Chapter 11 – [Object-Oriented Programming] Polymorphism, Interfaces & Operator Overloading

Many slides modified by Prof. L. Lilien (even many without explicit message).

11.1. Introduction

- Polymorphism
  - Poly = many , morph = form
  - Polymorphic = having many forms
- Example:
  - Base class: `Quadrilateral`
  - Derived classes: `Rectangle, Square, Trapezoid`
  - Polymorhpic "`Quadrilateral.DrawYourself`" method
    - 3 ("many") forms of `DrawYourself` for 3 classes derived from `Quadrilateral`
  - "`Quadrilateral.DrawYourself`" morphs (re-forms, reshapes) to fit the class of the object for which it is called
    - Becomes `Quadrilateral.Rectangle` for a `Rectangle` object
    - Becomes `Quadrilateral.Square` for a `Square` object
    - Becomes `Quadrilateral.Trapezoid` for a `Trapezoid` object

Slide added by L. Lilien

- Polymorphism
  - allows programmers to write:
  - Programs that handle a wide variety of related classes in a generic manner
  - Systems that are easily extensible

[10.2 in textbook ed.1]
Derived-Class-Object to Base-Class-Object Conversion

- Class hierarchies
  - Can assign derived-class objects to base-class references
    - A fundamental part of programs that process objects polymorphically
  - Can explicitly cast between types in a class hierarchy
- An object of a derived-class can be treated as an object of its base-class

  (The reverse is NOT true, i.e.: base-class object is NOT an object of any of its derived classes)
  - Can have arrays of base-class references that refer to objects of many derived-class types (we'll see in 10.4 & 10.6)

Point.cs

```
// Fig. 10.1: Point.cs [in textbook ed.1]
// Point class represents an x-y coordinat

using System;

// Point class definition implicitly inherits from Object
public class Point
{
   // point coordinate
   private int x, y;

   // default constructor
   public Point()
   {
      // implicit call to Object constructor occurs here
   }

   // constructor
   public Point( int xValue, int yValue )
   {
      // implicit call to Object constructor occurs here
      X = xValue;
      Y = yValue;
   }

   // property X
   public int X
   {
      get
      {
         return x;
      }

```

Definition of class Point

Point.cs

```
      set
      {
         x = value;  // no need for validation
      }

   } // end property X

   // property Y
   public int Y
   {
      get
      {
         return y;
      }

      set
      {
         y = value;  // no need for validation
      }

   } // end property Y

   // return string representation of Point
   public override string ToString()
   {
      return "[" + X + ", " + Y + "]";
   }

} // end class Point
```

Circle.cs

```
// Fig. 10.2: Circle.cs [in textbook ed.1]
// Circle class that inherits f

using System;

// Circle class definition inherits from Point
public class Circle : Point
{
   private double radius; // circle's radius

   // default constructor
   public Circle()
   {
      // implicit call to Point constructor occurs here
   }

   // constructor
   public Circle( int xValue, int yValue, double radiusValue )
      : base( xValue, yValue )
   {
      Radius = radiusValue;
   }

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

```

Definition of class Circle which inherits from class Point

Outline

Circle.cs

```
         set
         {
            if ( value >= 0 ) // validate radius
               radius = value;
         }

      } // end property Radius

      // calculate Circle diameter
      public double Diameter()
      {
         return Radius * 2;
      }

      // calculate Circle circumference
      public double Circumference()
      {
         return Math.PI * Diameter();
      }

      // calculate Circle area
      public virtual double Area()
      {
         return Math.PI * Math.Pow( Radius, 2 );
      }

      // return string representation of Circle
      public override string ToString()
      {
         return "Center = " + base.ToString() +
            "; Radius = " + Radius;
      }

   } // end class Circle
```

Outline

PointCircleTest.cs

```
// Fig. 10.3: PointCircleTest.cs [in textbook ed.1]
// Demonstrating inheritance and polymorphism.

using System;
using System.Windows.Forms;

// PointCircleTest class definition
class PointCircleTest
{
   // main entry point for application.
   static void Main( string[] args )
   {
      Point point1 = new Point( 30, 50 );
      Circle circle1 = new Circle( 120, 89,

      string output = "Point point1: " + po
         "\nCircle circle1: " + circle1.ToS

      // use 'is a' relationship to assign
      // Circle circle1 to Point reference
      Point point2 = circle1;

      output += "\n\nCCircle circle1 (via p
         point2.ToString(); // Circle's ToS
  // ToString, since point2 references now a
 // (Object type determines method version, not reference type.)

      // downcast (cast base-class reference to derived-class
      // data type) point2 to Circle circle2
      Circle circle2 = ( Circle ) point2;

      output += "\n\nCircle circle1 (via circle2 [and point2]): " +
         circle2.ToString();

      output += "\nArea of circle1 (via circle2): " +
         circle2.Area().ToString( "F" );
```

Create a Point object

Creat

Assign `Circle` `circle1` (derived-class object) to `Point`

Casts ("downcasts") `P` object `point2` (which references a `Circle` object `circle1`) to a `Circle` and then assigns the result to the `Circle` reference `circle2`. (Cast would be dangerous if `point2` were referencing a `Point`! It is OK since `point2` references a `Circle`)

Use base-class reference to access a derived-class object (see slide 11)

Slide modified by L. Lilien

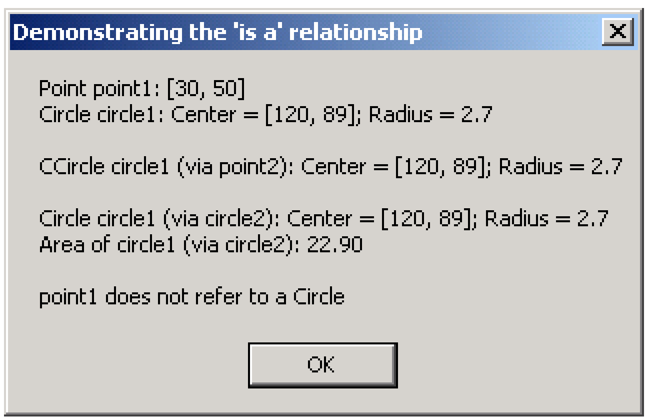
Outline

PointCircleTest.cs

```
36         // attempt to assign point1 object to Circle reference
37         if ( point1 is Circle )
38         {
39            circle2 = ( Circle ) point1; // Incorrect! Objects may be
              // cast only to their own type or their base-class types.
              // Here, point1's own type is Point, and Circle
              // is not a base-class type for Point.
40            output += "\n\ncast successful";
41         }
42         else
43         {
44            output += "\n\npoint1 does not refer to a Circle";
45         }
46
47         MessageBox.Show( output,
48            "Demonstrating the 'is a' relatio
49
50      } // end method Main
51
52   } // end class PointCircleTest
```

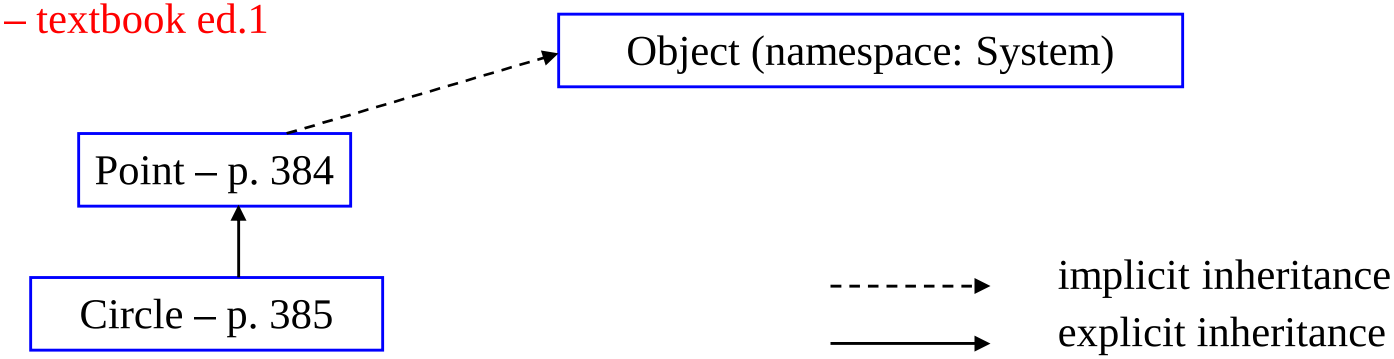
Test if point1 references a Circle object – it does not since `point1` references a `Point` object (l.13)

Program Output



Slide modified by L. Lilien

NOTE: All pages – textbook ed.1



Notes on PointCircleTest (p. 387):

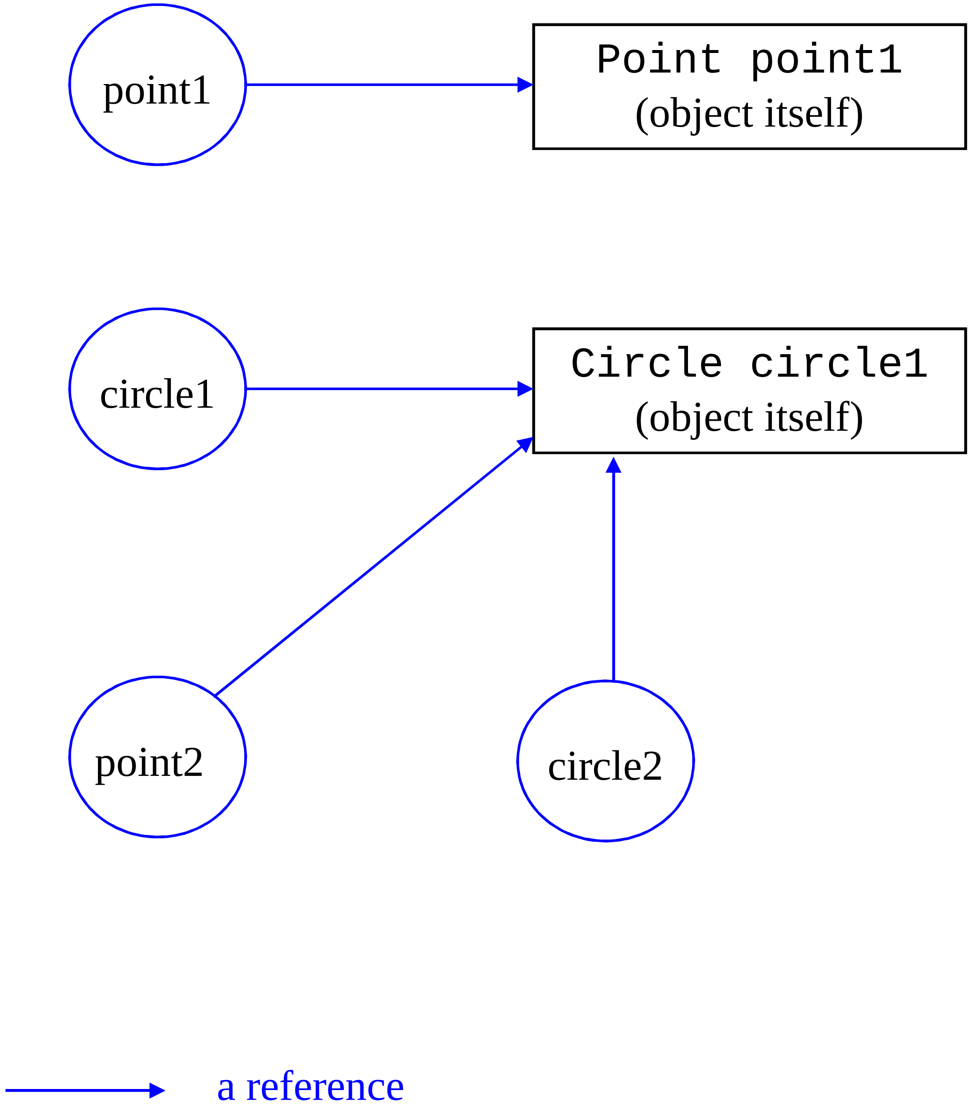
1) l.21: **`Point point2 = circle1;`**
- uses 'is a' relationship to assign `Circle circle1` (derived-class object) to `Point point2` reference (base-class object reference).

