Inheritance and Polymorphism

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.

System.Object

- 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 defintion.

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

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

The Main Program

```
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
        }
    }
}
```

Creating Derived Classes

```
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;
    }
}
```

This makes Circle inherit from Shape.

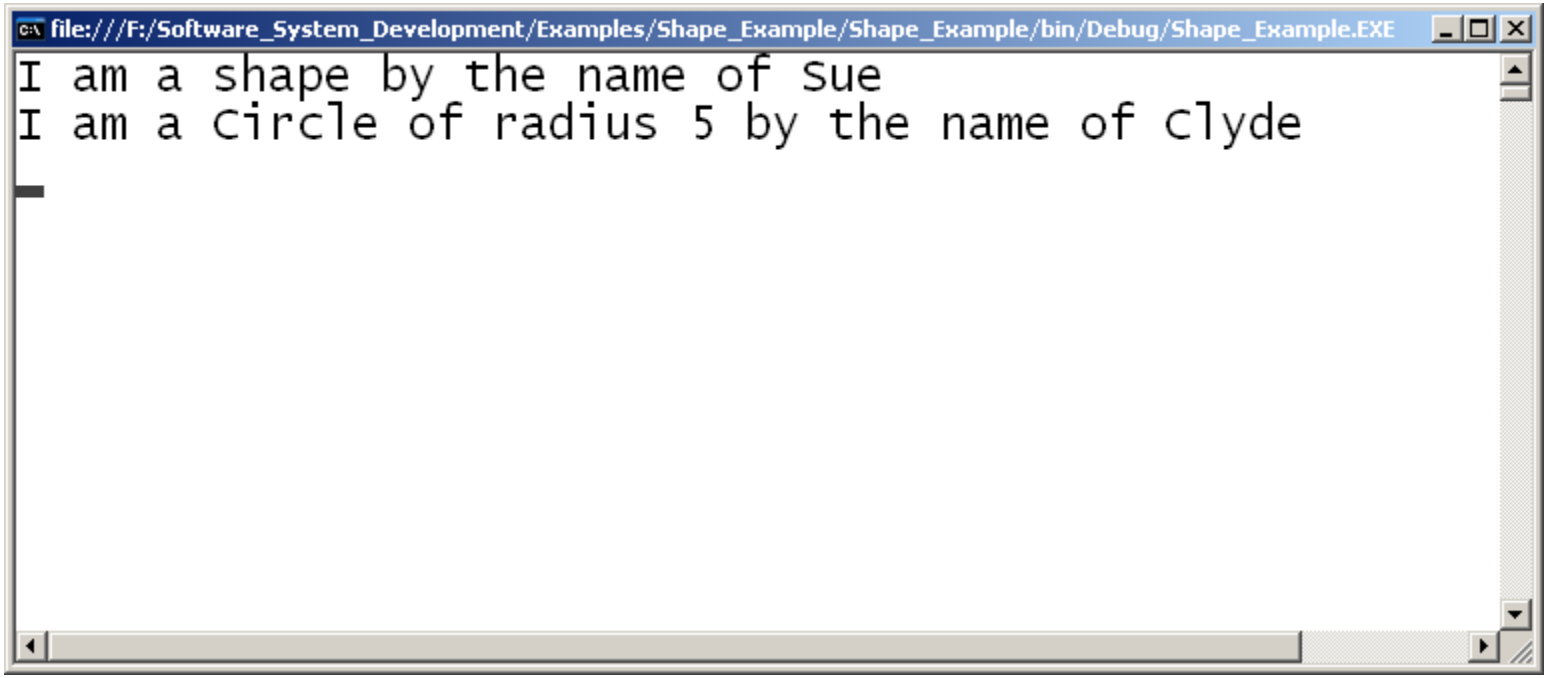
Call constructor for Shape passing name_.

Replace the ToString method inherited from Shape.

