Derived Classes and Inheritance

Chapter 12
Objectives

You will be able to:

- Create derived classes in C#.
- Understand polymorphism.
- Write polymorphic methods.
- Use the “protected” access specifier appropriately.
- Define methods to extend existing classes and structs without defining a derived class.
Inheritance

- Derived classes and inheritance in C# are similar to C++.
  - Virtual methods
  - Polymorphism

- If you are comfortable with the concept of inheritance from using C++, you can skip over (or just skim) this section.

- Continue with Extension Methods
  - New in C#.
Derived Classes

- A derived class *extends* the definition of an existing class, called the base class.
  - Can add new fields.
  - Can add new methods.

- Objects of a derived class *inherit* the methods and fields of the base class.
  - Can potentially provide a new definition for an inherited method.
  - *Override* the original definition.
All classes in C# are derived (directly or indirectly) from System.Object.

- Inherit methods
  - Equals
  - GetType
  - ToString
  - (others)

- Inheritance from System.Object is implicit.
  - Need not be specified in class definition.
Derived Classes

- Objects of a derived class can be used anywhere objects of the base class could be used.
  - Variables
  - Arguments to methods

- "Is a" relationship
  - The Liskov Substitution Principle (1987)
Barbara Liskov

- Recipient of the 2008 Turing Award
- First woman in the US to receive a PhD in computer science.
  - Stanford 1968

Protected Access

- Another value like public and private
- “protected” means that the method or field can be accessed from inside the same class or a derived class but not from elsewhere.
- Normal for a class that will have derived classes (instead of private.)
- If protection is “private” code in derived classes cannot access the member.
Example

- In Visual Studio 2008, create a new C# Console project called Shape_Example.

- Add new class, Shape.
using System;
namespace Shape_Example
{
    public class Shape
    {
        protected String name;

        public String Name
        {
            get { return name; }
        }

        public Shape(String name_)
        {
            name = name_; 
        }

        public override string ToString()
        {
            return "I am a shape by the name of " + name;
        }
    }
}

Replace the ToString method inherited from class Object
using System;

namespace Shape_Example
{
    class Program
    {
        static void Main(string[] args)
        {
            Shape s1 = new Shape("Sue");

            Console.WriteLine(s1.ToString());
            Console.ReadLine();     // Keep the window open
        }
    }
}
I am a shape by the name of Sue
To save space on the slides, I will omit the wrappings:

```csharp
using System;

namespace Shape_Example
{
    ...
}
```

Generated automatically by Visual Studio when a class is added to a project
Add new class Circle

- Don't download
- This Circle class will be somewhat different from the previous one.
Add Class Circle
class Circle : Shape
{
    double radius;

    public Circle(double radius_, String name_) : base (name_)
    {
        this.radius = radius_;  
    }

    public double Radius
    {
        get { return radius; } 
    }

    public override String ToString()
    {
        return "I am a Circle of radius " + radius +
              " by the name of " + name;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Shape s1 = new Shape("Sue");
        Console.WriteLine(s1.ToString());

        Circle c1 = new Circle(5, "Clyde");
        Console.WriteLine(c1.ToString());
        Console.ReadLine();
    }
}
Program in Action

I am a shape by the name of Sue
I am a circle of radius 5 by the name of clyde
class Rectangle : Shape
{
    private double length;
    private double width;

    public Rectangle(double length_, double width_, String name_) : base (name_)
    {
        length = length_;
        width = width_;
    }
}
public double Length
{
    get { return length; }
}

public double Width
{
    get { return width; }
}

public override String ToString()
{
    return "I am a " +
        length + " by " + width +
        " rectangle by the name of " + name;
}
}
using System;
class Class1
{
    static void Main(string[] args)
    {
        Shape s1 = new Shape("Sue");
        Console.WriteLine(s1.ToString());

        Circle c1 = new Circle(5, "Clyde");
        Console.WriteLine(c1.ToString());

        Rectangle r1 = new Rectangle(2, 4, "Rhonda");
        Console.WriteLine(r1.ToString());
        Console.ReadLine();
    }
}
I am a shape by the name of Sue
I am a Circle of radius 5 by the name of Clyde
I am a 2 by 4 rectangle by the name of Rhonda
Class Shape
{
    ...
    public virtual string Method_Name()
    {
        ...
    }
}

Says that derived classes can provide their own version of this method.

Automatically virtual in all derived classes.

ToString() was declared as virtual in the definition of class System.Object.

ToString() is automatically virtual in every class.
Each class can provide its own version of ToString().
Virtual Methods

- In C# methods are NOT virtual by default.
  - Like C++
  - Unlike Java

- You must DECLARE methods as virtual if you want them to be virtual
  - which is the normal case!
Virtual Methods

- If a base class method is virtual, a derived class can provide its own implementation.
  - Must use the "override" keyword.
  - public override void draw()
  - Must have same signature.
  - Must have same accessibility (e.g. public)

- Calls to the method using an object of the derived class will invoke the derived class’s implementation.
  - Polymorphism
public class Shape
{
    protected String name;

    public Shape (String name_)
    {
        name = name_;  
    }

    public virtual double Area()
    {
        return 0.0;
    }
}
public class Circle : Shape
{
    double radius;

    public Circle(double radius_arg, String name_arg) :
        base (name_arg)
    {
        this.radius = radius_arg;
    }

    public override double Area()
    {
        return Math.PI*radius*radius;
    }

    ...

