12

Polymorphism, Interfaces & Operator Overloading

12.6 `sealed` Methods and Classes

- A method declared **`sealed`** in a base class cannot be overridden in a derived class.
- Methods that are declared `private` are implicitly `sealed`.
- Methods that are declared `static` also are implicitly `sealed`, because `static` methods cannot be overridden either.
- A derived-class method declared both `override` and `sealed` can override a base-class method, but cannot be overridden in classes further down the inheritance hierarchy.
- Calls to `sealed` methods are resolved at compile time—this is known as **static binding**.

12.6 `sealed` Methods and Classes (Cont.)

Performance Tip 12.1

The compiler can decide to inline a `sealed` method call and will do so for small, simple `sealed` methods. Inlining does not violate encapsulation or information hiding, but does improve performance, because it eliminates the overhead of making a method call.

12.7 Case Study: Creating and Using Interfaces

- Interfaces define and standardize the ways in which people and systems can interact with one another.
- A C# interface describes a set of methods that can be called on an object—to tell it, for example, to perform some task or return some piece of information.
- An **interface declaration** begins with the keyword `interface` and can contain only abstract methods, properties, indexers and events.
- All interface members are implicitly declared both `public` and `abstract`.
- An interface can extend one or more other interfaces to create a more elaborate interface that other classes can implement.

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

- An interface is typically used when disparate (i.e., unrelated) classes need to share common methods so that they can be processed polymorphically
- A programmer can create an interface that describes the desired functionality, then implement this interface in any classes requiring that functionality.
- An interface often is used in place of an `abstract` class when there is no default implementation to inherit—that is, no fields and no default method implementations.
- Like `abstract` classes, interfaces are typically `public` types, so they are normally declared in files by themselves with the same name as the interface and the `.cs` file-name extension.

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

12.7.1 Developing an `IPayable` Hierarchy

- To build an application that can determine payments for employees and invoices alike, we first create an interface named `IPayable`.
- Interface `IPayable` contains method `GetPaymentAmount` that returns a `decimal` amount to be paid for an object of any class that implements the interface.

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

- The UML class diagram in Fig. 12.10 shows the interface and class hierarchy used in our accounts-payable application.

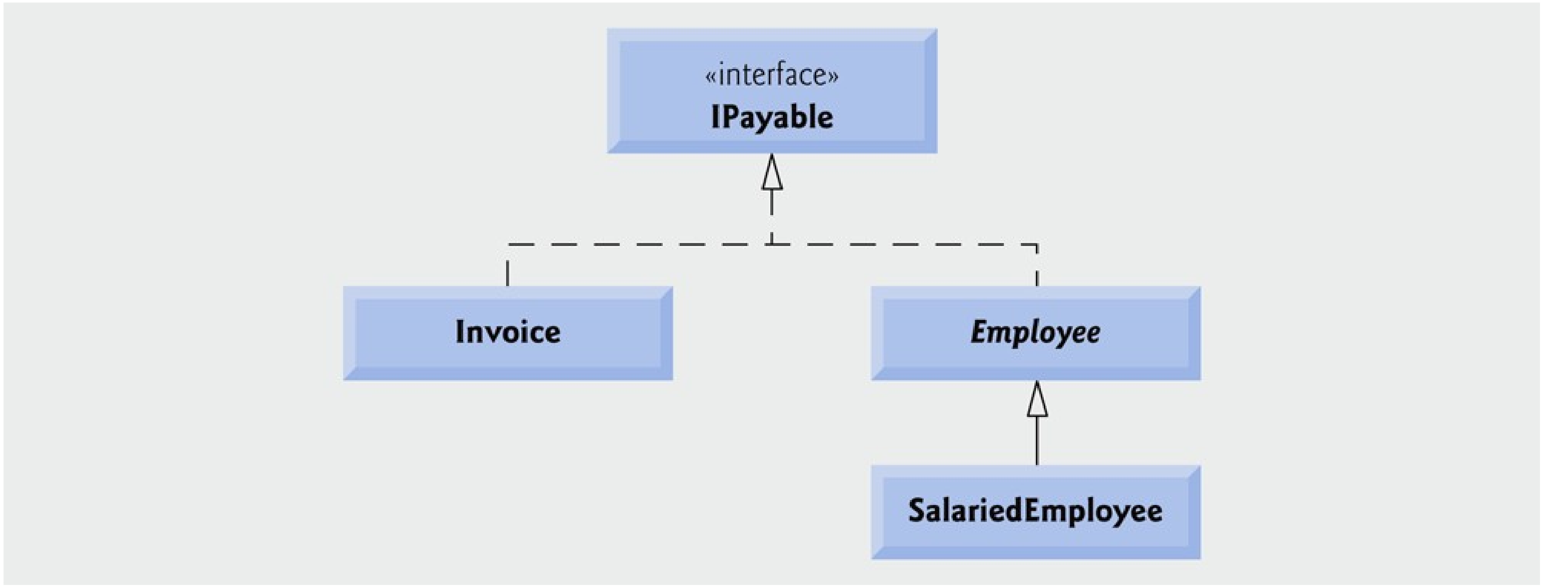


Fig. 12.10 | `IPayable` interface and class hierarchy UML class diagram.

