Reflection, Conversion, and Exceptions

Hussam Abu-Libdeh
CS 2026, Spring 2009

Before we begin

- Assignment #2 released
  - Emailed to you & put on CMS
  - Questions ?
  - Due Friday 11:59 PM
- Update:
  - Do not use Hash and/or Set like classes in your assignment
  - Your data structure should optimize for repeated access.
  - Check the new version on CMS

Today's Agenda

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

Reflection

- The ability to refer to the type system in code at run-time
  - `Type t = Type.GetType(“System.Int32”);`
  - `bool b = t.IsSubClassOf(typeof(object));`
- Construct types from strings
- Have classes that represent type
- Can explicitly compare types and determine subclassing and other relationships

Reflection Example

- We want to get 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 about the type
  - All methods, members, properties ..etc
  - Whether or not it is an array
  - All nested types
- Check out `System.Reflection`

Reflection; `is` operator

- How do we get/check type information?
  - Use is operator: `if (c is C) { … }`
    - Like `instanceOf` in Java
  - Return true if it is the class or subclass
  - Reflects dynamic type information

    -
      ```
      Base a = new Derived();
        if (a is Derived) { .. }
      ```

Reflection; as keyword

- Instead of a cast, can use 'as' keyword
  - string o = c as string
  - Returns an reference of the right type
    - Null if not possible
  - Can only use to convert to reference types
    - May perform boxing
- Does not throw exception like casting
  - May still need to cast if using a value type

Reflection; Code Generation

- `System.Reflection.Emit` namespace
- Can dynamically generate CIL code
- e.g. `System.Reflection.Emit.FooMethod`
  - Allows the replacement of a body with another

Attributes

- Declarative information about program entities
  - public, private, protected …
- Attributes are new kinds of declarative info
  - Authorship, Serializability, URLs of help documents

```
[System.Serializable]
public class SampleClass { .. }
```

- Can be retrieved at run-time through reflection

Attributes

- Declaration
  - Any class derived from `System.Attribute`
  - Naming convention: Attribute suffix
    - can be dropped in usage
- Three reserved attributes
  - `AttributeUsage`
    - Describes how a custom attribute can be used
  - `Conditional`
    - Describes a conditional method whose execution depends on a preprocessor identifier
  - `Obsolete`
    - Marks program entities that should not be used

AttributeUsage

-
  ```
  [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 ..etc
- `AllowMultiple`
  - Multi-use or single-use attributes
- `Inherited`
  - Inherited by derived class?
- Default value

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

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 Foo { ... }
  ```

Data Types of Parameters

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

Reserved Attributes

- `Conditional("SYMBOL")`
  - In `System.Diagnostics`
  - Calls to methods are included only if the symbol is defined at the method entry point
    - Example: `#define SYMBOL`
  - Useful in compiling different versions of a product from the same source code

Reserved Attributes

- `Obsolete(“error or warning msg”)`
  - Can return compiler errors or warnings
  - Useful for long-standing code
- `DllImport`
  - Pinvoke: can import functions from native API
  - `[DllImport(“kernel”)] NtCreateFile (..)`
  - Allows direct access to OS

Conversions

- Implicit
  - To a “larger” type
  - int x = 0; long y = x;
- Explicit
  - May fail
  - Can be to a “smaller” type
  - long y = 0; int x = (int) y;
- Boxing/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 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 bool(A a){..}
}
```

- C# will only take one jump to convert
  - If you have conversion from S to X and X to T, C# will not convert from S to T automatically

Exceptions

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

Exception Example

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

- Exceptions are costly, do not use them as your main control flow mechanism