Introduction to Microsoft .NET ...

- .NET initiative
  - Introduced by Microsoft (June 2000)
    - Vision for embracing the Internet in software development
  - Independence from specific language or platform
    - Applications developed in any .NET compatible language
      - Visual Basic .NET, Visual C++ .NET, C# and more
    - Programmers can contribute to applications using the language in which they are most competent
  - Architecture capable of existing on multiple platforms
  - New program development process
    - Provides increased productivity
Introduction to Microsoft .NET …

- Key components of .NET
  - Web services
    - Applications used over the Internet
  - Software reusability
    - Web services provide solutions for wide variety of companies
      - Cheaper than developing one-time solutions that can’t be reused
      - Single applications perform all operations for a company via various Web services
        » Manage taxes, bills, investments, etc
    - Pre-packaged components
      - Make application development quicker and easier
      - Developers no longer need to be concerned with details of components
Introduction to Microsoft .NET …

- Keys to interaction
  - XML and SOAP
    - “Glue” that combines various Web services to form applications
      - XML gives meaning to data
      - SOAP allows communication to occur easily

- Additional information available at Microsoft Web site
  www.microsoft.com/net
.NET Framework and the Common Language Runtime …

• .NET Framework
  – Heart of .NET
    • Manages and executes applications and Web services
    • Provides security, memory management and other programming capabilities
  – Includes Framework class library (FCL)
    • Pre-packaged classes ready for reuse
    • Used by any .NET language
  – Executes programs by Common Language Runtime (CLR) (analogous to Java’s Virtual Machine, JVM)

  – Details contained in Common Language Specification (CLS)
    • Submitted to European Computer Manufacturers Association to make the framework easily converted to other platforms
.NET Framework and the Common Language Runtime …

- Common Language Runtime (CLR)
  - Central part of framework
    - Executes Visual Basic .NET programs
  - Compilation process
    - Two compilations take place
      - Programs compiled to Microsoft Intermediate Language (MSIL)
        » Defines instructions for CLR
      - MSIL code translated into machine code
        » Machine code for a particular platform
.NET Framework and the Common Language Runtime …/

• Why two compilations?
  – Platform independence
    • .NET Framework can be installed on different platforms
    • Execute .NET programs without any modifications to code
  – Language independence
    • .NET programs not tied to particular language
    • Programs may consist of several .NET-compliant languages
    • Old and new components can be integrated
• Other advantages of CLR
  – Execution-management features
    • Manages memory, security and other features
      – Relieves programmer of many responsibilities
      – More concentration on program logic
JVM (Java) vs CLR (.NET) : multiple platforms vs multiple languages.
VISUAL STUDIO.NET Introduction

• Visual Studio .NET (VS.NET -- current version is VS2008)
  – Microsoft’s Integrated Development Environment (IDE)
  – Program in a variety of .NET languages
  – Tools to edit and manipulate several file types
Visual Studio .NET Integrated Development Environment (IDE) Overview

Start Page in Visual Studio .NET.
How to obtain Visual Studio (VS)

• Available to use in dept. Labs.
• Can download “express editions” (see links in course web site) [each express edition is good for specific tasks, e.g., C# programming, VB.NET programming, ASP.NET apps, etc]
• Can get a copy (per group) of full VS2008 from the dept.
C# (C-sharp)

(many of these slides are extracted and adapted from Deitel’s book and slides, “How to Program in C#”. They are provided for CSE3403 students only. Not to be published or publicly distributed without permission by the publisher).
C#

- Developed at Microsoft by a team led by Anders Hejlsberg and Scott Wiltamuth (June 2000)
- Event driven, object oriented, visual programming language.
- Based on C, C++ and Java.
- Incorporated into .NET platform
  - Web based applications can be distributed
    - Devices and desktop computers
  - Programs that can be accessed by anyone through any device
  - Allows communicating with different computer languages
What you need to develop C# programs

• Either Visual Studio .NET, or
• The .NET SDK
  – Command line development
  – Download from (http://www.microsoft.com/downloads)
• Or the “C# express edition”.

Introduction

• **Console applications**
  - No visual components
  - Only text output
  - Two types
    • MS-DOS prompt
      - Used in Windows pre-Win2000 (95/98/ME)
    • Command prompt
      - Used in windows post-Win2000 (2000/NT/XP/Vista)
  - Windows applications
    • Forms with several output types
    • Contain Graphical User Interfaces (GUIs)
Building **non-windows applications** (programs that only output to the command line and contain no GUI components).
Simple Program: Printing a Line of Text

Visual Studio .NET-generated console application.
Or can use any text editor and type your program (good for when new to programming in general … but not really for now)