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

- The UML distinguishes an interface from a class by placing the word “interface” in guillemets (« and ») above the interface name.
- The UML expresses the relationship between a class and an interface through a **realization**.

Outline

- Interface `IPayable` is declared in Fig. 12.11.

`IPayable.cs`

```
// Fig12.11: IPayable.cs
// IPayable interface declaration.
public interface IPayable
{
  decimal GetPaymentAmount(); // calculate payment; no implementation
} // end interface IPayable
```

Fig. 12.11 | `IPayable` interface declaration.

Outline

Invoice.cs

(1 of 3)

- We now create class `Invoice` (Fig. 12.12) represents a simple invoice that contains billing information for one kind of part.

```
// Fig12.12: Invoice.cs
// Invoice class implements IPayable.
public class Invoice : IPayable
{
   private int quantity;
   private decimal pricePerItem;

    // property that gets and sets the part number on the invoice
   public string PartNumber { get; set; }

    // property that gets and sets the part description on the invoice
   public string PartDescription { get; set; }

    // four-parameter constructor
   public Invoice( string part, string description, int count,
      decimal price )
   {
      PartNumber = part;
      PartDescription = description;
      Quantity = count; // validate quantity via property
      PricePerItem = price; // validate price per item via property
   } // end four-parameter Invoice constructor
```

Class `Invoice` implements interface `IPayable`. Like all classes, class `Invoice` also implicitly inherits from class `object`.

Fig. 12.12 | `Invoice` class implements `IPayable`. (Part 1 of 3.)

Outline

Invoice.cs

(2 of 3)

```

   //  property  that  gets  and sets  the  quantity  on the  invoice
   public int Quantity
   {
      get
      {
         return quantity;
      } //  end  get
      set
      {
         quantity = ( value < 0 ) ? 0 : value; // validate quantity
      } // end set
   } // end property Quantity

   // property that gets and sets the price per item
   public decimal PricePerItem
   {
      get
      {
         return pricePerItem;
      } // end get
```

Fig. 12.12 | `Invoice` class implements `IPayable`. (Part 2 of 3.)

Outline

Invoice.cs

(3 of 3)

```
      set
         {
            pricePerItem = ( value < 0 ) ? 0 : value; // validate price
      } // end set
   } // end property PricePerItem

   // return string representation of Invoice object
   public override string ToString()
   {
      return string.Format(
         "{0}: \n{1}: {2} ({3}) \n{4}: {5} \n{6}: {7:C}",
         "invoice", "part number", PartNumber, PartDescription,
         "quantity", Quantity, "price per item", PricePerItem );
   } // end method ToString

   // method required to carry out contract with interface IPayable
   public decimal GetPaymentAmount()
   {
      return Quantity * PricePerItem; // calculate total cost
   } // end method GetPaymentAmount
} //  end class Invoice
```

Invoice implements the IPayable interface by declaring a GetPaymentAmount method.

Fig. 12.12 | Invoice class implements IPayable. (Part 3 of 3.)

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

- C# does not allow derived classes to inherit from more than one base class, but it does allow a class to implement any number of interfaces.
- To implement more than one interface, use a comma-separated list of interface names after the colon (`:`) in the class declaration.
- When a class inherits from a base class and implements one or more interfaces, the class declaration must list the base-class name before any interface names.

Outline

- Figure 12.13 contains the `Employee` class, modified to implement interface `IPayable`.

`Employee.cs`

(1 of 2)

```
// Fig12.13: Employee.cs
// Employee abstract base class.
public abstract c Employee : IPayable
{
   // read-only property that gets employee's first name
   public string FirstName { get; private set; }

   // read-only property that gets employee's last name
   public string LastName { get; private set; }

   // read-only property that gets employee's social security number
   public string SocialSecurityNumber { get; private set; }

   // three-parameter constructor
   public Employee( string first, string last, string ssn )
   {
      FirstName = first;
      LastName = last;
      SocialSecurityNumber = ssn;
   } // end three-parameter Employee constructor
```

Class `Employee` now implements interface `IPayable`.

