MVA jumpstart

02 | Features of C#, Part 1

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

Module Overview

- Constructing Complex Types
- Object Interfaces and Inheritance
- Generics

Constructing Complex Types

Classes and Structs

- A class or struct defines the template for an object.
- A class represents a reference type
- A struct represents a value type
- Reference and value imply memory strategies

Struct

- A struct defines a value type.

```
public struct Point
{
    public int X { get; set; }

    public int Y { get; set; }
}
```

When to use Structs

- Use structs if:
  - instances of the type are small
  - the struct is commonly embedded in another type
  - the struct logically represent a single value
  - the values don't change (immutable)
  - It is rarely "boxed" (see later)
- Note: structs can have performance benefits in computational intensive applications.

Class

- A class defines a reference type (or object).
- Classes can optionally be declared as:
  - static – cannot ever be instantiated
  - abstract – incomplete class; must be completed in a derived class
  - sealed – cannot be inherited from

```
public class Animal
{
}
```

Partial Classes

- A class definition can be split into multiple source files.
- At compile time, these multiple parts are combined.
- Often used in code generation

```
// File: Animal.gen.cs

/// <summary>
/// Autogenerated do not edit
/// </summary>
public partial class Animal
{
}

// File Animal.cs
public partial class Animal
{
}
```

Access Modifiers

- All types and members have an accessibility level:

Keyword	Accessibility Level
public	can be accessed by any code in the same assembly or any other assembly that references it
private	can only be accessed by code in the same class or struct
protected	can only be accessed by code in the same class or struct or in a class derived from the class
internal	can be accessed by any code in the same assembly, but not from any other assembly

Properties

- Properties encapsulate access to an element of an object's data.
  - If a Animal *has* a Name, the value of the Name is accessed via the get accessor.
- Properties may explicitly encapsulate access to a backing field
- Properties may use the short-hand syntax for simple field access.

```
public class Animal
{
    public string Name { get; set; }
    public double Weight { get; private set; }

    private string _color;
    public string Color
    {
        get { return _color; }
        set { _color = value; }
    }
}
```

Methods

- Methods expose the functions that an object performs; what the object *does*.
  - If a Lion can make a sound, then the Lion class exposes the MakeSound method.

```
public class Lion()
{
    public string Sound {get; set;}

    public void MakeSound()
    {
        Console.WriteLine(Sound);
    }
}
```

Method Signature

- Methods are declared in a class or struct by specifying:
  - the access level
  - any modifiers
  - the return type
  - the name
  - any parameters
- The declaration of MakeSound states it is publically accessible, has no return type (void) and has no parameters.

```
public class Lion()
{
    public string Sound {get; set;}

    public void MakeSound()
    {
        Console.WriteLine(Sound);
    }
}
```

Method Parameters

- An *argument* is data passed to a method's *parameter* when it is invoked.
- When calling methods, we must supply arguments that match the parameter type.

```
public class Customer
{
    public int AddPoints(int pointsToAdd)
    {
        points += pointsToAdd;
        return points;
    }

    public int AddPoints(int pointsToAdd,
                         int maxPoints)
    {
        points += pointsToAdd;
        if (points > maxPoints)
        {
            points = maxPoints;
        }

        return points;
    }
}
```

Parameter Default Values

- Default parameter values allow developers to supply typical values
- Default parameter values allow developers to hint at usage
- Default parameters reduces the number of method overrides
- Default parameters allow parameters to be optional to the caller

```
public class Customer
{
    public int AddPoints(int pointsToAdd = 1)
    {
        points += pointsToAdd;
        return points;
    }
}

Usage:
 // Add 4 points
 int points = 4;
 int newPoints1 = AddPoints(points);

 // Add 5 points
 int newPoints2 = AddPoints(5);

 // Add 1 point (default value)
 int newPoints3 = AddPoints();
```

Named Arguments

- Named arguments allow developers to specify arguments in any order
- Named arguments allow developers to easily handle optional arguments
- Positional and named arguments can be mixed
  - positional arguments must be supplied first.

```
public class Customer
{
    public int AddPoints(int pointsToAdd,
                         int maxPoints)
    {
        points += pointsToAdd;
        if (points > maxPoints)
        {
            points = maxPoints;
        }

        return points;
    }
}

Usage:
 int newPoints1 = AddPoints(1, 10);
 int newPoints2 = AddPoints(maxPoints:10,
                            pointsToAdd:1);
 int newPoints3 = AddPoints(1, maxPoints:10);
```