Upon typing the ‘.’, VS gives options available …
The final program

```csharp
using System;

namespace ConsoleApplication1
{
    // <summary>
    // Summary description for Class1.
    // </summary>
    class Class1
    {
        // <summary>
        // The main entry point for the application.
        // </summary>
        [STAThread]
        static void Main(string[] args)
        {
            Console.WriteLine("Hello C# !!!");
        }
    }
}
```
Compile and Run

Compile (and produce .exe)

Run
The output

Hello C# !!!
Press any key to continue.
// Welcome1.cs
// A first program in C#.

using System;

class Welcome1 //optionally public class Welcome1
{
    static void Main( string[] args )
    // or public static void Main(…)
    {
        Console.WriteLine( "Welcome to C# Programming!" );
    }
}
Simple Program: Printing a Line of Text

Execution of the Welcome1 program.
Simple Program: Printing a line of text

• Comments
  – Comments can be created using //…
  – Multi-lines comments use /* … */
  – Comments are ignored by the compiler
  – Used only for human readers

• Namespaces
  – Groups related C# features into a categories
  – Allows the easy reuse of code
  – Many namespaces are found in the .NET framework library
  – Must be referenced in order to be used

• White Space
  – Includes spaces, newline characters and tabs
Simple Program: Printing a line of text

• Keywords
  – Words that cannot be used as variable or class names or any other capacity
  – Have a specific unchangeable function within the language
  – Example: `class`
  – All keywords are lowercase

• Classes
  – Class names can only be one word long (i.e. no white space in class name)
  – Class names are capitalized, with each additional English word capitalized as well (e.g., `MyFirstProgram`)
  – Each class name is an identifier
    • Can contain letters, digits, and underscores ( `_ `)
    • Cannot start with digits
    • Can start with the at symbol (` @ `)
Simple Program: Printing a line of text

- Class bodies start with a left brace ({)
- Class bodies end with a right brace (})

• Methods
  - Building blocks of programs
  - The `Main` method
    - Each console or windows application must have exactly one
    - All programs start by executing the `Main` method
  - Braces are used to start ({) and end (}) a method

• Statements
  - Anything in quotes (") is considered a string
  - Every statement must end in a semicolon (;)
Welcome2.cs

```csharp
// Welcome2.cs
// Printing a line with multiple statements.

using System;

class Welcome2
{
    static void Main(string[] args)
    {
        Console.Write("Welcome to ");
        Console.WriteLine("C# Programming!");
    }
}
```

Welcome to C# Programming!

Program Output
Welcome3.cs

```csharp
// Welcome3.cs
// Printing multiple lines with a single statement.

using System;

class Welcome3
{
    static void Main( string[] args )
    {
        Console.WriteLine( "Welcome\nto\nC#\nProgramming!" );
    }
}
```

Welcome
to
C#
Programming!

Program Output
Simple Program: Printing a Line of Text

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Newline. Position the screen cursor to the beginning of the next line.</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab. Move the screen cursor to the next tab stop.</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line. Any characters output after the carriage return overwrite the previous characters output on that line.</td>
</tr>
<tr>
<td>\</td>
<td>Backslash. Used to print a backslash character.</td>
</tr>
<tr>
<td>\\</td>
<td>Double quote. Used to print a double quote (&quot; ) character.</td>
</tr>
</tbody>
</table>

Some common escape sequences.
Another Simple Program: Adding Integers

- Primitive data types
  - Data types that are built into C#
    - String, Int, Double, Char, Long
    - 15 primitive data types
    - Each data type name is a C# keyword
    - Same type variables can be declared on separate lines or on one line
- `Console.ReadLine()`
  - Used to get a value from the user input
- `Int32.Parse()`
  - Used to convert a string argument to an integer
  - Allows math to be performed once the string is converted
using System;

class Addition
{
    static void Main(string[] args)
    {
        string firstNumber,  // first string entered
                        secondNumber;  // second string entered

        int number1,          // first number to add
                        number2,         // second number to add
                        sum;             // sum of number1 and
number2

        // prompt for and read first number from user
        Console.Write("Please enter the first integer: ");
        firstNumber = Console.ReadLine();
    }
}
```csharp
// read second number from user as string
Console.Write( "\nPlease enter the second integer: " );
secondNumber = Console.ReadLine();

// convert numbers from type string to type int
number1 = Int32.Parse( firstNumber ); // Parse converts the given string to integer
number2 = Int32.Parse( secondNumber );

// add numbers
sum = number1 + number2;
```
Addition.cs (cont’d)

```csharp
32         // display results
33             Console.WriteLine( "\nThe sum is {0}.", sum );
34
35         } // end method Main
36
37         } // end class Addition
```

Program Output

Please enter the first integer: 45
Please enter the second integer: 72

The sum is 117.
Decision Making: Equality and Relational Operators

- The `if` structure
  - Used to make a decision based on the truth of the condition
    - True: a statement is performed
    - False: the statement is skipped over
  - The start of an `if` statement should not end in a semicolon (;)
## Decision Making: Equality and Relational Operators

