Polymorphism, Interfaces & Operator Overloading [optional]

- Polymorphism
  - Poly = many , morph = form
    - DIGRESSION: Examples of morphing:
      1) Animation:
      - Bill Gates<--> Steve Jobs http://www.mukimuki.fr/flashblog/2009/05/10/morphing/

      2) Still images:
      - (Star Wars) http://www.cs.wustl.edu/cs/playground/euphoria/morph.gif
      - (Bush → Obama) http://paulbakaus.com/wp-content/uploads/2009/10/bush-obama-morphing.jpg
      - (esp. horse → dinosaur): http://www.cad.zju.edu.cn/home/xudong/Projects/mesh_morphing/main.htm
      - (bottom, face) http://www.cc.gatech.edu/grads/y/yliu88/fancyworks_page.html
      - (pretzel → fist) http://www.fp.utm.my/projek/psm/juzclick/morphing.htm
      - (hand → Tar Monster) https://www.cs.drexel.edu/node/11594
  - Polymorphic = having many forms

    Examples of polymorphism– next slide

12.2 Polymorphism Examples (Cont.)

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

12.3 Demonstrating Polymorphic Behavior (Cont.)

Derived-Class-Object to Base-Class-Object Conversion

- Class hierarchies
  - Can explicitly cast between types in a class hierarchy (next)
  - Can assign derived-class objects to base-class references
    - A fundamental part of programs that process objects polymorphically
  - This means that:
    - 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 can not be treated as an object of any of its derived classes)
      - That is why we can have arrays of base-class references that refer to objects of many derived-class types (as shown in examples above)

Slide modified by L. Lilien

12.4. Abstract Classes and Methods

- **Abstract** classes
  - Cannot be instantiated
  - Are 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 (citing from ed.1, p.395/1)

Slide modified by L. Lilien

12.4. Abstract Classes and Methods (Cont.)

- To define an abstract class, use keyword **abstract** in the declaration
- To declare an abstract method or property, use keyword **abstract** in the declaration
- Any class with at least one abstract method or property must be declared **abstract**

12.4. Abstract Classes and Methods (Cont.)

- Abstract classes can *not* be instantiated (used for creating, by *instantiation*, real objects)
  - They are too generic , too abstract
- Instead, abstract classes are used to provide appropriate base classes
  - Till this lecture, we considered only **concrete classes**
    - We called them "classes " (not "concrete classes")
  - A concrete class may inherit from an abstract base class
    - Concrete classes can (of course!) be instantiated

12.4. Abstract Classes and Methods (Cont.)

- **Concrete** classes use the keyword **override** to provide implementations for all the abstract methods and properties of the base-class
- Abstract classes are very useful, even though they can *not* be instantiated:
  - We can use abstract class references to refer to instances of any concrete class derived from the abstract class
  - We can use abstract base classes to declare variables that can hold references to objects of any concrete classes derived from those abstract classes
    - You can use such variables to manipulate derived-class objects polymorphically and to invoke `static` methods declared in those abstract base classes

12.4 Abstract Classes and Methods (Cont.)

- **Abstract property** declarations have the form:

```
public abstract PropertyType MyProperty
{
   get;
   set;
} // end abstract property
```

- An abstract property may omit implementations for the `get` accessor, the `set` accessor or both (as shown above)

C# Property Syntax

```
[access modifiers] return type property name
{
   get
   {
      ... sequence of statements ending in a return (or throw)
   }
   set
   {
      ... sequence of statements
   }
}
```

get accessor

set accessor

- Get accessor returns value of same type as "return type"
- Set accessors have implicit parameter named "value"
  - Use to set internal data representation
- Properties can be public, private, protected
  - Public: any class can access, private: only that class, protected: that class and children
- By convention, property names have initial capital ("X" to access "x")

C# Property Example

```
public class GameInfo
{
   private string gameName;
   public string Name;
   {
      get
      {   return gameName;   }
      set
      {   gameName = value;   }
   }
}
```

```
// Test code
GameInfo g = new GameInfo();

// Call set accessor
g.Name = "Radiant Silvergun";

// Call get accessor
System.Console.Write(g.Name);
```


12.4 Abstract Classes and Methods (Cont.)

- Constructors and `static` methods can not be declared `abstract`.

An abstract class declares common attributes and behaviors of the various classes that inherit from it, either directly or indirectly, in a class hierarchy.

An abstract class typically contains one or more abstract methods or properties that concrete derived classes must override.

Case Study: Payroll System Using Polymorphism

- Create an enhanced employee hierarchy to solve the following problem:

A company pays its employees on a weekly basis.
The employees are of four types:

1) salaried employees are paid a fixed weekly salary regardless of the number of hours worked,
2) hourly employees are paid by the hour and receive overtime pay for all hours worked in excess of 40 hours,
3) commission employees are paid a percentage of their sales,
4) salaried-commission employees receive a base salary plus a percentage of their sales.

