Object Oriented Programming in C#

February 13, 2008

1. The C# object model
2. Implementing an object oriented language

Goals of the C# object system.

polymorphism
: the ability to write code which operates on many types—realized by inheritance, interfaces, and overloading

encapsulation
: the ability to make separate a class's behavior from its implementation details—realized with access modifiers

extensibility
: the ability to extend class functionality—realized with inheritance and virtual methods.

Terminology

- Field: a variable declared in a class.
- Method: a procedure associated with a class.
- Member: a field or method.

- Instance of <class>: an object of type <class>

Inheritance

- All classes inherit from a base class (default is System.Object).
- Derived classes automatically include the members of their base classes.
- Child classes *extend* base classes by adding new members, and overriding virtual methods.

- Can treat an instance of a derived class as an instance of its base class.

Basic inheritance example

```
using System;

class BaseSimple{
    public void Print(){
      Console.Out.WriteLine("BaseSimple");}
}

class ChildSimple : BaseSimple { }

class Runner{
  public static void Main(string[] s){
     (new BaseSimple()).Print();   // "BaseSimple"
     (new ChildSimple()).Print();  // "BaseSimple"
  }
}
```

Static Dispatch: New *overloaded* methods are called using an object's compile-time type.

```
class BaseNew{
    public void Print(){
        Console.Out.WriteLine("BaseNew");}  }

class ChildNew : BaseNew {
    new public void Print(){
        Console.Out.WriteLine("ChildNew");}  }

class Runner{
  public static void Main(string[] s){
      ChildNew c = new ChildNew();
      BaseNew b = c;

      c.Print();   // "ChildNew"
      b.Print();   // "BaseNew"
  } }
```

Dynamic Dispatch: *Virtual* methods called using an object's run-time type.

```
class BaseVirt{
    public virtual void Print(){
        Console.Out.WriteLine("BaseVirt");} }

class ChildVirt : BaseVirt {
    public override void Print(){
        Console.Out.WriteLine("ChildVirt");} }

class Runner{
  public static void Main(string[] s){
      ChildVirt c = new ChildVirt();
      BaseVirt b = c;

      c.Print();   // "ChildVirt"
      b.Print();   // "ChildVirt"
  } }
```

Overriding Rules

- Base classes may mark methods with virtual . Such methods are virtual and may be overridden by derived classes.
- Derived classes must mark methods with override to override them.
- Derived classes can mark methods with sealed prevent subclasses from overriding the methods.
  - By default methods are sealed.
  - A derived class can seal a virtual method to stop further overriding.
- Compiler with raise an error if there's a chance of ambiguity.

Calling base class methods with base

Sometimes we need to call a base class's methods explicitly.

```
class ChildVirt : BaseVirt {
  public override void Print(){
      Console.Out.WriteLine("ChildVirt␣says␣Hi!")
      Console.Out.WriteLine("Base␣virt␣says:");
      // calls base method
      base.Print();
      }

  // calls base constructor
  public ChildVirt(int x): base(x) {}
}
```

Without the base keyword, there would be no way to access such methods!

Class Modifiers and Static Members

- Class modifiers
  - Marking a class abstract means it can't be instantiated, only derived from.
  - Marking a class sealed means it can't be derived from, only instantiated.
  - Marking a class static means a class is both sealed and abstract. (Can only contain static members.)
- Static members
  - One copy of member per class (as opposed to per instance).
  - Can be initialized with a zero-argument static constructor.

Static class example

```
using System.Collections.Generic;

public static class Logger{
  private static List<string> myList;

  static Logger() { myList = new List<string>(); }

  public static void Append(string s) {
      myList.Add(s); }
  }
```

Interfaces declare contracts that a class must follow.

- Interfaces list methods which much a appear in a class.
- Methods may use interface names for argument and result types (bounded polymorphism).
- Classes can implement interfaces in two ways
  - Implicitly (the normal way), interface methods added directly to class and accessed as usual.
  - Explicitly, interface members are declared with special syntax and accessed through casts. Useful in the case where two interfaces declare methods with the same name.

Example: Implicit Interface Implementation

```
interface IWindow {
  void Draw();
}

public class Display: IWindow {
  // Implicit Interface Implementation
  public void Draw(){ Console.Out.WriteLine ("A");
}

class Runner{
  static void Main(string[] args){
    Display c = new Display();
    d.Draw(); // "A"
} }
```

Multiple interfaces can conflict.

```
interface IWindow {
  // Implementations should print to the screen
  void Draw();
}

interface ICowboy {
  // Implementations should get out a gun
  void Draw();
}

// Trouble!
public class WesternGame: IWindow, ICowboy {...}
```