2) l.24: **`point2.ToString();`**
- Circle's ToString() method called, not Point's ToString(), since point2 references at that point a Circle object (circle1).
Object type, not reference type, determines method version. This is an example of **polymorphism**.

3) l.28: **`Circle circle 2 = (Circle) point2;`**
– casts ("downcasts") `Point` object `point2` (which references a `Circle` object `circle1`) to a `Circle` and then assigns the result to the `Circle` reference `circle2.`
Cast would be dangerous if `point2` were referencing a Point! (but it references a Circle – see l.21)

4) l.37: **`if ( point1 is Circle )`**
- tests if the object referenced by `point1` is-a Circle (i.e., is an object of the `Circle` class)

5) l.37: **`circle2 = ( Circle ) point1;`**
- Incorrect! Objects may be cast only to their own type or the base-class types.
Here, `point1`'s own type is Point, and Circle is not a base-class type for Point. Line 37 would cause an execution error if it were executed. But it is never executed (do you know why?).


point1
Point point1
(object itself)
circle1
Circle circle1
(object itself)
point2
circle2
a reference

[10.3 in textbook ed.1] Type Fields and switch Statements

- Using **switch** to determine the type of an object
  - Distinguish between object types, then perform appropriate action depending on object type
- Potential problems using **switch**
  - Programmer may forget to include a type test
  - Programmer may forget to test all possible cases in a switch
  - When new types are added, programmer may forget to modify all relevant **switch** structures
  - Every addition or deletion of a class requires modification of every **switch** statement determining object types in the system; tracking this is time consuming and error prone

11.2. Polymorphism Examples (Example 1)

- **Quadrilateral** base-class contains method `perimeter`
  - **Rectangle** derived-class (implements `perimeter`)
  - **Square** derived-class (implements `perimeter`)
  - **Parallelogram** derived-class (implements `perimeter`)
  - **Trapezoid** derived-class (implements `perimeter`)

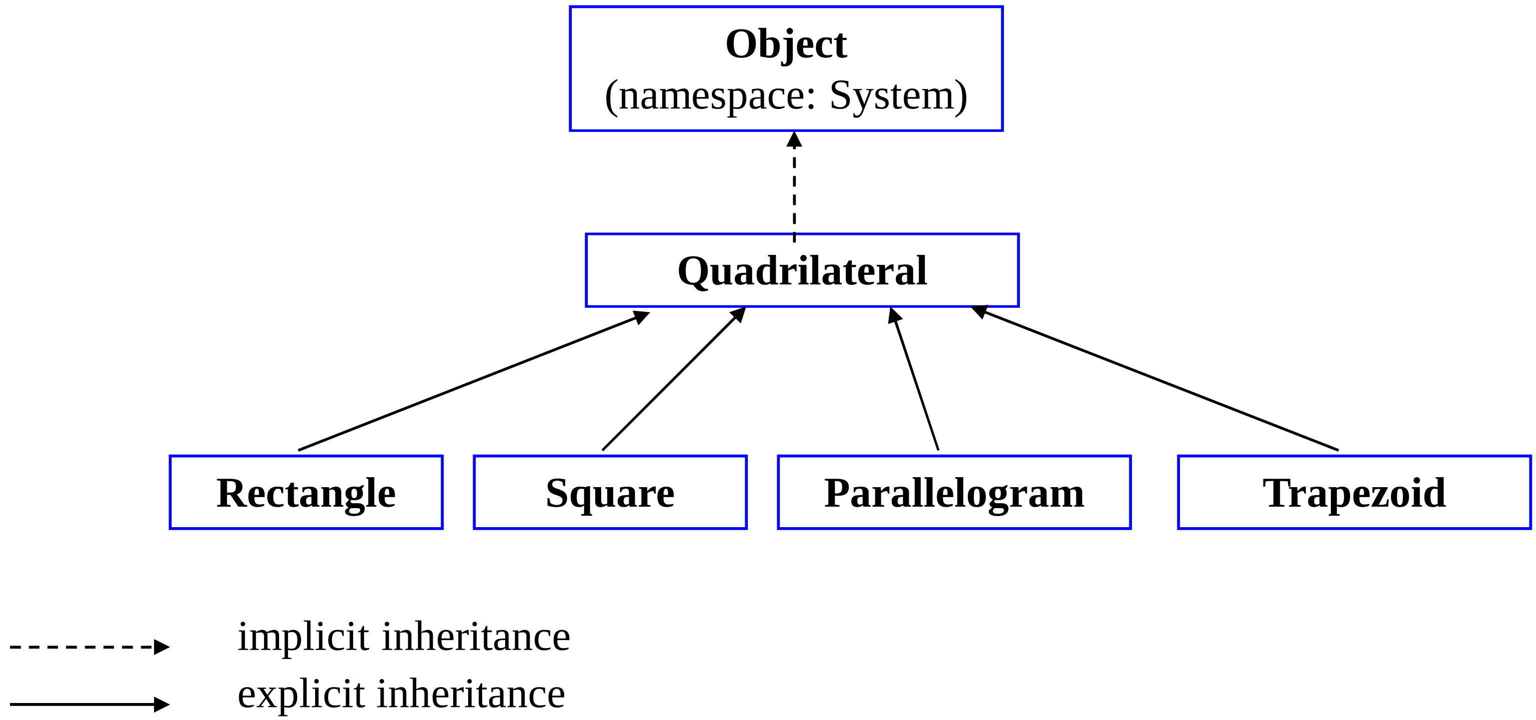


Figure added by L. Lilien

- Any other operation (e.g., **`perimeter`**) that can be performed on a **Quadrilateral** object can also be performed on a **Rectangle, Square, Parallelogram**, or **Trapezoid** object
- Suppose that: [added by L. Lilien]
  - Program P instantiated a **Rectangle/ Square/…** object referenced via `rectangle1/square1/…`:
    - `Rectangle rectangle1  = new Rectangle (…);`
      `Square square1 = new Square (…);`
  - P can use a base-class reference `quad1` to invoke derived-class methods `Rectangle.perimeter, Square.perimeter, …`
    -
```
Quadrilateral  quad1 = rectangle1;
… quad1.perimeter …   // Method perimeter is executed
          // on the derived-class object rectangle1 via the
          // base-class reference quad1.perimeter
Quadrilateral  quad1 = square1;
… quad1.perimeter …   // Method perimeter is executed
          // on the derived-class object square1 via the
          // same base-class reference quad1.perimeter
```
  - With such a **Quadrilateral** reference, C# **polymorphically** chooses the correct overriding method in any of these derived-classes from which the object is instantiated

11.2 Polymorphism Examples (Example 2)

- **SpaceObject** base-class – contains method **DrawYourself**
  - **Martian** derived-class (implements **DrawYourself**)
  - **Venutian** derived-class (implements **DrawYourself**)
  - **Plutonian** derived-class (implements **DrawYourself**)
  - **SpaceShip** derived-class (implements **DrawYourself**)

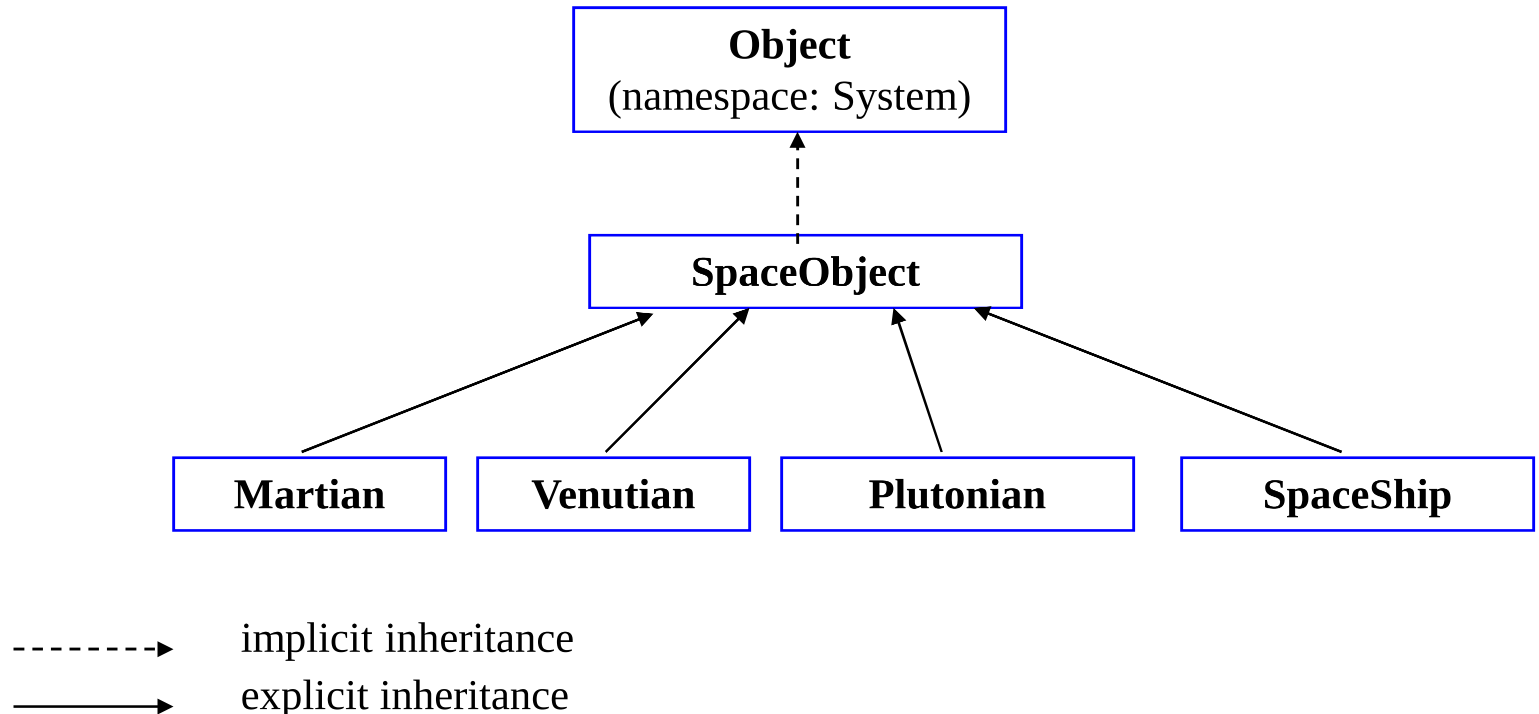


Figure added by L. Lilien

- A screen-manager program may contain a **SpaceObject** array of references to objects of various classes that derive from **SpaceObject**

