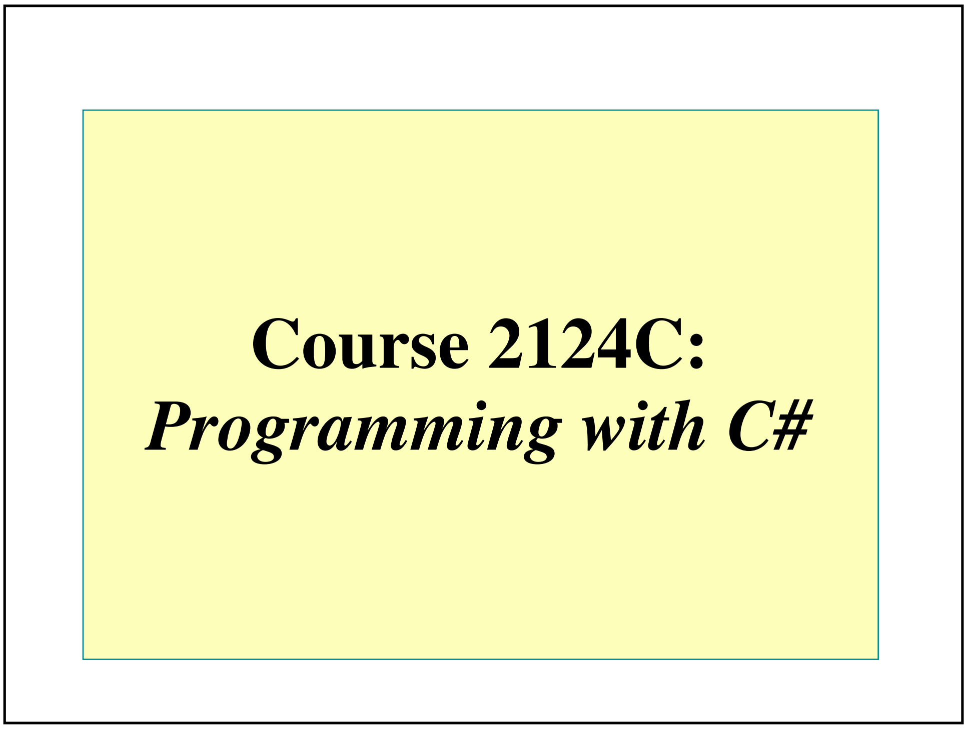
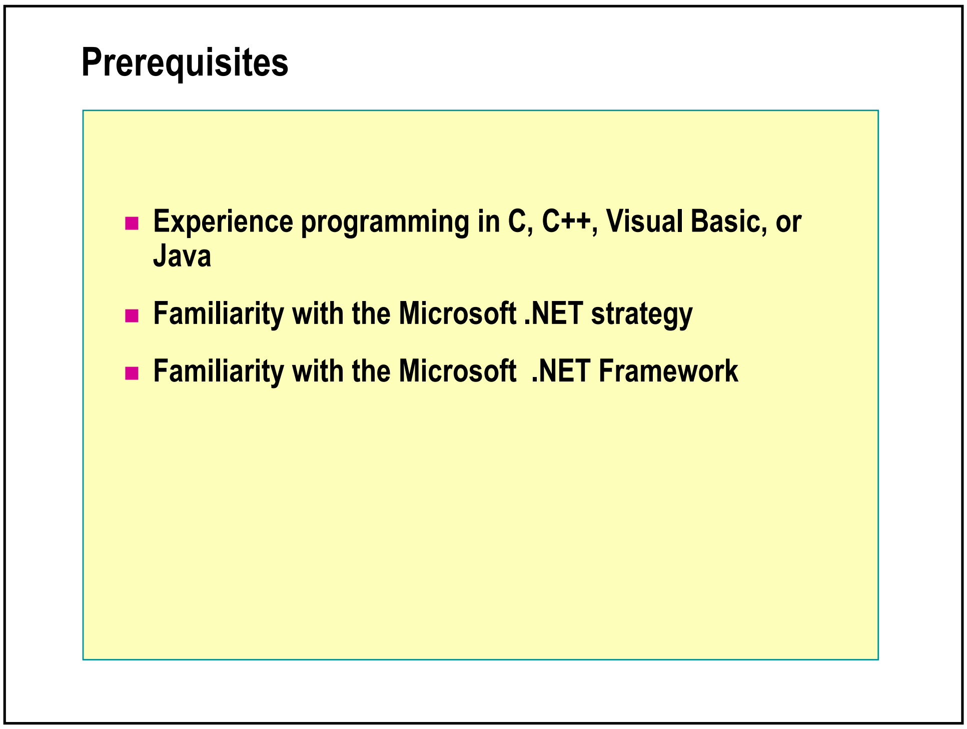
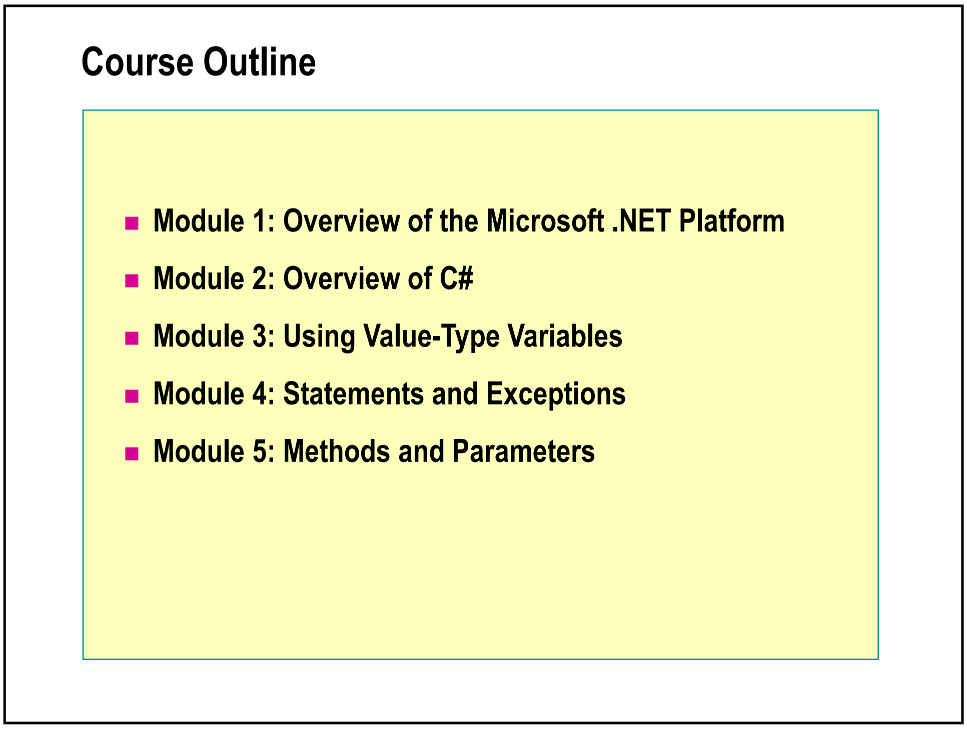