Fig. 12.13 | `Employee` abstract base class. (Part 1 of 2.)

Outline

Employee.cs

(2 of 2)

```
   // return string representation of Employee object
   public override string ToString()
   {
      return string.Format( "{0} {1}\nsocial security number: {2}",
         FirstName, LastName, SocialSecurityNumber );
   } // end method ToString

   // Note: We do not implement IPayable method GetPaymentAmount here, so
   // this class must be declared abstract to avoid a compilation error.
   public abstract decimal GetPaymentAmount();
} // end abstract class Employee
```

`Earnings` has been renamed to `GetPaymentAmount` to match the interface's requirements.

Fig. 12.13 | `Employee` abstract base class. (Part 2 of 2.)

Outline

SalariedEmployee
.cs

(1 of 2)

- Figure 12.14 contains a modified version of class `SalariedEmployee` that extends `Employee` and implements method `GetPaymentAmount`.

```
// Fig12.14: SalariedEmployee.cs
// SalariedEmployee class that extends Employee.
public class SalariedEmployee : Employee
{
   private decimal weeklySalary;

   // four-parameter constructor
   public SalariedEmployee( string first, string last, string ssn,
      decimal salary ) : base( first, last, ssn )
   {
      WeeklySalary = salary; // validate salary via property
   } // end four-parameter SalariedEmployee constructor

   // property that gets and sets salaried employee's salary
   public decimal WeeklySalary
   {
      get
      {
         return weeklySalary;
      } // end get
```

Fig. 12.14 | SalariedEmployee class that extends Employee. (Part 1 of 2.)

Outline

SalariedEmployee.cs

(2 of 2)

```
      set
         {
            weeklySalary = value < 0 ? 0 : value; // validation
      } // end set
   } // end property WeeklySalary

   // calculate earnings; implement interface IPayable method
   // that was abstract in base class Employee
   public override decimal GetPaymentAmount()
   {
      return WeeklySalary;
   } // end method GetPaymentAmount

   // return string representation of SalariedEmployee object
   public override string ToString()
   {
      return string.Format( "salaried employee: {0}\n{1}: {2:C}",
         base.ToString(), "weekly salary", WeeklySalary );
   } // end method ToString
} // end class SalariedEmployee
```

Method `GetPaymentAmount` replaces method `Earnings`, keeping the same functionality.

Fig. 12.14 | `SalariedEmployee` class that extends `Employee`. (Part 2 of 2.)

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

- The remaining `Employee` derived classes also must be modified to contain method `GetPaymentAmount` in place of `Earnings` to reflect the fact that `Employee` now implements `IPayable`.
- When a class implements an interface, the same *is-a* relationship provided by inheritance applies.

Outline

- PayableInterfaceTest (Fig. 12.15) illustrates that interface IPayable can be used to process a set of Invoices and Employees polymorphically in a single application.

PayableInterface
Test.cs

(1 of 3)

```
// Fig12.15: PayableInterfaceTest.cs
// Tests interface IPayable with disparate classes.
using System;

public class PayableInterfaceTest
{
  public static void Main( string[] args )
  {
    // create four-element IPayable array
    IPayable[] payableObjects = new IPayable[ 4];

    // populate array with objects that implement IPayable
    payableObjects[ 0] = new Invoice( "01234", "seat", 2, 375.00M );
    payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95M );
    payableObjects[ 2 ] = new SalariedEmployee( "John", "Smith",
      "111-11-1111", 800.00M );
    payableObjects[ 3 ] = new SalariedEmployee( "Lisa", "Barnes",
      "888-88-8888", 1200.00M );
```

Fig. 12.15 | Tests interface IPayable with disparate classes. (Part 1 of 3.)

Outline

PayableInterfaceTest.cs

(2 of 3)

```

      Console.WriteLine(
        "Invoices and Employees processed polymorphically:\n" );

    // generically process each element in array payableObjects
    foreach( var currentPayable in payableObjects )
    {
      // output currentPayable and its appropriate payment amount
      Console.WriteLine( "payment due \n{0}: {1:C}\n",
        currentPayable, currentPayable.GetPaymentAmount() );
    } // end foreach
  } // end Main
} // end class PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00
```

(continued on next page...)

Fig. 12.15 | Tests interface `IPayable` with disparate classes. (Part 2 of 3.)