Using the Derived Class

```
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



Class Rectangle

```
class Rectangle : Shape
{
    private double length;
    private double width;

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

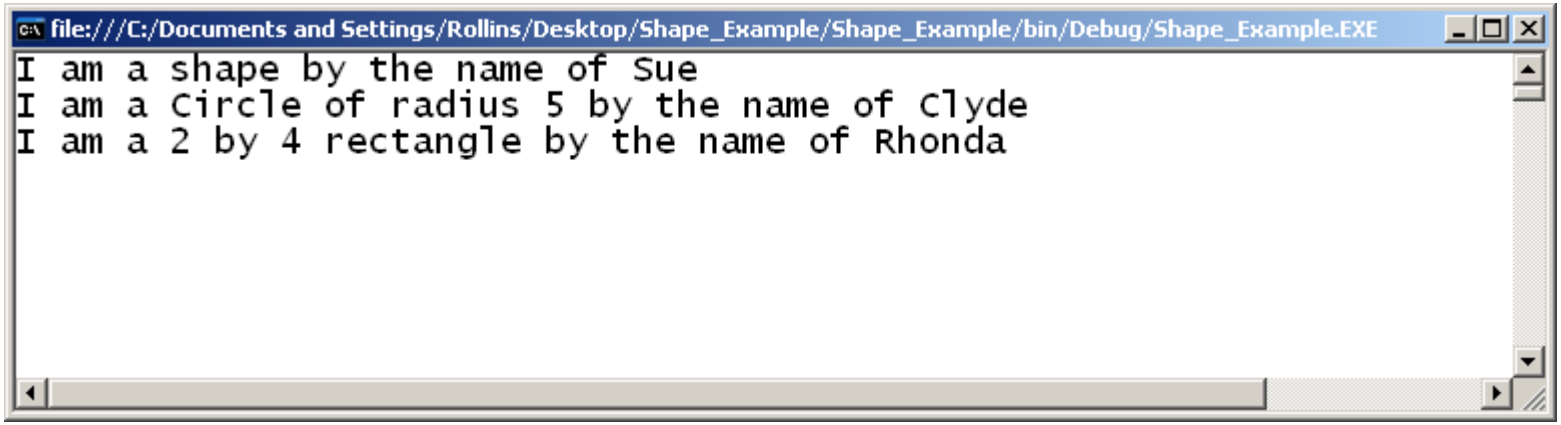
Class Rectangle

```
        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 Class Rectangle

```
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();
 }
}
```

Program in Action



Virtual Methods

```
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*

Virtual Methods

```
public class Shape
{
    protected String name;

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

    public virtual double Area()
    {
        return 0.0;
    }
```

Overriding the Base Class's Area Method

```
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;
    }

…
```

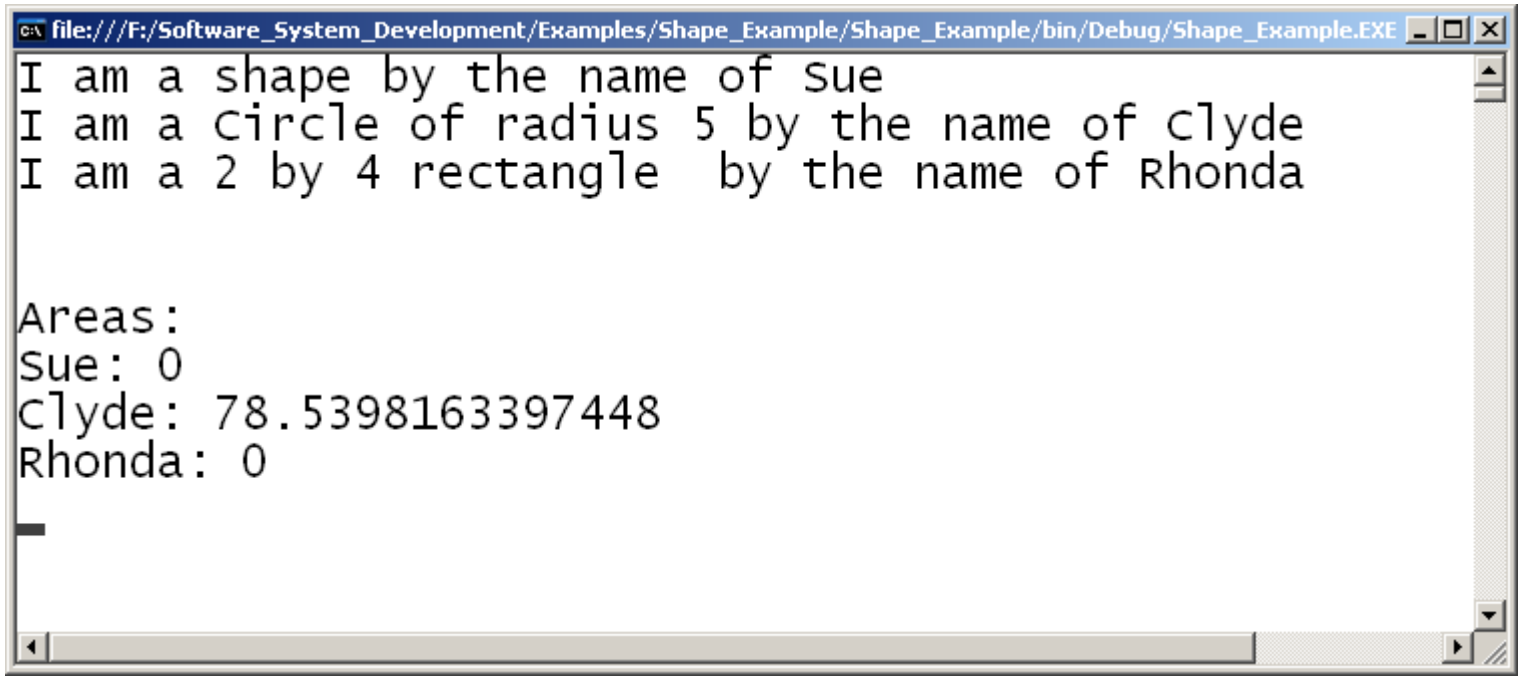
For now, we don't implement Area() in the other derived class.