<table>
<thead>
<tr>
<th>Standard algebraic equality operator or relational operator</th>
<th>C# equality or relational operator</th>
<th>Example of C# condition</th>
<th>Meaning of C# condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Equality operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>==</td>
<td>x == y</td>
<td>x is equal to y</td>
</tr>
<tr>
<td>≠</td>
<td>!=</td>
<td>x != y</td>
<td>x is not equal to y</td>
</tr>
<tr>
<td><strong>Relational operators</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>x &gt; y</td>
<td>x is greater than y</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>x &lt; y</td>
<td>x is less than y</td>
</tr>
<tr>
<td>≥</td>
<td>&gt;=</td>
<td>x &gt;= y</td>
<td>x is greater than or equal to y</td>
</tr>
<tr>
<td>≤</td>
<td>&lt;=</td>
<td>x &lt;= y</td>
<td>x is less than or equal to y</td>
</tr>
</tbody>
</table>

*Equality and relational operators.*
using System;

class Comparison
{
    static void Main( string[] args )
    {
        int number1, number2;

        // read in first number from user
        Console.Write( "Please enter first integer: " );
        number1 = Int32.Parse( Console.ReadLine() );

        // read in second number from user
        Console.Write( "\nPlease enter second integer: " );
        number2 = Int32.Parse( Console.ReadLine() );

        if ( number1 == number2 )
            Console.WriteLine( number1 + " == " + number2 );

        if ( number1 != number2 )
            Console.WriteLine( number1 + " != " + number2 );

        if ( number1 < number2 )
            Console.WriteLine( number1 + " < " + number2 );

        if ( number1 > number2 )
            Console.WriteLine( number1 + " > " + number2 );
    }
}
Comparison.cs
cont’d

34    if ( number1 <= number2 )
35        Console.WriteLine( number1 + " <= " + number2 );
36
37    if ( number1 >= number2 )
38        Console.WriteLine( number1 + " >= " + number2 );
39
40       } // end method Main
41
42   } // end class Comparison
Please enter first integer: 2000

Please enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000

Please enter first integer: 1000

Please enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000

Please enter first integer: 1000

Please enter second integer: 1000
1000 == 1000
1000 <= 1000
1000 >= 1000
Decision Making: Equality and Relational Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>left to right</td>
<td>parentheses</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
</tbody>
</table>

Precedence and associativity of operators discussed in this chapter.
while Repetition Structure

• Repetition Structure
  – An action is to be repeated
    • Continues while statement is true
    • Ends when statement is false
  – Contain either a line or a body of code
    • Must alter conditional
      – Endless loop
```csharp
// Average1.cs
// Class average with counter-controlled repetition.

using System;

class Average1
{
    static void Main(string[] args)
    {
        int total, // sum of grades
            gradeCounter, // number of grades entered
            gradeValue, // grade value
            average; // average of all grades

        // initialization phase
        total = 0; // clear total
        gradeCounter = 1; // prepare to loop

        // processing phase
        while (gradeCounter <= 10) // loop 10 times
        {
            // prompt for input and read grade from user
            Console.Write("Enter integer grade: ");

            // read input and convert to integer
            gradeValue = Int32.Parse(Console.ReadLine());
```
// add gradeValue to total
total = total + gradeValue;

// add 1 to gradeCounter
gradeCounter = gradeCounter + 1;

// termination phase
average = total / 10;  // integer division

// display average of exam grades
Console.WriteLine("\nClass average is {0}", average);

} // end Main

} // end class Average1

Enter integer grade: 100
Enter integer grade: 88
Enter integer grade: 93
Enter integer grade: 55
Enter integer grade: 68
Enter integer grade: 77
Enter integer grade: 83
Enter integer grade: 95
Enter integer grade: 73
Enter integer grade: 62

Class average is 79
**for** Repetition Structure

- The **for** repetition structure
  - Syntax: `for (Expression1, Expression2, Expression3)`
    - Expression1 = names the control variable
      - Can contain several variables
    - Expression2 = loop-continuation condition
    - Expression3 = incrementing/decrementing
      - If Expression1 has several variables, Expression3 must have several variables accordingly
      - `++counter` and `counter++` are equivalent
  - Variable scope
    - Expression1 can only be used in the body of the **for** loop
    - When the loop ends the variable expires
for Repetition Structure

```
for ( int counter = 1; counter <= 5; counter++ )
```

Components of a typical for header.
// ForCounter.cs
// Counter-controlled repetition with the for structure.

using System;

class ForCounter
{
    static void Main(string[] args)
    {
        // initialization, repetition condition and incrementing
        // are all included in the for structure
        for (int counter = 1; counter <= 5; counter++)
            Console.WriteLine(counter);
    }
}
**switch** Multiple-Selection Structure

