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

Section [order of the slides consistent with ed. 1 of the textbook]

11.1 Introduction
10.2 in ed.1 Derived-Class-Object to Base-Class-Object Conversion
10.3 in ed.1 Type Fields and switch Statements

11.2 Polymorphism Examples

11.4 Abstract Classes and Methods
10.6 in ed.1 Case Study: Inheriting from an Abstract Class and Implementation

11.6 sealed Methods and Classes

11.5 Case Study: Payroll System Using Polymorphism

11.7 Case Study: Creating and Using Interfaces
10.10 in ed.1 Delegates

11.8 Operator Overloading

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

Slides added by L.Lilien are © 2006 Leszek T. Lilien.
Permission to use for non-commercial purposes slides added by L.Lilien’s will be gladly granted upon a written (e.g., emailed) request.
11.1. Introduction

- **Polymorphism**
  - Poly = many, morph = form
  - Polymorphic = having many forms

- **Example:**
  - Base class: Quadrilateral
  - Derived classes: Rectangle, Square, Trapezoid
  - Polymorphic "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
• Polymorphism
  – allows programmers to write:
  – Programs that handle a wide variety of related classes in a generic manner
  – Systems that are easily extensible
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)
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;
        }
    }
}
```csharp
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
```
// Fig. 10.2: Circle.cs [in textbook ed 1]
// Circle class that inherits from class Point.
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;
        }
    }
}
```csharp
    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
```
// 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, 2.7);

        string output = "Point point1: " + point1.ToString() +
                 "\nCircle circle1: " + circle1.ToString();

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

        output += "\nCCircle circle1 (via point2): " +
                point2.ToString(); // Circle’s ToString, since point2 references now a Circle object (circle1).
        // (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
Create a Circle circle1 (derived-class object) to Point
Assign Circle circle1 (derived-class object) to Point reference
(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! It is OK since point2 references a Circle)

Slide modified by L. Lilien
// attempt to assign point1 object to Circle reference
if ( point1 is Circle )
{
    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.
    output += "\n\ncast successful";
}
else
{
    output += "\n\npoint1 does not refer to a Circle";
}

MessageBox.Show( output,
    "Demonstrating the 'is a' relationship"
);

} // end method Main

} // end class PointCircleTest

Test if point1 references a Circle object – it does not since point1 references a Point object (1.13)
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 circle2 = (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?).
Point point1
(object itself)

Circle circle1
(object itself)
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`)

![Inheritance Diagram]

© 2002 Prentice Hall. All rights reserved.  

Figure added by L. Lilien
• Any other operation (e.g., \texttt{perimeter}) that can be performed on a \textit{Quadrilateral} object can also be performed on a \textit{Rectangle, Square, Parallelogram,} or \textit{Trapezoid} object

• Suppose that:
  – Program \textit{P} instantiated a \textit{Rectangle/ Square/…} object referenced via \texttt{rectangle1/square1/…}:
    • \texttt{Rectangle rectangle1 = new Rectangle (...);}
      \texttt{Square square1 = new Square (...);}
  – \textit{P} can use a base-class reference \texttt{quad1} to invoke derived-class methods \texttt{Rectangle.perimeter, Square.perimeter, …}
    • \texttt{Quadrilateral quad1 = rectangle1;}
      \texttt{... quad1.perimeter ...} // Method \texttt{perimeter} is executed
      // on the derived-class object \texttt{rectangle1} via the
      // base-class reference \texttt{quad1.perimeter}
      \texttt{Quadrilateral quad1 = square1;}
      \texttt{... quad1.perimeter ...} // Method \texttt{perimeter} is executed
      // on the derived-class object \texttt{square1} via the
      // same base-class reference \texttt{quad1.perimeter}

  – With such a \textit{Quadrilateral} reference, \textit{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`)

![Class hierarchy diagram]

© 2002 Prentice Hall. All rights reserved.  

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**

```csharp
- 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

```csharp
- 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
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**
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)  
Overridess 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;
    }
}
// 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
        }
    }
Point2.cs

```csharp
// 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
```
// 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;
        }
    }

```csharp
    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)
// override property Name from class Point2
public override string Name
{
    get
    {
        return "Circle2";
    }
}
} // end class Circle2
// 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
{
    // Cylinder2 height
    private double 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;
            }
        }
    }
}
Override Area implementation of class Circle2

Override Volume implementation of class Shape

Override read-only property Name
// 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 Cylinder2 objects
        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[2] refers to Cylinder2 object

        string output = point.Name + ": " + point + "\n" +
        circle.Name + ": " + circle + "\n" +
        cylinder.Name + ": " + cylinder;
// 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");
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
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
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;
        }
    }
}
// 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
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;
        }
    }
}
// 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();
}
// Fig. 10.11: CommissionWorker.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;
        }
    }
}
40

CommisionWorker.cs

```csharp
30    set
31    {
32        // ensure non-negative salary value
33        if ( value > 0 )
34            salary = value;
35    }
36
37    // property Commission
38    public decimal Commission
39    {
40        get
41        {
42            return commission;
43        }
44
45        set
46        {
47            // ensure non-negative commission value
48            if ( value > 0 )
49                commission = value;
50        }
51    }
52
53    // property Quantity
54    public int Quantity
55    {
56        get
57        {
58            return quantity;
59        }
60    }
61
© 2002 Prentice Hall.
All rights reserved.
```csharp
41
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 CommissionWorker
```