For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

- We use `abstract` class `Employee` to represent the general concept of an employee.
- `SalariedEmployee`, `CommissionEmployee` and `HourlyEmployee` extend `Employee`.
- Class `BasePlusCommissionEmployee`—which extends `CommissionEmployee`—represents the last employee type.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

- The UML class diagram in Fig. 12.2 shows the inheritance hierarchy for our polymorphic employee payroll application.

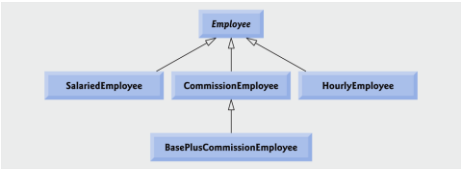


Fig. 12.2 | `Employee` hierarchy UML class diagram

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

12.5.1 Creating Abstract Base Class `Employee`

- Class `Employee` provides methods `Earnings` and `ToString`, in addition to the properties that manipulate `Employee`'s instance variables.
- Each earnings calculation depends on the employee's class, so we declare `Earnings` as `abstract`.
- The application iterates through the array and calls method `Earnings` for each `Employee` object. C# processes these method calls polymorphically.
- Each derived class overrides method `ToString` to create a string representation of an object of that class.

12.5 Case Study: Payroll System Using Polymorphism (Cont.)

- The diagram in Fig. 12.3 shows each of the five classes in the hierarchy down the left side and methods `Earnings` and `ToString` across the top.

	Earnings	ToString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklySalary*
Hourly-Employee	*if hours <= 40* wage * hours *if hours > 40* 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage* hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales* commission rate: *commissionRate* base salary: *baseSalary*

Fig. 12.3 | Polymorphic interface for the `Employee` hierarchy classes.

Outline

- The `Employee` class's declaration is shown in Fig. 12.4.

`Employee.cs`

(1 of 2)

```
1  // Fig. 12.4: Employee.cs
2  // Employee abstract base class.
3  public abstract class Employee
4  {
5     // read-only property that gets employee's first name
6     public string FirstName { get; private set; }
7
8     // read-only property that gets employee's last name
9     public string LastName { get; private set; }
10
11    // read-only property that gets employee's social security number
12    public string SocialSecurityNumber { get; private set; }
13
14    // three-parameter constructor
15    public Employee( string first, string last, string ssn )
16    {
17       FirstName = first;
18       LastName = last;
19       SocialSecurityNumber = ssn;
20    } // end three-parameter Employee constructor
```

Fig. 12.4 | `Employee` abstract base class. (Part 1 of 2.)

Employee.cs

(1 of 2)

```
21
22    // return string representation of Employee object, using properties
23    public override string ToString()
24    {
25       return string.Format( "{0} {1}\nsocial security number: {2}",
26          FirstName, LastName, SocialSecurityNumber );
27    } // end method ToString
28
29    // abstract method overridden by derived classes
30    public abstract decimal Earnings(); // no implementation here
31 } // end abstract class Employee
```

The `Employee` class includes an `abstract` method `Earnings`, which must be implemented by concrete derived classes.

Fig. 12.4 | `Employee` abstract base class. (Part 2 of 2.)

SalariedEmployee.cs

(1 of 2)

```
1  // Fig. 12.5: SalariedEmployee.cs
2  // SalariedEmployee class that extends Employee.
3  public class SalariedEmployee : Employee
4  {
5     private decimal weeklySalary;
6
7     // four-parameter constructor
8     public SalariedEmployee( string first, string last, string ssn,
9        decimal salary ) : base( first, last, ssn )
10    {
11       WeeklySalary = salary; // validate salary via property
12    } // end four-parameter SalariedEmployee constructor
13
14    // property that gets and sets salaried employee's salary
15    public decimal WeeklySalary
16    {
17       get
18       {
19          return weeklySalary;
20       } // end get
```

`SalariedEmployee` extends `Employee`.

Using the base class constructor to initialize the `private` variables not inherited from the base class.

Fig. 12.5 | `SalariedEmployee` class that extends `Employee`. (Part 1 of 2.)

SalariedEmployee.cs

(2 of 2)

```
21       set
22       {
23          weeklySalary = ( ( value >= 0 ) ? value : 0 ); // validation
24       } // end set
25    } // end property WeeklySalary
26
27    // calculate earnings; override abstract method Earnings in Employee
28    public override decimal Earnings()
29    {
30       return WeeklySalary;
31    } // end method Earnings
32
33    // return string representation of SalariedEmployee object
34    public override string ToString()
35    {
36       return string.Format( "salaried employee: {0}\n{1}: {2:C}",
37          base.ToString(), "weekly salary", WeeklySalary );
38    } // end method ToString
39 } // end class SalariedEmployee
```

Method `Earnings` overrides `Employee`'s abstract method `Earnings` to provide a concrete implementation that returns the `SalariedEmployee`'s weekly salary.

Method `ToString` overrides `Employee` method `ToString`.

