Reflection in C#

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

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

Semester 1 — 2021/22

[0] No proprietary software has been used in producing these slides

Motivation

- **Sometimes you want to get access to *concepts* in C# that are not usually explicit.**
- **This is handy if you want to manipulate program constructs.**
- **Technically, you'll need to access the *meta-data* of a program, i.e. the data that gives additional information about the program, but is not part of its semantics.**

Case study: Implicit Serialisation

- One instance of reflection is *implicit serialisation.*
- The goal is to turn the data in an object into a linear, or serial, form so that it can be written to disk or transferred to another machine.
- There are two ways to achieve this: explicit or implicit serialisation.
  - *explicit serialisation* means that the programmer writes the code for serialisation
  - *implicit serialisation* means that the system tries to automatically generate the code for it

Implicit serialisation is achieved by attaching an attribute as meta-data to a class definition:

```
[Serializable]
class Lecturer: Person {
  ...
```

Case study: Implicit Serialisation

- One instance of reflection is *implicit serialisation.*
- The goal is to turn the data in an object into a linear, or serial, form so that it can be written to disk or transferred to another machine.
- There are two ways to achieve this: explicit or implicit serialisation.
  - *explicit serialisation* means that the programmer writes the code for serialisation
  - *implicit serialisation* means that the system tries to automatically generate the code for it

Implicit serialisation is achieved by attaching an attribute as meta-data to a class definition:

```
[Serializable]
class Lecturer: Person {
 ...
```

Using implicit serialisation

This automatically generates a function `Serialize` for serialisation before writing an object to file.
We can use this function like this:

```
IFormatter formatter = new BinaryFormatter();
Stream streamOut = new FileStream("ThisPerson.bin",
    FileMode.Create, FileAccess.Write, FileShare.None)
    ;
formatter.Serialize(streamOut, 1);
streamOut.Close();
```

Using implicit serialisation

We read the serialised data from file like this:

```
IFormatter formatter = new BinaryFormatter();
Stream streamIn = new FileStream("ThisPerson.bin",
    FileMode.Open, FileAccess.Read, FileShare.Read);
Lecturer l1 = (Lecturer) formatter.Deserialize(
    streamIn);
streamIn.Close();
```

This uses a *binary* formatter. For compatibility across platforms, you often want other formats such as SOAP (`System.Runtime.Serialization.Formatters.Soap`) or XML (`System.Xml.Serialization`).

Using implicit serialisation

We read the serialised data from file like this:

```
IFormatter formatter = new BinaryFormatter();
Stream streamIn = new FileStream("ThisPerson.bin",
    FileMode.Open, FileAccess.Read, FileShare.Read);
Lecturer l1 = (Lecturer) formatter.Deserialize(
    streamIn);
streamIn.Close();
```

This uses a *binary* formatter. For compatibility across platforms, you often want other formats such as SOAP (`System.Runtime.Serialization.Formatters.Soap`) or XML (`System.Xml.Serialization`).

Explicit Serialisation

- Writing your own serialisation function is easy, and useful in many different contexts, e.g. implementing `ToString()`.
- To serialise an object of class A:
  - Serialise all value type attributes, by directly writing the data into the result buffer
  - Serialise all reference type attributes by recursively calling serialisation on them.

Naive Serialisation

We implement `ToString()` for our Student example as one special case of serialisation:

```
public string ToString() {
 return String.Format
   ("Name: {0} {1}\tAddress: {2}\nMatricNo: {3}\tDegree: {4}",
    this.GetfName(), this.GetlName(), this.GetAddress(),
    this.matricNo, this.degree);
}
```

Serialisation using overloading

A better way to implement serialisation is to use the class hierarchy and *overloading*:

```
public override string ToString() {
  string base_str = base.ToString();
  string this_str = String.Format("MatricNo: {0}\tDegree: {1}", this.matricNo, this.degree);
  return base_str+"\n"+this_str;
}
```

This way, any change in `ToString()` as defined in the base class is picked up without further code changes.

The implementor of the `Student` class no longer needs to know details of the base class (`Person`), i.e. he/she *abstracts* over implementation details and makes the code more re-usable.

Serialisation using overloading

A better way to implement serialisation is to use the class hierarchy and *overloading*:

```
public override string ToString() {
  string base_str = base.ToString();
  string this_str = String.Format("MatricNo:␣{0}\
      tDegree:␣{1}", this.matricNo, this.degree);
  return base_str+"\n"+this_str;
}
```

This way, any change in `ToString()` as defined in the base class is picked up without further code changes.
The implementor of the `Student` class no longer needs to know details of the base class (`Person`), i.e. he/she *abstracts* over implementation details and makes the code more re-usable.

Using properties rather than fields

We now use properties, rather than fields, to better control access to the data:

```
class Student: Person{
  // the private data for Student
  private string _matricNo;
  private string _degree;

  // the properties to access the data
  public string degree {
    get { return _degree; }
    set { _degree = value; } }

  public string matricNo {
    get { return _matricNo; }
    set { _matricNo = value; } }
  // constructor
  ...
}
```

Serialisation using reflection

We can further improve the code by abstracting over the concrete property-names as well, by using *reflection*:

```
public override string Serialise() {
 string str = "";
 Type type = this.GetType();  // reflection!
 PropertyInfo[] props = type.GetProperties();
 foreach(PropertyInfo propertyInfo in props) {
   str += String.Format("\t{0}:␣{1}",
              propertyInfo.Name, propertyInfo.GetValue(this, null));
 }
 return str;
}
```

NB: this code doesn't mention the concrete names of the properties at all!

