Reflection, Conversions, and Exceptions

Mingsheng Hong
CS 215, Spring 2008

Review

- Questions on second assignment?
- Quiz 2
  - Reference types and ref function parameters
  - Boxing: convert value types to reference types

Roadmap for Today's Lecture

- Reflection
- Conversion
  - Explicit and implicit conversions
  - User-defined conversions
- Exceptions

Reflection

- The ability to refer to the type system in code
  - eg. the `Type` class

```
Type t = Type.GetType("System.Int32");
bool b = t.IsSubclassOf(typeof(object));
```

  - construct types from strings
  - have classes that represent types
  - can explicitly compare types and determine subclassing (and other) relationships

Reflection – Example

- We want to get a methods dynamically:

```
C c = new C();
Type t = c.GetType();
for(int i = 0; i < 10; i++) {
    MethodInfo m = t.GetMethod("m" + i);
    m.Invoke(c, null);
}
```

- Type contains information about the type
  - all methods, members, properties, etc
  - whether or not it is an array
  - all nested types
- Check out `System.Reflection`

Reflection – is

- How do we get/check type information?
  - use `is` operator: `if (c is C) { … }`
    - like `instanceof` in Java
  - returns `true` if it is this class
  - If it is a subclass, `is` returns `true`
  - reflects dynamic type information
    - `Base a = new Derived(); if (a is Derived) {…}`
  - if compiler can decide statically, it will warn
    - `int i = 0; if (i is object) { … }`

Reflection – as

- Instead of a cast, use `as` keyword
  - eg. `object o = c as object`
  - returns an object of the right type
  - or `null` if not possible (no conversion exists)
  - can only use to convert to reference types
    - may perform boxing
- Better than type cast
  - may still need to cast if using a value type

Attributes

- Declarative information about program entities: public, private…
- Attributes are new kinds of declarative info
  - authorship
  - serializability
  - URLs of help documents
- Can be retrieved at run-time through reflection

Attributes

- Declaraction
  - any class derived from `System.Attribute`
  - naming convention: `Attribute` suffix
- Usage
  - `Attribute` suffix can be omitted

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute  { ... }
[Simple] class Class1 {...}
[Simple] interface Interface1 {...}
```

Params of `AttributeUsage`

- `ValidOn`
  - Of type `AttributeTargets`
  - Class, Struct, Enum, Method, All, …
- `AllowMultiple`
  - Multi-use or single-use attributes
- `Inherited`
  - Inherited by derived classes?
- Default value

```
[AttributeUsage(AttributeTargets.All,
AllowMultiple = false, Inherited = true)]
```

Attribute Parameters

- Positional and named parameters
  - constructors define positional parameters
  - non-static public RW fields define named ones

```
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute  {
    public HelpAttribute(string url) {…}
    public string Topic { get{…} set{…} }
    public string Url { get{…} }
}
[Help("http://…", Topic = "Programming")]
class Class1 {…}
```

Data Types of Parameters

- Parameters limited in type
  - numeric, string, and enum types
  - object and System.Type
  - single dimensional arrays of the above

```
public class HelpAttribute: Attribute  {
    public HelpAttribute(string url) {…}
    public string Topic { get{…} set{…} }
    public string Url { get{…} }
}
```

Reserved Attributes

- `AttributeUsage`
- `Conditional("SYMBOL")`
  - in `System.Diagnostics`
  - calls to method are included only if the symbol is defined at the method call point
    - E.g. `#define SYMBOL`
- `Obsolete("error or warning msg")`
  - can return compiler errors or warnings
  - useful for long-standing code

Reserved Attributes

- `DllImport`
  - Pinvoke: can import functions from native API
  - `[DllImport("kernel")] NtCreateFile(…)`
  - allows direct access to OS and others
- We will discuss this more when we discuss unsafe mode

Reflection – Code Generation

- `System.Reflection.Emit` namespace
- Can dynamically generate (CIL) code
- eg. `System.Reflection.Emit.MethodRental`
  - allows the replacement of a body with another
- Could write a program that generates another
  - why?

Conversions

- Implicit
  - never fails
  - to a "larger" type
  - eg. `int x = 0; long y = x;`
- Explicit
  - may fail
  - *usually* to a "smaller" type
    - Note: explicit conversions include all implicit conversions
  - eg. `long y = 0; int x = (int)y;`
- Boxing and unboxing?

User-Defined Conversions

- Can define a conversion operator
  - if not already defined
- Can be implicit or explicit

```
public class A {
  public static explicit operator B(A a) {…}
  …
}
public class B { … }
```

- Note: can be placed in either class `A` or `B`

Conversion Operators

- Can be overloaded

```
public class A {
  public static explicit operator short(A a) {…}
  public static explicit operator int(A a) {…}
  public static explicit operator double(A a) {…}
  …
}
```

- C# will only take one jump to convert
  - eg. if have conversion S to X and X to T, will not convert S to T automatically

Exceptions

- Dynamic exceptions can occur at runtime
  - eg. NullReference, DivideByZero
  - necessary to catch them
- Control structure same as Java
  - `try, catch, finally`
- `throw` statement can propagate exceptions
- Can implement own exceptions
  - Inherit from `System.Exception`

Exception Example

```
try {
   int x=5, y=0; x/=y;
} catch(DivideByZeroException except) {
   Console.WriteLine("Exception "+except.Message);
} catch(System.ArithmeticException except) {
   Console.WriteLine("Exception "+except.Message);
} catch(Exception except) {
   Console.WriteLine("Exception "+except.Message);
}
```

- Not a substitute for transfer of control or `goto` statement!
  - E.g. `as` operator versus conversion