- **The switch statement**
  - Constant expressions
    - String
    - Integral
  - Cases
    - Case ‘x’ :
      - Use of constant variable cases
    - Empty cases
    - The default case
- **The break statement**
  - Exit the switch statement
SwitchTest.cs

```csharp
// SwitchTest.cs
// Counting letter grades.

using System;

class SwitchTest
{
    static void Main(string[] args)
    {
        char grade;      // one grade
        int aCount = 0,  // number of As
            bCount = 0,  // number of Bs
            cCount = 0,  // number of Cs
            dCount = 0,  // number of Ds
            fCount = 0;  // number of Fs
```
for ( int i = 1; i <= 10; i++ )
{
    Console.Write( "Enter a letter grade: " );
    grade = Char.Parse( Console.ReadLine() );

    switch ( grade )
    {
        case 'A': // grade is uppercase A
        case 'a':  // or lowercase a
            ++aCount;
            break;

        case 'B': // grade is uppercase B
        case 'b':  // or lowercase b
            ++bCount;
            break;
    
}
case 'C':  // grade is uppercase C
    case 'c':   // or lowercase c
        ++cCount;
        break;

    case 'D':  // grade is uppercase D
    case 'd':  // or lowercase d
        ++dCount;
        break;

    case 'F':  // grade is uppercase F
    case 'f':  // or lowercase f
        ++fCount;
        break;

    default:    // processes all other characters
        Console.WriteLine("Incorrect letter grade entered."
                         + "\nGrade not added to totals."");
        break;

} // end switch

} // end for
Console.WriteLine("\nTotals for each letter grade are:\nA: {0}\nB: {1}\nC: {2}\nD: {3}\nF: {4}", aCount, bCount, cCount, dCount, fCount);

} // end method Main

} // end class SwitchTest

SwitchTest.cs .../
Enter a letter grade: a
Enter a letter grade: A
Enter a letter grade: c
Enter a letter grade: F
Enter a letter grade: z
Incorrect letter grade entered.
Grade not added to totals.
Enter a letter grade: D
Enter a letter grade: d
Enter a letter grade: B
Enter a letter grade: a
Enter a letter grade: C

Totals for each letter grade are:
A: 3
B: 1
C: 2
D: 2
F: 1
Statements `break` and `continue`

- **Use**
  - Used inside loops
  - Used to alter the flow of control
  - The `break` statement
    - Used to exit a loop early
  - The `continue` statement
    - Used to skip the rest of the statements and begin the loop at the first statement in the loop
  - Programs can be completed without their usage
// BreakTest.cs
// Using the break statement in a for structure.

using System;

class BreakTest
{
    static void Main(string[] args)
    {
        string output = "";
        int count;

        for (count = 1; count <= 10; count++)
        {
            if (count == 5)
            {
                break;
            }
            output += count + " ";
        } // end for loop
        output += "\nBroke out of loop at count = " + count;
    } // end method Main
} // end class BreakTest
using System;
using System.Windows.Forms;

class ContinueTest
{
    static void Main(string[] args)
    {
        string output = "";

        for (int count = 1; count <= 10; count++)
        {
            if (count == 5)
            {
                continue;
            }
            output += count + " ";
        }
        output += "\nUsed continue to skip printing 5";
    } // end method Main
} // end class ContinueTest
Logical and Conditional Operators

• Operators
  – Conditional AND (&&)
  – Conditional OR (||)
  – Logical exclusive OR or XOR (^)
  – Logical NOT (!)
    • Can be avoided if desired by using other conditional operators

• Used to add multiple conditions to a statement
# Logical and Conditional Operators

<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 &amp;&amp; expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

*Truth table for the `&&` (logical AND) operator.*

| expression1 | expression2 | expression1 || expression2 |
|-------------|-------------|-----------------------------|
| false       | false       | false                       |
| false       | true        | true                        |
| true        | false       | true                        |
| true        | true        | true                        |

*Truth table for the `||` (logical OR) operator.*
Logical and Conditional Operators

<table>
<thead>
<tr>
<th>expression1</th>
<th>expression2</th>
<th>expression1 ^ expression2</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

Truth table for the logical exclusive OR (^) operator.