Course 2124C: *Programming with C#*



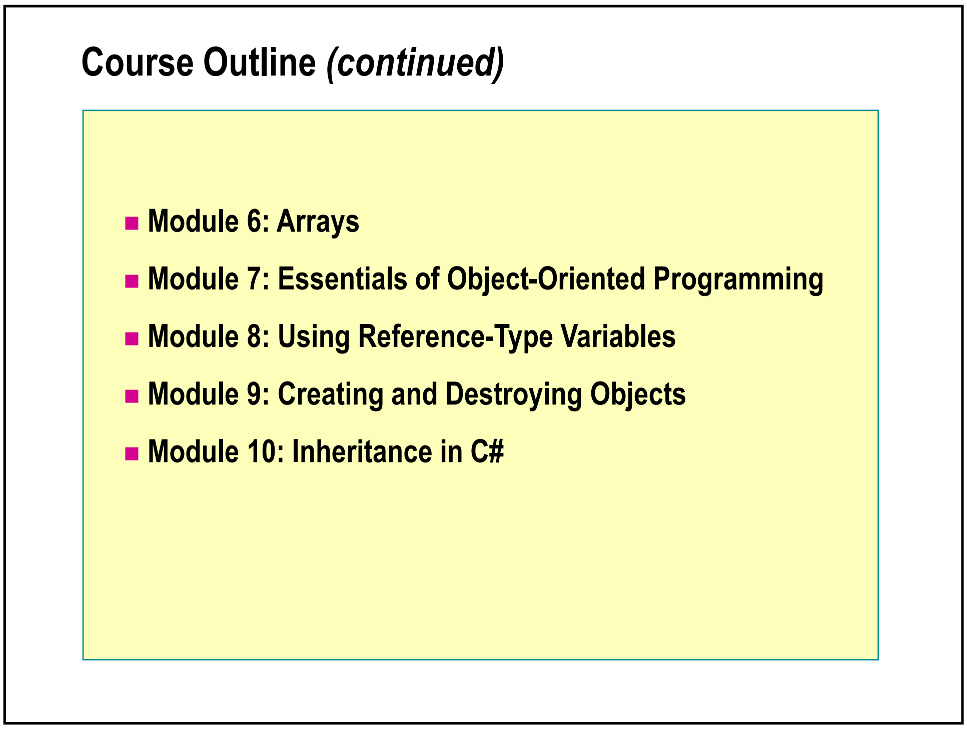
Prerequisites

- Experience programming in C, C++, Visual Basic, or Java
- Familiarity with the Microsoft .NET strategy
- Familiarity with the Microsoft .NET Framework



Course Outline

- Module 1: Overview of the Microsoft .NET Platform
- Module 2: Overview of C#
- Module 3: Using Value-Type Variables
- Module 4: Statements and Exceptions
- Module 5: Methods and Parameters



Course Outline *(continued)*

- Module 6: Arrays
- Module 7: Essentials of Object-Oriented Programming
- Module 8: Using Reference-Type Variables
- Module 9: Creating and Destroying Objects
- Module 10: Inheritance in C#