Example: Explicit Interface Implementation

```
class WesternGame: IWindow, ICowboy {
  // Explicit Interface Implementations
  void IWindow.Draw(){
    Console.Out.WriteLine ("Drawing␣Picture"); }
  void ICowboy.Draw(){
    Console.Out.WriteLine ("Drawing␣Six␣Shooter");
}

class Runner{
static void Main(string[] args){
  WesternGame w = new WesternGame();

  // Error: w.Draw();
  ((ICowboy) w).Draw(); // "Drawing Picture"
  ((IWindow) w).Draw(); // "Drawing Six Shooter"
}}
```

Casting

```
string x = (string) someObject
```

- Up-casts:
  - Convert instances of a child class to a parent class or interface.
  - Always succeeds.
- Down-casts:
  - Convert instances of a parent class to a child class.
  - May fail and throw InvalidCastException
  - Use as or is to check if a cast is safe.
- Generics provide an elegant way to write (for example) collection classes without casting.

Access modifiers protect class implementation details.

Access modifiers may be attached to class, field, and method declarations.

Modifier	Meaning
public	No visibility restrictions.
protected[1]	Visible to classes derived from the defining class
internal[2]	Visible anywhere in the same assembly.
protected internal[1]	Visible according to protected. Also, member visible according to internal.
private[1]	Visible only within defining class

[1] Only applicable to elements defined in a class (i.e. not to classes defined only in a namespace).

[2] internal is the default access modifier.

1. The C# object model
2. Implementing an object oriented language

Methods

- C#: Methods, generics, objects, interfaces…
- Machine code:
  - operators: add, subtract, xor…
  - conditionals: if
  - jump
  - take CIS 371 for more details.
- Common Intermediate Language
  - object oriented byte code
  - .Net equivalent to Java byte code
  - closer to C# than machine code

How do we compile an object oriented program to machine code?

Functions and Methods

Functions

- take arguments, compute, and return a result.
- have access to arguments and global variables.
- always “means the same thing” (static dispatch).
- easy to implement in machine code.

Methods

- take arguments, compute, and return a result.
- has access to arguments, global variables, and object members.
- have context dependent meanings (dynamic dispatch).
- are be implemented in terms of functions.

From functions to methods

Translating methods to functions requires emulating two key method behaviors

- Access to object members:

- Dynamic dispatch:

Will also need simpleClasses (or records) which contain multiple fields but no methods.

From functions to methods

Translating methods to functions requires emulating two key method behaviors

- Access to object members:
  Represent methods as a functions that takes special argument, this, that contains an object reference.
- Dynamic dispatch:

Will also need simpleClasses (or records) which contain multiple fields but no methods.

From functions to methods

Translating methods to functions requires emulating two key method behaviors

- Access to object members:
  Represent methods as a functions that takes special argument, this, that contains an object reference.

- Dynamic dispatch:
  Lookup the right function to call in a table (the *vtable*) at runtime.

Will also need simpleClasses (or records) which contain multiple fields but no methods.

Example: Adding a this argument

```
class Counter{
    int C;
    void inc(int x) {C += x;}
    void incTwice(int x) {inc(x); inc(x)}
}
```

⇝

```
simpleClass Counter{ int C; }

function void Counter_inc(Counter this, int x){
    this.C += x;}

function void Counter_incTwice(Counter this,int x){
    call Counter_inc(this, x);
    call Counter_inc(this, x)}
```

Example: Virtual methods through v-tables

```
class Counter{
    int C;
    virtual void inc(int x) {C += x;} }

class FastCounter: Counter{
    override void inc(int x) {C += 2*x;} }

class Runner{
    static void Main(string[] args)
    {
      Counter c = new FastCounter();

      // Should call FastCounter method and get 6
      c.inc(3);
    }
}
```

Example: Virtual methods through v-tables

```
class Counter{
    int C;
    virtual void inc(int x) {C += x;}
}
```

⇝

```
simpleClass Counter{
    int C;
    // compiler remembers 0 -> Counter_inc
    function[] vtable = {Counter_inc};
}

function void Counter_inc(Counter this, int x){
    this.C += x;}
```

Example: Virtual methods through v-tables

```
class FastCounter: Counter{
    override void inc(int x) {C += 2*x;} }
```

$\rightsquigarrow$

```
simpleClass FastCounter{
    // copied from base class
    int C;
    // compiler remembers 0 -> FastCounter_inc
    function[] vtable = {FastCounter_inc};
}

function void FastCounter_inc(Counter this, int x){
    C += 2*x;}
```

Example: Virtual methods through v-tables

```
static void Main(string[] args)
{
    Counter c = new FastCounter();

    c.inc(3);
}

// static methods can compile to functions w/o this
function void Runner_Main(string[] args){
     // call FastCounter's default constructor
     c = call FastCounter_ctor();

     // do the virtual call
     function f = c.vtable[0];
     call f (c, 3)
}
```

Other features

Interfaces
: Each interface gets an interface table—analogous to a vtable.

Constructors
: Implemented like static methods—return the this pointer.