TR1

Test Review #1

1.10 Visual C#

- Visual C# is an event-driven, visual programming language.
- You'll write programs that respond to **events** such as mouse clicks and keystrokes.
- You'll also use Visual Studio's graphical user interface to drag and drop predefined objects like buttons and textboxes into place.
- The **.NET platform** allows applications to be distributed to a variety of devices.

1.16 Introduction to Microsoft .NET

- Microsoft's **.NET initiative** uses the Internet and the web in the development, engineering, distribution and use of software.
- Applications in any .NET-compatible language can interact with each other.
- Microsoft's **ASP.NET** technology allows you to create web applications.
- The .NET strategy allows programmers to concentrate on their specialties without having to implement every component of every application.

1.17 The .NET Framework and the Common Language Runtime

- The Microsoft **.NET Framework**:
  - manages and executes applications and web services
  - contains a class library (called the .NET Framework Class Library)
  - provides security and other programming capabilities.
- The **Common Language Runtime (CLR)**:
  - Programs are compiled first into **Microsoft Intermediate Language** (**MSIL**).
  - When the application executes, the **just-in-time compiler** translates the MSIL in the executable file into machine-language code.

2

Dive Into® Visual C# 2008 Express

2.2 Overview of the Visual Studio 2008 IDE (Cont.)

- A **project** is a group of related files, such as the code files and any images that make up a program.
- **Solutions** contain one ore more projects.
- To begin programming in Visual C#, select **File** > **New Project**

2.2 Overview of the Visual Studio 2008 IDE (Cont.)

- The **New Project dialog** (Fig. 2.3) displays.
- **Templates** are project types users can create in Visual C#.
  - A **Windows Forms application** executes within a Windows operating system and has a **graphical user interface** (**GUI**).

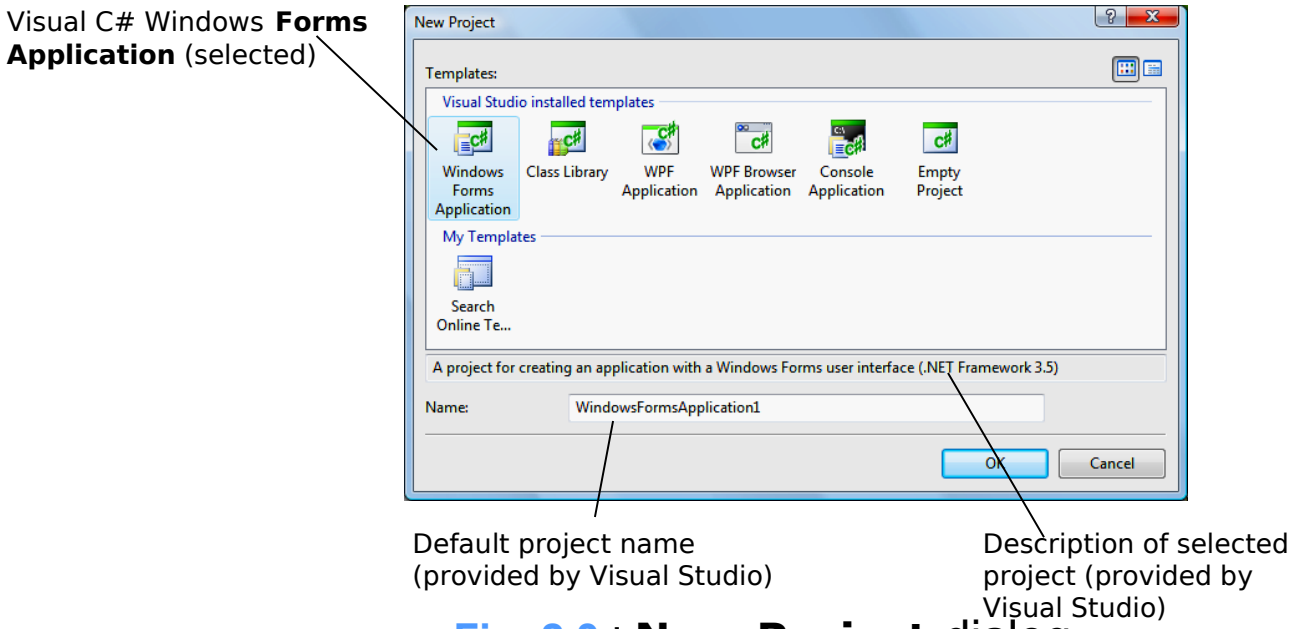


Fig. 2.3 | New Project dialog.

2.2 Overview of the Visual Studio 2008 IDE (Cont.)

- Figure 2.5 shows where the `Form`'s name can be modified in the **Properties** window.