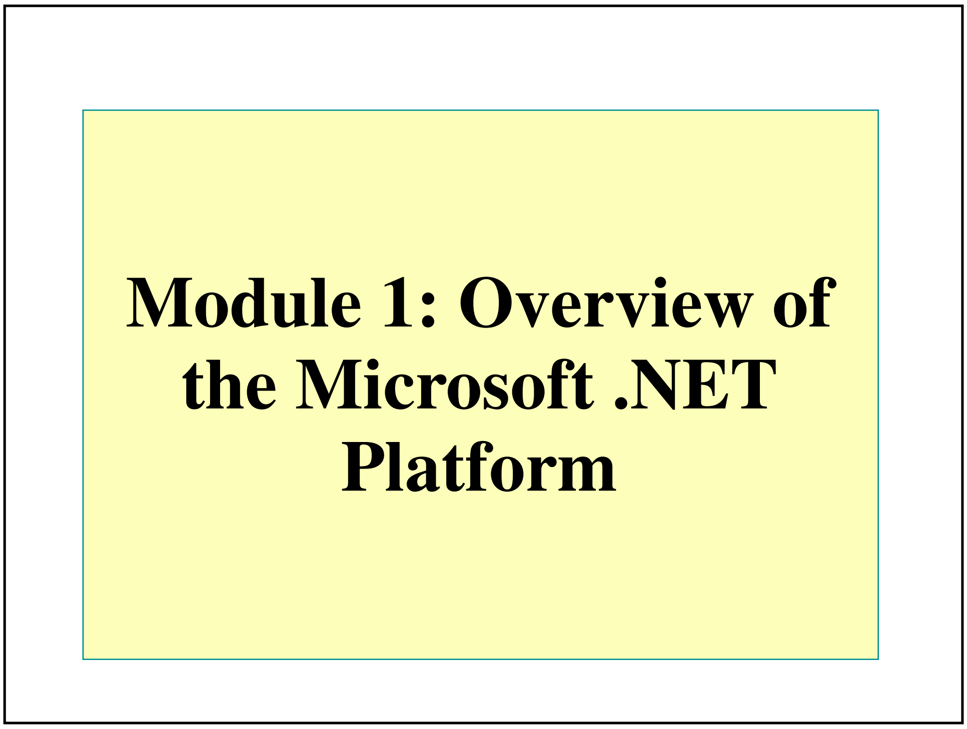
Course Outline *(continued)*

- Module 11: Aggregation, Namespaces, and Advanced Scope
- Module 12: Operators, Delegates, and Events
- Module 13: Properties and Indexers
- Module 14: Attributes
- Appendix A: Resources for Further Study

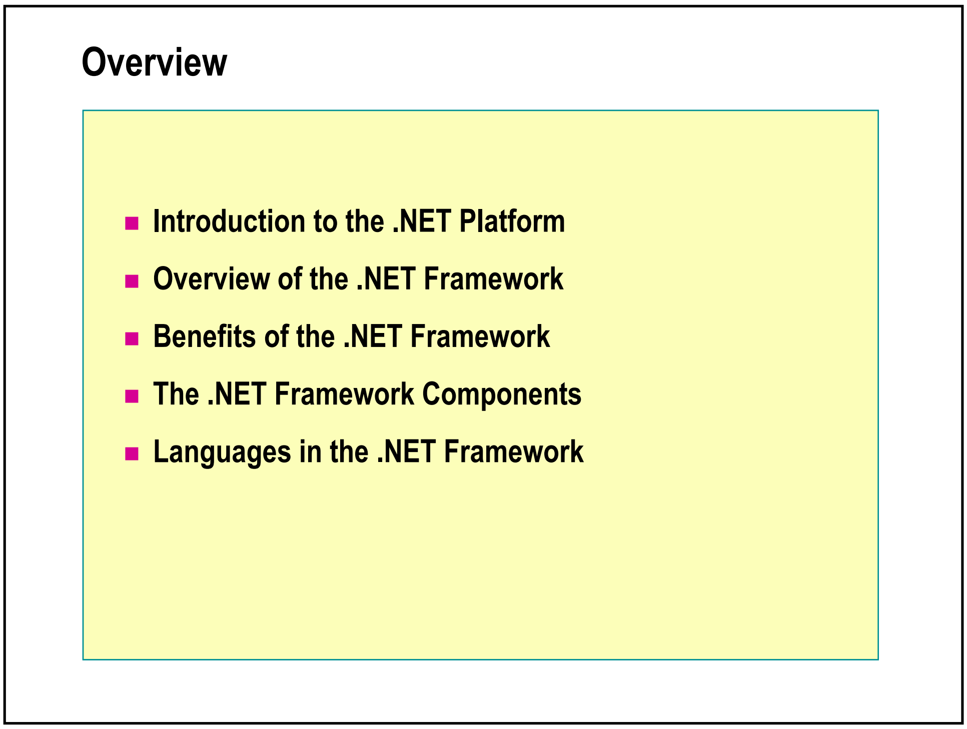
Microsoft Certified Professional Program