Listing added by L. Lilien

  - ```
    arrOfSpaceObjects[ 0 ] = martian1;
    arrOfSpaceObjects[ 1 ] = martian2;
    arrOfSpaceObjects[ 2 ] = venutian1;
    arrOfSpaceObjects[ 3 ] = plutonian1;
    arrOfSpaceObjects[ 4 ] = spaceShip1;
    …
    ```

- To refresh the screen, the screen-manager calls **DrawYourself** on each object in the array

Listing added by L. Lilien

  - ```
    foreach(SpaceObject spaceObject in arrOfSpaceObjects)
    {
      spaceObject.DrawYourself
    }
    ```

- The program **polymorphically** calls the appropriate version of **DrawYourself** on each object, based on the type of that object

11.4. Abstract Classes and Methods

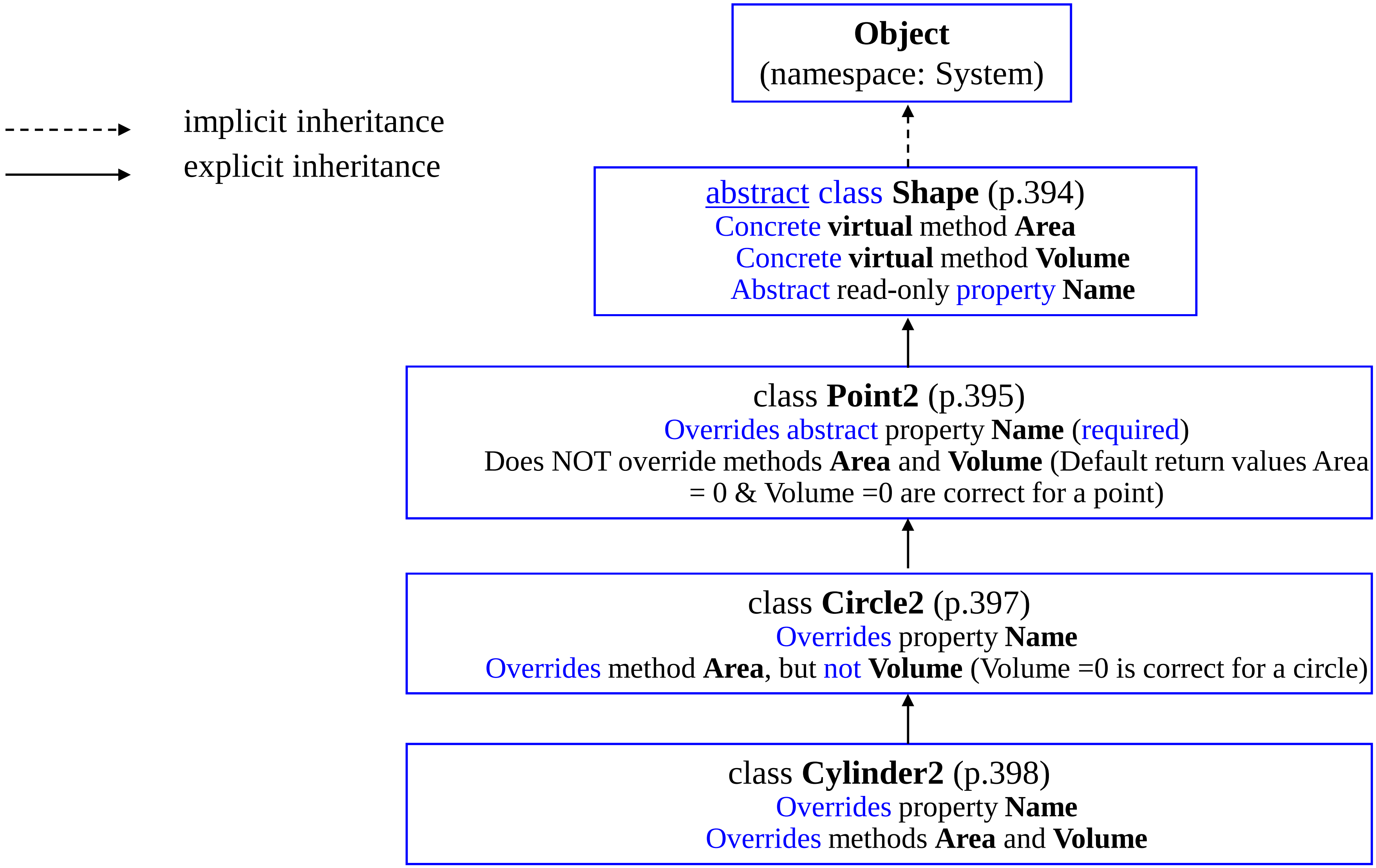
- Abstract classes
  - Cannot be instantiated
  - Used as base classes
  - Class definitions are not complete
    - Derived classes must define the missing pieces
  - Can contain abstract methods and/or abstract properties
    - Have no implementation
    - Derived classes must override inherited abstract methods and abstract properties to enable instantiation
      - Abstract methods and abstract properties are implicitly virtual (p.395/1)

- An abstract class is used to provide an appropriate base class from which other classes may inherit (concrete classes)
- Abstract base classes are too generic to define (by instantiation) real objects
- To define an abstract class, use keyword **abstract** in the declaration
- To declare a method or property abstract, use keyword **abstract** in the declaration
- Abstract methods and properties have no implementation

- Concrete classes use the keyword **override** to provide implementations for all the abstract methods and properties of the base-class
- Any class with an abstract method or property must be declared **abstract**
- Even though abstract classes cannot be instantiated, we can use abstract class references to refer to instances of any concrete class derived from the abstract class

[10.6 in textbook ed.1]
Case Study: Inheriting from an Abstract Class & Implementation [see Fig. - next slide]

- Abstract base class **Shape**
  - Concrete **virtual** method **Area** (default return value is 0)
  - Concrete **virtual** method **Volume** (default return value is 0)
  - Abstract read-only property **Name**
- Class **Point2** inherits from **Shape**
  - Overrides abstract property **Name** (required)
  - Does NOT override methods **Area** and **Volume**
    - Area = 0 and Volume =0 are correct for a point
- Class **Circle2** inherits from **Point2**
  - Overrides property **Name**
  - Overrides method **Area**, but not **Volume**
    - Volume =0 is correct for a circle
- Class **Cylinder2** inherits from **Circle2**
  - Overrides property **Name**
  - Overrides methods **Area** and **Volume**


implicit inheritance
explicit inheritance
Object
(namespace: System)
abstract class Shape (p.394)
Concrete virtual method Area
Concrete virtual method Volume
Abstract read-only property Name
class Point2 (p.395)
Overrides abstract property Name (required)
Does NOT override methods Area and Volume (Default return values Area = 0 & Volume =0 are correct for a point)
class Circle2 (p.397)
Overrides property Name
Overrides method Area, but not Volume (Volume =0 is correct for a circle)
class Cylinder2 (p.398)
Overrides property Name
Overrides methods Area and Volume

```
// Fig. 10.4: Shape.cs [in textbook ed.1]
// Demonstrate a shape hierarchy using an abstract base class.
using System;

public abstract class Shape
{
   // return Shape's area
   public virtual double Area()
   {
      return 0;
   }

   // return Shape's volume
   public virtual double Volume()
   {
      return 0;
   }

   // return Shape's name
   public abstract string Name
   {
      get;
   }
}
```

Declaration of virtual methods Area and Volume with default implementations

Declaration of read-only (`get`) abstract property Name; implementing classes will have to provide an implementation for this property

Declaration of abstract class Shape

Point2.cs

```
// Fig. 10.5: Point2.cs [in textbook ed.1]
// Point2 inherits from abstract class Shape and represents
// an x-y coordinate pair.
using System;

// Point2 inherits from abstract class Shape
public class Point2 : Shape
{
   private int x, y; // Point2 coordinates

   // default constructor
   public Point2()
   {
      // implicit call to Object constructor occurs here
   }

   // constructor
   public Point2( int xValue, int yValue )
   {
      X = xValue;
      Y = yValue;
   }

   // property X
   public int X
   {
      get
      {
         return x;
      }

      set
      {
         x = value; // no validation needed
      }
```

Class Point2 inherits from class Shape

Point2.cs

```
   }

   // property Y
   public int Y
   {
      get
      {
         return y;
      }

      set
      {
         y = value; // no validation needed
      }
   }

   // return string representation of Point2 object
   public override string ToString()
   {
      return "[" + X + ", " + Y + "]";
   }

   // implement abstract property Name of class Shape
   public override string Name
   {
      get
      {
         return "Point2";
      }
   }

} // end class Point2
```

Point2's implementation of the read-only Name property

Circle2.cs

```
// Fig. 10.6: Circle2.cs [in textbook ed.1]
// Circle2 inherits from class Point2 and overrides key members.
using System;

// Circle2 inherits from class Point2
public class Circle2 : Point2
{
   private double radius; // Circle2 radius

   // default constructor
   public Circle2()
   {
      // implicit call to Point2 constructor occurs here
   }

   // constructor
   public Circle2( int xValue, int yValue, double radiusValue )
      : base( xValue, yValue )
   {
      Radius = radiusValue;
   }

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

```

Definition of class Circle2 which inherits from class Point2

Outline

Circle2.cs

```
         set
         {
            // ensure non-negative radius value
            if ( value >= 0 )
               radius = value;
         }
      }

      // calculate Circle2 diameter
      public double Diameter()
      {
         return Radius * 2;
      }

      // calculate Circle2 circumference
      public double Circumference()
      {
         return Math.PI * Diameter();
      }

      // calculate Circle2 area
      public override double Area()
      {
         return Math.PI * Math.Pow( Radius, 2 );
      }

      // return string representation of Circle2 object
      public override string ToString()
      {
         return "Center = " + base.ToString() +
            "; Radius = " + Radius;
      }
```

Override the Area method (defined in class Shape)

Circle2.cs

```

    // override property Name from class Point2
    public override string Name
    {
       get
       {
          return "Circle2";
       }
    }

 } // end class Circle2
```