Using Area() Method

Add to Main()

```
Console.WriteLine();
Console.WriteLine("Areas:");
Console.WriteLine(s1.Name + ": " + s1.Area());
Console.WriteLine(c1.Name + ": " + c1.Area());
Console.WriteLine(r1.Name + ": " + r1.Area());
```

Program in Action

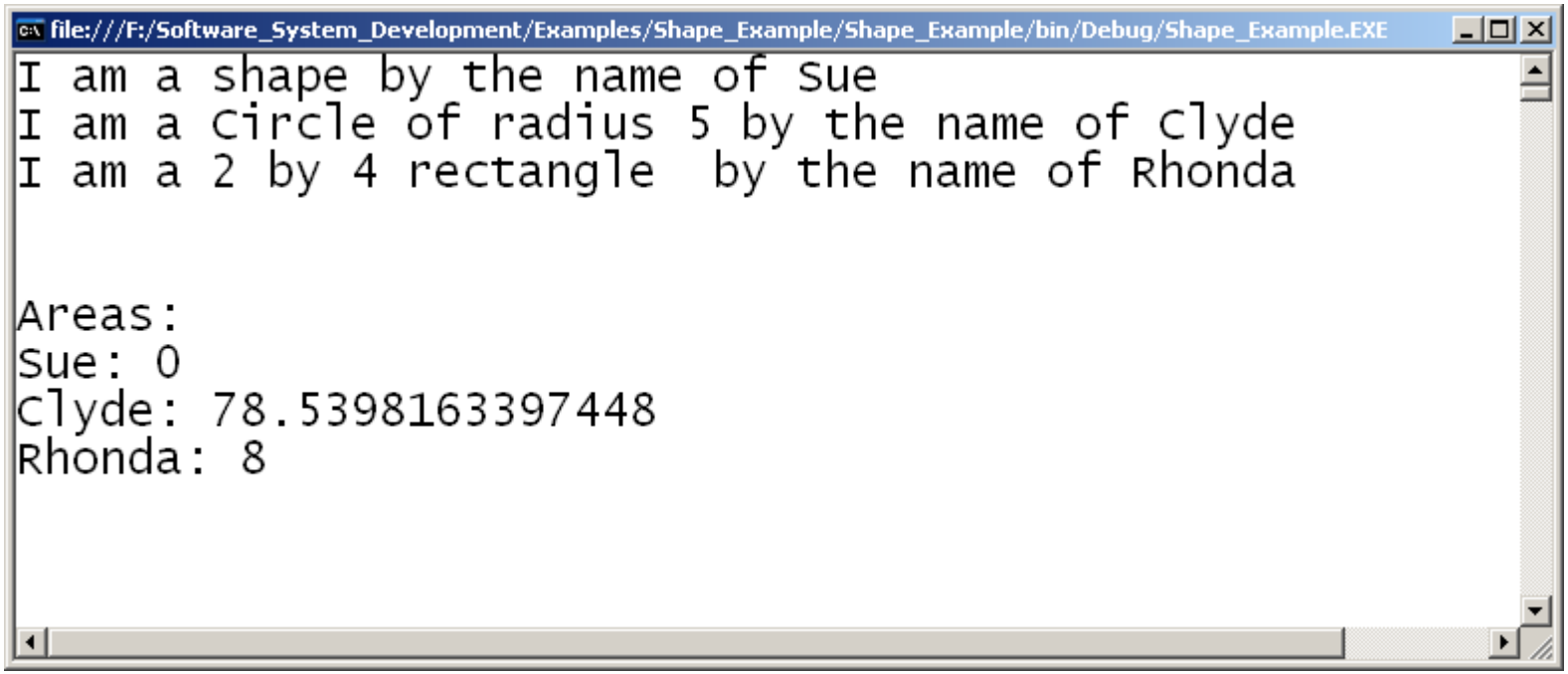


Add Area() to class Rectangle

```