Microsoft Certified Professional

http://www.microsoft.com/traincert/



Module 1: Overview of the Microsoft .NET Platform



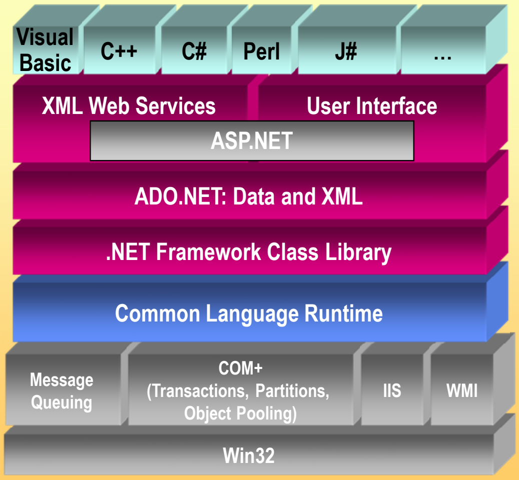
Overview

- Introduction to the .NET Platform
- Overview of the .NET Framework
- Benefits of the .NET Framework
- The .NET Framework Components
- Languages in the .NET Framework

Introduction to the .NET Platform

- The .NET Framework
- .NET My Services
- The .NET Enterprise Servers
- Visual Studio .NET

Overview of the .NET Framework



Visual Basic

C++

C#

Perl

J#

…

XML Web Services

User Interface

ASP.NET

ADO.NET: Data and XML

.NET Framework Class Library

Common Language Runtime

Message Queuing

COM+ (Transactions, Partitions, Object Pooling)

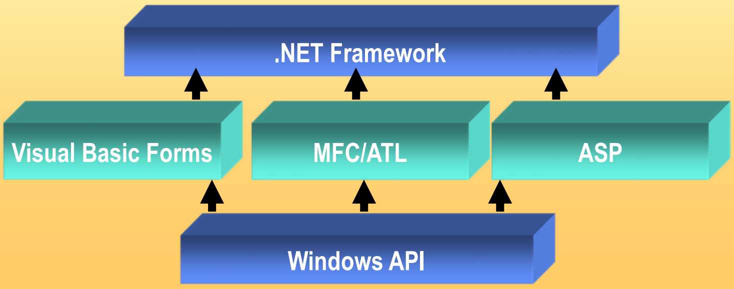
IIS

WMI

Win32

Benefits of the .NET Framework

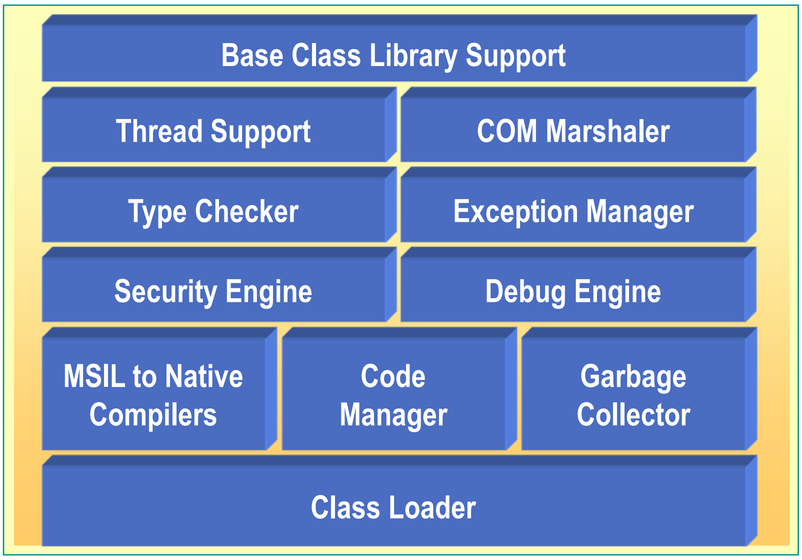
- Based on Web standards and practices
- Designed using unified application models
- Easy for developers to use
- Extensible classes



The .NET Framework Components

- Common Language Runtime
- .NET Framework Class Library
- ADO.NET: Data and XML
- Web Forms and XML Web Services
- User Interface for Windows