Fig. 12.5 | `SalariedEmployee` class that extends `Employee`. (Part 2 of 2.)

- Class `HourlyEmployee` also extends class `Employee`.

HourlyEmployee.cs

(1 of 3)

```
1  // Fig. 12.6: HourlyEmployee.cs
2  // HourlyEmployee class that extends Employee.
3  public class HourlyEmployee : Employee
4  {
5     private decimal wage; // wage per hour
6     private decimal hours; // hours worked for the week
7
8     // five-parameter constructor
9     public HourlyEmployee( string first, string last, string ssn,
10       decimal hourlyWage, decimal hoursWorked )
11       : base( first, last, ssn )
12    {
13       Wage = hourlyWage; // validate hourly wage via property
14       Hours = hoursWorked; // validate hours worked via property
15    } // end five-parameter HourlyEmployee constructor
16
17    // property that gets and sets hourly employee's wage
18    public decimal Wage
19    {
20       get
21       {
22          return wage;
23       } // end get
```

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 1 of 3.)

HourlyEmployee.cs

(2 of 3)

```
24       set
25       {
26          wage = ( value >= 0 ) ? value : 0; // validation
27       } // end set
28    } // end property Wage
29
30    // property that gets and sets hourly employee's hours
31    public decimal Hours
32    {
33       get
34       {
35          return hours;
36       } // end get
37       set
38       {
39          hours = ( ( value >= 0 ) && ( value <= 168 ) ) ?
40             value : 0; // validation
41       } // end set
42    } // end property Hours
```

Method `ToString` overrides `Employee` method `ToString`.

The `set` accessor in property `Hours` ensures that `hours` is in the range 0–168 (the number of hours in a week).

HourlyEmployee.cs

(3 of 3)

```
43
44    // calculate earnings; override Employee's abstract method Earnings
45    public override decimal Earnings()
46    {
47       if ( Hours <= 40 ) // no overtime
48          return Wage * Hours;
49       else
50          return ( 40 * Wage ) + ( ( Hours - 40 ) * Wage * 1.5M );
51    } // end method Earnings
52
53    // return string representation of HourlyEmployee object
54    public override string ToString()
55    {
56       return string.Format(
57          "hourly employee: {0}\n{1}: {2:C}; {3}: {4:F2}",
58          base.ToString(), "hourly wage", Wage, "hours worked", Hours );
59    } // end method ToString
60 } // end class HourlyEmployee
```

Fig. 12.6 | `HourlyEmployee` class that extends `Employee`. (Part 3 of 3.)

- Class `CommissionEmployee`) extends class `Employee`.

`CommissionEmployee.cs` (1 of 3)

```
1  // Fig. 12.7: CommissionEmployee.cs
2  // CommissionEmployee class that extends Employee.
3  public class CommissionEmployee : Employee
4  {
5     private decimal grossSales; // gross weekly sales
6     private decimal commissionRate; // commission percentage
7
8     // five-parameter constructor
9     public CommissionEmployee( string first, string last, string ssn,
10       decimal sales, decimal rate ) : base( first, last, ssn )
11    {
12       GrossSales = sales; // validate gross sales via property
13       CommissionRate = rate; // validate commission rate via property
14    } // end five-parameter CommissionEmployee constructor
15
16    // property that gets and sets commission employee's commission rate
17    public decimal CommissionRate
18    {
19       get
20       {
21          return commissionRate;
22       } // end get
```

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 1 of 3.)

`CommissionEmployee.cs` (2 of 3)

```
23       set
24       {
25          commissionRate = ( value > 0 && value < 1 ) ?
26             value : 0; // validation
27       } // end set
28    } // end property CommissionRate
29
30    // property that gets and sets commission employee's gross sales
31    public decimal GrossSales
32    {
33       get
34       {
35          return grossSales;
36       } // end get
37       set
38       {
39          grossSales = ( value >= 0 ) ? value : 0; // validation
40       } // end set
41    } // end property GrossSales
```

Fig. 12.7 | `CommissionEmployee` class that extends `Employee`. (Part 2 of 3.)

`CommissionEmployee.cs` (3 of 3)

```
42
43    // calculate earnings; override abstract method Earnings in Employee
44    public override decimal Earnings()
45    {
46       return CommissionRate * GrossSales;
47    } // end method Earnings
48
49    // return string representation of CommissionEmployee object
50    public override string ToString()
51    {
52       return string.Format( "{0}: {1}\n{2}: {3:C}\n{4}: {5:F2}",
53          "commission employee", base.ToString(),
54          "gross sales", GrossSales, "commission rate", CommissionRate );
55    } // end method ToString
56 } // end class CommissionEmployee
```

Calling base-class method `ToString` to obtain the `Employee`-specific information.

- Class `BasePlusCommissionEmployee` (Fig. 12.8) extends class `Commission-Employee` and therefore is an indirect derived class of class `Employee`.