Override the read-only Name property

Cylinder2.cs

```
// Fig. 10.7: Cylinder2.cs [in textbook ed.1]
// Cylinder2 inherits from class Circle2 and overrides key members.
using System;

// Cylinder2 inherits from class Circle2
public class Cylinder2 : Circle2
{
   private double height; // Cylinder2 height

   // default constructor
   public Cylinder2()
   {
      // implicit call to Circle2 constructor occurs here
   }

   // constructor
   public Cylinder2( int xValue, int yValue, double radiusValue,
      double heightValue ) : base( xValue, yValue, radiusValue )
   {
      Height = heightValue;
   }

   // property Height
   public double Height
   {
      get
      {
         return height;
      }

      set
      {
         // ensure non-negative height value
         if ( value >= 0 )
            height = value;
```

Class Cylinder2 derives from Circle2

```
         }
      }

      // calculate Cylinder2 area
      public override double Area()
      {
         return 2 * base.Area() + base.Circumference() * Height;
      }

      // calculate Cylinder2 volume
      public override double Volume()
      {
         return base.Area() * Height;
      }

      // return string representation of Circle2 object
      public override string ToString()
      {
         return base.ToString() + "; Height = " + Height;
      }

      // override property Name from class Circle2
      public override string Name
      {
         get
         {
            return "Cylinder2";
         }
      }

   } // end class Cylinder2
```

Override read-only property Name

Override Area implementation of class Circle2

Override Volume implementation of class Shape

```
// Fig. 10.8: AbstractShapesTest.cs [in textbook ed.1]
// Demonstrates polymorphism in Point-Circle-Cylinder hierarchy.
using System;
using System.Windows.Forms;

public class AbstractShapesTest
{
   public static void Main( string[] args )
   {
      // instantiate Point2, Circle2 and Cylin
      Point2 point = new Point2( 7, 11 );
      Circle2 circle = new Circle2( 22, 8, 3.5 );
      Cylinder2 cylinder = new Cylinder2( 10, 10, 3.3, 10 );

      // create empty array of Shape base-class references
      Shape[] arrayOfShapes = new Shape[ 3 ];

      // arrayOfShapes[ 0 ] refers to Point2 object
      arrayOfShapes[ 0 ] = point;

      // arrayOfShapes[ 1 ] refers to Circle2 object
      arrayOfShapes[ 1 ] = circle;

      // arrayOfShapes[ 1 ] refers to Cylinder2 object
      arrayOfShapes[ 2 ] = cylinder;

      string output = point.Name + ": " + point + "\n" +
         circle.Name + ": " + circle + "\n" +
         cylinder.Name + ": " + cylinder;

```

Assign a Shape reference to reference a Point2 object

bjects

Assign a Shape reference to reference a Cylinder2 object

reference a Circle2 object

Slide modified by L. Lilien

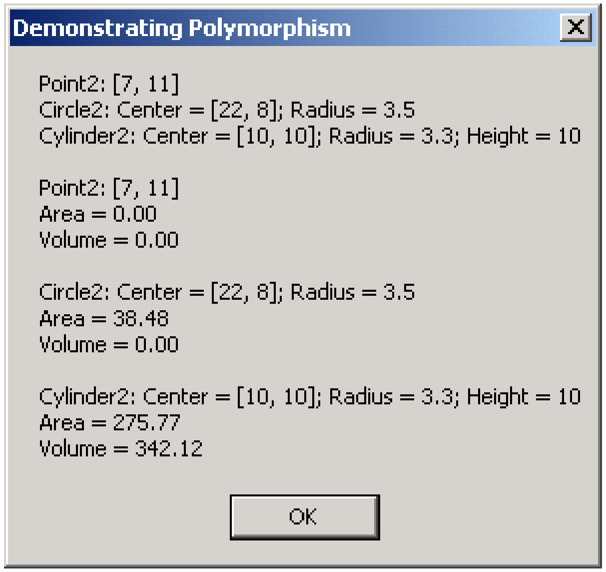
AbstractShapesTest.cs

```
      // display Name, Area and Volume for each object
      // in arrayOfShapes polymorphically
      foreach( Shape shape in arrayOfShapes )
      {
         output += "\n\n" + shape.Name + ": " + shape +
            "\nArea = " + shape.Area().ToString( "F" ) +
            "\nVolume = " + shape.Volume().ToString( "F" );
      }

      MessageBox.Show( output, "Demonstrating Polymorphism" );
   }
}
```

Use a foreach loop to access every element of the array

Rely on polymorphism to call appropriate version of methods

Program Output



11.6. `sealed` Methods and Classes

- **`sealed`** is a keyword in C#
- **`sealed`** methods and **`sealed`** classes
  1) **`sealed`** methods cannot be overridden in a derived class
     - Methods that are declared **`static`** or **`private`** are implicitly sealed
  2) **`sealed`** classes cannot have any derived-classes
     - Creating **`sealed`** classes can allow some runtime optimizations
       – e.g., **virtual** method calls can be transformed into non-**virtual** method calls

Slide modified by L. Lilien

STUDY ON YOUR OWN:

11.5. Case Study: Payroll System Using Polymorphism

- Base-class **Employee**
  - **abstract**
  - **abstract** method **Earnings**
- Classes that derive from Employee
  - **Boss**
  - **CommissionWorker**
  - **PieceWorker**
  - **HourlyWorker**
- All derived-classes implement method **Earnings**
- Driver program uses **Employee** base-class references to refer to instances of derived-classes
- Polymorphically calls the correct version of **Earnings**

Employee.cs

```
// Fig. 10.9: Employee.cs [in textbook ed.1]
// Abstract base class for company employees.
using System;

public abstract class Employee
{
   private string firstName;
   private string lastName;

   // constructor
   public Employee( string firstNameValue,
      string lastNameValue )
   {
      FirstName = firstNameValue;
      LastName = lastNameValue;
   }

   // property FirstName
   public string FirstName
   {
      get
      {
         return firstName;
      }

      set
      {
         firstName = value;
      }
   }

```

Definition of abstract class Employee

Employee.cs

```
   // property LastName
   public string LastName
   {
      get
      {
         return lastName;
      }

      set
      {
         lastName = value;
      }
   }

   // return string representation of Employee
   public override string ToString()
   {
      return FirstName + " " + LastName;
   }

   // abstract method that must be implemented for each derived
   // class of Employee to calculate specific earnings
   public abstract decimal Earnings();

} // end class Employee
```

Declaration of abstract class Earnings – implementation must be provided by all derived classes

Outline

Boss.cs

```
// Fig. 10.10: Boss.cs [in textbook ed.1]
// Boss class derived from Employee.
using System;

public class Boss : Employee
{
   private decimal salary; // Boss's salary

   // constructor
   public Boss( string firstNameValue, string lastNameValue,
      decimal salaryValue)
      : base( firstNameValue, lastNameValue )
   {
      WeeklySalary = salaryValue;
   }

   // property WeeklySalary
   public decimal WeeklySalary
   {
      get
      {
         return salary;
      }

      set
      {
         // ensure positive salary value
         if ( value > 0 )
            salary = value;
      }
   }

```

Definition of class Boss – derives from Employee

Outline

Boss.cs

```
   // override base-class method to calculate Boss's earnings
   public override decimal Earnings()
   {
      return WeeklySalary;
   }

   // return string representation of Boss
   public override string ToString()
   {
      return "Boss: " + base.ToString();
   }
}
```

Implementation of abstract base-class method Earnings (required by classes deriving from Employee)

CommisionWorker.cs

```
// Fig. 10.11: CommisionWorker.cs [in textbook ed.1]
// CommissionWorker class derived from Employee
using System;

public class CommissionWorker : Employee
{
   private decimal salary;     // base weekly salary
   private decimal commission; // amount paid per item sold
   private int quantity;       // total items sold

   // constructor
   public CommissionWorker( string firstNameValue,
      string lastNameValue, decimal salaryValue,
      decimal commissionValue, int quantityValue )
      : base( firstNameValue, lastNameValue )
   {
      WeeklySalary = salaryValue;
      Commission = commissionValue;
      Quantity = quantityValue;
   }

   // property WeeklySalary
   public decimal WeeklySalary
   {
      get
      {
         return salary;
      }

```

Definition of class CommissionWorker – derives from Employee

Outline

CommisionWorker.cs

```
         set
         {
            // ensure non-negative salary value
            if ( value > 0 )
               salary = value;
         }
      }

      // property Commission
      public decimal Commission
      {
         get
         {
            return commission;
         }

         set
         {
            // ensure non-negative commission value
            if ( value > 0 )
               commission = value;
         }
      }

      // property Quantity
      public int Quantity
      {
         get
         {
            return quantity;
         }

```

CommisionWorker.cs

```
         set
         {
            // ensure non-negative quantity value
            if ( value > 0 )
               quantity = value;
         }
      }

      // override base-class method to calculate
      // CommissionWorker's earnings
      public override decimal Earnings()
      {
         return WeeklySalary + Commission * Quantity;
      }

      // return string representation of CommissionWorker
      public override string ToString()
      {
         return "CommissionWorker: " + base.ToString();
      }

   } // end class CommissionWor
```

Implementation of method Earnings (required by classes deriving from Employee)

```
// Fig. 10.12: PieceWorker.cs [in textbook ed.1]
// PieceWorker class derived from Employee.
using System;

public class PieceWorker : Employee
{
   private decimal wagePerPiece; // wage per piece produced
   private int quantity;         // quantity of pieces produced

   // constructor
   public PieceWorker( string firstNameValue,
      string lastNameValue, decimal wagePerPieceValue,
      int quantityValue )
      : base( firstNameValue, lastNameValue )
   {
      WagePerPiece = wagePerPieceValue;
      Quantity = quantityValue;
   }

   // property WagePerPiece
   public decimal WagePerPiece
   {
      get
      {
         return wagePerPiece;
      }

      set
      {
         if ( value > 0 )
            wagePerPiece = value;
      }
   }

```

Definition of class PieceWorker – derives from Employee

PieceWorker.cs

PieceWorker.cs

```
      // property Quantity
      public int Quantity
      {
         get
         {
            return quantity;
         }

         set
         {
            if ( value > 0 )
               quantity = value;
         }
      }

      // override base-class method to calculate
      // PieceWorker's earnings
      public override decimal Earnings()
      {
         return Quantity * WagePerPiece;
      }

      // return string representation of PieceWorker
      public override string ToString()
      {
         return "PieceWorker: " + base.ToString();
      }
   }
```

Implementation of method Earnings (required by classes deriving from Employee)

HourlyWorker.cs

```
// Fig. 10.13: HourlyWorker.cs [in textbook ed.1]
// HourlyWorker class derived from Employee.
using System;

public class HourlyWorker : Employee
{
   private decimal wage;       // wage per hour of work
   private double hoursWorked; // hours worked during week

   // constructor
   public HourlyWorker( string firstNameValue, string LastNameValue,
      decimal wageValue, double hoursWorkedValue )
      : base( firstNameValue, LastNameValue )
   {
      Wage = wageValue;
      HoursWorked = hoursWorkedValue;
   }

   // property Wage
   public decimal Wage
   {
      get
      {
         return wage;
      }

      set
      {
         // ensure non-negative wage value
         if ( value > 0 )
            wage = value;
      }
   }