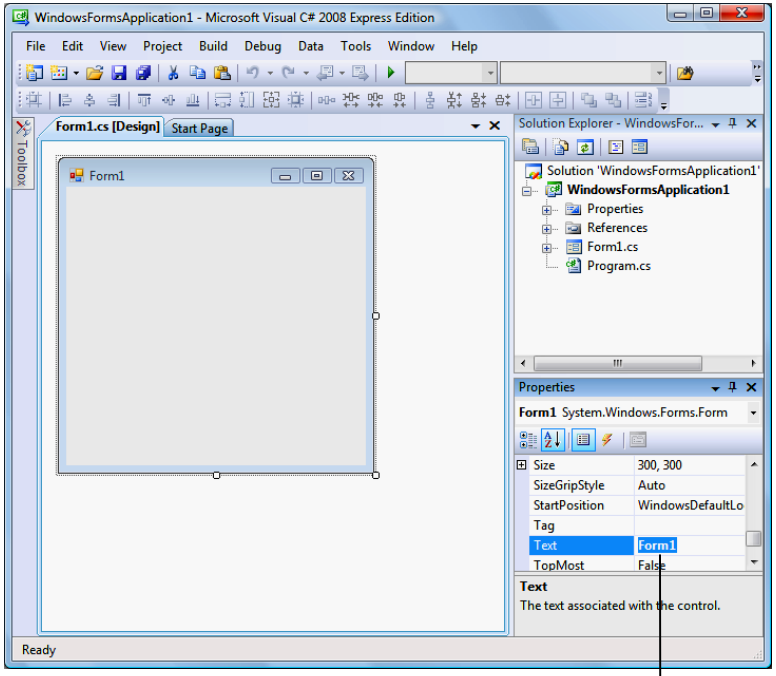


Fig. 2.5 | Textbox control for modifying a property in the Visual Studio IDE.

2.4 Navigating the Visual Studio IDE (Cont.)

2.4.2 Toolbox

- The **Toolbox** contains icons representing controls used to customize `Forms` (Fig. 2.21).
- The **Toolbox** groups the prebuilt controls into categories.

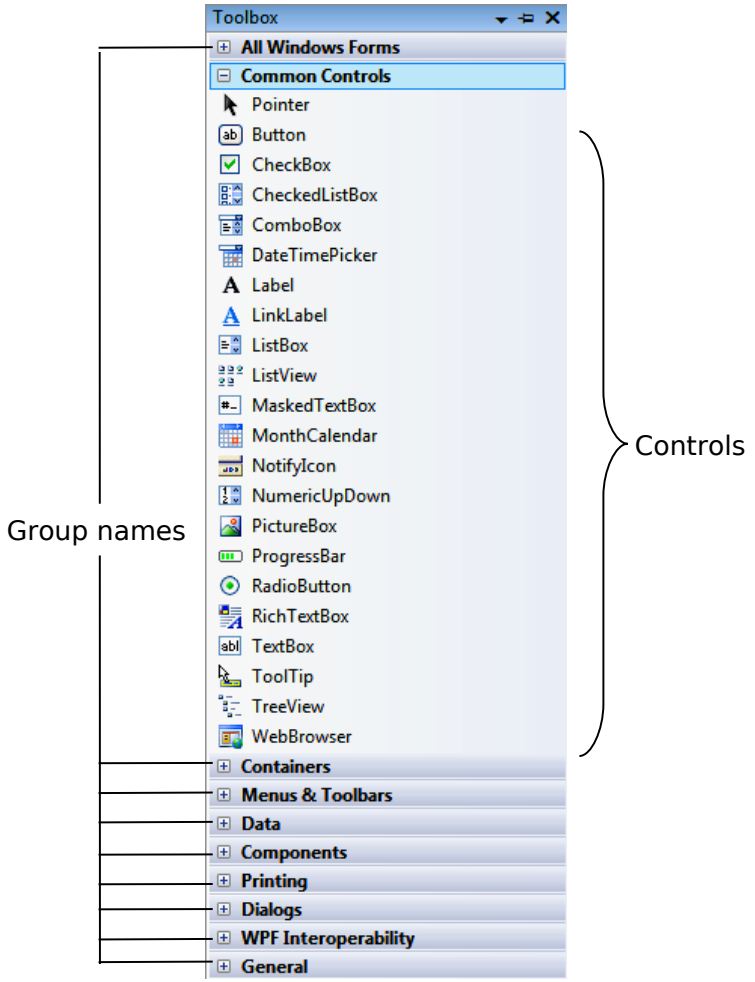


Fig. 2.21 | **Toolbox** window displaying controls for the **Common Controls** group.

3

Introduction to C# Applications

3.3 Creating a Simple Application in Visual C# Express (Cont.)

- ***IntelliSense*** lists a class's **members**, which include method names.
- As you type characters, Visual C# Express highlights the first member that matches all the characters typed, then displays a tool tip containing a description of that member.
- You can either type the complete member name, double click the member name in the member list or press the *Tab* key to complete the name.
- While the *IntelliSense* window is displayed pressing the *Ctrl* key makes the window transparent so you can see the code behind the window.

3.6 Another C# Application: Adding Integers (Cont.)

- Variables of type **`int`** store **integer** values (whole numbers such as `7`, `–11`, `0` and `31914`).
- Types **`float`**, **`double`** and **`decimal`** specify real numbers (numbers with decimal points).
- Type **`char`** represents a single character.
- These types are called **simple types**. Simple-type names are keywords and must appear in all lowercase letters.

3.6 Another C# Application: Adding Integers (Cont.)

- The `Console`'s **`ReadLine`** method waits for the user to type a string of characters at the keyboard and press the *Enter* key.
- `ReadLine` returns the text the user entered.
- The **`Convert`** class's **`ToInt32`** method converts this sequence of characters into data of type `int`.
- `ToInt32` returns the `int` representation of the user's input.

4

Introduction to Classes and Objects

4.3 Declaring a Class with a Method and Instantiating an Object of a Class (Cont.)

- Any class that contains a `Main` method can be used to execute an application.
- A `static` method can be called without creating an object of the class.