Events

- Events notify an observer that something has occurred in the object.
  - A Processor object exposes Completed event and raises it whenever the Process method completes.

```
void Start()
{
    var processor = new Processor();
    processor.Completed += processor_Completed;
    processor.Process();
}

void processor_Completed(object sender, EventArgs e)
{
    Console.WriteLine("Completed");
}

class Processor
{
    public event EventHandler Completed;
    public void Process()
    {
        // TODO: do something
        if (Completed != null)
            Completed(this, EventArgs.Empty);
    }
}
```

Delegates, Multicasting & EventHandler

- A delegate is an object that knows how to call a method
- A delegate type defines the method signature.
- An event is a list of delegates
  - Raising an event invokes every delegate (multicasting)
- An EventHandler is a delegate. It matches the expected method signature for an event.

Object Instances and Inheritance

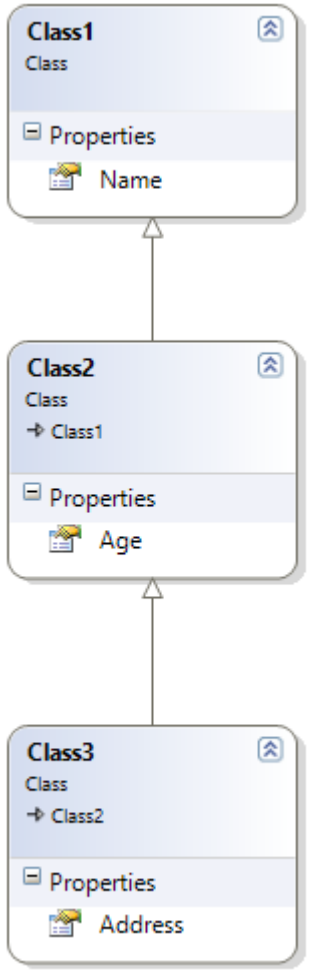
Inheritance

- Classes can optionally be declared as:
  - static – can never be instantiated
  - abstract – can never be instantiated; it is an incomplete class
  - sealed – all classes can be inherited unless marked as sealed
- Virtual Methods
  - Virtual methods have implementations
  - They can be overridden in derived class.

Example Inheritance Hierarchy

- Class1 defines one property: Name
- Class2 inherits Name from Class1 and defines Age.
- Class3 inherits Name from Class1 (via Class2) and Age from Class2, as well as defines Address.
- So an instance of Class3 has 3 properties:
  - Name
  - Age
  - Address

```
public class Class1
{
    public string Name { get; set; }
}
public class Class2 : Class1
{
    public int Age { get; set; }
}
public class Class3 : Class2
{
    public string Address { get; set; }
}
```