<table>
<thead>
<tr>
<th>expression</th>
<th>!expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>True</td>
<td>false</td>
</tr>
</tbody>
</table>

Truth table for operator! (logical NOT).
C# Namespaces

• Namespace (analogous to “Package” of Java)
  – A group of classes and their methods
  – The Framework Class Library (FCL) is composed of namespaces
  – Namespaces are stored in .dll files called assemblies
  – A list of the FCL namespaces are shown next
  – Included in a program with the using keyword
# Some C# & .NET Namespaces

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>Contains essential classes and data types (such as <code>int</code>, <code>double</code>, <code>char</code>, etc.). Implicitly referenced by all C# programs.</td>
</tr>
<tr>
<td>System.Data</td>
<td>Contains classes that form ADO .NET, used for database access and manipulation.</td>
</tr>
<tr>
<td>System.Drawing</td>
<td>Contains classes used for drawing and graphics.</td>
</tr>
<tr>
<td>System.IO</td>
<td>Contains classes for the input and output of data, such as with files.</td>
</tr>
<tr>
<td>System.Threading</td>
<td>Contains classes for multithreading, used to run multiple parts of a program simultaneously.</td>
</tr>
<tr>
<td>System.Windows.Forms</td>
<td>Contains classes used to create graphical user interfaces.</td>
</tr>
<tr>
<td>System.Xml</td>
<td>Contains classes used to process XML data.</td>
</tr>
</tbody>
</table>

*Some Namespaces in the Framework Class Library.*
The C# (and .NET in general) API
System is a major namespace for C#
How to access the C# API (and the .NET API, in general)
Microsoft .NET Framework SDK

Welcome to the Microsoft .NET Framework Software Development Kit. The .NET reference for developers who use the .NET Framework technologies.

Getting Started

For those who are new to the .NET Framework technologies, the Getting Started with the .NET Framework section of the .NET Framework SDK documentation is designed to point you in the right direction.

For known issues and late-breaking information see the Release Notes.

Documentation

The .NET Framework SDK documentation provides a wide range of overviews, programming tasks, and class library reference information that is designed to help you build efficient, powerful, and scalable applications based on the .NET Framework technologies.

QuickStarts, 1

The .NET Framework and samples are with the program components that

Tools and Debug

The .NET Framework enable you to create applications and Framework.
System.Exception Hierarchy

System.Object
  System.Exception
    System.SystemException
      System.AppDomainUnloadedException
      System.ArgumentException
      System.ArithmeticException
      System.ArrayTypeMismatchException
      System.BadImageFormatException
      System.CannotUnloadAppDomainException
      System.ComponentModel.LicenseException
      System.ComponentModel.WarningException
      System.Configuration.ConfigurationException
      System.Configuration.Install.InstallException
      System.ContextMarshalException
      System.Data.DataException
      System.Data.DBConcurrencyException
      System.Data.SqlClient.SqlException
      System.Data.SqlTypes.SqlTypeException
      System.Drawing.Printing.InvalidPrinterException
      System.EnterpriseServices.RegistrationException
      System.EnterpriseServices.ServiceComponentException
      System.ExecutionEngineException
      System.FormatException
      System.IndexOutOfRangeException
      System.InvalidCastException
      System.InvalidOperationException
      System.InvalidProgramException
      System.IO.InternalBufferOverflowException
      System.IO.IOException
      System.Management.ManagementException
Program Modules in C#

- Modules
  - Class
  - Method

- To display to screen from within program (for now):
  - Console
  - MessageBox
Math Class Methods

• The **Math** class
  – Allows the user to perform common math calculations
  – Using methods
    • `ClassName.MethodName( argument1, argument2, ... )`
  – Constants
    • `Math.PI = 3.1415926535...`
    • `Math.E = 2.7182818285...`
# Math Class Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs( x )</td>
<td>absolute value of x</td>
<td>Abs( 23.7 ) is 23.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abs( 0 ) is 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Abs( -23.7 ) is 23.7</td>
</tr>
<tr>
<td>Ceiling( x )</td>
<td>rounds x to the smallest integer not less than x</td>
<td>Ceiling( 9.2 ) is 10.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ceiling( -9.8 ) is -9.0</td>
</tr>
<tr>
<td>Cos( x )</td>
<td>trigonometric cosine of x (x in radians)</td>
<td>Cos( 0.0 ) is 1.0</td>
</tr>
<tr>
<td>Exp( x )</td>
<td>exponential method $e^x$</td>
<td>Exp( 1.0 ) is approximately 2.7182818284590451</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exp( 2.0 ) is approximately 7.3890560989306504</td>
</tr>
<tr>
<td>Floor( x )</td>
<td>rounds x to the largest integer not greater than x</td>
<td>Floor( 9.2 ) is 9.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Floor( -9.8 ) is -10.0</td>
</tr>
<tr>
<td>Log( x )</td>
<td>natural logarithm of x (base e)</td>
<td>Log( 2.7182818284590451 ) is approximately 1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Log( 7.3890560989306504 ) is approximately 2.0</td>
</tr>
<tr>
<td>Max( x, y )</td>
<td>larger value of x and y (also has versions for float, int and long values)</td>
<td>Max( 2.3, 12.7 ) is 12.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max( -2.3, -12.7 ) is -2.3</td>
</tr>
<tr>
<td>Min( x, y )</td>
<td>smaller value of x and y (also has versions for float, int and long values)</td>
<td>Min( 2.3, 12.7 ) is 2.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Min( -2.3, -12.7 ) is -12.7</td>
</tr>
<tr>
<td>Pow( x, y )</td>
<td>x raised to power y ($x^y$)</td>
<td>Pow( 2.0, 7.0 ) is 128.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pow( 9.0, .5 ) is 3.0</td>
</tr>
<tr>
<td>Sin( x )</td>
<td>trigonometric sine of x (x in radians)</td>
<td>Sin( 0.0 ) is 0.0</td>
</tr>
<tr>
<td>Sqrt( x )</td>
<td>square root of x</td>
<td>Sqrt( 900.0 ) is 30.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sqrt( 9.0 ) is 3.0</td>
</tr>
<tr>
<td>Tan( x )</td>
<td>trigonometric tangent of x (x in radians)</td>
<td>Tan( 0.0 ) is 0.0</td>
</tr>
</tbody>
</table>