Outline

(continued from previous page…)

```
invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

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

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

`PayableInterfaceTest.cs`

(3 of 3)

Fig. 12.15 | Tests interface `IPayable` with disparate classes. (Part 3 of 3.)

Software Engineering Observation 12.8

All methods of class `object` can be called by using a reference of an interface type—the reference refers to an object, and all objects inherit the methods of class `object`.

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

- **12.7.7 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.

Software Engineering Observation 12.9

Use operator overloading when it makes an application clearer than accomplishing the same operations with explicit method calls.

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

Outline

ComplexNumber.cs

(2 of 4)

```
// Fig12.17: 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
```

Fig. 12.17 | Class that overloads operators for adding, subtracting and multiplying complex numbers. (Part 1 of 3.)

Outline

`ComplexNumber.cs`

(3 of 4)

```

   // 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

   // 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 plus operator (+) to perform addition of `ComplexNumber`s

Fig. 12.17 | Class that overloads operators for adding, subtracting and multiplying complex numbers. (Part 2 of 3.)

Outline

`ComplexNumber.cs`

(4 of 4)

```
   // 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
```

Fig. 12.17 | Class that overloads operators for adding, subtracting and multiplying complex numbers. (Part 3 of 3.)

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, and the second argument is the right operand.
- Overloaded operator methods must be `public` and `static`.

Outline

- Class `ComplexTest` (Fig. 12.18) demonstrates the overloaded operators for adding, subtracting and multiplying `ComplexNumber`s.

`OperatorOverloading.cs`

(1 of 2)

```
// Fig12.18: OperatorOverloading.cs
// Overloading operators for complex numbers.
using System;

public class ComplexTest
{
   public static void Main( string[] args )
   {
      // declare two variables to store complex numbers
      // to be entered by user
      ComplexNumber x, y;

      // prompt the user to enter the first complex number
      Console.Write( "Enter the real part of complex number x: " );
      double realPart = Convert.ToDouble( Console.ReadLine() );
      Console.Write(
         "Enter the imaginary part of complex number x: " );
      double imaginaryPart = Convert.ToDouble( Console.ReadLine() );
      x = new ComplexNumber( realPart, imaginaryPart );

```

Fig. 12.18 | Overloading operators for complex numbers. (Part 1 of 2.)

Outline

```
      // 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
```

`OperatorOverloading.cs`

(2 of 2)

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

```
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)
```

Fig. 12.18 | Overloading operators for complex numbers. (Part 2 of 2.)

13

Exception Handling

13.1 Introduction

- An **exception** is an indication of a problem that occurs during a program’s execution.
- Exception handling enables applications to resolve exceptions.
- Exception handling enables clear, **robust** and more **fault-tolerant programs**.

Error-Prevention Tip 13.1

Exception handling helps improve a program’s fault tolerance.

13.2 Exception Handling Overview (Cont.)

- Exception handling enables programmers to remove error-handling code from the “main line” of the program’s execution.
- Programmers can decide to handle all exceptions, all exceptions of a certain type or all exceptions of related types.
- Such flexibility reduces the likelihood that errors will be overlooked.

Outline

- Figure 13.1's application divides one input integer by a second to obtain an `int` result.
- In this example, we'll see that an exception is **thrown** when a method detects a problem.

DivideByZeroNoExceptionHandling.cs

(1 of 3)

```
// Fig. 13.1: DivideByZeroNoExceptionHandling.cs
// Integer division without exception handling.
using System;

class DivideByZeroNoExceptionHandling
{
   static void Main()
   {
      // get numerator and denominator
      Console.Write( "Please enter an integer numerator: " );
      int numerator = Convert.ToInt32( Console.ReadLine() );
      Console.Write( "Please enter an integer denominator: " );
      int denominator = Convert.ToInt32( Console.ReadLine() );

      // divide the two integers, then display the result
```

Converting values can cause a `FormatException`.

Converting values can cause a `FormatException`.

Fig. 13.1 | Integer division without exception handling. (Part 1 of 3.)

Outline

DivideByZeroNoExceptionHandling.cs

(2 of 3)

```
16       int result = numerator / denominator;
17       Console.WriteLine( "\nResult:  {0:D} /  {1:D}  = {2:D}",
18          numerator, denominator, result );
19     } // end Main
20  } // end class DivideByZeroNoExceptionHandling
```

Division can cause a `DivideByZeroException`.