`BasePlusCommissionEmployee.cs` (1 of 2)

```
1  // Fig. 12.8: BasePlusCommissionEmployee.cs
2  // BasePlusCommissionEmployee class that extends CommissionEmployee.
3  public class BasePlusCommissionEmployee : CommissionEmployee
4  {
5     private decimal baseSalary; // base salary per week
6
7     // six-parameter constructor
8     public BasePlusCommissionEmployee( string first, string last,
9        string ssn, decimal sales, decimal rate, decimal salary )
10       : base( first, last, ssn, sales, rate )
11    {
12       BaseSalary = salary; // validate base salary via property
13    } // end six-parameter BasePlusCommissionEmployee constructor
14
15    // property that gets and sets
16    // base-salaried commission employee's base salary
17    public decimal BaseSalary
18    {
19       get
20       {
21          return baseSalary;
22       } // end get
```

Fig. 12.8 | `BasePlusCommissionEmployee` class that extends `CommissionEmployee`. (Part 1 of 2.)

`BasePlusCommissionEmployee.cs` (2 of 2)

```
23       set
24       {
25          baseSalary = ( value >= 0 ) ? value : 0; // validation
26       } // end set
27    } // end property BaseSalary
28
29    // calculate earnings; override method Earnings in CommissionEmployee
30    public override decimal Earnings()
31    {
32       return BaseSalary + base.Earnings();
33    } // end method Earnings
34
35    // return string representation of BasePlusCommissionEmployee object
36    public override string ToString()
37    {
38       return string.Format( "base-salaried {0}; base salary: {1:C}",
39          base.ToString(), BaseSalary );
40    } // end method ToString
41 } // end class BasePlusCommissionEmployee
```

Method `Earnings` calls the base class's `Earnings` method to calculate the commission-based portion of the employee's earnings.

`BasePlusCommissionEmployee`'s `ToString` method creates a string that contains `"base-salaried"`, followed by the `string` obtained by invoking base class `CommissionEmployee`'s `ToString` method (a good example of code reuse) then the base salary.

Fig. 12.8 | `BasePlusCommissionEmployee` class that extends `CommissionEmployee`. (Part 2 of 2.)

- The application in Fig. 12.9 tests our `Employee` hierarchy.

`PayrollSystemTest.cs` (1 of 6)

```
1  // Fig. 12.9: PayrollSystemTest.cs
2  // Employee hierarchy test application.
3  using System;
4
5  public class PayrollSystemTest
6  {
7     public static void Main( string[] args )
8     {
9        // create derived-class objects
10       SalariedEmployee salariedEmployee =
11          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00M );
12       HourlyEmployee hourlyEmployee =
13          new HourlyEmployee( "Karen", "Price",
14          "222-22-2222", 16.75M, 40.0M );
15       CommissionEmployee commissionEmployee =
16          new CommissionEmployee( "Sue", "Jones",
17          "333-33-3333", 10000.00M, .06M );
18       BasePlusCommissionEmployee basePlusCommissionEmployee =
19          new BasePlusCommissionEmployee( "Bob", "Lewis",
20          "444-44-4444", 5000.00M, .04M, 300.00M );
21
```

Create objects of each of the four concrete `Employee` derived classes.

Fig. 12.9 | `Employee` hierarchy test application. (Part 1 of 6.)

PayrollSystemTest.cs (2 of 6)

```
22      Console.WriteLine( "Employees processed individually:\n" );
23
24      Console.WriteLine( "{0}\nearned: {1:C}\n",
25         salariedEmployee, salariedEmployee.Earnings() );
26      Console.WriteLine( "{0}\nearned: {1:C}\n",
27         hourlyEmployee, hourlyEmployee.Earnings() );
28      Console.WriteLine( "{0}\nearned: {1:C}\n",
29         commissionEmployee, commissionEmployee.Earnings() );
30      Console.WriteLine( "{0}\nearned: {1:C}\n",
31         basePlusCommissionEmployee,
32         basePlusCommissionEmployee.Earnings() );
33
34      // create four-element Employee array
35      Employee[] employees = new Employee[ 4 ];
36
37      // initialize array with Employees of derived types
38      employees[ 0 ] = salariedEmployee;
39      employees[ 1 ] = hourlyEmployee;
40      employees[ 2 ] = commissionEmployee;
41      employees[ 3 ] = basePlusCommissionEmployee;
42
```

Each object's `ToString` method is called implicitly.

Fig. 12.9 | `Employee` hierarchy test application. (Part 2 of 6.)

PayrollSystemTest.cs (2 of 6)

```
43      Console.WriteLine( "Employees processed polymorphically:\n" );
44
45      // generically process each element in array employees
46      foreach ( var currentEmployee in employees )
47      {
48         Console.WriteLine( currentEmployee ); // invokes ToString
49
50         // determine whether element is a BasePlusCommissionEmployee
51         if ( currentEmployee is BasePlusCommissionEmployee )
52         {
53            // downcast Employee reference to
54            // BasePlusCommissionEmployee reference
55            BasePlusCommissionEmployee employee =
56               ( BasePlusCommissionEmployee ) currentEmployee;
57
58            employee.BaseSalary *= 1.10M;
59            Console.WriteLine(
60               "new base salary with 10% increase is: {0:C}",
61               employee.BaseSalary );
62         } // end if
```

Method calls are resolved at execution time, based on the type of the object referenced by the variable.