4.5 Instance Variables and Properties

- Variables declared in the body of a method are known as **local variables**.
- When a method terminates, the values of its local variables are lost.
- Attributes are represented as variables in a class declaration.
- When each object of a class maintains its own copy of an attribute, the field is known as an instance variable.

4.5 Instance Variables and Properties (Cont.)

- Variables, properties or methods declared with access modifier `private` are accessible only within the class in which they are declared.

- Declaring instance variables with access modifier `private` is known as **information hiding**.

4.5 Instance Variables and Properties (Cont.)

- The `set` accessor begins with the identifier **`set`** and is delimited by braces.

```
gradeBook.CourseName = "CS100 Introduction to Computers";
```

- The text `"CS100 Introduction to Computers"` is assigned to the `set` accessor's keyword named **`value`** and the `set` accessor executes.
- A `set` accessor does not return any data.

4.5 Instance Variables and Properties (Cont.)

- Unlike local variables, every instance variable has a **default initial value**.
- The default value for an instance variable of type `string` is `null`.
- When you display a `string` variable that contains the value `null`, no text is displayed.

Outline

- Figure 4.10 redefines class `GradeBook` with an auto-implemented `CourseName` property.

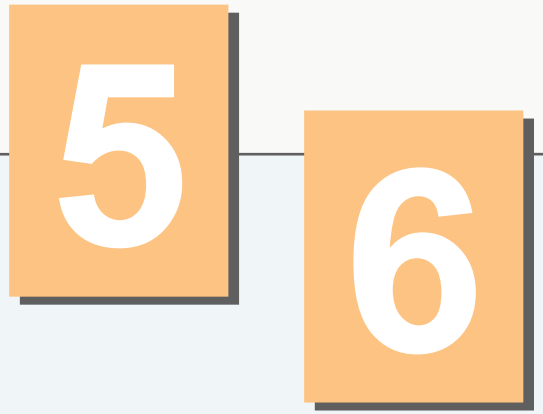
GradeBook.cs

```
1  // Fig4.10: GradeBook.cs
2  // GradeBook class with an auto-implemented property.
3  using System;
4
5  public class GradeBook
6  {
7     // auto-implemented property CourseName implicitly creates
8     // an instance variable for this GradeBook's course name
9     public string CourseName { get; set; }
10
11    // display a welcome message to the GradeBook user
12    public void DisplayMessage()
13    {
14       // use auto-implemented property CourseName to get the
15       // name of the course that this GradeBook represents
16       Console.WriteLine( "Welcome to the grade book for\n{0}!"
17          CourseName ); // display auto-implemented property CourseName
18    } // end method DisplayMessage
19 } // end class GradeBook
```

Declaring the auto-implemented property.

Implicitly obtaining the property's value.

Fig. 4.10 | GradeBook class with an auto-implemented property.



Control Statements: Part 1 & 2

6.4 Examples Using the `for` Statement (Cont.)

- Format item `{0,20}` indicates that the value output should be displayed with a **field width** of 20.
  - To indicate that output should be **left justified**, use a negative field width.

5.13 Simple Types

- The table in Appendix B, Simple Types, lists the 13 **simple types** in C#.
- C# requires all variables to have a type.
- Instance variables of types `char`, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` are all given the value `0` by default.
- Instance variables of type `bool` are given the value `false` by default.

7

Methods: A Deeper Look

7.3 `static` Methods, `static` Variables and Class `Math`

- A method that applies to the class in which it is declared as a whole is known as a `static` method.
  - Static methods are different because they do not depend on any particular object instance
- To declare a method as `static`, place the keyword `static` before the return type in the method's declaration.
- You call any `static` method by specifying the name of the class in which the method is declared, followed by the member access (`.`) operator and the method name.

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

- Class `Math` (from System namespace) provides a collection of `static` methods that enable you to perform common mathematical calculations.
- You do not need to create a `Math` object before calling method `Sqrt`.
- Method arguments may be constants, variables or expressions.

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

- Figure 7.2 summarizes several `Math` class methods. In the figure, *x* and *y* are of type `double`.

Method	Description	Example
`Abs(` *x* `)`	absolute value of *x*	`Abs( ٢٣.٧ )` is ٢٣.٧ `Abs( ٠.٠ )` is ٠.٠ `Abs( -٢٣.٧ )` is ٢٣.٧
`Ceiling(` *x* `)`	rounds *x* to the smallest integer not less than *x*	`Ceiling( ٩.٢ )` is ١٠.٠ `Ceiling( -٩.٨ )` is -٩.٠
`Cos(` *x* `)`	trigonometric cosine of *x* (*x* in radians)	`Cos( ٠.٠ )` is ١.٠

Fig. 7.2 | `Math` class methods. (Part 1 of 3.)

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

Method	Description	Example
`Exp(`*x*`)`	exponential method e^x	`Exp( 1.0 )` is `2.71828` `Exp( 2.0 )` is `7.38906`
`Floor(`*x*`)`	rounds *x* to the largest integer not greater than *x*	`Floor( 9.2 )` is `9.0` `Floor( -9.8 )` is `-10.0`
`Log(`*x*`)`	natural logarithm of *x* (base *e*)	`Log( Math.E )` is `1.0` `Log( Math.E * Math.E )` is `2.0`
`Max(`*x*`,`*y*`)`	larger value of *x* and *y*	`Max( 2.3, 12.7 )` is `12.7` `Max( -2.3, -12.7 )` is `-2.3`

Fig. 7.2 | `Math` class methods. (Part 3 of 3.)