Common Language Runtime


Base Class Library Support

Thread Support

COM Marshaler

Type Checker

Exception Manager

Security Engine

Debug Engine

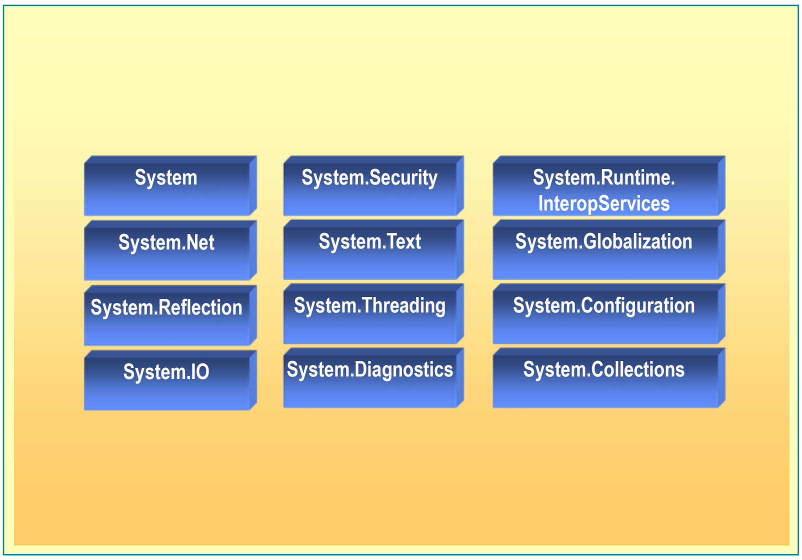
MSIL to Native Compilers

Code Manager

Garbage Collector

Class Loader

.NET Framework Class Library


System

System.Security

System.Runtime.InteropServices

System.Net

System.Text

System.Globalization

System.Reflection

System.Threading

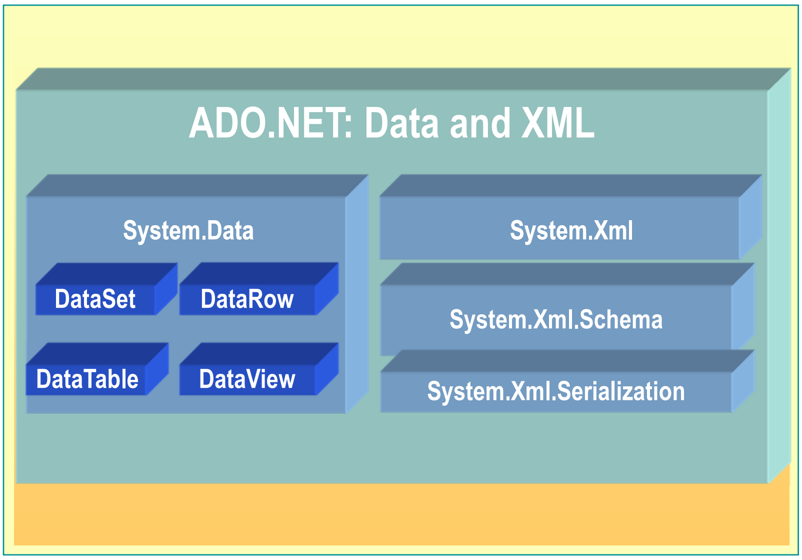
System.Configuration

System.IO

System.Diagnostics

System.Collections

ADO.NET: Data and XML



ADO.NET: Data and XML

System.Data

DataSet

DataRow

DataTable

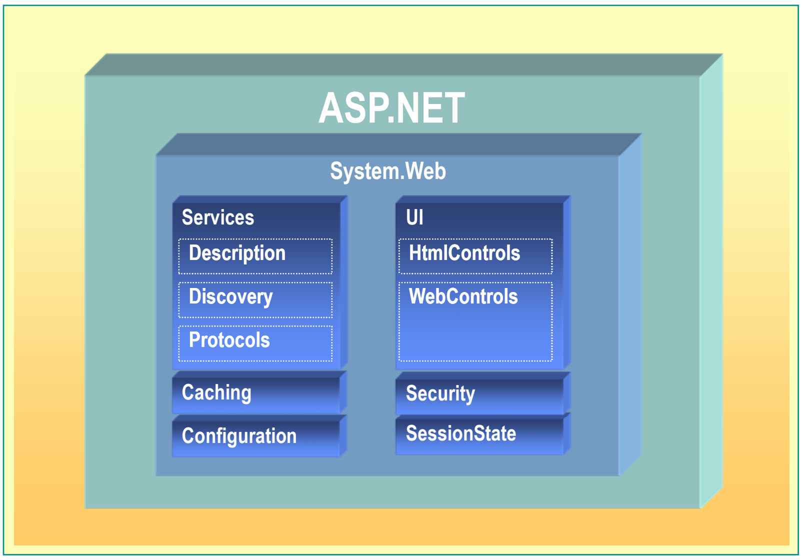
DataView

System.Xml

System.Xml.Schema

System.Xml.Serialization

Web Forms and XML Web Services