Commonly used Math class methods.
Method Definitions

• Writing a custom method
  – Header
    • *ReturnType Properties Name*(Param1, Param2, …)
  – Body
    • Contains the code of what the method does
    • Contains the return value if necessary
  – For uses call elsewhere in program
    • Pass parameters if needed
  – All methods must be defined inside of a class
MaximunValue.cs

// Finding the maximum of three doubles.
using System;

class MaximumValue
{
    static void Main( string[] args )
    {
        // obtain user input and convert to double
        Console.Write( "Enter first floating-point value: " );
        double number1 = Double.Parse( Console.ReadLine() );
        Console.Write( "Enter second floating-point value: " );
        double number2 = Double.Parse( Console.ReadLine() );
        Console.Write( "Enter third floating-point value: " );
        double number3 = Double.Parse( Console.ReadLine() );
        // call method Maximum to determine largest value
        double max = Maximum( number1, number2, number3 );
        // display maximum value
        Console.WriteLine("\nmaximum is: " + max );
    } // end method Main

    static double Maximum( double number1, double number2, double number3 )
    {
        if (number1 >= number2 && number1 >= number3)
        {
            return number1;
        }
        else if (number2 >= number1 && number2 >= number3)
        {
            return number2;
        }
        else
        {
            return number3;
        }
    }
} // end class MaximumValue
// Maximum method uses method Math.Max to help determine
// the maximum value

static double Maximum( double x, double y, double z )
{
    return Math.Max( x, Math.Max( y, z ) );
}

} // end class MaximumValue

Enter first floating-point value: 37.3
Enter second floating-point value: 99.32
Enter third floating-point value: 27.1928

maximum is: 99.32
Argument Promotion

- Implicit Conversion
  - Object is converted to a needed type implicitly
  - Only done if compiler knows no data will be lost
- Explicit Conversion
  - Object is manually converted
  - Required if there could be a loss of data
  - Widening
    - Make an object that of a derived class and more complex
  - Narrowing
    - Make an object that of a base class and cause some data loss
### Argument Promotion

<table>
<thead>
<tr>
<th>Type</th>
<th>Can be Converted to Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>object</td>
</tr>
<tr>
<td>byte</td>
<td>decimal, double, float, int, uint, long, ulong, object, short or ushort</td>
</tr>
<tr>
<td>sbyte</td>
<td>decimal, double, float, int, long, object or short</td>
</tr>
<tr>
<td>char</td>
<td>decimal, double, float, int, uint, long, ulong, object or ushort</td>
</tr>
<tr>
<td>decimal</td>
<td>object</td>
</tr>
<tr>
<td>double</td>
<td>object</td>
</tr>
<tr>
<td>float</td>
<td>double or object</td>
</tr>
<tr>
<td>int</td>
<td>decimal, double, float, long or object</td>
</tr>
<tr>
<td>uint</td>
<td>decimal, double, float, long, ulong, or object</td>
</tr>
<tr>
<td>long</td>
<td>decimal, double, float or object</td>
</tr>
<tr>
<td>ulong</td>
<td>decimal, double, float or object</td>
</tr>
<tr>
<td>object</td>
<td>None</td>
</tr>
<tr>
<td>short</td>
<td>decimal, double, float, int, long or object</td>
</tr>
<tr>
<td>ushort</td>
<td>decimal, double, float, int, uint, long, ulong or object</td>
</tr>
<tr>
<td>string</td>
<td>object</td>
</tr>
</tbody>
</table>

*Allowed implicit conversions.*
Value Types and Reference Types

• Value types
  – Contains **data** of the specified type
  – Programmer created
    • **structs**
    • **enumerations**

• Reference types
  – Contain an **address** to a spot in memory where the data is
  – Programmer created
    • Classes
    • Interfaces
    • Delegates

• All values are 32bit allowing cross-platform use
Passing Arguments: Call-By-Value vs. Call-By-Reference

• Passing by value
  – Send a method a copy of the object
  – When returned are always returned by value
  – Set by value by default