The `is` operator is used to determine whether a particular `Employee` object's type is `BasePlusCommissionEmployee`.

Downcasting `currentEmployee` from type `Employee` to type `BasePlusCommissionEmployee`.

Fig. 12.9 | `Employee` hierarchy test application. (Part 3 of 6.)

PayrollSystemTest.cs (3 of 6)

```
63
64         Console.WriteLine(
65            "earned {0:C}\n", currentEmployee.Earnings() );
66      } // end foreach
67
68      // get type name of each object in employees array
69      for ( int j = 0; j < employees.Length; j++ )
70         Console.WriteLine( "Employee {0} is a {1}", j,
71            employees[ j ].GetType() );
72   } // end Main
73 } // end class PayrollSystemTest
```

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00
```

(continued on next page...)

Method calls are resolved at execution time, based on the type of the object referenced by the variable.

Method `GetType` returns an object of class `Type`, which contains information about the object's type.

Fig. 12.9 | `Employee` hierarchy test application. (Part 4 of 6.)

PayrollSystemTest.cs (5 of 6)

(continued from previous page...)

```
commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00
commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00
commission rate: 0.04; base salary: $300.00
earned: $500.00


Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111


weekly salary: $800.00
earned $800.00
```

(continued on previous page...)

Fig. 12.9 | `Employee` hierarchy test application. (Part 5 of 6.)

PayrollSystemTest.cs (6 of 6)

(continued from previous page...)

```
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00
commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00
commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Fig. 12.9 | `Employee` hierarchy test application. (Part 6 of 6.)

** OPTIONAL ** 12.5 Case Study: Payroll System Using Polymorphism (Cont.)

- You can avoid a potential `InvalidCastException` by using the `as` operator to perform a downcast rather than a cast operator.
  - If the downcast is invalid, the expression will be null instead of throwing an exception.
- Method `GetType` returns an object of class Type (of namespace `System`), which contains information about
  the object's type, including its class name, the names of its methods, and the name of its base class.
- The `Type` class's `ToString` method returns the class name.

12.6. `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 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

12.7. Case Study: Creating and Using Interfaces

- 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, a tree a car, …, are disparate objects
    An interface to define age and name for these disparate objects:

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

  – Enables polymorphic processing of age and name for these disparate objects

Slide modified by L. Lilien

Case Study: Creating and Using Interfaces – Cont.

- 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 be/are defined, do *not* use an abstract class, use an interface instead

Slide modified by L. Lilien

12.7. Case Study: Creating and Using Interfaces (Cont.)

- 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 : BaseClassName, InterfaceName1, InterfaceName2* (object list must precedes interface list)
    - Only one *BaseClassName* in C# (no multiple class inheritance)
- A class that implements an interface, must provide implementations for every method and property in the interface definition

Slide modified by L. Lilien

12.7 Case Study: Creating and Using Interfaces (Cont.)

- A programmer creates an interface that describes the desired functionality
  – Then implement this interface in any classes requiring that functionality
- Like `abstract` classes, interfaces are typically `public` types
  – Normally declared in files by themselves
    - File name = interface name plus the `.cs` extension

12.7 Case Study: Creating and Using Interfaces (Cont.)

- Example 1: Class hierarchy with interface **IAge**
  – (notice "**I**" in "**IAge**"– this is a convention used for naming interfaces)
  – Used polymorphically:
    - by class **Person**
    - by class **Tree**

next slide

Outline

IAge.cs

```
1   // Fig. 10.15: IAge.cs [in textbook ed.1]
2   // Interface IAge declares property for setting and getting age.
3
4   public interface IAge
5   {
6      int Age { get; }
7      string Name { get; }
8   }
```

Definition of interface IAge

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

Outline

Person.cs

```
1   // Fig. 10.16: Person.cs [in textbook ed.1]
2   // Class Person has a birthday.
3   using System;
4
5   public class Person : IAge
6   {
7      private string firstName;
8      private string lastName;
9      private int yearBorn;
10
11     // constructor
12     public Person( string firstNameValue, string lastNameValue,
13        int yearBornValue )
14     {
15        firstName = firstNameValue;
16        lastName = lastNameValue;
17
18        if ( yearBornValue > 0 && yearBornValue <= DateTime.Now.Year )
19           yearBorn = yearBornValue;
20        else
21           yearBorn = DateTime.Now.Year;
22     }
23
24     // property Age implementation of interface IAge
25     public int Age
26     {
27        get
28        {
29           return DateTime.Now.Year - yearBorn;
30        }
31     }
32
```

Definition of Age property (required)