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

- Class `Math` also declares two `static` constants that represent commonly used mathematical values: `Math.PI` and `Math.E`.
  - Pi ~3.14 (ratio of circumference to diameter)
  - E ~2.72 (base value for natural log)
- These constants are declared in class `Math` as `public` and `const`.
  - `public` allows other programmers to use these variables in their own classes.
  - Keyword `const` prevents its value from being changed after the constant is declared.

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

Why Is Method `Main` Declared `static`?

- The `Main` method is sometimes called the application's **entry point**.
- Declaring `Main` as `static` allows the execution environment to invoke `Main` without creating an instance of the class.
- When you execute your application from the command line, you type the application name, followed by **command-line arguments** that specify a list of `string`s separated by spaces.
- The execution environment will pass these arguments to the `Main` method of your application.

7.4 Declaring Methods with Multiple Parameters (Cont.)

Assembling Strings with String Concatenation

- `string` concatenation allows you to combine `string`s using operator `+` .
- When one of the `+` operator's operands is a `string`, the other is implicitly converted to a `string`, then the two are concatenated.
- If a `bool` is concatenated with a `string`, the `bool` is converted to the `string` `"True"` or `"False"`.

7.4 Declaring Methods with Multiple Parameters (Cont.)

- All objects have a `ToString` method that returns a `string` representation of the object.
- When an object is concatenated with a `string`, the object's `ToString` method is implicitly called to obtain the `string` representation of the object.
- A large `string` literal in a program can be broken into several smaller `strings` and placed them on multiple lines for readability, and reassembled using string concatenation or string

7.6 Method-Call Stack and Activation Records (Cont.)

- The program-execution stack also stores local variables. This data is known as the **activation record** or **stack frame** of the method call.
  - When a method call is made, its activation record is pushed onto the program-execution stack.
  - When the method call is popped off the stack, the local variables are no longer known to the application.
- If so many method calls occur that the stack runs out of memory, an error known as a **stack overflow** occurs.

7.9 Case Study: Random-Number Generation

- Objects of class **`Random`** can produce random `byte`, `int` and `double` values.
- Method `Next` of class `Random` generates a random `int` value.
- The values returned by `Next` are actually **pseudorandom numbers**—a sequence of values produced by a complex mathematical calculation.
- The calculation uses the current time of day to **seed** the random-number generator.

7.9 Case Study: Random-Number Generation (Cont.)

- If you supply the `Next` method with an argument—called the **scaling factor**—it returns a value from 0 up to, but not including, the argument's value.
- You can also **shift** the range of numbers produced by adding a **shifting value** to the number returned by the `Next` method.
- Finally, if you provide `Next` with two `int` arguments, it returns a value from the first argument's value up to, but not including, the second argument's value.

Outline

Rolling a Six-Sided Die

- Figure 7.7 shows two sample outputs of an application that simulates 20 rolls of a six-sided die and displays each roll's value.

RandomIntegers.cs

(1 of 2)

```
1  // Fig7.7: RandomIntegers.cs
2  // Shifted and scaled random integers.
3  using System;
4
5  public class RandomIntegers
6  {
7    public static void Main( string[] args )
8    {
9      Random randomNumbers = new Random(); // random-number generator
10     int face; // stores each random integer generated
11
12       // loop 20 times
13     for ( int counter = 1; counter <= 20; counter++ )
14     {
15       // pick random integer from 1 to 6
```

Create the `Random` object `randomNumbers` to produce random values.

Fig. 7.7 | Shifted and scaled random integers. (Part 1 of 2.)

Outline

```
16      face = randomNumbers.Next( 1, 7 );
17
18      Console.Write( "{0} ", face ); // display generated value
19
20         // if counter is divisible by 5, start a new line of output
21      if ( counter % 5 == 0 )
22        Console.WriteLine();
23    } // end for
24  } // end Main
25 } // end class RandomIntegers
```

```
3  3  3  1  1
2  1  2  4  2
2  3  6  2  5
3  4  6  6  1
```

```
6  2  5  1  3
5  2  1  6  5
4  1  6  1  3
3  1  4  3  4
```

RandomIntegers.cs

(2 of 2)

Call `Next` with two arguments.

Fig. 7.7 | Shifted and scaled random integers. (Part 2 of 2.)

7.9 Case Study: Random-Number Generation (Cont.)

7.9.1 Scaling and Shifting Random Numbers

- Given two arguments, the next method allows scaling and shifting as follows:

`number` = `randomNumbers.Next(` *shiftingValue*, *shiftingValue* + *scalingFactor*);

  - *shiftingValue* specifies the first number in the desired range of consecutive integers.
  - *scalingFactor* specifies how many numbers are in the range.

7.9 Case Study: Random-Number Generation (Cont.)

- To choose integers at random from sets of values other than ranges of consecutive integers, it is simpler to use the version of the `Next` method that takes only one argument:

`number` = *shiftingValue* +
differenceBetweenValues * `randomNumbers.Next(` *scalingFactor*);

  - *shiftingValue* specifies the first number in the desired range of values.
  - *differenceBetweenValues* represents the difference between consecutive numbers in the sequence.
  - *scalingFactor* specifies how many numbers are in the range.