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.

Program in Action



Virtual Method Example

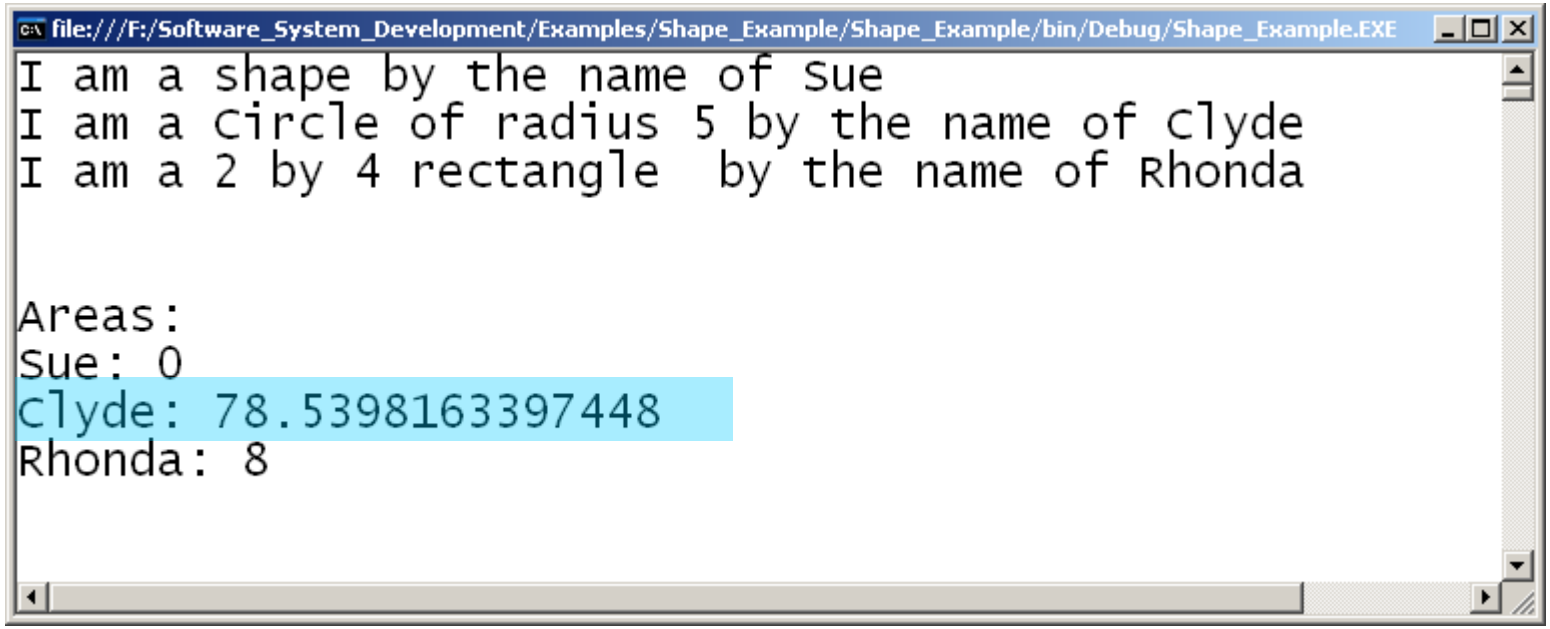
- What if we had defined c1 as type Shape rather than type Circle?

c1 as a Shape

```
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

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.