```

Definition of class HourlyWorker – derives from Employee

HourlyWorker.cs

```
      // property HoursWorked
      public double HoursWorked
      {
         get
         {
            return hoursWorked;
         }

         set
         {
            // ensure non-negative hoursWorked value
            if ( value > 0 )
               hoursWorked = value;
         }
      }

      // override base-class method to calculate
      // HourlyWorker earnings
      public override decimal Earnings()
      {
         // compensate for overtime (paid "time-and-a-half")
         if ( HoursWorked <= 40 )
         {
            return Wage * Convert.ToDecimal( HoursWorked );
         }        // Above: clas

         else
         {
            // calculate base and overtime pay
            decimal basePay = Wage * Convert.ToDecimal( 40 );
            decimal overtimePay = Wage * 1.5M *
               Convert.ToDecimal( HoursWorked - 40 );

```

Implementation of method Earnings (required by classes deriving from Employee)

HourlyWorker.cs

```
            return basePay + overtimePay;
         }
      }

      // return string representation of HourlyWorker
      public override string ToString()
      {
         return "HourlyWorker: " + base.ToString();
      }
   }
```

Outline

EmployeesTest.cs

```
// Fig. 10.14: EmployeesTest.cs [in textbook ed.1]
// Demonstrates polymorphism by displaying earnings
// for various Employee types.
using System;
using System.Windows.Forms;

public class EmployeesTest
{
   public static void Main( string[] args )
   {
      Boss boss = new Boss( "John", "Smith",

      CommissionWorker commissionWorker =
         new CommissionWorker( "Sue", "Jones",

      PieceWorker pieceWorker = new PieceWorker( "Bob", "Lewis",
         Convert.ToDecimal( 2.5 ), 200 );

      HourlyWorker hourlyWorker = new HourlyWorker( "Karen",
         "Price", Convert.ToDecimal( 13.75 ), 50 );

      Employee employee = boss;
          // GetString used below is defined in Lines 51-55.
         // "C" produces default currency format (2 decimal places)
      string output = GetString( employee ) + boss + " earned " +
         boss.Earnings().ToString( "C" ) + "\n\n";
            // GetString() prepares first output line, the rest
           // produces identical second line - see output.
      employee = commissionWorker;

      output += GetString( employee ) + commissionWorker +
         " earned " +
         commissionWorker.Earnings().ToString( "C" ) + "\n\n";

      employee = pieceWorker;

```

Assign Employee reference to reference a Boss object

Use method GetString to polymorphically obtain salary information. Then use the original Boss reference to obtain the information

Outline

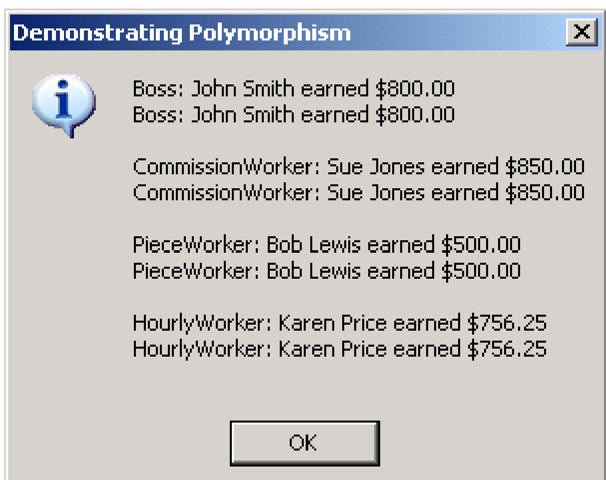
EmployeesTest.cs

```
         output += GetString( employee ) + pieceWorker +
            " earned " + pieceWorker.Earnings().ToString( "C" ) +
            "\n\n";

         employee = hourlyWorker;

         output += GetString( employee ) + hourlyWorker +
            " earned " + hourlyWorker.Earnings().ToString( "C" ) +
            "\n\n";

         MessageBox.Show( output, "De
            MessageBoxButtons.OK, Mes                          );

      } // end method Main

      // return string that contains Employee information
      public static string GetString( Employee worker )
      {
         return worker.ToString() + " earned " +
            worker.Earnings().ToString( "C" ) + "\n";
      }
   }
```

Definition of method GetString, which takes as an argument an Employee object.

Polymorphically call the method of the appropriate derived class

Program Output



11.7. Case Study: Creating and Using Interfaces

- Interfaces specify the public services (methods and properties) that classes must implement
- Interfaces vs. abstract classes w.r.t default implementations
  - Interfaces provide no default implementations
  - Abstract classes may provide some default implementations
    - If no default implementations can/are defined – do *not* use an abstract class, use an interface instead
- Interfaces are used to "bring together" or relate to each other disparate objects that relate to one another only through the interface
  - I.e., provide uniform set of methods and properties for disparate objects
  - E.g.: A person and a tree are disparate objects

    Interface can define age and name for these disparate objects
  - Enables polymorphic processing of age and name for person & tree objects

- Interfaces are defined using keyword **`interface`**
- Use inheritance notation to specify that a class implements an interface

  ClassName : *InterfaceName*
- Classes may implement more then one interface:

  e.g.: *ClassName* : *InterfaceName1*, *InterfaceName2*
  - Can also have:
    - *ClassName : ClassName1*, *ClassName2*, *InterfaceName1*, *InterfaceName2* (object list must precedes interface list)
- A class that implement an interface, must provide implementations for every method and property in the interface definition
- Example: interface **IAge** that returns information about an object's age
  - Can be used by classes for people, cars, trees (all have an age)

Outline

IAge.cs

```
// Fig. 10.15: IAge.cs [in textbook ed.1]
// Interface IAge declares property for setting and getting age.

public interface IAge
{
   int Age { get; }
   string Name { get; }
}
```

Definition of interface IAge

Classes implementing this interface will have to define read-only properties Age and Name

Person.cs

```
// Fig. 10.16: Person.cs [in textbook ed.1]
// Class Person has a birthday.
using System;

public class Person : IAge
{
   private string firstName;
   private string lastName;
   private int yearBorn;

   // constructor
   public Person( string firstNameValue, string lastNameValue,
      int yearBornValue )
   {
      firstName = firstNameValue;
      lastName = lastNameValue;

      if ( yearBornValue > 0 && yearBornValue <= DateTime.Now.Year )
         yearBorn = yearBornValue;
      else
         yearBorn = DateTime.Now.Year;
   }

   // property Age implementation of interface IAge
   public int Age
   {
      get
      {
         return DateTime.Now.Year - yearBorn;
      }
   }

```

Definition of Age property (required)

Class Person implements the IAge interface

Person.cs

```
    // property Name implementation of interface IAge
    public string Name
    {
       get
       {
          return firstName + " " + lastName;
       }
    }

 } // end class Person
```

Definition of Name property (required)

Outline

Tree.cs

```
// Fig. 10.17: Tree.cs [in textbook ed.1]
// Class Tree contains number of rings corresponding to its age.
using System;

public class Tree : IAge
{
   private int rings; // number of rings in tree trunk

   // constructor
   public Tree( int yearPlanted )
   {
      // count number of rings in Tree
      rings = DateTime.Now.Year - yearPlanted;
   }

   // increment rings
   public void AddRing()
   {
      rings++;
   }

   // property Age implementation of interface IAge
   public int Age
   {
      get
      {
         return rings;
      }
   }
```

Implementation of Age property (required)

Class Tree implements the IAge interface

Outline

Tree.cs

```

   // property Name implementation of interface IAge
   public string Name
   {
      get
      {
         return "Tree";
      }
   }

} // end class Tree
```

Definition of Name property (required)

InterfacesTest.cs

```
// Fig. 10.18: InterfacesTest.cs [in textbook ed.1]
// Demonstrating polymorphism with interfaces (on the objects of
// disparate classes Tree and Person).
using System.Windows.Forms;

public class InterfacesTest
{
   public static void Main( string[] args )
   {
      Tree tree = new Tree( 1978 );
      Person person = new Person( "Bob", "Jones", 1971 );

      // create array of IAge references
      IAge[] iAgeArray = new IAge[ 2 ];

      // iAgeArray[ 0 ] refers to Tree object polymorphically
      iAgeArray[ 0 ] = tree;

      // iAgeArray[ 1 ] refers to Person object polymorphically
      iAgeArray[ 1 ] = person;

      // display tree information
      string output = tree + ": " + tree.Name + "\nAge is " +
         tree.Age + "\n\n";

      // display person information
      output += person + ": " + person.Name + "\nAge is: "
         + person.Age + "\n\n";

```

Create array of IAge references

Assign an IAge reference to reference a Person object

reference a Tree object

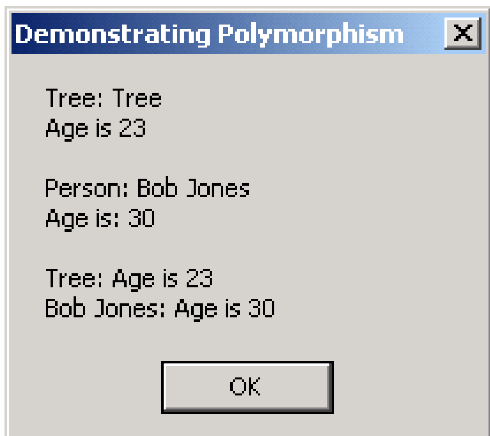
InterfacesTest.cs

```
         // display name and age for each IAge object in iAgeArray
         foreach ( IAge ageReference in iAgeArray )
         {
            output += ageReference.Name + ": Age is " +
               ageReference.Age + "\n";
         }

         MessageBox.Show( output, "Demonstrating Polymorphism" );

      } // end method Main

   } // end class InterfacesTest
```

Use foreach loop t
each element of the array

Use polymorphism to call the property of the appropriate class

Program Output



Outline

IShape.cs

```
// Fig. 10.19: IShape.cs [in textbook ed.1]
// Interface IShape for Point, Circle, Cylinder Hierarchy.

public interface IShape
{
   // classes that implement IShape must implement these methods
   // and this property
   double Area();
   double Volume();
   string Name { get; }
}
```

Definition of IShape interface

Classes implementing the interface must define methods Area and Volume (each of which take no arguments and return a double) and a read-only string property Name

Point3.cs

```
// Fig. 10.20: Point3.cs [in textbook ed.1]
// Point3 implements interface IShape and represents
// an x-y coordinate pair.
using System;

// Point3 implements IShape interface
public class Point3 : IShape
{
   private int x, y; // Point3 coordinates

   // default constructor
   public Point3()
   {
      // implicit call to Object constructor occurs here
   }

   // constructor
   public Point3( int xValue, int yValue )
   {
      X = xValue;
      Y = yValue;
   }

   // property X
   public int X
   {
      get
      {
         return x;
      }