    For now, we don’t implement Area() in the other derived class.
Using Area() Method

Add to Main()

```csharp
Console.WriteLine();
Console.WriteLine("Areas:");
Console.WriteLine("{0}: {1}", s1.Name, s1.Area());
Console.WriteLine("{0}: {1}", c1.Name, c1.Area());
Console.WriteLine("{0}: {1}", r1.Name, r1.Area());
```
I am a shape by the name of Sue
I am a circle of radius 5 by the name of Clyde
I am a 2 by 4 rectangle by the name of Rhonda

Areas:
Sue: 0
Clyde: 78.5398163397448
Rhonda: 0
public class Rectangle : Shape
{
    private double length;
    private double width;

    public Rectangle(double length_,
                     double width_,
                     String name_):
        base (name_)
    {
        length = length_;
        width = width_;
    }

    public override double Area()
    {
        return length*width;
    }
}

Main is unchanged.
I am a shape by the name of Sue
I am a circle of radius 5 by the name of Clyde
I am a 2 by 4 rectangle by the name of Rhonda

Areas:
Sue: 0
Clyde: 78.5398163397448
Rhonda: 8
Virtual Method Example

- What if we had defined c1 as type Shape rather than type Circle?
static void Main(string[] args)
{
    Shape s1 = new Shape("Sue");
    Console.WriteLine(s1.ToString());

    Shape c1 = new Circle(5, "Clyde");
    Console.WriteLine(c1.ToString());

    Rectangle r1 = new Rectangle(2, 4, "Rhonda");
    Console.WriteLine(r1.ToString());
    Console.WriteLine();
    Console.WriteLine("Areas:");
    Console.WriteLine(s1.Name() + ": " + s1.Area());
    Console.WriteLine(c1.Name() + ": " + c1.Area());
    Console.WriteLine(r1.Name() + ": " + r1.Area());
    Console.ReadLine();
}
Virtual Method Example

Output is the same!

This is **polymorphism**.
Virtual Method Example

How did the compiler know to call Circle.Area() rather than Shape.Area?

Ans: It didn’t!

The linkage to the Area method() was not resolved until run time.

This is know as *late binding.*
Late binding is the key to polymorphism.

Virtual methods are called indirectly through a pointer in an overhead area of the object. (not accessible to the programmer.)

The specific object used for the call determines which version of the method is invoked.
If the method had not been declared as virtual, the call to `c1.Area()` would have been resolved at compile time.

The *declaration* of `c1` would have determined which version of `Area()` was invoked by the call `c1.Area()`;
Extension Methods

- New feature in C# 3.0
  - Pages 233 – 236 in textbook

- Permits us to add methods to existing classes (or structs) without creating a derived class.

- Extends the class definition for the program in which it is defined.
Extension Methods

- Let’s extend class String with a method that determines whether a string is a palindrome.

- Create a new C# console application
  - Extension_Method_Demo
Extension Methods

```csharp
using System;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("This program determines whether strings\n");
        Console.WriteLine("that you type are palindromes\n");
        Console.ReadLine();
    }
}
```
Defining an Extension Method

- An extension method must be a static method defined within a static class.

- The type to be extended is the type of a parameter of the method
  - Preceeded by the keyword *this*. 
Defining an Extension Method

- Let’s extend the String class with a method that says whether or not a given string is a palindrome.
  - Same letters forward and backward.
  - Ignoring case and non-letter characters.

- Strategy:
  - Build strings consisting of only the letters, all in upper case, in the original string.
  - Forward and backward copies.
  - Check if they are identical.
Add Static Class Utils

```csharp
using System;

class static class Utils {
}
```
An Extension Method

```csharp
using System;

static class Utils
{
    public static bool Is_Palindrome(this String s)
    {
        return true;
    }
}
```
using System;
using System.Text;
using System.Collections.Generic;

static class Utils
{
    public static bool Is_Palindrome(this String s)
    {
        StringBuilder forward = new StringBuilder();
        StringBuilder reverse = new StringBuilder();

        Because Strings are immutable in C#, modifying a String is inefficient. StringBuilder is the preferred way to build up a string. Convert to String when finished.
foreach (char c in s)
{
    if (char.IsLetter(c))
    {
        forward.Append(char.ToUpper(c));
        reverse.Insert(0, char.ToUpper(c));
    }
}

String forward_string = forward.ToString();
String reverse_string = reverse.ToString();

return forward_string == reverse_string;
static void Main(string[] args)
{
    Console.WriteLine("This program determines whether strings");
    Console.WriteLine("that you type are palindromes");

    while (true)
    {
        Console.WriteLine("\n\nType a sentence.");
        String s = Console.ReadLine();
        if (s.Is_Palindrome())
        {
            Console.WriteLine(@"" + s + @"" is a palindrome");
        }
        else
        {
            Console.WriteLine(@"" + s + @"" is not a palindrome");
        }
    }
    Console.ReadLine();  // Hold window open.
}
This program determines whether strings that you type are palindromes.

Type a sentence.
Able was I ere I saw Elba.
"Able was I ere I saw Elba." is a palindrome.

Type a sentence.
Now is the time.
"Now is the time." is not a palindrome.

Type a sentence.
Summary

- Inheritance is one of the key concepts of object oriented programming.
  - Permits us to extend existing classes without modifying the original code.

- Objects of a derived class can be used anywhere an object of the base class could be used.
  - Circle “IS A” Shape
A derived class can provide its own version of any virtual method defined in its base class.

Methods in other classes need not be aware of which derived class they are dealing with.
- Can invoke any public method declared in the base class, using a reference to the base class.
- Objects of a derived class will do the right thing.
  - Polymorphism.
Summary

- Extension methods provide an alternative to creating a derived class.

- Permits us to define new static methods to existing classes and structs without creating a derived class.

End of Presentation