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Unhandled Exception: System.DivideByZeroException:
   Attempted to divide by zero.
      at DivideByZeroNoExceptionHandling.Main()
         in C:\examples\ch13\Fig13_01\DivideByZeroNoExceptionHandling\
         DivideByZeroNoExceptionHandling\
      DivideByZeroNoExceptionHandling.cs: line 16
```

Fig. 13.1 | Integer division without exception handling. (Part 2 of 3.)

Outline

DivideByZeroNo ExceptionHandling .cs

(3 of 3)

```
Please enter an integer numerator:100
Please enter an integer denominator: hello

Unhandled Exception: System.FormatException:
   Input string was not in a correct format.
    at System.Number.StringToNumber(String str, NumberStyles options,
       NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
    at System.Number.ParseInt32(String s, NumberStyles style,
       NumberFormatInfo info)
    at System.Convert.ToInt32(String value)
    at DivideByZeroNoExceptionHandling.Main()
       in C:\examples\ch13\Fig13_01\DivideByZeroNoExceptionHandling\
       DivideByZeroNoExceptionHandling\
     DivideByZeroNoExceptionHandling.cs: line 13
```

Fig. 13.1 | Integer division without exception handling. (Part 3 of 3.)

13.3 Example: Divide by Zero without Exception Handling

- If you run using **Debug > Start Debugging**, the program pauses at the line where an exception occurs.
- Try executing the application from a **Command Prompt** window.
- When an error arises, a dialog indicates that the application has encountered a problem and needs to close.
- An error message describing the problem is displayed in the **Command Prompt**.

13.3 Example: Divide by Zero without Exception Handling (Cont.)

- Additional information—known as a **stack trace**—displays the exception name and the path of execution that led to the exception.
- Each "`at`" line in the stack trace indicates a line of code in the particular method that was executing when the exception occurred.
- This information tells where the exception originated, and what method calls were made to get to that point.

Outline

DivideByZeroTest.cs

(1 of 4)

- This application (Fig. 13.2) uses exception handling to process `DivideByZeroException`s and `FormatException`s.
- This program demonstrates how to **catch** and **handle** (i.e., deal with) such exceptions.

```
// Fig. 13.2: DivideByZeroTest.cs
// FormatException and DivideByZeroException handlers.
using System;
using System.Windows.Forms;

namespace DivideByZeroTest
{
  public partial class DivideByZeroTestForm : Form
  {
    public DivideByZeroTestForm()
    {
      InitializeComponent();
    } // end constructor

```

Fig. 13.2 | `FormatException` and `DivideByZeroException` handlers. (Part 1 of 4.)

Outline

DivideByZeroTest.cs

(2 of 4)

```
      // obtain 2 integers from the user
      // and divide numerator by denominator
      private void divideButton_Click( object sender, EventArgs e )
      {
         outputLabel.Text = ""; // clear Label OutputLabel

         // retrieve user input and calculate quotient
         try
         {
            // Convert.ToInt32 generates FormatException
            // if argument cannot be converted to an integer
            int numerator = Convert.ToInt32( numeratorTextBox.Text );
            int denominator = Convert.ToInt32( denominatorTextBox.Text );

            // division generates DivideByZeroException
            // if denominator is 0
            int result = numerator / denominator;

            // display result in OutputLabel
            outputLabel.Text = result.ToString();
         } // end try
```

Fig. 13.2 | `FormatException` and `DivideByZeroException` handlers. (Part 2 of 4.)

Outline

DivideByZeroTest.cs

(3 of 4)