• Passing by reference
  – Send to the method the actual reference point
    • Causes the variable to be changed throughout the program
  – When returned are always returned by reference
  – The ref keyword specifies by reference
  – The out keyword means a called method will initialize it
class RefOutTest
{
    // x is passed as a ref int (original value will change)
    static void SquareRef( ref int x )
    {
        x = x * x;
    }

    // original value can be changed and initialized
    static void SquareOut( out int x )
    {
        x = 6; // method initializes x ("out" keyword)
        x = x * x;
    }

    // x is passed by value (original value not changed)
    static void Square( int x )
    {
        x = x * x;
    }

    static void Main( string[] args )
    {
        // create a new integer value, set it to 5
        int y = 5;
        int z; // declare z, but do not initialize it
// display original values of y and z
string output1 = "The value of y begins as "
    + y + ", z begins uninitialized.\n\n\n";

// y and z are passed by reference
RefOutTest.SquareRef( ref y );
RefOutTest.SquareOut( out z );

// display values of y and z after modified by methods
// SquareRef and SquareOut
string output2 = "After calling SquareRef with y as an " +
    "argument and SquareOut with z as an argument,\n" +
    "the values of y and z are:\n\n" +
    "y: " + y + "\nz: " + z + "\n\n";

// values of y and z are passed by value
RefOutTest.Square( y );
RefOutTest.Square( z );

// values of y and z will be same as before because Square
// did not modify variables directly
string output3 = "After calling Square on both x and y, " +
    "the values of y and z are:\n\n" +
    "y: " + y + "\nz: " + z + "\n\n";

MessageBox.Show( output1 + output2 + output3,
    "Using ref and out Parameters", MessageBoxButtons.OK,
    MessageBoxIcon.Information );

} // end method Main

} // end class RefOutTest
RefOutTest.cs
Program Output

The value of y begins as 5, z begins uninitialized.

After calling SquareRef with y as an argument and SquareOut with z as an argument, the values of y and z are:

y: 25
z: 36

After calling Square on both x and y, the values of y and z are:

y: 25
z: 36
Random Number Generation

• Class Random
  – Within namespace System
  – Truly random
    • The numbers are generated using an equation with a seed
      – The seed is usually the exact time of day
  – randomObject.Next()
    • Returns a number from 0 to Int32.MaxValue
      – Int32.MaxValue = 2,147,483,647
  – randomObject.Next(x)
    • Returns a value from 0 up to but not including x
  – randomObject.Next(x, y)
    • Returns a number between x and up to but not including y
// RandomInt.cs; // Random integers.
using System;
using System.Windows.Forms;

// calculates and displays 20 random integers
class RandomInt
{
    // main entry point for application
    static void Main( string[] args )
    {
        int value;
        string output = "";

        Random randomInteger = new Random(); // create a new Random object

        // loop 20 times
        for ( int i = 1; i <= 20; i++ )
        {
            // pick random integer between 1 and 6
            value = randomInteger.Next( 1, 7 ); // Will set value to a random number from 1 up to but not including 7
            output += value + " "; // append value to output
            // if counter divisible by 5, append newline
            if ( i % 5 == 0 ) //Format the output to only have 5 numbers per line
            {
                output += "\n";
            }
        } // end for
    }
}
31        MessageBox.Show(output, "20 Random Numbers from 1 to 6",
32            MessageBoxButtons.OK, MessageBoxIcon.Information );
33
34        } // end Main
35
36        } // end class RandomInt
Duration of Identifiers

- **Duration**
  - The amount of time an identifier exist in memory

- **Scope**
  - The section of a program in which the object can be referenced

- **Local variables**
  - Created when declared
  - Destroyed when the block exits
  - Not initialized
    - Most variables are set to 0
    - All `bool` variables are set to `false`
    - All reference variables are set to `null`
Scope Rules

• **Scope**
  – Portion of a program in which a variable can be accessed
  – **Class scope**
    • From when created in class
    • Until end of class (})
    • Global to all methods in that class
      – Direct modification
    • Repeated names causes previous to be hidden until scope ends
  – **Block scope**
    • From when created
    • Until end of block (})
    • Only used within that block
      – Must be passed and modified indirectly
    • Cannot repeat variable names
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class Scoping : System.Windows.Forms.Form
{
    private System.ComponentModel.Container components = null;
    private System.Windows.Forms.Label outputLabel;
    public int x = 1;

    public Scoping()
    {
        InitializeComponent();
        int x = 5;     // variable local to constructor
        outputLabel.Text = outputLabel.Text +
                           "local x in method Scoping is " + x;