Late Binding

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

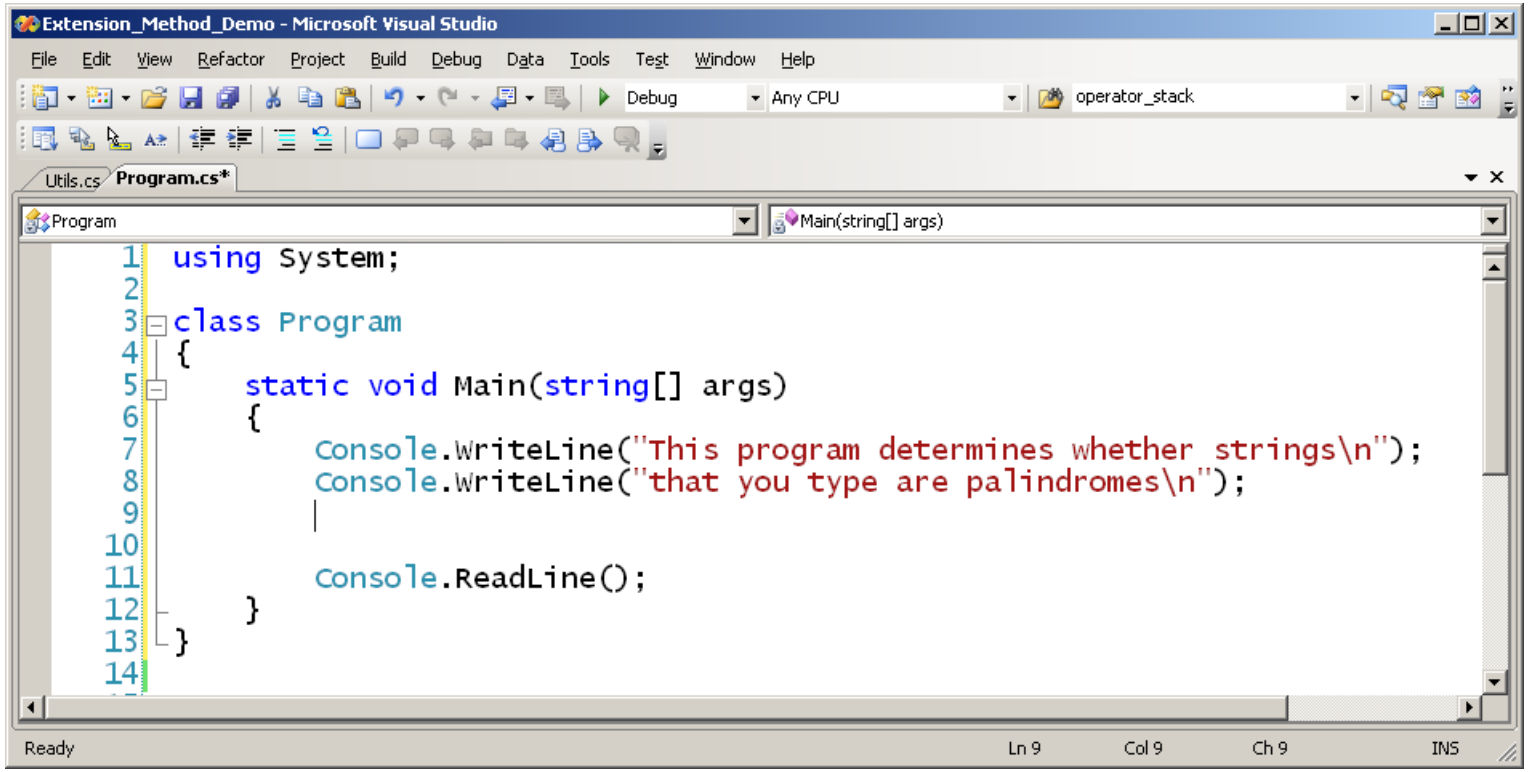
- New feature in C# 3.0
- Permits us to add methods to existing classes (or structs) without creating a derived class.
- Extends the class definition *for the program in which it is defined*.