```
         catch ( FormatException )
         {
            MessageBox.Show( "You must enter two integers.",
               "Invalid Number Format", MessageBoxButtons.OK,
               MessageBoxIcon.Error );
         } // end catch
         catch ( DivideByZeroException divideByZeroExceptionParameter )
         {
            MessageBox.Show( divideByZeroExceptionParameter.Message,
               "Attempted to Divide by Zero", MessageBoxButtons.OK,
               MessageBoxIcon.Error );
         } // end catch
      } // end method divideButton_Click
   } // end class DivideByZeroTestForm
} // end namespace DivideByZeroTest
```

This block catches and handles a `FormatException`.

This block catches and handles a `DivideByZeroException`.

Fig. 13.2 | `FormatException` and `DivideByZeroException` handlers. (Part 3 of 4.)

13.4 Example: Handling DivideByZeroExceptions and FormatExceptions

- The Int32.TryParse method converts a string to an int value if possible.
- The method requires two arguments—one is the string to parse and the other is the variable in which the converted value is to be stored.
- The method returns true if the string was parsed successfully.
- If the string could not be converted, the value 0 is assigned to the second argument.

13.4 Example: Handling `DivideByZeroException`s and `FormatException`s (Cont.)

13.4.1 Enclosing Code in a `try` Block

- A **`try` block** encloses code that might throw exceptions and code that is skipped when an exception occurs.

13.4 Example: Handling `DivideByZeroExceptions` and `FormatExceptions` (Cont.)

13.4.2 Catching Exceptions

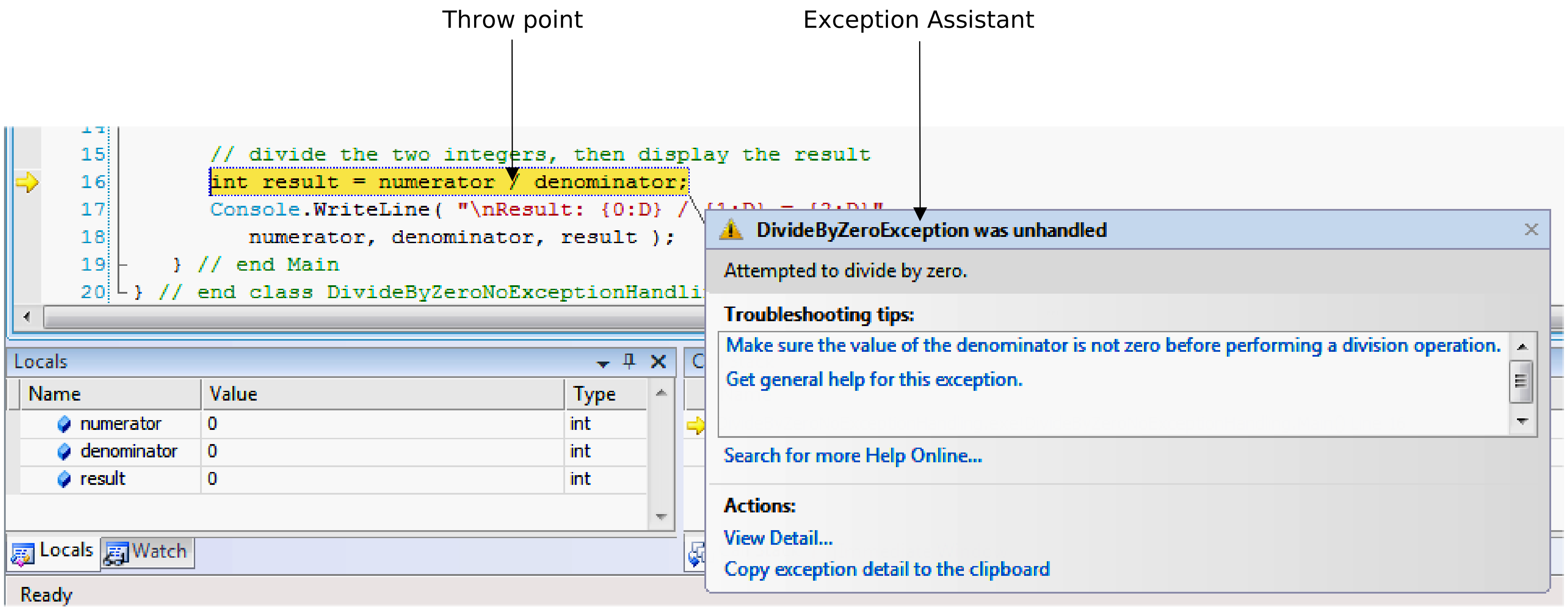
- When an exception occurs in a `try` block, a corresponding **`catch` block** catches the exception and handles it.
- At least one `catch` block must immediately follow a `try` block.
- A `catch` block specifies an exception parameter representing the exception that the `catch` block can handle.
- Optionally, you can include a `catch` block that does not specify an exception type to catch all exception types.

13.4 Example: Handling `DivideByZeroException`s and `FormatException`s (Cont.)

13.4.3 Uncaught Exceptions

- An **uncaught exception** (or **unhandled exception**) is an exception for which there is no matching **`catch`** block.
- If you run the application from Visual Studio with debugging, a window called the **Exception Assistant** (Fig. 13.3) appears.

13.4 Example: Handling `DivideByZeroExceptions` and `FormatExceptions` (Cont.)