Serialisation using reflection

We can further improve the code by abstracting over the concrete property-names as well, by using *reflection*:

```
public override string Serialise() {
 string str = "";
 Type type = this.GetType();  // reflection!
 PropertyInfo[] props = type.GetProperties();
 foreach(PropertyInfo propertyInfo in props) {
   str += String.Format("\t{0}:␣{1}",
             propertyInfo.Name, propertyInfo.GetValue(
                this, null));
 }
 return str;
}
```

NB: **this code doesn't mention the concrete names of the properties at all!**

Serialisation using reflection

Running this code will show the names and values for all properties visible in an object, e.g.

```
Lecturer: Name: Hans-Wolfgang Loidl Address: Edinburgh
OfficeNo: G51

Now we use Reflection to implement generic serialisation:

Doing Lecturer.Serialise() ...
Found 4 fields ...
Lecturer:
officeNo: G51 fName: Hans-Wolfgang lName: Loidl address: Edinl

Doing Lecturer.Serialise1() ...
Lecturer: Name: Hans-Wolfgang Loidl Address: Edinburgh
OfficeNo: G51
Person: Person.Serialise: To be implemented
```

Fields vs Properties

You can use either fields or properties, but the reflective code needs to know whether to look for one or the other.

Properties are usually safer, even if you use the default getter and setter, because you can later modify e.g. the setter to *trace calls* to it like this:

```
public string officeNo {
  get { return _officeNo; }
  // set { _officeNo = value; } } // default
  set {  // setter prints changes and callers
    StackTrace stackTrace = new StackTrace();
    MethodBase methodBase = stackTrace.GetFrame(1).
        GetMethod();
    Console.WriteLine("setter called by : " +
        methodBase.Name);
    _officeNo = value;
  }
```

You can also use attributes to achieve the same thing.

Fields vs Properties

You can use either fields or properties, but the reflective code needs to know whether to look for one or the other.

Properties are usually safer, even if you use the default getter and setter, because you can later modify e.g. the setter to *trace calls* to it like this:

```
public string officeNo {
  get { return _officeNo; }
  // set { _officeNo = value; } } // default
  set {  // setter prints changes and callers
    StackTrace stackTrace = new StackTrace();
    MethodBase methodBase = stackTrace.GetFrame(1).GetMethod();
    Console.WriteLine("setter called by : " + methodBase.Name);
    _officeNo = value;
  }
```

You can also use attributes to achieve the same thing.

Custom attributes

Now we want to define our own attributes and attach it to code.
We want to define a `BugFix` attribute which captures *information of bug-fixes* during development.
This is better than capturing changes in comments, because the meta-data is machine-readable.
E.g. we want to use an attribute like this[1]:

```
[BugFixAttribute(121, "Jesse Liberty", "01/03/08")]
[BugFixAttribute(107, "Jesse Liberty", "01/04/08",
                 Comment = "Fixed off by one errors")]
public class MyMath
```

[1] Example code from Chapter 20 of "Programming C#", Jesse Liberty, Donald Xie (fifth ed.)

Defining custom attributes

We first need to define a *class* for our attributes, deriving from `System.Attribute`

```
public class BugFixAttribute : System.Attribute
```

Then, we need to specify, to which language constructs this attribute can be attached to. To do this, we use an attribute:

```
[AttributeUsage(AttributeTargets.Class |
                AttributeTargets.Constructor |
                AttributeTargets.Field |
                AttributeTargets.Method |
                AttributeTargets.Property,
                AllowMultiple = true)]
```

Defining custom attributes

We first need to define a *class* for our attributes, deriving from `System.Attribute`

```
public class BugFixAttribute : System.Attribute
```

Then, we need to specify, to which language constructs this attribute can be attached to. To do this, we use an attribute:

```
[AttributeUsage(AttributeTargets.Class |
                AttributeTargets.Constructor |
                AttributeTargets.Field |
                AttributeTargets.Method |
                AttributeTargets.Property,
                AllowMultiple = true)]
```

Defining custom attributes (cont'd)

The constructor of the attribute is fairly conventional:

```
// attribute constructor for positional parameters
public BugFixAttribute
(
    int bugID,
    string programmer,
    string date
)
{
    this.BugID = bugID;
    this.Programmer = programmer;
    this.Date = date;
}
```

Defining custom attributes (cont'd)

We want to use both positional and named arguments to the constructor. To do this we introduce properties:

```
// accessors
public int BugID { get; private set; }
public string Date { get; private set; }
public string Programmer { get; private set; }
// property for named parameter
public string Comment { get; set; }
```

NB: The positional parameters are *read-only*, by specifying the setter as `private`

NB: The named parameter `Comment` is implemented as a property

The complete example is in sample source file: `CustomAttrib.cs`

Defining custom attributes (cont'd)

We want to use both positional and named arguments to the constructor. To do this we introduce properties:

```
// accessors
public int BugID { get; private set; }
public string Date { get; private set; }
public string Programmer { get; private set; }
// property for named parameter
public string Comment { get; set; }
```

NB: **The positional parameters are *read-only,* by specifying the setter as** `private`

NB: **The named parameter** `Comment` **is implemented as a property**

The complete example is in sample source file: `CustomAttrib.cs`

Summary

- **Reflection allows the programmer to put data ("*meta-data*") onto language constructs.**
- **A common example is the use of the `[Serializable]` attribute, needed to e.g. write to file.**
- **Reflection can be used to make code more abstract and therefore more general, e.g. iterating over all properties.**
- **The programmer can define own custom attributes to attach meta-data, e.g. info about code changes.**