7.9 Case Study: Random-Number Generation (Cont.)

7.9.2 Random-Number Repeatability for Testing and Debugging

- The calculation that produces the pseudorandom numbers uses the time of day as a **seed value** to change the sequence's starting point.
- You can pass a seed value to the `Random` object's constructor.
- Given the same seed value, the `Random` object will produce the same sequence of random numbers.

7.11 Scope of Declarations

- The scope of a declaration is the portion of the application that can refer to the declared entity by its unqualified name.
- The basic scope rules are as follows:
  - The scope of a parameter declaration is the body of the method in which the declaration appears.
  - The scope of a local-variable declaration is from the point at which the declaration appears to the end of the block containing the declaration.
  - The scope of a non-`static` method, property or field of a class is the entire body of the class.
- If a local variable or parameter in a method has the same name as a field, the field is hidden until the block terminates.

7.12 Method Overloading

- Methods of the same name can be declared in the same class, or **overloaded**, as long as they have different sets of parameters.
- When an **overloaded method** is called, the C# compiler selects the appropriate method by examining the number, types and order of the arguments in the call.
- Method overloading is used to create several methods with the same name that perform the same tasks, but on different types or numbers of arguments.

7.13 Recursion

- A **recursive method** is a method that calls itself.
- A recursive method is capable of solving only the **base case(s)**.
- Each method call divides the problem into two conceptual pieces: a piece that the method knows how to do and a **recursive call**, or **recursion step** that solves a smaller problem.
- A sequence of returns ensues until the original method call returns the result to the caller.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference

- Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**.
- When an argument is passed by value (the default in C#), a *copy* of its value is made and passed to the called function.
- When an argument is passed by reference, the caller gives the method the ability to access and modify the caller's original variable.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

- Applying the **`ref`** keyword to a parameter declaration allows you to pass a variable to a method by reference
- The `ref` keyword is used for variables that already have been initialized in the calling method.
- Preceding a parameter with keyword **`out`** creates an **output parameter**.
- This indicates to the compiler that the argument will be passed by reference and that the called method will assign a value to it.
- A method can return multiple output parameters.

8

Arrays

8.3 Declaring and Creating Arrays (Cont.)

Difference from C++

- The number of elements can also be specified as an expression that is calculated at execution time.
- When an array is created, each element of the array receives a default value:
  - `0` for the numeric simple-type elements.
  - `false` for `bool` elements.
  - `null` for references.
- An application can create several arrays in a single declaration.
- For readability, it is better to write each array declaration in its own statement.

8.4 Examples Using Arrays (Cont.)

- In many programming languages, like C and C++, writing outside the bounds of an array is allowed, but often causes disastrous results.
- In C#, accessing any array element forces a check on the array index to ensure that it is valid. This is called **bounds checking**.
- If an application uses an invalid index, the Common Language Runtime generates an `IndexOutOfRangeException` to indicate that an error occurred in the application at execution time.

8.6 `foreach` Statement

- The **`foreach` statement** iterates through the elements of an entire array or collection.
- The syntax of a `foreach` statement is:

  `foreach` (*type identifier* **`in`** *arrayName*)
  statement

  - *type* and *identifier* are the type and name (e.g., `int number`) of the **iteration variable**.
  - *arrayName* is the array through which to iterate.
- The type of the iteration variable must match the type of the elements in the array.
- The iteration variable represents successive values in the array on successive iterations of the `foreach` statement.

8.6 `foreach` Statement (Cont.)

Implicitly Typed Local Variables

- C# provides a new feature—called **implicitly typed local variables**—that enables the compiler to infer a local variable's type based on the type of the variable's initializer.
- To distinguish such an initialization from a simple assignment statement, the `var` keyword is used in place of the variable's type.
- The compiler assumes that floating-point number values are of type `double`.
- You can use local type inference with control variables in the header of a `for` or `foreach` statement.
- For example, the following `for` statement headers are equivalent:

```
for ( int counter = 1; counter < 10; counter++ )
for ( var counter = 1; counter < 10; counter++ )
```

8.10 Multidimensional Arrays

- **Multidimensional arrays** with two dimensions are often used to represent **tables of values** consisting of information arranged in **rows** and **columns**.
- To identify a particular table element, we must specify two indices. By convention, the first identifies the element's row and the second its column.
- Arrays that require two indices to identify a particular element are called **two-dimensional arrays**.

8.10 Multidimensional Arrays (Cont.)

Jagged Arrays

- A **jagged array** is a one-dimensional array whose elements are one-dimensional arrays.
- The lengths of the rows in the array need not be the same.
- Elements in a jagged array are accessed using an array-access expression of the form *arrayName*[*row*][*column*].
- A jagged array with three rows of different lengths could be declared and initialized as follows:

```
int[][] jagged = { new int[] { 1, 2 },
                   new int[] { 3 },
                   new int[] { 4, 5, 6 } };
```

9

Introduction to LINQ and Generic Collections

LINQWithSimpleTypeArray.cs

(3 of 5)

```
34        // sort the filtered results into descending order
35     var sortFilteredResults =
36        from value in filtered
37        orderby value descending
38        select value;
39
40        // display the sorted results
41     Display( sortFilteredResults,
42        "Values greater than 4, descending order (separately):" );
43
44        // filter original array and sort in descending order
45     var sortAndFilter =
46        from value in values
47        where value > 4
48        orderby value descending
49        select value;
50
```

The `descending` modifier in the `orderby` clause sorts the results in descending order.