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



Defining an Extension Method

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

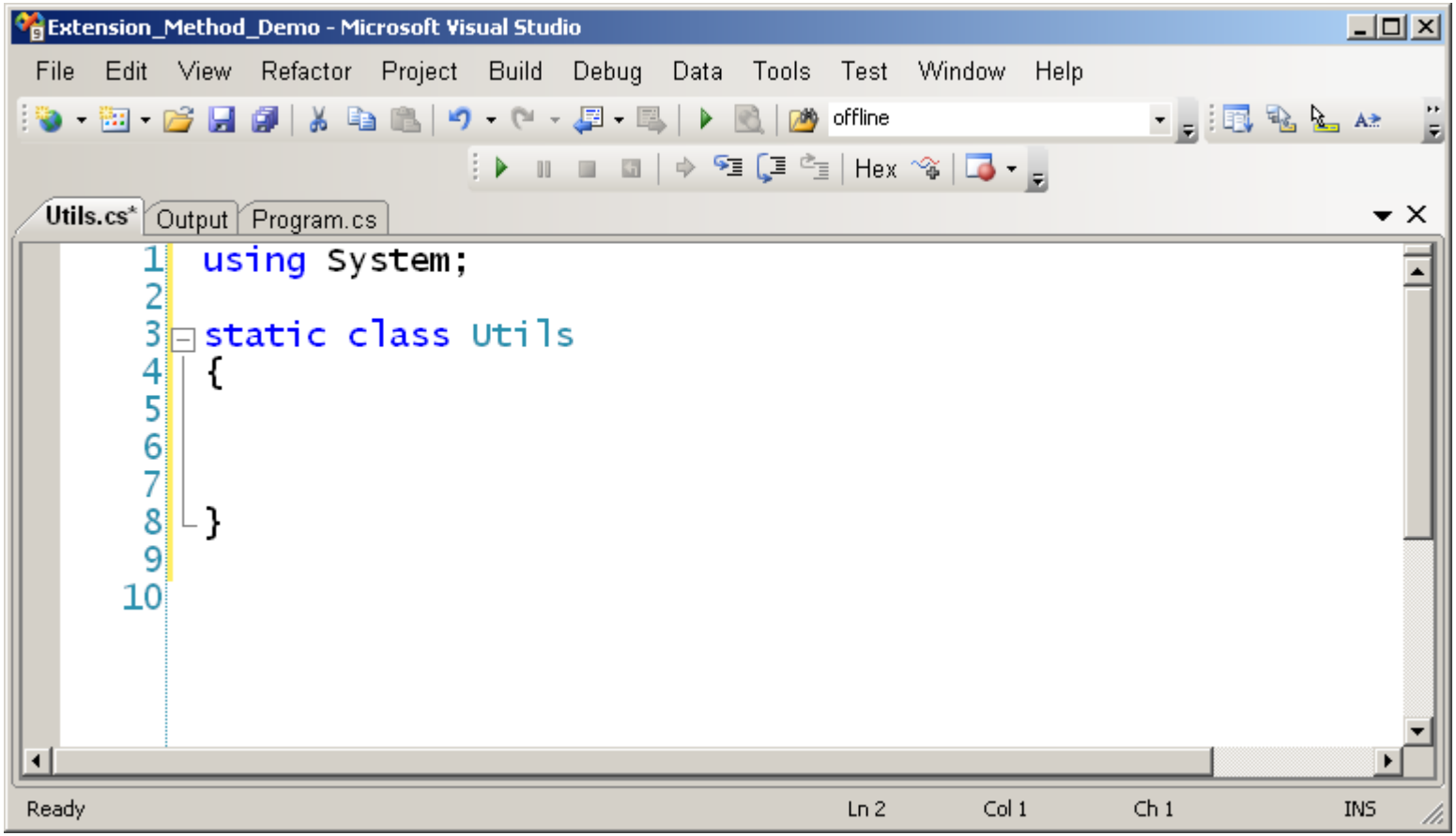
- The type to be extended is the type of a parameter of the method
  - Preceeded by the keyword *this*.

