
        case 2: case 22: result = d.ToString() + "nd"; break;
        case 3: case 23: result = d.ToString() + "rd"; break;
        default: result = d.ToString() + "th"; break;
    }
    return result;
}
```

```
static void Main(string[] args)
{

    ValueTester delegateVariable = Day;
     Console.WriteLine(delegateVariable(13));
     Console.WriteLine(delegateVariable(22));
     Console.WriteLine(delegateVariable(31));
}
```

Indexers

- Allow **index values** to be used with *our own classes, similar to accessing array elements*, e.g., **`variable[index]`**
  - *index can be any data type*
  - *Several indexes can be used, separated by comma*
- Syntax:

```
[access_modifier ]return_type this[[parameter[, ...]]]
{
    [get
    {
        accessor_body
    }]
    [set
    {
        mutator_body
    }]
}
```

```
using System;
using System.Collections.Generic;

public class MyDictionary
{
      private List<string> _Data = new List<string>();
      public void StoreDefinition(params string[] data)
      {   foreach (string s in data) _Data.Add(s);
          _Data.Sort(); // arrange Dictionary in ascending for faster search of sort
      }
      public string this[int position]
      {   get { return _Data.Count != 0 && position >= 0 && position < _Data.Count ? _Data[position] : ""; }
          set { if(_Data.Count != 0 && position >= 0 && position < _Data.Count) _Data[position] = value; }
      }
      public string this[string value]
      {   get
          {  string result = "";
             int i;
             for (i = 0; i < _Data.Count; i++)
                if (_Data[i].ToLower().Contains(value.ToLower()))
                {   result = "Found :: \"" + _Data[i] + "\"";
                    break;
                }
             if (i == _Data.Count) result = "Do NOT have \"" + value +"\"";
             return result;
          }
      }
      public List<string> MyDictionaryData { get { return _Data; } }
}
```

```
public class Program
{
    static void Main()
    {
        MyDictionary myDictionary = new MyDictionary();
        myDictionary.StoreDefinition("word", "definition");
        Console.WriteLine("Using index location to retrieve and update data in the list:");
        Console.WriteLine(myDictionary[1]);
        myDictionary[0] = "Important Terms";

        Console.WriteLine("\nSearch for data which is stored on the list or NOT:\n");
        Console.WriteLine(myDictionary["definition"]);
        Console.WriteLine(myDictionary["important"]);

        Console.WriteLine("\nThe current myDictionary status:");
        //string[] dictionary = myDictionary.MyDictionaryData.ToArray();
        foreach (string s in myDictionary.MyDictionaryData) Console.WriteLine(s);
    }
}
```

Namespaces

- Consider **files on a disk**…
  - You can create as many files as you like
  - Each file must have a different name
- **The problem**…
  - Creating *different names can be a problem once a certain number of files are created*
- **The solution**…
  - Apply subdirectories – the scope of a *file name is limited to an individual subdirectory*

Namespaces

- Consider the **data types you create**…
  - You can create as many data types as you like
  - Each data type *must have a different name*
- **The problem**…
  - Creating *different names can be a problem once a certain number of data types are created*
- **The solution**…
  - Apply namespaces – the scope of a data type name is limited to an *individual namespace*

Namespaces

```
/*************************************************************
** File: FirstProgram.cs
** Author/s: Justin Rough
** Description:
**      A simple program used to introduce the basic structure
**      of a C# application.
*************************************************************/
using System;

namespace FirstProgram
{
        public class FirstProgram
        {
                // Main function, where the program's execution begins
                static void Main(string[] args)
                {
                        Console.WriteLine("Welcome to OO Development!");
                }
        }
}
```

File: FirstProgram.cs

Namespaces

- We have **already seen namespaces**…

```
using System;
```

- *Imports members from a namespace*, e.g.,

```
System.Console.WriteLine(…);
```

- *Becomes*

```
Console.WriteLine(…);
```

```
namespace FirstProgram
{
    …
}
```

- Used to *define members of a namespace*…

Namespaces

- Syntax:

```
namespace namespace_name
{
        namespace_member …
}
```

- Note that *all members of a namespace do not need to be declared in one file*
  - Can still have each class in a separate file as part of the same namespace

Report writing

- **Report writing** is often required when *dealing with text devices/streams*, e.g.,
  - Console input/output
  - Text files
  - Text printers
- The **requirements are minimal**:
  - Composite formatting
  - Ability to align various fields
  - Use appropriate characters to divide rows/columns
    - #, *, +, -, |, =, etc.
- Examples...

Report writing

```
#################################################################
# Qty # Description                                  #    Cost    #
#################################################################
#   5 # Visual C#: How to Program                    #    $134.95 #
#  50 # Database Systems                             #    $119.95 #
# 500 # HTML & XHTML: The Complete Reference         #     $72.00 #
#################################################################

*****************************************************************
* Qty * Description                                  *    Cost    *
*****************************************************************
*   5 * Visual C*: How to Program                    *    $134.95 *
*  50 * Database Systems                             *    $119.95 *
* 500 * HTML & XHTML: The Complete Reference         *     $72.00 *
*****************************************************************
```

Report writing

```
+-----+-----------------------------------------------+------------+
| Qty | Description                                   |    Cost    |
+-----+-----------------------------------------------+------------+
|   5 | Visual C#: How to Program                     |    $134.95 |
|  50 | Database Systems                              |    $119.95 |
| 500 | HTML & XHTML: The Complete Reference          |     $72.00 |
+-----+-----------------------------------------------+------------+
```

Qty	Description	Cost
5	Visual C#: How to Program	$134.95
50	Database Systems	$119.95
500	HTML & XHTML: The Complete Reference	$72.00

Report writing

- Need to decide **how to deal with extra-long strings**, examples...
  - `Data is too long`
    - No change, report looks ugly, no lost data
  - `Data is to`
    - Straight truncation
  - `Data is >>`
    - Truncation with an indication of truncation (>>)
  - `**********`
    - Replace data with a pattern clearly indicating data too big/lost

Report writing

- **Handy features of the string class**:
  - To truncate a string:

```
someString = someString.Substring(0, 10);
```

  - To fill a string with a character:

```
string asterisks = new string('*', 40);
```

Summary

- Session 04. Relationships
  - Objectives
  - Object and class relationships
  - Relationships in C#
  - Delegates
  - Indexers
  - Namespaces
  - Report Writing

Summary

- Training Videos:
  - C#: Constructors, Destructors, and Dispose
  - C#: ToString
  - T&T: Cheating with Visual Studio