```

Class Point3 implements the IShape interface

Outline

Point3.cs

```
        set
        {
           x = value;
        }
     }

     // property Y
     public int Y
     {
        get
        {
           return y;
        }

        set
        {
           y = value;
        }
     }

     // return string representation of Point3 object
     public override string ToString()
     {
        return "[" + X + ", " + Y + "]";
     }

     // implement interface IShape method Area
     public virtual double Area()
     {
        return 0;
     }

```

Implementation of IShape method Area (required), declared virtual to allow deriving classes to override

Outline

Point3.cs

```
   // implement interface IShape method Volume
   public virtual double Volume()
   {
      return 0;
   }

   // implement property Name of IShape interface
   public virtual string Name
   {
      get
      {
         return "Point3";
      }
   }

} // end class Point3
```

Implementation of IShape method Volume (required), declared virtual to allow deriving classes to override

Implementation of IShape property Name (required), declared virtual to allow deriving classes to override

Circle3.cs

```
// Fig. 10.21: Circle3.cs [in textbook ed.1]
// Circle3 inherits from class Point3 and overrides key members.
using System;

// Circle3 inherits from class Point3
public class Circle3 : Point3
{
   private double radius; // Circle3 radius

   // default constructor
   public Circle3()
   {
      // implicit call to Point3 constructor occurs here
   }

   // constructor
   public Circle3( int xValue, int yValue, double radiusValue )
      : base( xValue, yValue )
   {
      Radius = radiusValue;
   }

   // property Radius
   public double Radius
   {
      get
      {
         return radius;
      }

```

Definition of class Circle3 which inherits from Point3

Outline

Circle3.cs

```
         set
         {
            // ensure non-negative Radius value
            if ( value >= 0 )
               radius = value;
         }
      }

      // calculate Circle3 diameter
      public double Diameter()
      {
         return Radius * 2;
      }

      // calculate Circle3 circumference
      public double Circumference()
      {
         return Math.PI * Diameter();
      }

      // calculate Circle3 area
      public override double Area()
      {
         return Math.PI * Math.Pow( Radius, 2 );
      }

      // return string representation of Circle3 object
      public override string ToString()
      {
         return "Center = " + base.ToString() +
            "; Radius = " + Radius;
      }

```

Override the Point3 implementation of Area

```
    // override property Name from class Point3
    public override string Name
    {
       get
       {
          return "Circle3";
       }
    }

 } // end class Circle3
```

Override the Point3 implementation of Name

Circle3.cs

Cylinder3.cs

```
// Fig. 10.22: Cylinder3.cs [in textbook ed.1]
// Cylinder3 inherits from class Circle2 and overrides key members.
using System;

// Cylinder3 inherits from class Circle3
public class Cylinder3 : Circle3
{
   private double height; // Cylinder3 height

   // default constructor
   public Cylinder3()
   {
      // implicit call to Circle3 constructor occurs here
   }

   // constructor
   public Cylinder3( int xValue, int yValue, double radiusValue,
      double heightValue ) : base( xValue, yValue, radiusValue )
   {
      Height = heightValue;
   }

   // property Height
   public double Height
   {
      get
      {
         return height;
      }

```

Declaration of class Cylinder3 which inherits from class Circle3

Cylinder3.cs

```
         set
         {
            // ensure non-negative Height
            if ( value >= 0 )
               height = value;
         }
      }

      // calculate Cylinder3 area
      public override double Area()
      {
         return 2 * base.Area() + base.Circumference() * Height;
      }

      // calculate Cylinder3 volume
      public override double Volume()
      {
         return base.Area() * Height;
      }

      // return string representation of Cylinder3 object
      public override string ToString()
      {
         return "Center = " + base.ToString() +
            "; Height = " + Height;
      }

```

Override the Point3 implementation of Volume

Override the Circle3 implementation of Area

Cylinder3.cs

```
   // override property Name from class Circle3
   public override string Name
   {
      get
      {
         return "Cylinder3";
      }
   }

} // end class Cylinder3
```

Override the Circle3 implementation of Name

```
// Fig. 10.23: Interfaces2Test.cs [in textbook ed.1]
// Demonstrating polymorphism with interfaces in
// Point-Circle-Cylinder hierarchy.

using System.Windows.Forms;

public class Interfaces2Test
{
   public static void Main( string[] args )
   {
      // instantiate Point3, Circle3 and Cylin
      Point3 point = new Point3( 7, 11 );
      Circle3 circle = new Circle3( 22, 8, 3.5
      Cylinder3 cylinder = new Cylinder3( 10, 1

      // create array of IShape references
      IShape[] arrayOfShapes = new IShape[ 3 ];

      // arrayOfShapes[ 0 ] references Point3 object
      arrayOfShapes[ 0 ] = point;

      // arrayOfShapes[ 1 ] references Circle3 object
      arrayOfShapes[ 1 ] = circle;

      // arrayOfShapes[ 2 ] references Cylinder3 object
      arrayOfShapes[ 2 ] = cylinder;

      string output = point.Name + ": " + point + "\n" +
         circle.Name + ": " + circle + "\n" +
         cylinder.Name + ": " + cylinder;

```

Create an array of IShape references (IShape is an inter

Assign an IShape reference to reference a Point3 object

Assign an IShape reference to reference a Circle3 object

Assign an IShape reference to reference a Cylinder3 object

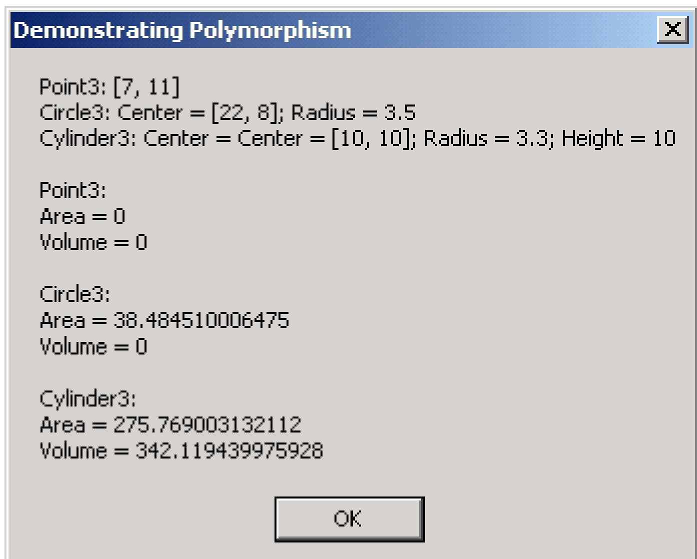
Outline

Interfaces2Test.cs

```
        foreach ( IShape shape in arrayOfShapes )
        {
           output += "\n\n\n" + shape.Name + ":\nArea = " +
              shape.Area() + "\nVolume = " + shape.Volume();
        }

        MessageBox.Show( output, "Demonstrating Polymorphism" );
     }
  }
```

Use polymorphism to call the appropriate class's method or property

Program Output



[10.10 in textbook ed.1] Delegates

- Sometimes useful to pass methods as arguments to other methods
- Example - sorting:
  - The same method can be used to sort in the ascending order and in the descending order
  - The only difference, when comparing elements:
    - swap them only if the first is larger than the second for ascending order (e.g., …, 9, 3, … => …, 3, 9,…)
    - swap them only if the first is smaller than the second for descending order (e.g., …, 4, 7, … => …, 7, 4,…)
- C# prohibits passing a method reference directly as an argument to another method. Must use delegates

  \- delegate = a class that <u>encapsulates</u> sets of references to methods

  – Analogy: Prof. Ninu Atluri requires that students submit homeworks in yellow envelopes
  She allows a few students to use a single yellow envelope, but does not accept any homeworks not "encapsulated" by an envelope.

  Prof. Atluri <--> "receiving method" (to which other methods' references are passed)

  homework <--> method reference (passed to another method)

  yellow envelope <--> delegate encapsulating references to passed methods

- Delegates must be declared before use
- Delegate declaration specifies:
  - the *parameter-list*
  - the return type of the methods the delegate can refer to

  - E.g. delegate Comparator:

    public delegate bool Comparator(**int** element1, **int** element2);

- Delegates (delegate objects) are sets of references to methods
  - E.g., delegate Comparator - a set of references to 2 methods:

    SortAscending and SortDescending

- Methods that can be referred to by a delegate, must have the same signature as the delegate
  - E.g.: **public delegate bool** Comparator(**int** el1, **int** el2);
    signature of Comparator is: (int, int) -> bool
  - Methods SortAscending or SortDescending referred to by Comparator must have the same signature ((int, int) -> bool)
    private bool SortAscending(**int** element1, **int** element2)
    private bool SortDescending(**int** element1, **int** element2)
  - Delegate instances can then be created to refer to the methods

- Delegates can be passed to methods
  - E.g., delegate Comparator can be passed to method SortArray
- Once a delegate instance is created (below: instance of delegate Comparator is created with **new**), the method it refers to (below: SortAscending) can be invoked by the method (below: SortArray) to which the delegate passed it
  - E.g.: DelegateBubbleSort.SortArray(elementArray,
    new DelegateBubbleSort.Comparator(SortAscending))
- The method to which a delegate passed methods can then invoke the methods the delegate object refers to
  - E.g., method SortArray can invoke method SortAscending to which delegate object Comparator refers to

- Example – sorting (continued from slide 70)
  - Declare delegate Comparator :

    public delegate bool Comparator(**int** element1, **int** element2);

    // delegate signature: **(int**, **int)** -> **bool**

  - Use delegate Comparator to pass the appropriate comparison methods (SortAscending or SortDescending) to method SortArray:
    - DelegateBubbleSort.SortArray(elementArray,

      new DelegateBubbleSort.Comparator(SortAscending)) [seen above]
    - DelegateBubbleSort.SortArray(elementArray,

      new DelegateBubbleSort.Comparator(SortDescending))

[The passed methods (SortAscending and SortDescending) have the same signatures ((int, int) -> bool) as delegate Comparator]

- Types of delegates
  - *singlecast delegate* - contains one method
    - created or derived from class **Delegate** (from System namespace)
  - *multicast delegate* - contains multiple methods
    - created or derived from class **MulticastDelegate** (from System namespace)
      - E.g., Comparator is a *singlecast delegate*
        - Bec. each instance contains a single method (either SortAscending or SortDescending)
- Delegates are very useful for event handling
  - We'll see how used for mouse click events
  - Used for event handling - Chapters 12-14
    - Not discussed in CS 1120

Outline

egateBubbleSo
s

```
// Fig. 10.24: DelegateBubbleSort.cs [in textbook ed.1]
// Demonstrating delegates for sorting numbers.

public class DelegateBubbleSort
{
   public delegate bool Comparator( in
      int element2 );  // Declare delegate Comparator, i.e., declare
                        // signature for the
                  // No implementation here

   // sort array using Comparator delegate
   public static void SortArray( int[] array,
      Comparator Compare )  // Reference to delegate Comparator
                             // passed as a parameter to SortArray method.
   {
      for ( int pass = 0; pass < array.Length

         for ( int i = 0; i < array.Length -

            if ( Compare( array[ i ], array [ i + 1 ] ) )
                    // if Compare returns true, elements are out of order
               Swap( ref array[ i ], ref array[ i + 1 ] );
   }

   // swap two elements
   private static void Swap( ref int firstElement,
      ref int secondElement )
   {
      int hold = firstElement;
      firstElement = secondElement;
      secondElement = hold;
   }
}
```