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 3 of 5.)

Outline

LINQWithSimpleTypeArray.cs

(4 of 5)

```
51        // display the filtered and sorted results
52        Display( sortAndFilter,
53           "Values greater than 4, descending order (one query):" );
54     } // end Main
55
56     // display a sequence of integers with the specified header
57     public static void Display(
58        IEnumerable< int > results, string header )
59     {
60        Console.Write( "{0}", header ); // display header
61
62        // display each element, separated by spaces
63        foreach ( var element in results )
64           Console.Write( " {0}", element );
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 4 of 5.)

Outline

```
11        // initialize array of employees
12        Employee[] employees = {
13          new Employee( "Jason", "Red", 5000M ),
14          new Employee( "Ashley", "Green", 7600M ),
15          new Employee( "Matthew", "Indigo", 3587.5M ),
16          new Employee( "James", "Indigo", 4700.77M ),
17          new Employee( "Luke", "Indigo", 6200M ),
18          new Employee( "Jason", "Blue", 3200M ),
19          new Employee( "Wendy", "Brown", 4236.4M ) }; // end init list
20
21        Display( employees, "Original array" ); // display all employees
22
23        // filter a range of salaries using && in a LINQ query
24        var between4K6K =
25          from e in employees
26          where e.MonthlySalary >= 4000M && e.MonthlySalary <= 6000M
27          select e;
28
29        // display employees making between 4000 and 6000 per month
30        Display( between4K6K, string.Format(
31          "Employees earning in the range {0:C}-{1:C} per month",
32          4000, 6000 ) );
```

LINQWithArrayOfObjects.cs

(2 of 5)

A `where` clause can access the properties of the range variable.

Fig. 9.4 | LINQ to Objects using an array of Employee objects. (Part 2 of 5.)

9.2 Querying an Array Using LINQ (Cont.)

- **Generic methods** enable you to create a single method definition that can be called with arguments of many types.
- To define a generic method, you must specify a **type parameter list** which contains one or more type parameters separated by commas.
- A **type parameter** is a placeholder for a type argument. They can be used to declare return types, parameter types and local variable types in generic method declarations.

9.3 Introduction to Collections

- The .NET Framework Class Library provides **collections**, which are used to store groups of related objects.
- Collections provide efficient methods that organize, store and retrieve your data without requiring knowledge of how the data is being stored.
- The collection class `List<T>` (from namespace `System.Collections.Generic`) does not need to be reallocated to change its size.

9.3 Introduction to Collections (Cont.)

- `List<T>` is called a **generic class** because it can be used with any type of object.
- `T` is a placeholder for the type of the objects stored in the list.
- Figure 9.5 shows some common methods and properties of class `List<T>`.

Method or property	Description
`Add`	Adds an element to the end of the `List`.
`Capacity`	Property that gets or sets the number of elements a `List` can store.
`Clear`	Removes all the elements from the `List`.
`Contains`	Returns `true` if the `List` contains the specified element; otherwise, returns `false`.
`Count`	Property that returns the number of elements stored in the `List`.

Fig. 9.5 | Some methods and properties of class `List<T>`. (Part 1 of 2.)

9.3 Introduction to Collections (Cont.)

Method or property	Description
`IndexOf`	Returns the index of the first occurrence of the specified value in the `List`
`Insert`	Inserts an element at the specified index.
`Remove`	Removes the first occurrence of the specified value.
`RemoveAt`	Removes the element at the specified index.
`RemoveRange`	Removes a specified number of elements starting at a specified index.
`Sort`	Sorts the `List`
`TrimExcess`	Sets the Capacity of the List to the number ofelements the List currently contains (`Count`).

Fig. 9.5 | Some methods and properties of class `List<T>`. (Part 2 of 2.)

10

Classes and Objects: A Deeper Look

- Every object can access a reference to itself with keyword **this**.
- When a non-`static` method is called, the method's body implicitly uses keyword `this` to refer to the object's instance variables and other methods.
- You can also use keyword `this` *explicitly* in a non-`static` method's body.

ThisTest.cs

(1 of 3)

```
1  // Fig10.4: ThisTest.cs
2  // this used implicitly and explicitly to refer to members of an object.
3  using System;
4
5  public class ThisTest
6  {
7     public static void Main( string[] args )
8     {
9        SimpleTime time = new SimpleTime( 15, 30, 19 );
10       Console.WriteLine( time.BuildString() );
11    } // end Main
12 } // end class ThisTest
13
```

Fig. 10.4 | `this` used implicitly and explicitly to refer to members of an object. (Part 1 of 3.)

10.5 Indexers

- A class that encapsulates lists of data can use keyword `this` to define property-like class members called **indexers** that allow array-style indexed access to lists of elements.
- You can define both integer indices and noninteger indices.
- Indexers can return any type, even one that is different from the type of the underlying data.
- Unlike properties, for which you can choose an appropriate property name, indexers must be defined with keyword `this`.

10.5 Indexers (Cont.)

- Indexers have the general form:

accessModifier returnType `this`[*IndexType1 name1*, *IndexType2 name2*,…]