Class Person implements the IAge interface

Outline

Person.cs

```
33     // property Name implementation of interface IAge
34     public string Name
35     {
36        get
37        {
38           return firstName + " " + lastName;
39        }
40     }
41
42  } // end class Person
```

Definition of Name property (required)

Outline

Tree.cs

```
1   // Fig. 10.17: Tree.cs [in textbook ed.1]
2   // Class Tree contains number of rings corresponding to its age.
3   using System;
4
5   public class Tree : IAge
6   {
7      private int rings; // number of rings in tree trunk
8
9      // constructor
10     public Tree( int yearPlanted )
11     {
12        // count number of rings in Tree
13        rings = DateTime.Now.Year - yearPlanted;
14     }
15
16     // increment rings
17     public void AddRing()
18     {
19        rings++;
20     }
21
22     // property Age implementation of interface IAge
23     public int Age
24     {
25        get
26        {
27           return rings;
28        }
29     }
```

Implementation of Age property (required)

Class Tree implements the IAge interface

Outline

Tree.cs

```
30
31     // property Name implementation of interface IAge
32     public string Name
33     {
34        get
35        {
36           return "Tree";
37        }
38     }
39
40  } // end class Tree
```

Definition of Name property (required)

Outline

InterfacesTest.c

```
1   // Fig. 10.18: InterfacesTest.cs [in textbook ed.1]
2   // Demonstrating polymorphism with interfaces (on the objects of
3   // disparate classes Tree and Person).
4   using System.Windows.Forms;
5
6   public class InterfacesTest
7   {
8      public static void Main( string[] args )
9      {
10        Tree tree = new Tree( 1978 );
11        Person person = new Person( "Bob", "Jones", 1971 );
12
13        // create array of IAge references
14        IAge[] iAgeArray = new IAge[ 2 ];
15
16        // iAgeArray[ 0 ] refers to Tree object polymorphically
17        iAgeArray[ 0 ] = tree;
18
19        // iAgeArray[ 1 ] refers to Person object polymorphically
20        iAgeArray[ 1 ] = person;
21
22        // display tree information
23        string output = tree + ": " + tree.Name + "\nAge is " +
24           tree.Age + "\n\n";
25
26        // display person information
27        output += person + ": " + person.Name + "\nAge is: "
28           + person.Age + "\n\n";
```

Create array of IAge references

Assign an IAge reference to reference a Person object

reference a Tree object

Outline

InterfacesTest.cs

```
29        // display name and age for each IAge object in iAgeArray
30        foreach ( IAge ageReference in iAgeArray )
31        {
32           output += ageReference.Name + ": Age is " +
33              ageReference.Age + "\n";
34        }
35
36        MessageBox.Show( output, "Demonstrating Polymorphism" );
37
38     } // end method Main
39
40  } // end class InterfacesTest
```

Use polymorphism to call the property of the appropriate class

Use foreach loop t… each element of the…

Program Output

12.7 Case Study: Creating and Using Interfaces (Cont.)

- **Common Interfaces of the .NET Framework Class Library**

Interface	Description
`IComparable`	Objects of a class that implements the interface can be compared to one another.
`IComponent`	Implemented by any class that represents a component, including Graphical User Interface (GUI) controls.
`IDisposable`	Implemented by classes that must provide an explicit mechanism for releasing resources.
`IEnumerator`	Used for iterating through the elements of a collection (such as an array) one element at a time.

Fig. 12.16 | Common interfaces of the .NET Framework Class Library.

12.8. Operator Overloading

- C# contains many operators that are defined for *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

12.8. Operator Overloading (Cont.)

- 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)

12.8. Operator Overloading (Cont.)

- C# enables you to overload most operators to make them sensitive to the context in which they are used.
- Example:

  Class `ComplexNumber` (**Fig. 12.17**) overloads the plus (+), minus (-) and multiplication (*) operators
  - To enable programs to add, subtract and multiply instances of class `ComplexNumber` using common mathematical notation.

12.8. Operator Overloading (Cont.)

Example (console implementation):

Class that overloads operators for adding, subtracting and multiplying complex numbers

Operations on complex numbers:

Let x = xR + xI * i and y= yR+ yI* i

Then:

x + y= (xR + yR) + (xI + yI) * i

x - y= (xR - yR) + (xI - yI) * i

x * y= (xR * yR – xI * yI) + (xR * yI + yR *xI) * i

Outline

ComplexNumber.cs

(2 of 4)

```
// Fig. 12.17: ComplexNumber.cs
// Class that overloads operators for adding, subtracting
// and multiplying complex numbers.
using System;

public class ComplexNumber
{
   // read-only property that gets the real component
   public double Real { get; private set; }

   // read-only property that gets the imaginary component
   public double Imaginary { get; private set; }

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

   // return string representation of ComplexNumber
   public override string ToString()
   {
      return string.Format( "({0} {1} {2}i)",
         Real, ( Imaginary < 0 ? "-" : "+" ), Math.Abs( Imaginary ) );
   } // end method ToString
```