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;
        }
    }
}
// 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();
}
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;
        }
    }
}
// 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()
{
    if ( HoursWorked <= 40 )
    {
        return Wage * Convert.ToDecimal( HoursWorked );
    }
    else
    {
        // calculate base and overtime pay
        decimal basePay = Wage * Convert.ToDecimal( 40 );
        decimal overtimePay = Wage * 1.5M * Convert.ToDecimal( HoursWorked - 40 );
    }
}
```csharp
return basePay + overtimePay;

// return string representation of HourlyWorker
public override string ToString()
{
    return "HourlyWorker: " + base.ToString();
}
```
// Fig. 10.14: EmployeesTest.cs [in textbook ed.1]
// Demonstrates polymorphism by displaying earnings
// for various Employee types.
using System;
using System.Windows.Forms;

class EmployeesTest
{
    public static void Main(string[] args)
    {
        Boss boss = new Boss("John", "Smith", 800);
        CommissionWorker commissionWorker = new CommissionWorker("Sue", "Jones", 400, 3, 150);
        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;
        string output = GetString(employee) + boss + " earned " + boss.Earnings().ToString("C") + "

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

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

Polymorphically call the method of the appropriate derived class

Program Output

Demonstrating Polymorphism

Boss: John Smith earned $800.00
CommissionWorker: Sue Jones earned $850.00
PieceWorker: Bob Lewis earned $500.00
HourlyWorker: Karen Price earned $756.25
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 than 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)
// 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
// 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;
        }
    }
}
// property Name implementation of interface IAge
public string Name
{
    get
    {
        return firstName + " " + lastName;
    }
}
} // end class Person

Definition of Name property (required)
```csharp
// 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++; // Implementation of Age property (required)
    }

    // property Age implementation of interface IAge
    public int Age
    {
        get
        {
            return rings; // Class Tree implements the IAge interface
        }
    }
}
```
// property Name implementation of interface IAge
public string Name
{
    get
    {
        return "Tree";
    }
}
} // end class Tree

Definition of Name property (required)
/ 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";
    }
}
// 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
// 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
using System;

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;
        }
    }
}
```csharp
    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 ";\]
```
// 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
// 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
```csharp
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
// 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;
        }
    }
}
```csharp
// 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 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 Cylinder3 objects
        Point3 point = new Point3( 7, 11 );
        Circle3 circle = new Circle3( 22, 8, 3.5 );
        Cylinder3 cylinder = new Cylinder3( 10, 10, 3.3, 10 );

        // 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

        string output = point.Name + ": " + point + "\n" +
            circle.Name + ": " + circle + "\n" +
            cylinder.Name + ": " + cylinder;
    }
}
foreach ( IShape shape in arrayOfShapes )
{
    output += "\n\n" + shape.Name + ":\nArea = " + 
shape.Area() + "\nVolume = " + shape.Volume();
    MessageBox.Show( output, "Demonstrating Polymorphism" );
}
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 encapsulates 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:

    ```csharp
    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)`

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

public class DelegateBubbleSort
{
    public delegate bool Comparator(int element1, 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; pass++)
            for (int i = 0; i < array.Length - 1;
                if (Compare(array[i], array[i + 1]))
                    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;
    }
}
// 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.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();
        // magic here
    

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

private bool SortAscending(int element1, int element2)
{
    return element1 > element2;
}

private void ascendingButton_Click(object sender, EventArgs e)
{
    // sort array, passing delegate for SortAscending
    DelegateBubbleSort.SortArray(elementArray,
        new DelegateBubbleSort.Comparator(SortAscending));

    DisplayResults();
}

private bool SortDescending(int element1, int element2)
{
    return element1 < element2;
}
// 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 + "\r
";
    }

    // main entry point for application
    public static void Main( string[] args )
    {
        Application.Run( new BubbleSortForm() );
            // new instance waits for button click
    }
}
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 than 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)
// 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;
        }
    }
}
```csharp
set
{
    real = value;
}
} // end property Real

// property Imaginary
public int Imaginary
{
    get
    {
        return imaginary;
    }
    set
    {
        imaginary = value;
    }
} // end property Imaginary

// overload the addition operator
public static ComplexNumber operator + ( ComplexNumber x, ComplexNumber y )
{
    return new ComplexNumber(
        x.Real + y.Real, x.Imaginary + y.Imaginary );
} // 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
```

Overload the addition (+) operator for ComplexNumbers.

Property Imaginary provides access to the imaginary part of a complex number.
// provide alternative to overloaded + operator
public static ComplexNumber Add( ComplexNumber x, ComplexNumber y )
{
    return x + y; // overloaded + operator defined in lines 59-64
}

// overload the subtraction operator
public static ComplexNumber operator - ( ComplexNumber x, ComplexNumber y )
{
    return new ComplexNumber( x.Real - y.Real, x.Imaginary - y.Imaginary );
    // 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
}

// provide alternative to overloaded - operator
// for subtraction
public static ComplexNumber Subtract( ComplexNumber x, ComplexNumber y )
{
    return x - y; // overloaded - operator defined in lines 75-80
}

// overload the multiplication operator
public static ComplexNumber operator * ( ComplexNumber x, ComplexNumber y )
{
    return new ComplexNumber( x.Real * y.Real - x.Imaginary * y.Imaginary, x.Real * y.Imaginary + y.Real * x.Imaginary );
    // non-overloaded '*' and '+' (for int's) are used above twice
    // since x.Real, y.Real, x.Imaginary, y.Imaginary are all int's - see Lines 7-8 & 28-56
}
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 multiplying 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.
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 ComplexNumber x = new ComplexNumber();
    private ComplexNumber y = new ComplexNumber();

    [STAThread]
    static void Main()
    {
        Application.Run( new ComplexTest() );
    }
}
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 multiplication operator to multiply two ComplexNumbers.
OperatorOverloading.cs

Program Output

First Complex Number is: \(1 + 2i\)

Second Complex Number is: \(5 + 9i\)

\((1 + 2i) + (5 + 9i) = (6 + 11i)\)

\((1 + 2i) \times (5 + 9i) = (-13 + 19i)\)
The End