DEMO

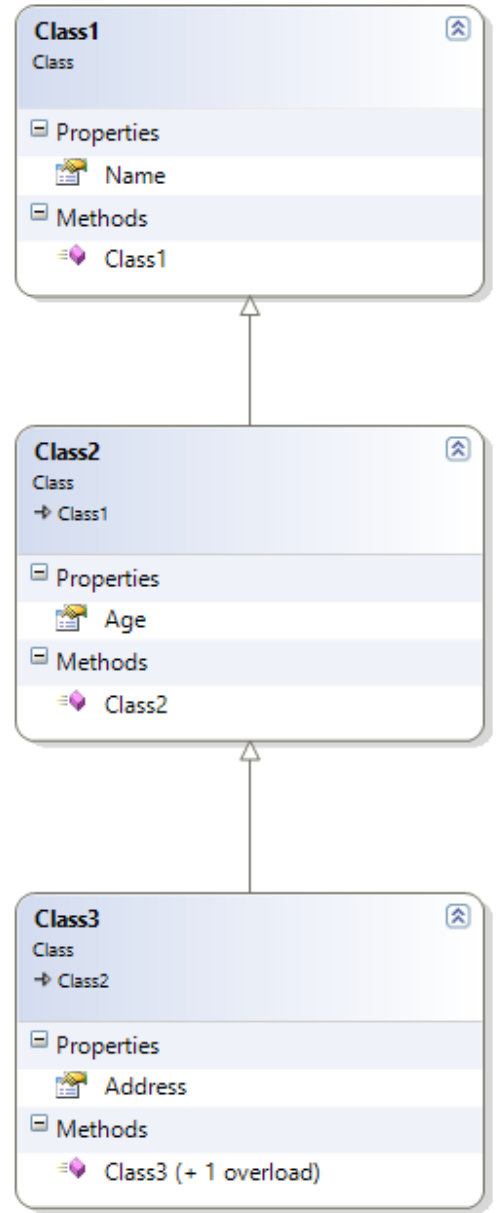
Microsoft

Understanding virtual, override and new (037)

Creating Object Instances

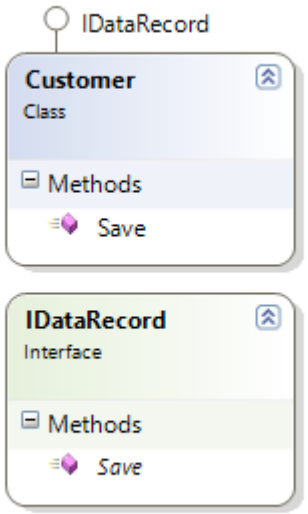
- When a class or struct is created, a constructor is called.
- Unless a class is static, the compiler generates a default constructor if not supplied in code.
- Constructors are invoked by using the *new* keyword.
- Constructors may require parameters.
- More than one constructor can be defined.
- Constructors can be chained together
- Base class constructors are always called (first)

```
public class Class1
{
    public string Name { get; set; }
    public Class1(string name)
    {
        Name = name;
    }
}

public class Class2 : Class1
{
    public int Age { get; set; }
    public Class2(string name, int age) : base(name)
    {
        Age = age;
    }
}

public class Class3 : Class2
{
    public string Address { get; set; }
    public Class3()
        : this("Fred", 1, "1 Microsoft Way") { }
    public Class3(string name, int age, string address) : base(name, age)
    {
        Address = address;
    }
}
```



Interfaces

- An interface defines a set of related functionality that can belong to one or more classes or structs.




```
interface IDataRecord
{
    // Defines the save method
    void Save();
}

public class Customer : IDataRecord
{
    // Actually implements the save record
    public void Save()
    {
        // Save the customer record here
    }
}
```

Generics

Generics

- Generics introduce to the .NET Framework the concept of type parameters
- Generics make it possible to design classes and methods that defer type specification until the class or method is declared
- One of the most common situations for using generics is when a strongly-typed collection is required.
  - lists, hash tables, queues, etc.

```
void List()
{
    var objects = new ArrayList();
    objects.Add(1);
    objects.Add(2);
    objects.Add("three");

    var list = new List<int>();
    list.Add(1);
    list.Add(2);
    list.Add("three");
}
```

Boxing / Unboxing

- Boxing is the act of converting a value type to a reference type.
- Unboxing is the reverse
  - Unboxing requires a cast.
- Boxing/Unboxing copies the value.
- Boxing is computationally expensive
  - avoid repetition
- Generics help avoid these scenarios

```
int count = 1;

// the value of count is copied and boxed
object countObject = count;

count += 1; // countObject is still 1

// the value of countObject (1) is unboxed
// and copied to count
count = (int)countObject;
```

Generic Classes, Interfaces and Methods

- You can create your own custom generic classes.
- When creating a generic class, consider:
  - Which types to generalize
  - What constraints to apply
  - Whether to create generic base classes
  - Whether to create generic interfaces
- Generic methods may also be created within non-generic classes

```
void Breakfast()
{
    var bird = new Animal<Egg>();
    var pig = new Animal<Piglet>();
}

public class Animal<T> where T : Offspring
{
    public T Offspring { get; set; }
}

public abstract class Offspring { }
public class Egg : Offspring { }
public class Piglet : Offspring { }
```

Module Recap

- Constructing Complex Types
- Object Interfaces and Inheritance
- Generics