++SKIP++

Outline

ComplexNumber.cs

(3 of 4)

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

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

   // 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 );
   } // end operator *
} // end class ComplexNumber
```

Overload the plus operator (+) to perform addition of ComplexNumbers

12.8 Operator Overloading (Cont.)

- Keyword **`operator`**, followed by an operator symbol, indicates that a method overloads the specified operator
- Methods that overload binary operators must take two arguments—
  - The first argument is the left operand
  - The second argument is the right operand
- Overloaded operator methods must be `public` and `static`

++SKIP++

Outline

OperatorOverloading.cs

(2 of 2)

```
      // prompt the user to enter the second complex number
      Console.Write( "\nEnter the real part of complex number y: " );
      realPart = Convert.ToDouble( Console.ReadLine() );
      Console.Write(
         "Enter the imaginary part of complex number y: " );
      imaginaryPart = Convert.ToDouble( Console.ReadLine() );
      y = new ComplexNumber( realPart, imaginaryPart );

      // display the results of calculations with x and y
      Console.WriteLine();
      Console.WriteLine( "{0} + {1} = {2}", x, y, x + y );
      Console.WriteLine( "{0} - {1} = {2}", x, y, x - y );
      Console.WriteLine( "{0} * {1} = {2}", x, y, x * y );
   } // end method Main
} // end class ComplexTest
```

```
Enter the real part of complex number x: 2
Enter the imaginary part of complex number x: 4

Enter the real part of complex number y: 4
Enter the imaginary part of complex number y: -2

(2 + 4i) + (4 - 2i) = (6 + 2i)
(2 + 4i) - (4 - 2i) = (-2 + 6i)
(2 + 4i) * (4 - 2i) = (16 + 12i)
```

Add, subtract and multiply x and y using the overloaded operators, then output the results.

OPTIONAL++ 12.8. Operator Overloading (Cont.)

Example (Windows Forms implementation):

The same class that overloads operators for adding, subtracting and multiplying complex numbers

The following slides

Slide modified by L. Lilien

OPTIONAL++

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();


   static void Main()
   {
      Application.Run( new ComplexTest() );
   }
```

OPTIONAL++

Outline

61

OperatorOverloading.cs

```
35    private void firstButton_Click(
36       object sender, System.EventArgs e )
37    {
38       x.Real = Int32.Parse( realTextBox.Text );
39       x.Imaginary = Int32.Parse( imaginaryTextBox.Text );
40       realTextBox.Clear();
41       imaginaryTextBox.Clear();
42       statusLabel.Text = "First Complex Number is: " + x;
43    }
44
45    private void secondButton_Click(
46       object sender, System.EventArgs e )
47    {
48       y.Real = Int32.Parse( realTextBox.Text );
49       y.Imaginary = Int32.Parse( imaginaryTextBox.Text );
50       realTextBox.Clear();
51       imaginaryTextBox.Clear();
52       statusLabel.Text = "Second Complex Number is: " + y;
53    }
54
55    // add complex numbers
56    private void addButton_Click( object sender, System.EventArgs e )
57    {
58       statusLabel.Text = x + " + " + y + " = " + ( x + y );
59    }
60
61    // subtract complex numbers
62    private void subtractButton_Click(
63       object sender, System.EventArgs e )
64    {
65       statusLabel.Text = x + " - " + y + " = " + ( x - y );
66    }
67
```

Use overloaded addition operator to add two ComplexNumbers

Use overloaded subtraction operator to subtract two ComplexNumbers

OPTIONAL++

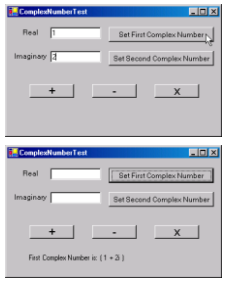
Outline

62

OperatorOverloading.cs

```
68    // multiply complex numbers
69    private void multiplyButton_Click(
70       object sender, System.EventArgs e )
71    {
72       statusLabel.Text = x + " * " + y + " = " + ( x * y );
73    }
74
75 } // end class ComplexTest
```

Use overloaded multiplication operator to multiply two ComplexNumbers



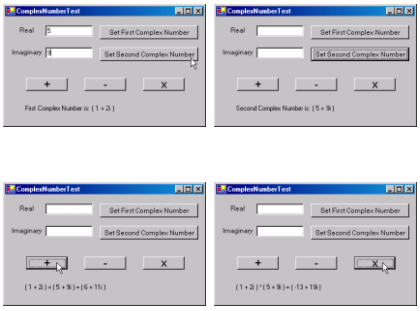
OPTIONAL++

Outline

63

OperatorOverloading.cs

Program Output



[10.10 in textbook ed.1] Delegates

64

- 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

Slide modified by L. Lilien

[10.10 in textbook ed.1] Delegates (Cont.)