Defining an Extension Method

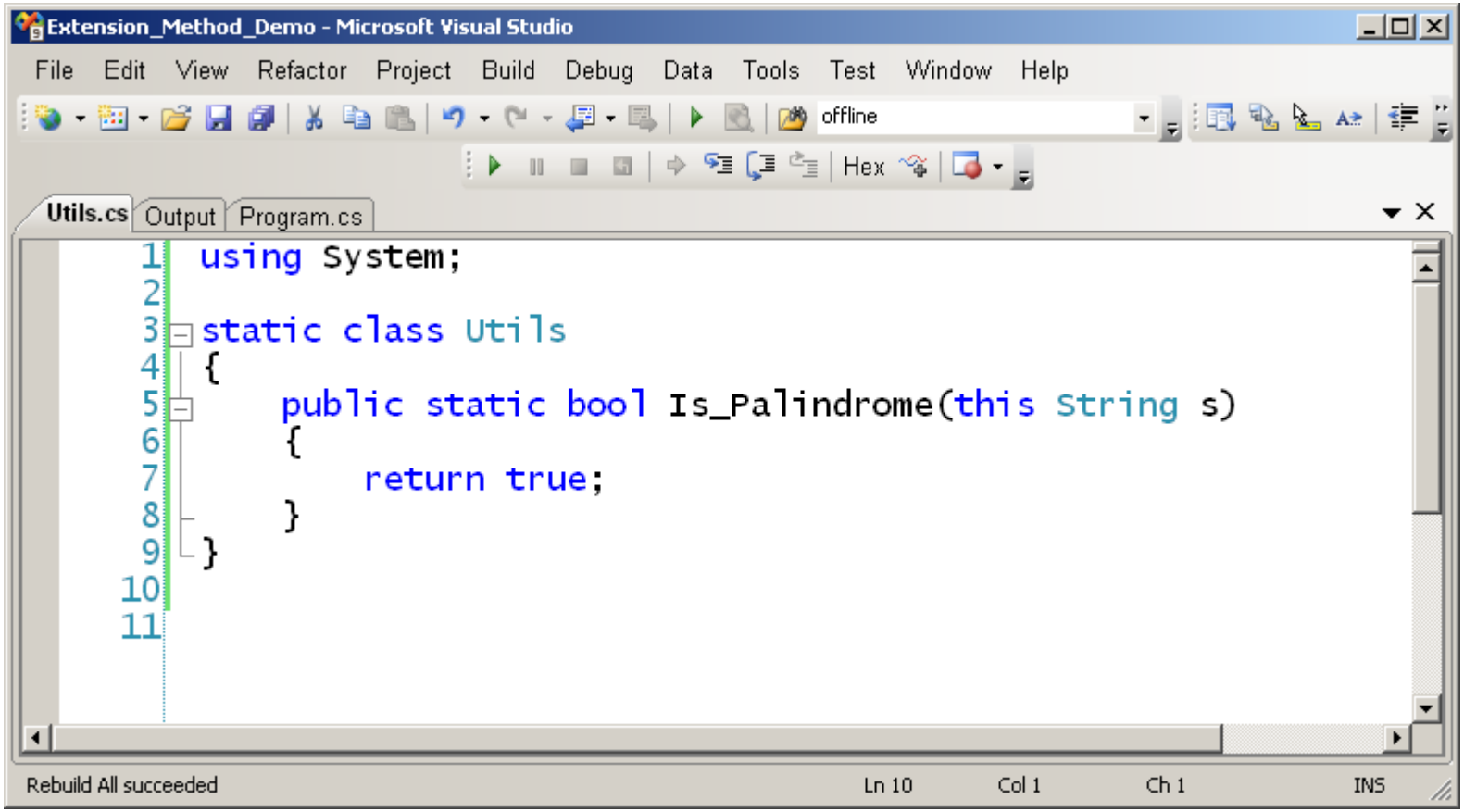
- Let's extend the String class with a method that says whether or not a given string is a palendrome.
  - Same letters forward and backward.
  - Ignoring case and non-letter characters.

