MVA jumpstart

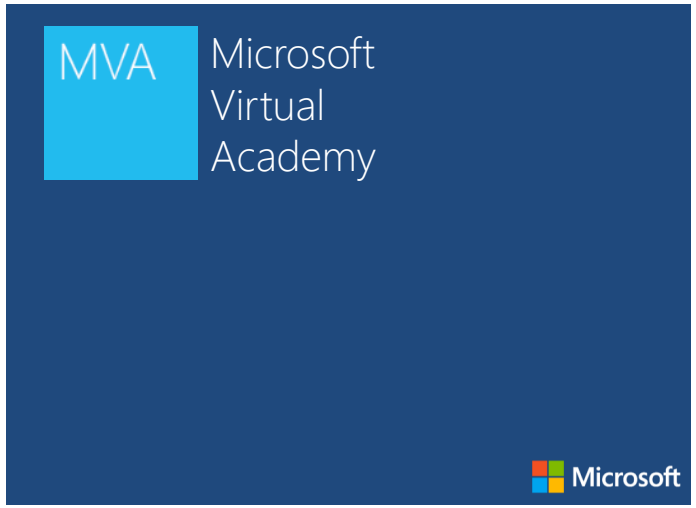
04 | Features of C#, Part 3

Jerry Nixon | Microsoft Developer Evangelist
Daren May | President & Co-founder, Crank211

Module Overview

- Code Reflection and Information
- Working with Garbage Collection

Code Reflection and Information



What is Reflection?

- Reflection inspects type metadata at runtime
- The type metadata contains information such as:
  - The type Name
  - The containing Assembly
  - Constructors
  - Properties
  - Methods
  - Attributes
- This data can be used to create instances, access values and execute methods dynamically at runtime

How do I get Type data?

- Two methods:
  - Statically at compile time
  - Dynamically at runtime

```
var dog = new Dog { NumberOfLegs = 4 };

// At compile time
Type t1 = typeof (Dog);

// At runtime
Type t2 = dog.GetType();

// output: Dog
Console.WriteLine(t2.Name);

// output: After002, Version=1.0.0.0,
//         Culture=neutral, PublicKeyToken=null
Console.WriteLine(t2.Assembly);
```

How can I create an instance of a Type?

- There are two ways to dynamically instantiate a type:
  - Activator.CreateInstance
  - Calling Invoke on a ConstructorInfo object (advanced scenarios)

```
var newDog =
    (Dog)Activator.CreateInstance(typeof(Dog));

var genericDog =
    Activator.CreateInstance<Dog>();

// uses default constructor
// with no defined parameters
var dogConstructor =
    typeof (Dog).GetConstructors()[0];

var advancedDog =
    (Dog)dogConstructor.Invoke(null);
```

Accessing a Property

```
void Property()
{
    var horse = new Animal() { Name = "Ed" };

    var type = horse.GetType();

    var property = type.GetProperty("Name");

    var value = property.GetValue(horse, null);
    // value == "Ed"
}

public class Animal
{
    public string Name { get; set; }
}
```

Invoking a Method

```
void Method()
{
    var horse = new Animal();

    var type = horse.GetType();

    var method = type.GetMethod("Speak");

    var value = (string)method.Invoke(horse, null);
    // value == "Hello"
}

public class Animal
{
    public string Speak() { return "Hello"; }
}
```

DEMO

Microsoft

Reflection (004)

Working with Garbage Collection

What is Garbage Collection?

- Garbage collection is automatic memory management.
- De-referenced objects (orphans) are not collected immediately but periodically.
  - Many factors influence Garbage Collection frequency
  - Not all orphans are collected at the same time
- Garbage Collection is computationally expensive

Forcing Garbage Collection

- In most cases, let the Garbage Collector do its thing.
- For a periodic activity it may make sense to force the collector to run:
  - Windows Service

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

But I want to help!

- If an object consumes many resources when instantiated.
- If you want to proactively free expensive resources
  - You don't want to force a full collection cycle.
- Force Garbage Collection?
  - Implement IDisposable.

Disposable Objects

- Some objects need explicit code to release resources.
- The IDisposable interface marks that these types implement the Dispose method.
- The simple dispose pattern works well for simple scenarios and sealed types
  - use the advanced pattern in most cases.

```
interface IDisposable
{
    void Dispose();
}

public class Demo : IDisposable
{
    public void Dispose()
    {
        // release resources
    }
}
```

Advanced Dispose Pattern

- Use for any non-trivial disposable object.

```
public class AdvancedDemo : IDisposable
{
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // release managed resources
        }
        // release unmanaged resources
    }

    ~AdvancedDemo()
    {
        Dispose(false);
    }
}
```

“Using” a shortcut…

- The *using* keyword provides a useful shortcut for invoking Dispose on types that implement IDisposable.

```
// open the file
using (Stream stream1 = File.Open(file, FileMode.Open))
{
    Console.WriteLine(stream1.Length);
}


// is the same as
Stream stream1 = File.Open(file, FileMode.Open);
try
{
    Console.WriteLine(stream1.Length);
}
finally
{
    if (stream1 != null)
    {
        ((IDisposable)stream1).Dispose();
    }
}
```

Dispose versus Close versus Stop

- Close
  - May be functionally the same as Dispose
  - May be a subset of the Dispose functionality
- A closed object may be reopened
  - IDbConnection
- Stop is similar to Close
  - May be restarted.
  - Timer, etc.

DEMO

Microsoft

Streams and Dispose (005)

Memory Leaks

- Despite having automatic memory management, it is still possible to create managed memory leaks.
- Objects that fall out of scope may be referenced by objects in scope, keeping them alive.
- Events can be a common source of memory leaks:
  - Events can hold references to objects
  - Solution! Unsubscribe from events proactively
- Weak references can be used to avoid some memory leak scenarios.

Weak References

- Weak references create a reference that the Garbage Collector ignores.
- The Garbage Collector will assume an object is eligible for collection if it is only referred to by weak references.
- To hold an object with only weak references, create a local variable referring to the weak reference value.
  - This prevents collection until the local variable is out of scope.

DEMO

Microsoft

Memory Leaks (036)

Module Recap

- Code Reflection and Information
- Working with Garbage Collection