```
{
    get
    {
        // use name1, name2, ... here to get data
    }
    set
    {
        // use name1, name2, ... here to set data
    }
}
```

- The *IndexType* parameters are accessible to the `get` and `set` accessors.

- Class Box (Fig. 10.5) represents a box with a length, a width and a height.

Box.cs

(1 of 3)

```
1  // Fig10.5: Box.cs
2  // Box class definition represents a box with length,
3  // width and height dimensions with indexers.
4  public class Box
5  {
6    private string[] names = { "length", "width", "height" };
7    private double[] dimensions = new double[ 3 ];
8
9     // constructor
10   public Box( double length, double width, double height )
11   {
12     dimensions[ 0 ] = length;
13     dimensions[ 1 ] = width;
14     dimensions[ 2 ] = height;
15   }
16
17    // indexer to access dimensions by integer index number
```

Fig. 10.5 | Box class definition represents a box with length, width and height dimensions with indexers. (Part 1 of 3.)

```
18    public double this[ int index ]
19    {
20       get
21       {
22          // validate index to get
23          if ( ( index < 0 ) || ( index >= dimensions.Length ) )
24             return -1;
25          else
26             return dimensions[ index ];
27       } // end get
28       set
29       {
30          if ( index >= 0 && index < dimensions.Length )
31             dimensions[ index ] = value;
32       } // end set
33    } // end numeric indexer
34
35    // indexer to access dimensions by their string names
36    public double this[ string name ]
37    {
38       get
39       {
```

Box.cs

(2 of 3)

Manipulate the array by index.

Manipulate the array by dimension name.

Fig. 10.5 | Box class definition represents a box with length, width and height dimensions with indexers. (Part 2 of 3.)

```
40       // locate element to get
41       int i = 0;
42       while ( ( i < names.Length ) &&
43         ( name.ToLower() != names[ i ] ) )
44         i++;
45
46       return ( i == names.Length ) ? -1 : dimensions[ i ];
47     } // end get
48     set
49     {
50       // locate element to set
51       int i = 0;
52       while ( ( i < names.Length ) &&
53         ( name.ToLower() != names[ i ] ) )
54         i++;
55
56       if ( i != names.Length )
57         dimensions[ i ] = value;
58     } // end set
59   } // end string indexer
60 } //  end class Box
```

Box.cs

(3 of 3)

Manipulate the array by dimension name.

Fig. 10.5 | Box class definition represents a box with length, width and height dimensions with indexers. (Part 3 of 3.)

Time2.cs

(2 of 5)

```
16 // Time2 constructor: hour and minute supplied, second defaulted to 0
17 public Time2( int h, int m ) : this( h, m, 0 ) { }
18
19 // Time2 constructor: hour, minute and second supplied
20 public Time2( int h, int m, int s )
21 {
22    SetTime( h, m, s ); // invoke SetTime to validate time
23 } // end Time2 three-argument constructor
24
25 // Time2 constructor: another Time2 object supplied
26 public Time2( Time2 time )
27    : this( time.Hour, time.Minute, time.Second ) { }
28
29 // set a new time value using universal time; ensure that
30 // the data remains consistent by setting invalid values to zero
```

Declare a `Time2` constructor with a single `int` parameter representing the `hour`. Pass the given hour and 0's to the three-parameter constructor.

Declare the `Time2` constructor that receives three `int` parameters representing the `hour`, `minute` and `second`. This constructor is used by all of the others.

Fig. 10.7 | `Time2` class declaration with overloaded constructors. (Part 2 of 5.)

10.9 Garbage Collection and Destructors

- Every object you create uses various system resources, such as memory.
- In many programming languages, these system resources are reserved for the object's use until they are explicitly released by the programmer.
- If all the references to the object that manages the resource are lost before the resource is explicitly released, it can no longer be released. This is known as a **resource leak**.
- The Common Language Runtime (CLR) uses a **garbage collector** to reclaim the memory occupied by objects that are no longer in use.
- When there are no more references to an object, the object becomes **eligible for destruction**.

10.9 Garbage Collection and Destructors (Cont.)

- Every object has a **destructor** that is invoked by the garbage collector to perform **termination housekeeping** before its memory is reclaimed.
- A destructor's name is the class name, preceded by a tilde, and it has no access modifier in its header.
- After an object's destructor is called, the object becomes **eligible for garbage collection**—the memory for the object can be reclaimed by the garbage collector.
- **Memory leaks** are less likely in C# than languages like C and C++ (but some can still happen in subtle ways).

10.11 `readonly` Instance Variables

- The **principle of least privilege** states that code should be granted only the amount of privilege and access needed to accomplish its designated task, but no more.
- Constants declared with `const` must be initialized to a constant value when they are declared.
- C# provides keyword **`readonly`** to specify that an instance variable of an object is not modifiable and that any attempt to mod-ify it after the object is constructed is an error.
- Like constants, `readonly` variables are declared with all capital letters by convention
- `readonly` instance variables can be initialized when they are declared, but this is not required.

- In Visual C# 2008, you can use **extension methods** to add functionality to an existing class without modifying the class's source code.
- Figure 10.25 uses extension methods to add functionality to class `Time` (from Section 10.17).

TimeExtensionsTest.cs

(1 of 3)