65

- 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

Slide modified by L. Lilien

[10.10 in textbook ed.1] Delegates (Cont.)

66

- 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

Slide modified by L. Lilien

[10.10 in textbook ed.1]
Delegates (Cont.)

- 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

[10.10 in textbook ed.1]
Delegates (Cont.)

- 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]

[10.10 in textbook ed.1]
Delegates (Cont.)

- 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

Delegates (Cont.)

Outline

```
1  // Fig. 10.24: DelegateBubbleSort.cs [in textbook ed.1]
2  // Demonstrating delegates for sorting numbers.
3
4  public class DelegateBubbleSort
5  {
6     public delegate bool Comparator(
7        int element2 );  // Declare delegate Comparator, i.e., declare
                          // signature for the
                       // No implementation here
8
9     // sort array using Comparator delegate
10    public static void SortArray( int[] array,
11       Comparator Compare )  // Reference to d
                               // passed as a p
12    {
13       for ( int pass = 1; pass < array.Length
14
15          for ( int i = 0; i < array.Length - 1; i++ )
16
17             if ( Compare( array[ i ], array [ i + 1 ] ) )
                   // if Compare returns true, elements are out of order
18                Swap( ref array[ i ], ref array[ i + 1 ] );
19    }
20
21    // swap two elements
22    private static void Swap( ref int firstElement,
23       ref int secondElement )
24    {
25       int hold = firstElement;
26       firstElement = secondElement;
27       secondElement = hold;
28    }
29 }
```

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

Delegates (Cont.)

Outline

BubbleSortForm.cs

```
1  // Fig. 10.25: BubbleSortForm.cs [in textbook ed.1]
2  // Demonstrates bubble sort using delegates to determine
3  // the sort order.
   // Some thing will be magic for you (e.g. button click handling)
4  using System;
5  using System.Drawing;
6  using System.Collections;
7  using System.ComponentModel;
8  using System.Windows.Forms;
9
10 public class BubbleSortForm : System.Windows.Forms.Form
11 {
12    private System.Windows.Forms.TextBox originalTextBox;
13    private System.Windows.Forms.TextBox sortedTextBox;
14    private System.Windows.Forms.Button createButton; // 3 buttons
15    private System.Windows.Forms.Button ascendingButton;
16    private System.Windows.Forms.Button descendingButton;
17    private System.Windows.Forms.Label originalLabel;
18    private System.Windows.Forms.Label sortedLabel;
19
20    private int[] elementArray = new int[ 10 ];
21
22    // create randomly generated set of numbers to sort
23    private void createButton_Click( object sender,
24       System.EventArgs e )          // magic here
25    {
26       // clear TextBoxes
27       originalTextBox.Clear();
28       sortedTextBox.Clear();
29
30       // create random-number generator
31       Random randomNumber = new Random();
32
```


Delegates (Cont.)

Outline

BubbleSortForm.cs

```
33       // populate elementArray with random integers
34       for ( int i = 0; i < elementArray.Length; i++ )
35       {
36          elementArray[ i ] = randomNumber.Next( 100 );
37          originalTextBox.Text += elementArray[ i ] + "\r\n";
38       }
39    }
40
41    // delegate implementation for ascending sort
   // More precisely: implementation of SortAscending method which can be
   // passed to another method via Comparator delegate (matches its signature)
42    private bool SortAscending( int element1, int element2 )
43    {
44       return element1 > element2;
45    }
46
47    // sort randomly generated numbers in ascend
48    private void ascendingButton_Click( object sender,
49       System.EventArgs e )          // magic here
50    {
51       // sort array, passing delegate for SortAs
52       DelegateBubbleSort.SortArray( elementArray,
53          new DelegateBubbleSort.Comparator(
54             SortAscending ) );
55
56       DisplayResults();
57    }
58
59    // delegate implementation for desc. sort
   // More precisely: implementation of SortDescending method which can be
   // passed to another method via Comparator delegate (matches its signature)
60    private bool SortDescending( int element1, int element2 )
61    {
62       return element1 < element2;
63    }
64
```

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

Delegates (Cont.)

Outline 73

BubbleSortForm.cs

```
65      // sort randomly generating numbers in descending order
66      private void descendingButton_Click( object sender,
67         System.EventArgs e )                  // magic here
68      {
69         // sort array, passing delegate for SortDescending
70         DelegateBubbleSort.SortArray( elementArray,
71            new DelegateBubbleSort.Comparator(
72               SortDescending ) );
73
74         DisplayResults();
75      }
76
77      // display the sorted array in sortedTextBox
78      private void DisplayResults()
79      {
80         sortedTextBox.Clear();
81
82         foreach ( int element in elementArray )
83            sortedTextBox.Text += element + "
84      }
85
86      // main entry point for application
87      public static void Main( string[] args )
88      {
89         Application.Run( new BubbleSortForm() );
                     // new instance waits for button click
90      }
91   }
```

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

Delegates (Cont.)

Outline 74

BubbleSortForm.cs

Program Output