Call delegate method to compare array elements

Method Swap, swaps the two arguments (passed by reference)

Delegate Comparator definition declaration; defines a delegate to a method that takes two integer parameters and returns a boolean

Method SortArray which takes an integer array and a Comparator delegate

Outline

BubbleSortForm.cs

```
// Fig. 10.25: BubbleSortForm.cs [in textbook ed.1]
// Demonstrates bubble sort using delegates to determine
// the sort order.
   // Some thing will be magic for you (e.g. button click handling)
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;

public class BubbleSortForm : System.Windows.Forms.Form
{
   private System.Windows.Forms.TextBox originalTextBox;
   private System.Windows.Forms.TextBox sortedTextBox;
   private System.Windows.Forms.Button createButton; // 3 buttons
   private System.Windows.Forms.Button ascendingButton;
   private System.Windows.Forms.Button descendingButton;
   private System.Windows.Forms.Label originalLabel;
   private System.Windows.Forms.Label sortedLabel;

   private int[] elementArray = new int[ 10 ];

   // create randomly generated set of numbers to sort
   private void createButton_Click( object sender,
      System.EventArgs e )          // magic here
   {
      // clear TextBoxes
      originalTextBox.Clear();
      sortedTextBox.Clear();

      // create random-number generator
      Random randomNumber = new Random();

```

Slide modified by L. Lilien

BubbleSortForm.cs

```
         // populate elementArray with random integers
         for ( int i = 0; i < elementArray.Length; i++ )
         {
            elementArray[ i ] = randomNumber.Next( 100 );
            originalTextBox.Text += elementArray[ i ] + "\r\n";
         }
      }

      // delegate implementation for ascending sort
   // More precisely: implementation of SortAscending method which can b
   // passed to another method via Comparator delegate (matches its sign
      private bool SortAscending( int element1, int element2 )
      {
         return element1 > element2;
      }

      // sort randomly generated numbers in ascendi
      private void ascendingButton_Click( object sender,
         System.EventArgs e )          // magic here
      {
         // sort array, passing delegate for SortAsc
         DelegateBubbleSort.SortArray( elementArray,
            new DelegateBubbleSort.Comparator(
               SortAscending ) );

         DisplayResults();
      }

     // delegate implementation for desc. sort
   // More precisely: implementation of SortDescending method which can be
   // passed to another method via Comparator delegate (matches its signature)
      private bool SortDescending( int element1, int element2 )
      {
         return element1 < element2;
      }

```

To sort in ascending order, send a delegate for the SortAscending method to method SortArray

Method Sort... the first argument is smaller then the second; returns false otherwise

Method SortAscending returns true if the first argument is larger then the second; returns false otherwise

Slide modified by L. Lilien

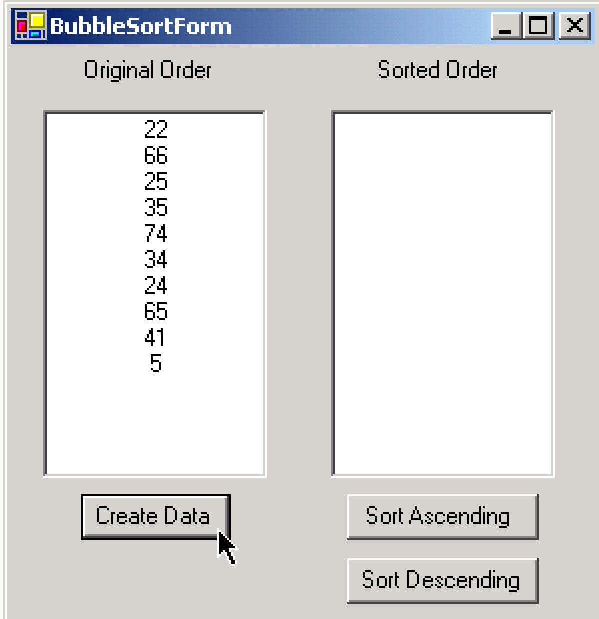
BubbleSortForm.cs

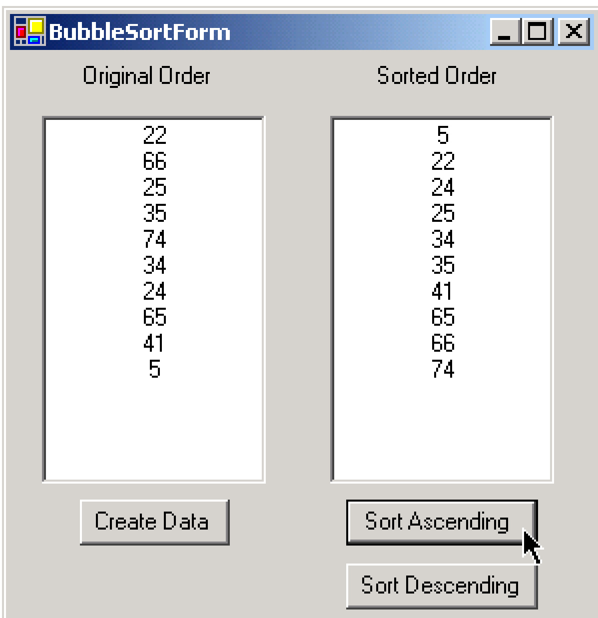
```
   // sort randomly generating numbers in descending order
   private void descendingButton_Click( object sender,
      System.EventArgs e )         // magic here
   {
      // sort array, passing delegate for SortDescending
      DelegateBubbleSort.SortArray( elementArray,
         new DelegateBubbleSort.Comparator(
            SortDescending ) );

      DisplayResults();
   }

   // display the sorted array in sortedTextBox
   private void DisplayResults()
   {
      sortedTextBox.Clear();

      foreach ( int element in elementArray
         sortedTextBox.Text += element + "\

   }

   // main entry point for application
   public static void Main( string[] args )
   {
      Application.Run( new BubbleSortForm() );
                 // new instance waits for button click
   }
}
```

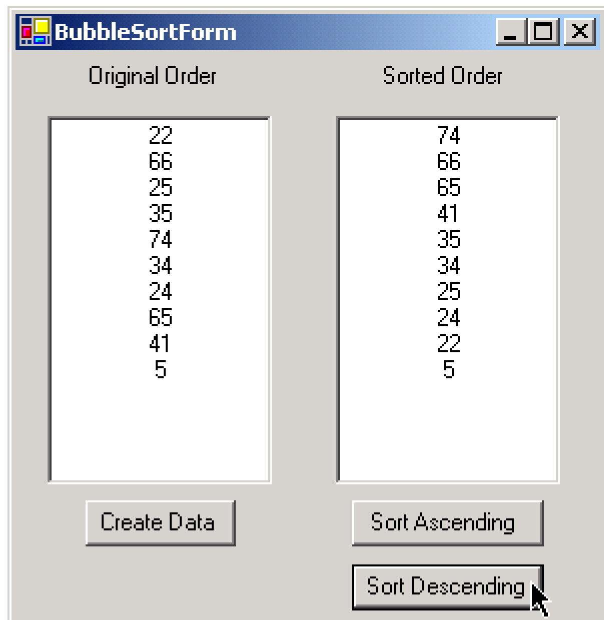
To sort in descending order, send a delegate to the SortDescending method to method SortArray

BubbleSortForm.cs

Program Output







11.8. Operator Overloading

- C# contains many operators that are defined for some primitive types
  - E.g., + - * /
- It is often useful to use operators with user-defined types
  - E.g., user-defined complex number class with + - *
- Operator notation may often be more intuitive then method calls
  - E.g., 'a+b' more intuitive than 'Myclass.AddIntegers(a, b)'
- C# allows programmers to overload operators to make them [*polymorphically*] sensitive to the context in which they are used
  - Overloading operators is a kind of polymorphism
    - What looks like the same operator used for different types
      - E.g., '+' for primitive type int and '+' for user-defined type ComplexNumber
      - Each time different actions performed – polymorphism at work

- Methods define the actions to be taken for the overloaded operator
- They are in the form:

  public static *ReturnType* **operator** *operator-to-be-overloaded*(*arguments*)

  - These methods must be declared **public** and **static**
  - The return type is the type returned as the result of evaluating the operation
  - The keyword **operator** follows the return type to specify that this method defines an operator overload
  - The last piece of information is the operator to be overloaded
    - E.g., operator '+':public static CompNr operator + (CompNr x, Int y, Int z)
  - If the operator is unary, one argument must be specified, if the operator is binary, then two, etc.
    - E.g., operator + shown above is ternary (3 parameters)

Outline

ComplexNumber.cs

```
// Fig. 10.26: ComplexNumber.cs [in textbook ed.1]
// Class that overloads operators for adding, subtracting
// and multiplying complex numbers.

public class ComplexNumber
{
   private int real;
   private int imaginary;

   // default constructor
   public ComplexNumber() {}

   // constructor
   public ComplexNumber( int a, int b )
   {
      Real = a;
      Imaginary = b;
   }

   // return string representation of ComplexNumber
   public override string ToString()
   {
      return "( " + real +
         ( imaginary < 0 ? " - " + ( imaginary * -1 ) :
         " + "  + imaginary ) + "i )";  // conditional operator (?:)
   }

   // property Real
   public int Real
   {
      get
      {
         return real;
      }

```

Property Real provides access to the real part of the complex number

Class ComplexNumber definition

Slide modified by L. Lilien

ComplexNumber.cs

```
36          set
37          {
38             real = value;
39          }
40
41       } // end property Real
42
43       // property Imaginary
44       public int Imaginary
45       {
46          get
47          {
48             return imaginary;
49          }
50
51          set
52          {
53             imaginary = value;
54          }
55
56       } // end property Imaginary
57
58       // overload the addition operator
59       public static ComplexNumber operator + (
60          ComplexNumber x, ComplexNumber y )
61       {
62          return new ComplexNumber(
63             x.Real + y.Real, x.Imaginary + y.Imaginary );
64       }  // non-overloaded '+' (for int's) is used above twice
65          // since x.Real, y.Real, x.Imaginary, y.Imaginary are all
            // int's - see Lines 7-8 & 28-56
```

Overload the addition (+) operator for ComplexNumbers.