```
        // divide the two integers, then display the result
        int result = numerator / denominator;
        Console.WriteLine( "\nResult: {0:D} / {1:D} = {2:D}",
           numerator, denominator, result );
     } // end Main
  } // end class DivideByZeroNoExceptionHandli
```

Name	Value	Type
numerator	0	int
denominator	0	int
result	0	int

DivideByZeroException was unhandled

Attempted to divide by zero.

Troubleshooting tips:

Make sure the value of the denominator is not zero before performing a division operation.

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

Fig. 13.3 | Exception Assistant.

13.4 Example: Handling `DivideByZeroExceptions` and `FormatExceptions` (Cont.)

13.4.4 Termination Model of Exception Handling

- When a method called in a program or the CLR detects a problem, the method or the CLR throws an exception.
- The point at which an exception occurs is called the throw point
- If an exception occurs in a `try` block, program control immediately transfers to the first `catch` block matching the type of the thrown exception.
- After the exception is handled, program control resumes after the last `catch` block.
- This is known as the **termination model of exception handling**.

13.5 .NET Exception Hierarchy

- In C#, only objects of class `Exception` and its derived classes may be thrown and caught.
- Exceptions thrown in other .NET languages can be caught with the general `catch` clause.

13.5 .NET Exception Hierarchy (Cont.)

13.5.1 Class `SystemException`

- Class **`Exception`** is the base class of .NET’s exception class hierarchy.
- The CLR generates **`SystemException`**`s`, derived from class `Exception`, which can occur at any point during program execution.
- If a program attempts to access an **out-of-range array index**, the CLR throws an exception of type **`IndexOutOfRangeException`**.
- Attempting to use a `null` reference causes a **`NullReferenceException`**.

13.5 .NET Exception Hierarchy (Cont.)

- A `catch` block can use a base-class type to catch a hierarchy of related exceptions.
- A `catch` block that specifies a parameter of type `Exception` can catch all exceptions.
- This technique makes sense only if the handling behavior is the same for a base class and all derived classes.

Common Programming Error 13.3

The compiler issues an error if a `catch` block that catches a base-class exception is placed before a `catch` block for any of that class's derived-class types. In this case, the base-class `catch` block would catch all base-class and derived-class exceptions, so the derived-class exception handler would never execute—a possible logic error.

13.5 .NET Exception Hierarchy (Cont.)

13.5.2 Determining Which Exceptions a Method Throws

- Search for "`Convert.ToInt32` method" in the **Index** of the Visual Studio online documentation.
- Select the document entitled **Convert.ToInt32 Method (System)**.
- In the document that describes the method, click the link **ToInt32(String)**.
- The **Exceptions** section indicates that method `Convert.ToInt32` throws two exception types.

13.6 `finally` Block

- Programs frequently request and release resources dynamically.
- Operating systems typically prevent more than one program from manipulating a file.
- Therefore, the program should close the file (i.e., release the resource) so other programs can use it.
- If the file is not closed, a resource leak occurs.

13.6 `finally` Block (Cont.)

- Exceptions often occur while processing resources.
- Regardless of whether a program experiences exceptions, the program should close the file when it is no longer needed.
- C# provides the `finally` block, which is guaranteed to execute regardless of whether an exception occurs.
- This makes the `finally` block ideal to release resources from the corresponding `try` block.

13.6 `finally` Block (Cont.)

- Local variables in a `try` block cannot be accessed in the corresponding `finally` block, so variables that must be accessed in both should be declared before the `try` block.

Error-Prevention Tip 13.3

A `finally` block typically contains code to release resources acquired in the corresponding `try` block, which makes the `finally` block an effective mechanism for eliminating resource leaks.

Outline

- The application in Fig. 13.4 demonstrates that the `finally` block always executes.

UsingExceptions.cs

(1 of 10)

```
// Fig. 13.4: UsingExceptions.cs
// Using finally blocks.
// finally blocks always execute, even when no exception occurs.
using System;

class UsingExceptions
{
  static void Main()
  {
     // Case 1: No exceptions occur in called method
     Console.WriteLine( "Calling DoesNotThrowException" );
     DoesNotThrowException();

     // Case 2: Exception occurs and is caught in called method
     Console.WriteLine( "\nCalling ThrowExceptionWithCatch" );
     ThrowExceptionWithCatch();

     // Case 3: Exception occurs, but is not caught in called method
     // because there is no catch block.
     Console.WriteLine( "\nCalling ThrowExceptionWithoutCatch" );
```

`Main` invokes method `DoesNotThrowException`.

`Main` invokes method `ThrowExceptionWithCatch`.

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 1 of 8.)

Outline

`UsingExceptions.cs`

(2 of 10)

```
        // call ThrowExceptionWithoutCatch
      try
        {
        ThrowExceptionWithoutCatch();
        //}end try
      catch
        {
        Console.WriteLine( "Caught exception from " +
          "ThrowExceptionWithoutCatch in Main" );
      } // end catch

      // Case 4: Exception occurs and is caught in called method,
      // then rethrown to caller.
      Console.WriteLine( "\nCalling ThrowExceptionCatchRethrow" );

      // call ThrowExceptionCatchRethrow
      try
      {
        ThrowExceptionCatchRethrow();
      } // end try
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 2 of 8.)