        MethodA();     // MethodA has automatic local x;
        MethodB();     // MethodB uses instance variable x
        MethodA();     // MethodA creates new automatic local x
        MethodB();     // instance variable x retains its value

        outputLabel.Text = outputLabel.Text +
                           "\n\nlocal x in method Scoping is " + x;
    }
}
```csharp
public void MethodA()
{
    int x = 25;  // initialized each time a is called
    outputLabel.Text = outputLabel.Text + "\n\nlocal x in MethodA is " + x + " after entering MethodA";
    ++x;
    outputLabel.Text = outputLabel.Text + "\nlocal x in MethodA is " + x + " before exiting MethodA";
}

public void MethodB()
{
    outputLabel.Text = outputLabel.Text + "\n\ninstance variable x is " + x + " on entering MethodB";
    x *= 10;
    outputLabel.Text = outputLabel.Text + "\ninstance variable x is " + x + " on exiting MethodB";
}

// main entry point for application
[STAThread]
static void Main()
{
    Application.Run( new Scoping() );
}

} // end of class Scoping
```

Scoping.cs

- Uses a new `x` variable that hides the value of the global `x`.
- Uses the global version of `x` (1).
- Will permanently change the value of `x` globally.

Uses the global version of `x` (1)
Scoping.cs
Program Output

local x in method Scoping is 5

local x in MethodA is 25 after entering MethodA
local x in MethodA is 26 before exiting MethodA

instance variable x is 1 on entering MethodB
instance variable x is 10 on exiting MethodB

local x in MethodA is 25 after entering MethodA
local x in MethodA is 26 before exiting MethodA

instance variable x is 10 on entering MethodB
instance variable x is 100 on exiting MethodB

local x in method Scoping is 5
Recursion

- Recursive methods
  - Methods that call themselves
    - Directly
    - Indirectly
      - Call others methods which call it
  - Continually breaks problem down to simpler forms
  - Must converge in order to end recursion
  - Each method call remains open (unfinished)
    - Finishes each call and then finishes itself
public long Factorial( long number )
{
    if ( number <= 1 ) // base case;
    //The recursion ends when the value is less than or equal to 1
    return 1;

    else
    return number * Factorial( number - 1 );
}
Method Overloading

• Methods with the same name
  - Can have the same name but need different arguments
    • Variables passed must be different
      - Either in type received or order sent
// first version, takes one integer
public int Square ( int x )
{
    return x * x;
}

// second version, takes one double
public double Square ( double y )
{
    return y * y;
}
Object-Based Programming
Introduction

• Object classes encapsulate (wrap together) data and methods
• Objects can hide implementation from other objects (information hiding)
• Methods : units of programming
• Classes have
  – Data members (member variable or instance variables)
  – Methods that manipulate the data members
• The opening left brace (\{) and closing right brace (\}) delimit the body of a class

• Variables inside the class definition but not a method definition are called *instance variables*

• *Member Access Modifiers*
  – *public*: member is accessible wherever an instance of the object exists
  – *private*: members is accessible only inside the class definition
  – (More later)
• Access methods: read or display data
• Predicate methods: test the truth of conditions
• Constructor
  – Initializes objects of the class
  – Can take arguments
  – Cannot return values
  – There may be more than one constructor per class (overloaded constructors)
• Operator **new** used to instantiate classes (as in Java)
• (in VS.NET Use **Project < Add Class** to add a new class to your project
// Time1.cs
// Class Time1 maintains time in 24-hour format.

using System;

// Time1 class definition
public class Time1
{
    private int hour;    // 0-23
    private int minute; // 0-59
    private int second;  // 0-59

    // Time1 constructor initializes instance variables to zero to set default time to midnight
    public Time1()
    {
        SetTime( 0, 0, 0 );
    }

    // Set new time value in 24-hour format. Perform validity checks on the data. Set invalid values to zero.
    public void SetTime(
        int hourValue, int minuteValue, int secondValue )
    {
        hour = ( hourValue >= 0 && hourValue < 24 ) ?
            hourValue : 0;
        minute = ( minuteValue >= 0 && minuteValue < 60 ) ?
            minuteValue : 0;
        second = ( secondValue >= 0 && secondValue < 60 ) ?
            secondValue : 0;
    }
}
// TimeTest1.cs
// Demonstrating class Time1.

using System;
using System.Windows.Forms;

// TimeTest1 uses creates and uses a Time1 object
class TimeTest1
{
    // main entry point for application
    static void Main( string[] args )
    {
        Time1 time = new Time1();  // calls Time1 constructor

        // attempt valid time settings
        time.SetTime( 13, 27, 6 );

        // attempt invalid time settings
        time.SetTime( 99, 99, 99 );
    }
}
Class Scope

- All members are accessible within the class’s methods and can be referenced by name.
- Instance variables may also be accessed by using the keyword `this` and the dot operator (such as `this.hour`).
- Outside a class, private members cannot be referenced by name, public members may be referenced using the dot operator (`referenceName.memberName`).
- Method-scope variables
  - Only accessible within the methods in which they are defined
  - Hide instance variables