ASP.NET

System.Web

Services

Description

Discovery

Protocols

Caching

Configuration

UI

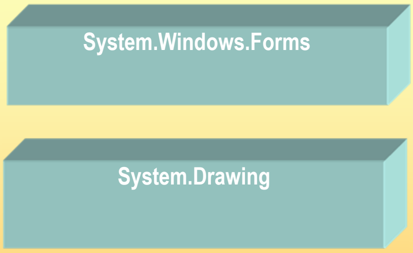
HtmlControls

WebControls

Security

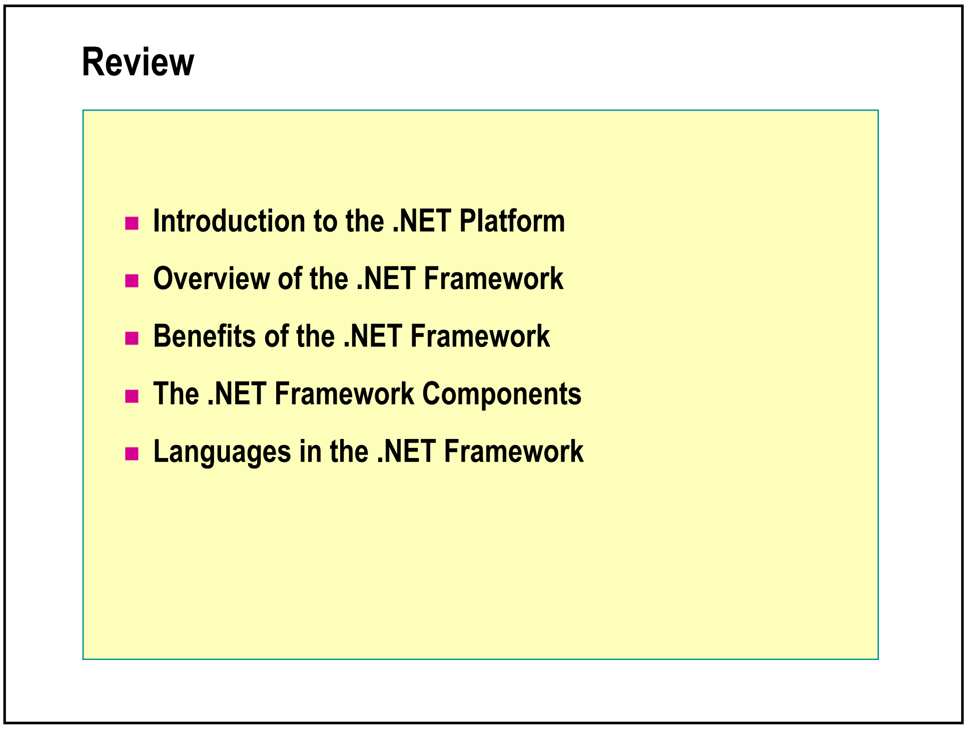
SessionState

User Interface for Windows



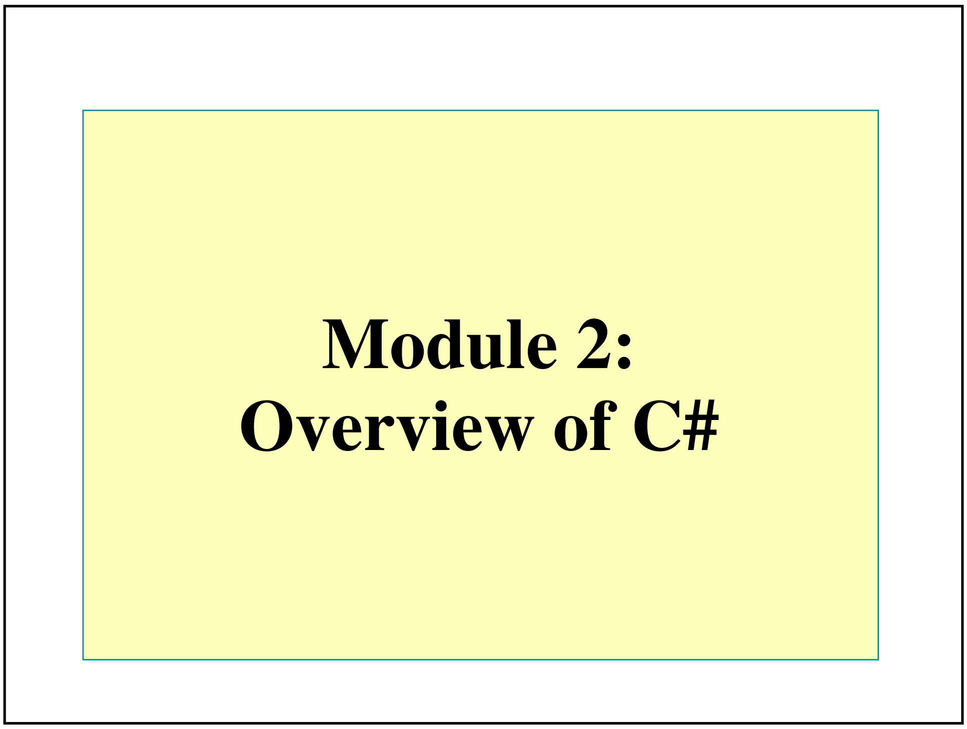
Languages in the .NET Framework

- **C# – Designed for .NET**
  New component-oriented language
- **Managed Extensions to C++**
  Enhanced to provide more power and control
- **Visual Basic .NET**
  New version of Visual Basic with substantial language innovations
- **JScript .NET**
  New version of JScript that provides improved performance and productivity
- **J# .NET**
  .NET Java-language support enabling new development and Java migration
- **Third-party Languages**