Property Imaginary provides access to the imaginary part of a complex number

Slide modified by L. Lilien

Outline

ComplexNumber.cs

```
66       // provide alternative to overloaded + operator
67       // for addition
68       public static ComplexNumber Add( ComplexNumber x,
69          ComplexNumber y )
70       {
71          return x + y; // overloaded
72       }                // this class
73
74       // overload the subtraction operator
75       public static ComplexNumber operator - (
76          ComplexNumber x, ComplexNumber y )
77       {
78          return new ComplexNumber(
79             x.Real - y.Real, x.Imaginary - y.Imaginary );
80       }  // non-overloaded '-' (for int's) is used above twice
            // since x.Real, y.Real, x.Imaginary, y.Imaginary are all
            // int's - see Lines 7-8 & 28-56
81
82       // provide alternative to overloaded - operator
83       // for subtraction
84       public static ComplexNumber Subtract( ComplexNumber x,
85          ComplexNumber y )
86       {
87          return x - y; // overloaded
88       }     // class) used to subtract
89
90       // overload the multiplication operato
91       public static ComplexNumber operator *
92          ComplexNumber x, ComplexNumber y )
93       {
94          return new ComplexNumber(
95             x.Real * y.Real - x.Imaginary * y.Imaginary,
96             x.Real * y.Imaginary + y.Real * x.Imaginary );
97       }   // non-overloaded '*', '-' and '+' (for int's) are used above
            // since x.Real, y.Real, x.Imaginary, y.Imaginary are all
            // int's - see Lines 7-8 & 28-56
```

Method Subtract – provides an alternative to the subtraction operator

Overloads the multiplication (*) operator for ComplexNumbers

Method Add – provides an alternative to the addition operator

Overload the subtraction (-) operator for ComplexNumbers

Slide modified by L. Lilien

```
98
99      // provide alternative to overloaded * operator
100     // for multiplication
101     public static ComplexNumber Multiply( ComplexNumber x,
102        ComplexNumber y )
103     {
104        return x * y; // overloaded '*' (just def. in lines 91-97 in this
105     }                // class) used to multiply 2 ComplexNumbers x & y

106
107  } // end class ComplexNumber
```

Method Multiply – provides an alternative to the multiplication operator

Note:

1) *Some* .NET languages do *not* support operator overloading.

2) The alternative methods:

public static ComplexNumber Add (ComplexNumber x, ComplexNumber y)
public static ComplexNumber Subtract (ComplexNumber x, ComplexNumber y)
public static ComplexNumber Multiply(ComplexNumber x, ComplexNumber y)

for adding, subtracting and mutliplying ComplexNumbers, respectively, are needed to ensure that the ComplexNumber class can be used in such languages.

(of course, it can be used only with 'ComplexNumber Add (x, y)', not 'x + y', etc.)

3) We do not use the alternative methods in our test below.

Outline

OperatorOverloading.cs

```
// Fig 10.27: OperatorOverloading.cs [in textbook ed.1]
// An example that uses operator overloading

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class ComplexTest : System.Windows.Forms.Form
{
   private System.Windows.Forms.Label realLabel;
   private System.Windows.Forms.Label imaginaryLabel;
   private System.Windows.Forms.Label statusLabel;

   private System.Windows.Forms.TextBox realTextBox;
   private System.Windows.Forms.TextBox imaginaryTextBox;

   private System.Windows.Forms.Button firstButton;
   private System.Windows.Forms.Button secondButton;
   private System.Windows.Forms.Button addButton;
   private System.Windows.Forms.Button subtractButton;
   private System.Windows.Forms.Button multiplyButton;

   private ComplexNumber x = new ComplexNumber();
   private ComplexNumber y = new ComplexNumber();

   [STAThread]
   static void Main()
   {
      Application.Run( new ComplexTest() );
   }

```

Outline

OperatorOverloading.cs

```
      private void firstButton_Click(
         object sender, System.EventArgs e )
      {
         x.Real = Int32.Parse( realTextBox.Text );
         x.Imaginary = Int32.Parse( imaginaryTextBox.Text );
         realTextBox.Clear();
         imaginaryTextBox.Clear();
         statusLabel.Text = "First Complex Number is: " + x;
      }

      private void secondButton_Click(
         object sender, System.EventArgs e )
      {
         y.Real = Int32.Parse( realTextBox.Text );
         y.Imaginary = Int32.Parse( imaginaryTextBox.Text );
         realTextBox.Clear();
         imaginaryTextBox.Clear();
         statusLabel.Text = "Second Complex Number is: " + y;
      }

      // add complex numbers
      private void addButton_Click( object sender, System.EventArgs e )
      {
         statusLabel.Text = x + " + " + y + " = " + ( x + y );
      }

      // subtract complex numbers
      private void subtractButton_Click(
         object sender, System.EventArgs e )
      {
         statusLabel.Text = x + " - " + y + " = " + ( x - y );
      }

```

Use overloaded addition operator to add two ComplexNumbers

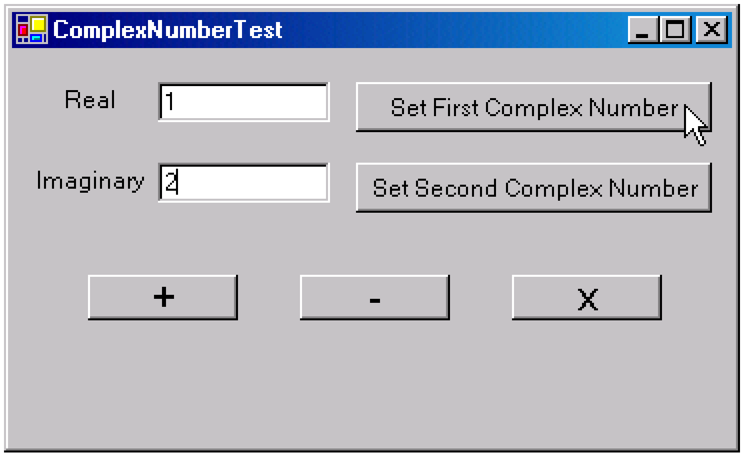
Use overloaded subtraction operator to subtract two ComplexNumbers

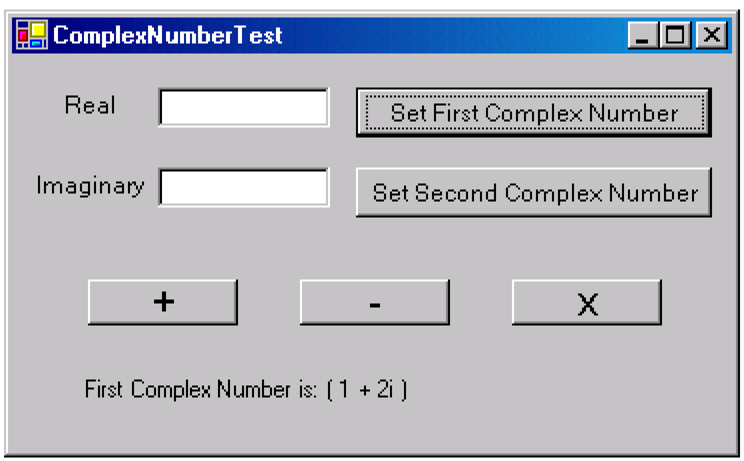
OperatorOverloading.cs

Program Output

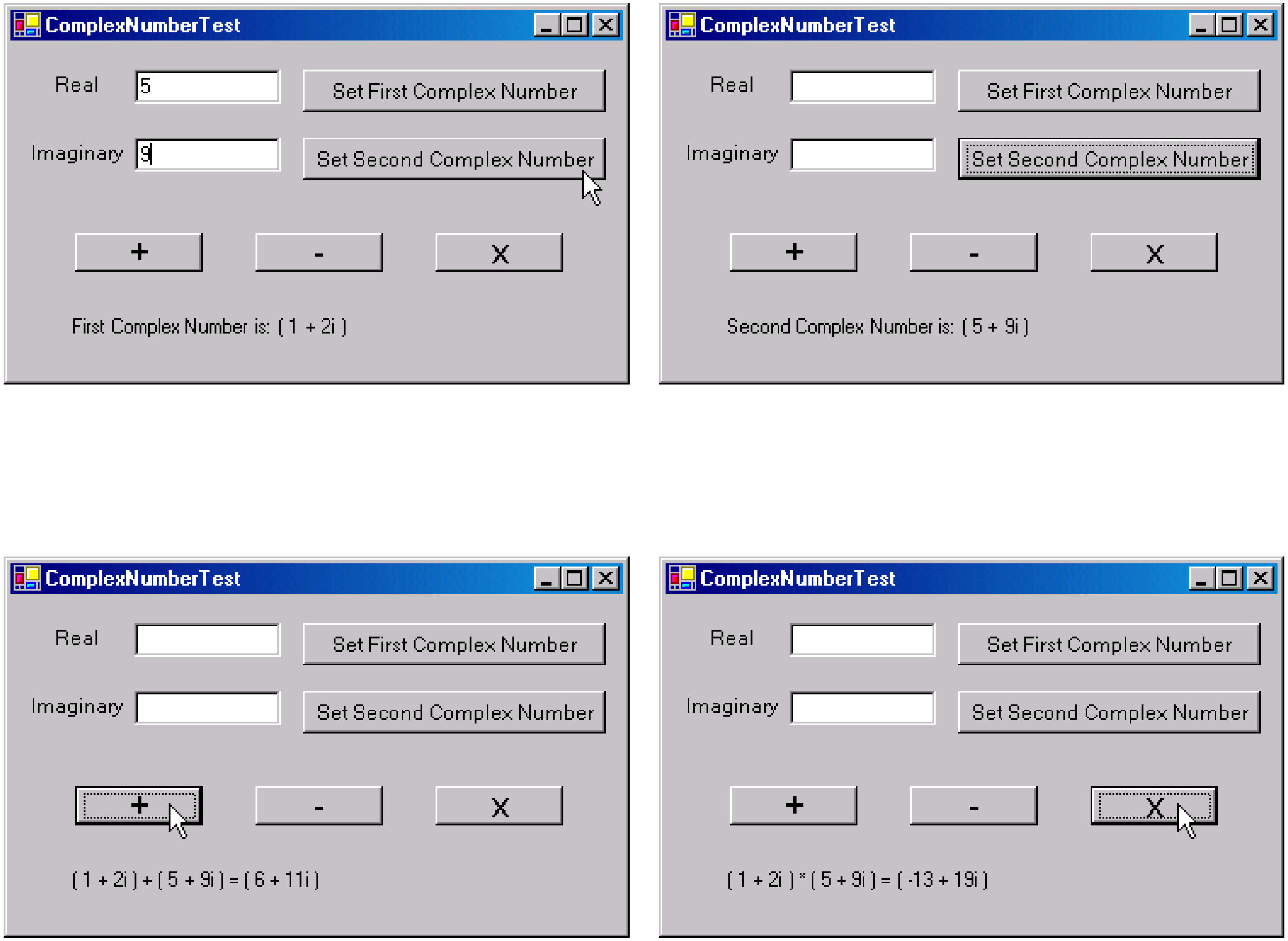
```
      // multiply complex numbers
      private void multiplyButton_Click(
         object sender, System.EventArgs e )
      {
         statusLabel.Text = x + " * " + y + " = " + ( x * y );
      }

   } // end class ComplexTest
```

Use overloaded multiplication operator to multiply two ComplexNumbers





OperatorOverloading.cs
Program Output



The End