```
1  // Fig10.25: TimeExtensionsTest.cs
2  // Demonstrating extension methods.
3  using System;
4
5  class TimeExtensionsTest
6  {
7     static void Main( string[] args )
8     {
9        Time myTime = new Time(); // call Time constructor
10       myTime.SetTime( 11, 34, 15 ); // set the time to 11:34:15
11
```

Fig. 10.25 | Demonstrating extension methods. (Part 1 of 3.)

- A **delegate** is an object that holds a reference to a method.
- Delegates allow you to treat methods as data—via delegates, you can assign methods to variables, and pass methods to and from other methods.
- You can also call methods through variables of delegate types.
- A delegate type is declared by preceeding a method header with keyword `delegate` (placed after any access specifiers, such as `public` or `private`).
- Figure 10.27 uses delegates to customize the functionality of a method that filters an `int` array.

10.20 Lambda Expressions

- A lambda expression begins with a parameter list, which is followed by the => **lambda operator** and an expression that represents the body of the function.
- The value produced by the expression is implicitly returned by the lambda expression.
- The return type can be inferred from the return value or, in some cases, from the delegate's return type.
- A delegate can hold a reference to a lambda expression whose signature is compatible with the delegate type.
- Lambda expressions are often used as arguments to methods with parameters of delegate types, rather than defining and referencing a separate method.

10.20 Lambda Expressions (Cont.)

- A lambda expression can be called via the variable that references it.
- A lambda expression's input parameter `number` can be explicitly typed.
- Lambda expressions that have an expression to the right of the lambda operator are called **expression lambdas**.
- **Statement lambdas** contain a statement block—a set of statements enclosed in braces (`{}`)—to the right of the lambda operator.
- Lambda expressions can help reduce the size of your code and the complexity of working with delegates.
- Lambda expressions are particularly powerful when combined with the `where` clause in LINQ queries.

11

Object-Oriented Programming: Inheritance

11.1 Introduction

- **Inheritance** allows a new class to absorb an existing class's members.
- A derived class normally adds its own fields and methods to represent a more specialized group of objects.
- Inheritance saves time by reusing proven and debugged high-quality software.

11.4 Relationship between Base Classes and Derived Classes (Cont.)

- The `virtual` and `abstract` keywords indicate that a base-class method can be overridden in derived classes.
- The `override` modifier declares that a derived-class method overrides a `virtual` or `abstract` base-class method.
- This modifier also implicitly declares the derived-class method `virtual`.
- We need to declare `CommissionEmployee`'s `Earnings` method `virtual`.

11.7 Class `object`

- All classes inherit directly or indirectly from the `object` class.
- Figure 11.19 summarizes `object`'s methods.

Method	Description
`Equals`	This method compares two objects for equality and returns `true` if they are equal and `false` otherwise.
`Finalize`	`Finalize` is called by the garbage collector before it reclaims an object's memory.
`GetHashCode`	The hashcode value returned can be used by a hashtable to determine the location at which to insert the corresponding value.

Fig. 11.19 | `object` methods that are inherited directly or indirectly by all classes. (Part 1 of 2.)

11.7 Class `object` (Cont.)

Method	Description
`GetType`	Returns an object of class `Type` that contains information about the object's type.
`MemberwiseClone`	This `protected` method makes a copy of the object on which it is called. Instance-variable values in one object are copied into another object of the same type. For reference types, only the references are copied.
`ReferenceEquals`	This `static` method returns `true` if two objects are the same instance or if they are `null` references.
`ToString`	Returns a `string` representation of an object. The default implementation returns the namespace and class name.

Fig. 11.19 | `object` methods that are inherited directly or indirectly by all classes. (Part 2 of 2.)

12

Polymorphism, Interfaces & Operator Overloading

12.4 Abstract Classes and Methods

- **Abstract classes**, or **abstract base classes** cannot be used to instantiate objects.
- Abstract base classes are too general to create real objects—they specify only what is common among derived classes.
- Classes that can be used to instantiate objects are called **concrete classes**.
- Concrete classes provide the specifics that make it reasonable to instantiate objects.

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.)

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.

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.

13

Exception Handling

Outline

```
15       // obtain 2 integers from the user
16       // and divide numerator by denominator
17       private void divideButton_Click( object sender, EventArgs e )
18       {
19          outputLabel.Text = ""; // clear Label OutputLabel
20
21          // retrieve user input and calculate quotient
22          try
23          {
24             // Convert.ToInt32 generates FormatException
25             // if argument cannot be converted to an integer
26             int numerator = Convert.ToInt32( numeratorTextBox.Text );
27             int denominator = Convert.ToInt32( denominatorTextBox.Text );
28
29             // division generates DivideByZeroException
30             // if denominator is 0
31             int result = numerator / denominator;
32
33             // display result in OutputLabel
34             outputLabel.Text = result.ToString();
35          } // end try
```

DivideByZeroTest.cs

(2 of 4)

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

Outline

DivideByZeroTest.cs

(3 of 4)

```
36          catch ( FormatException )
37          {
38             MessageBox.Show( "You must enter two integers.",
39                "Invalid Number Format", MessageBoxButtons.OK,
40                MessageBoxIcon.Error );
41          } // end catch
42          catch ( DivideByZeroException divideByZeroExceptionParameter )
43          {
44             MessageBox.Show( divideByZeroExceptionParameter.Message,
45                "Attempted to Divide by Zero", MessageBoxButtons.OK,
46                 MessageBoxIcon.Error );
47          } // end catch
48       } // end method divideButton_Click
49    } // end class DivideByZeroTestForm
50 } // 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.)

13.6 `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.