- 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



An Extension Method



Utils.cs

```
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.

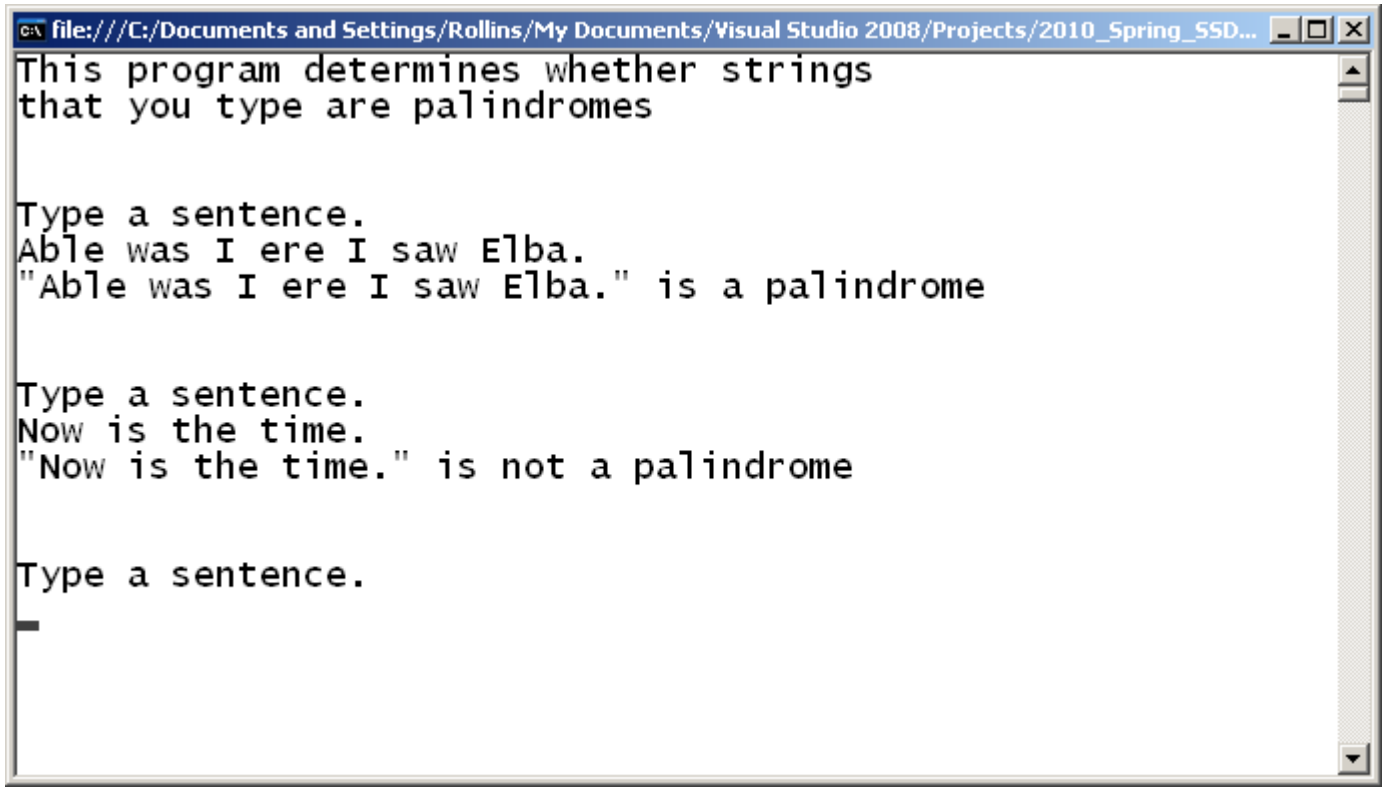
Utils.cs continued

```
        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;
    }
}
```

Program.cs

```
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.
}
```

Program in Action



The *as* Operator

- The as operator tries to convert a variable to a specified type; if no such conversion is possible the result is null
- More efficient than using `is` operator
  - Can test and convert in one operation

```
private static void DoSomething(object o) {
  Car c = o as Car;
  if (c != null) c.Drive();
}
```

The *is* Operator

- The is operator is used to dynamically test if the run-time type of an object is compatible with a given type

```
private static void DoSomething(object o) {
  if (o is Car)
    ((Car)o).Drive();
}
```

Classes and Structs
`typeof` Operator

- The `typeof` operator returns the `System.Type` object for a specified type
- Can then use reflection to dynamically obtain information about the type

```
Console.WriteLine(typeof(int).FullName);
Console.WriteLine(typeof(System.Int).Name);
Console.WriteLine(typeof(float).Module);
Console.WriteLine(typeof(double).IsPublic);
Console.WriteLine(typeof(Car).MemberType);
```