Outline

UsingExceptions.cs

(3 of 10)

```
      catch
        {
         Console.WriteLine( "Caught exception from " +
            "ThrowExceptionCatchRethrow in Main" );
      } // end catch
   } // end method Main

    // no exceptions thrown
   static void DoesNotThrowException()
   {
        // try block does not throw any exceptions
      try
      {
         Console.WriteLine( "In DoesNotThrowException" );
      } // end try
      catch
      {
         Console.WriteLine( "This catch never executes" );
      } // end catch
```

Because the `try` block does not throw any exceptions, the `catch` block is ignored.

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 3 of 8.)

Outline

UsingExceptions.cs

(4 of 10)

```
      finally
      {
         Console.WriteLine( "finally executed in DoesNotThrowException" );
      } // end finally

      Console.WriteLine( "End of DoesNotThrowException" );
   } // end method DoesNotThrowException

   // throws exception and catches it locally
   static void ThrowExceptionWithCatch()
   {
      // try block throws exception
      try
      {
         Console.WriteLine( "In ThrowExceptionWithCatch" );
         throw new Exception( "Exception in ThrowExceptionWithCatch" );
      } // end try
      catch ( Exception exceptionParameter )
      {
         Console.WriteLine( "Message: " + exceptionParameter.Message );
      } // end catch
```

The `finally` block always executes.

The `try` block throws an exception.

The `catch` and `finally` blocks execute when the exception occurs.

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 4 of 8.)

Outline

UsingExceptions.cs

(5 of 10)

```
      finally
      {
         Console.WriteLine(
            "finally executed in ThrowExceptionWithCatch" );
      } // end finally

      Console.WriteLine( "End of ThrowExceptionWithCatch" );
   } // end method ThrowExceptionWithCatch

   // throws exception and does not catch it locally
   static void ThrowExceptionWithoutCatch()
   {
      // throw exception, but do not catch it
      try
      {
         Console.WriteLine( "In ThrowExceptionWithoutCatch" );
         throw new Exception( "Exception in ThrowExceptionWithoutCatch" );
      } // end try
```

The `catch` and `finally` blocks execute when the exception occurs.

The `try` block throws an exception.

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 5 of 8.)

Outline

UsingExceptions.cs

(6 of 10)

```
      finally
      {
         Console.WriteLine( "finally executed in " +
            "ThrowExceptionWithoutCatch" );
      } //  end finally

      // unreachable code; logic error
      Console.WriteLine( "End of ThrowExceptionWithoutCatch" );
   } // end method ThrowExceptionWithoutCatch

   // throws exception, catches it and rethrows it
   static void ThrowExceptionCatchRethrow()
   {
      // try block throws exception
      try
      {
         Console.WriteLine( "In ThrowExceptionCatchRethrow" );
         throw new Exception( "Exception in ThrowExceptionCatchRethrow" );
      } // end try
```

The `finally` block executes but the exception remains uncaught until after control returns to `Main`.

The `try` block throws an exception.

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 6 of 8.)

Outline

UsingExceptions.cs

(7 of 10)

```
      catch ( Exception exceptionParameter )
      {
         Console.WriteLine( "Message: " + exceptionParameter.Message );

        // rethrow exception for further processing
         throw;

         // unreachable code; logic error
      } // end catch
      finally
      {
         Console.WriteLine( "finally executed in " +
            "ThrowExceptionCatchRethrow" );
      } // end finally

     // any code placed here is never reached
      Console.WriteLine( "End of ThrowExceptionCatchRethrow" );
   } // end method ThrowExceptionCatchRethrow
} // end class UsingExceptions
```

The `catch` block rethrows the exception, which is then caught after control returns to `Main`.

The `catch` and `finally` blocks execute when the exception occurs.

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 7 of 8.)

```
Calling DoesNotThrowException
In DoesNotThrowException
finally executed in DoesNotThrowException
End of DoesNotThrowException

Calling ThrowExceptionWithCatch
In ThrowExceptionWithCatch
Message: Exception in ThrowExceptionWithCatch
finally executed in ThrowExceptionWithCatch
End of ThrowExceptionWithCatch
```

```
Calling ThrowExceptionWithoutCatch
In ThrowExceptionWithoutCatch
finally executed in ThrowExceptionWithoutCatch
Caught exception from ThrowExceptionWithoutCatch in Main

Calling ThrowExceptionCatchRethrow
In ThrowExceptionCatchRethrow
Message: Exception in ThrowExceptionCatchRethrow
finally executed in ThrowExceptionCatchRethrow
Caught exception from ThrowExceptionCatchRethrow in Main
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 8 of 8.)