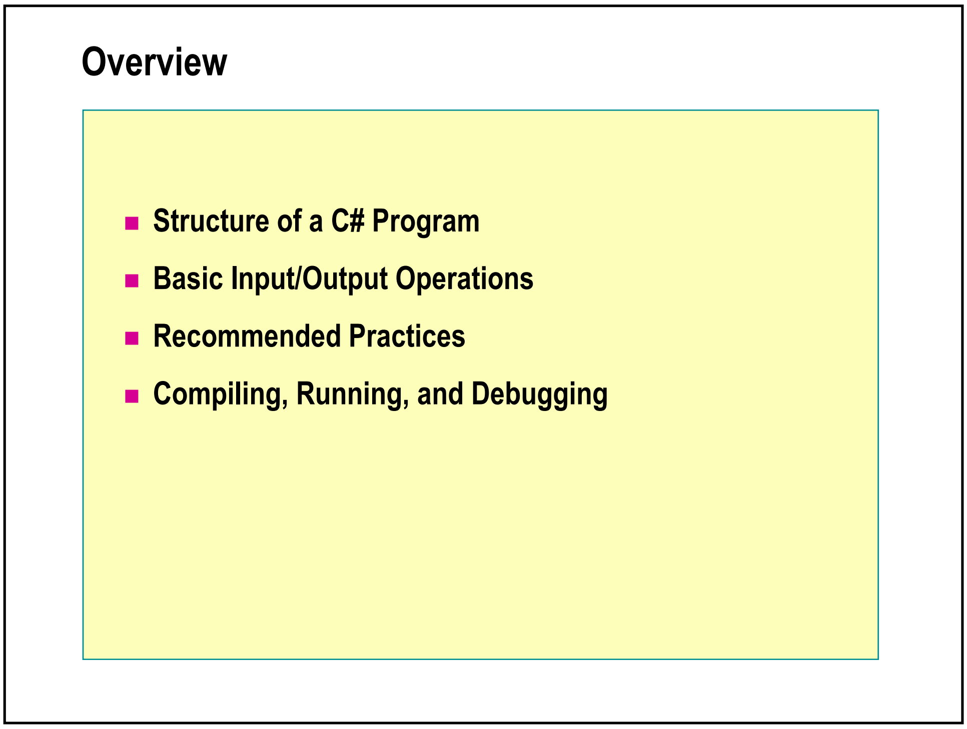


Review

- Introduction to the .NET Platform
- Overview of the .NET Framework
- Benefits of the .NET Framework
- The .NET Framework Components
- Languages in the .NET Framework

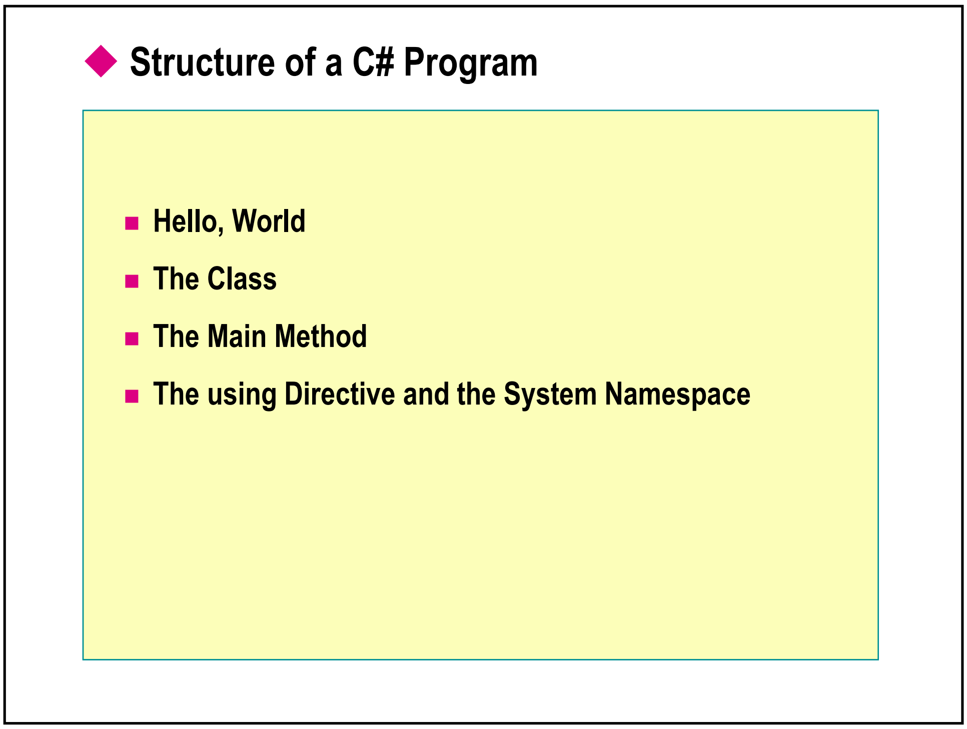


Module 2:
Overview of C#



Overview

- Structure of a C# Program
- Basic Input/Output Operations
- Recommended Practices
- Compiling, Running, and Debugging



◆ Structure of a C# Program

- Hello, World
- The Class
- The Main Method
- The using Directive and the System Namespace

Hello, World

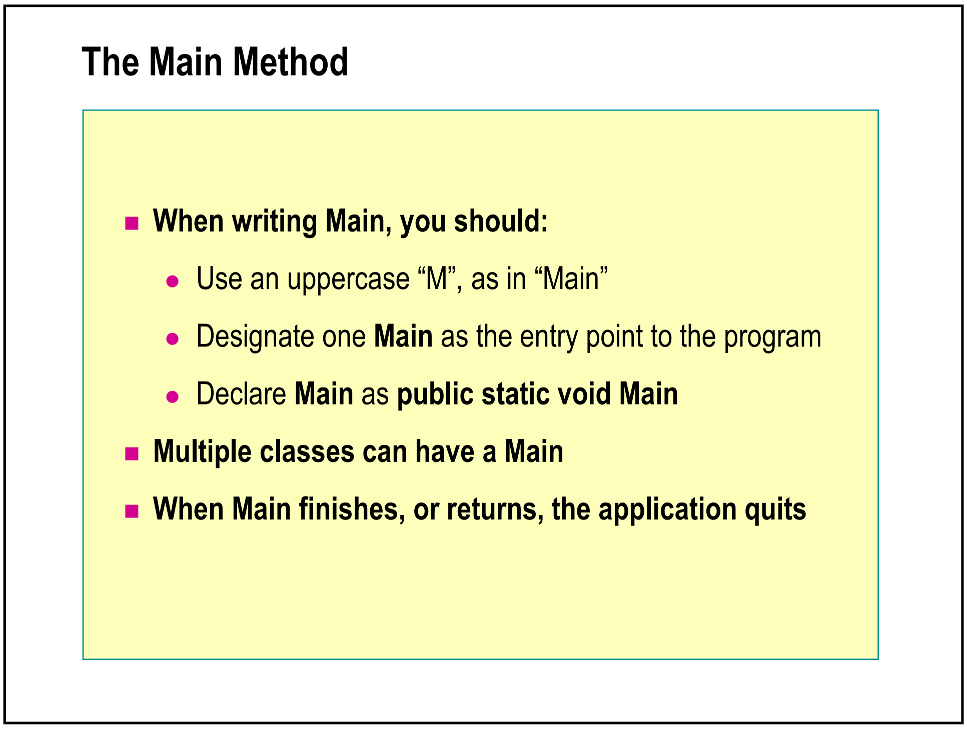
```
using System;

class Hello
{
  public static void Main()
  {
    Console.WriteLine("Hello, World");
  }
}
```

The Class

- **A C# application is a collection of classes, structures, and types**
- **A class Is a set of data and methods**
- **Syntax**

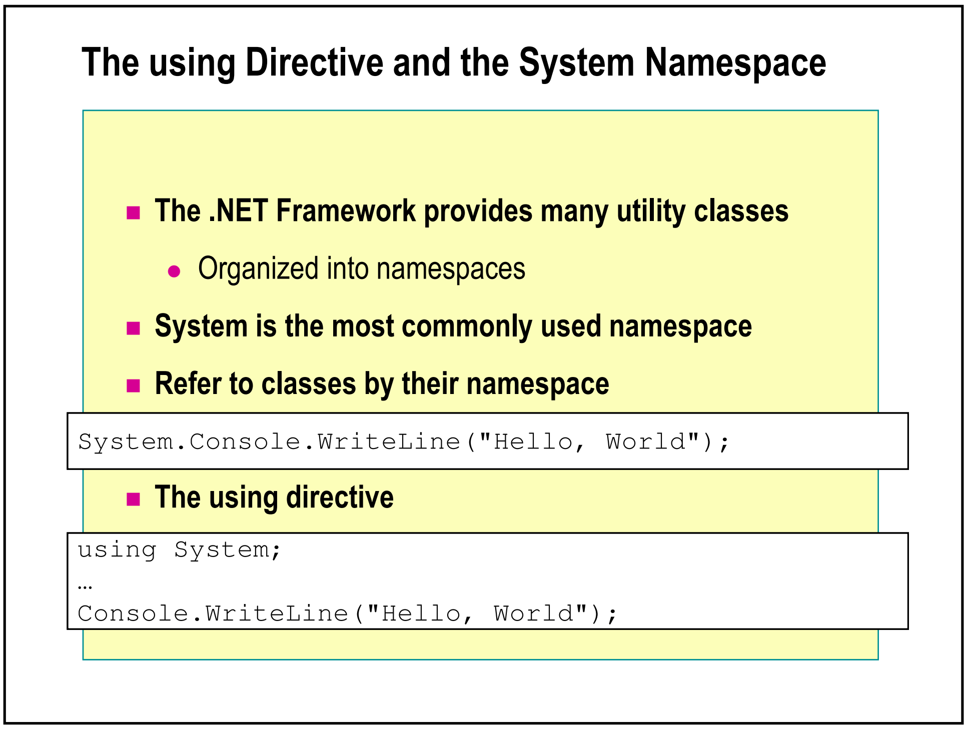
```
class name
{
     ...
}
```

- **A C# application can consist of many files**
- A class cannot span multiple files



The Main Method

- **When writing Main, you should:**
  - Use an uppercase “M”, as in “Main”
  - Designate one **Main** as the entry point to the program
  - Declare **Main** as **public static void Main**
- **Multiple classes can have a Main**
- **When Main finishes, or returns, the application quits**



The using Directive and the System Namespace

- **The .NET Framework provides many utility classes**
  - Organized into namespaces
- **System is the most commonly used namespace**
- **Refer to classes by their namespace**

```
System.Console.WriteLine("Hello, World");
```

- **The using directive**

```
using System;
…
Console.WriteLine("Hello, World");
```

◆ Basic Input/Output Operations

- The Console Class
- Write and WriteLine Methods
- Read and ReadLine Methods

The Console Class

- Provides access to the standard input, standard output, and standard error streams
- Only meaningful for console applications
  - Standard input – keyboard
  - Standard output – screen
  - Standard error – screen
- All streams may be redirected

Write and WriteLine Methods

- **Console.Write and Console.WriteLine display information on the console screen**
  - **WriteLine** outputs a line feed/carriage return
- **Both methods are overloaded**
- **A format string and parameters can be used**
  - Text formatting
  - Numeric formatting

Read and ReadLine Methods

- **Console.Read and Console.ReadLine read user input**
  - **Read** reads the next character
  - **ReadLine** reads the entire input line

◆ Recommended Practices

- Commenting Applications
- Generating XML Documentation
- Exception Handling

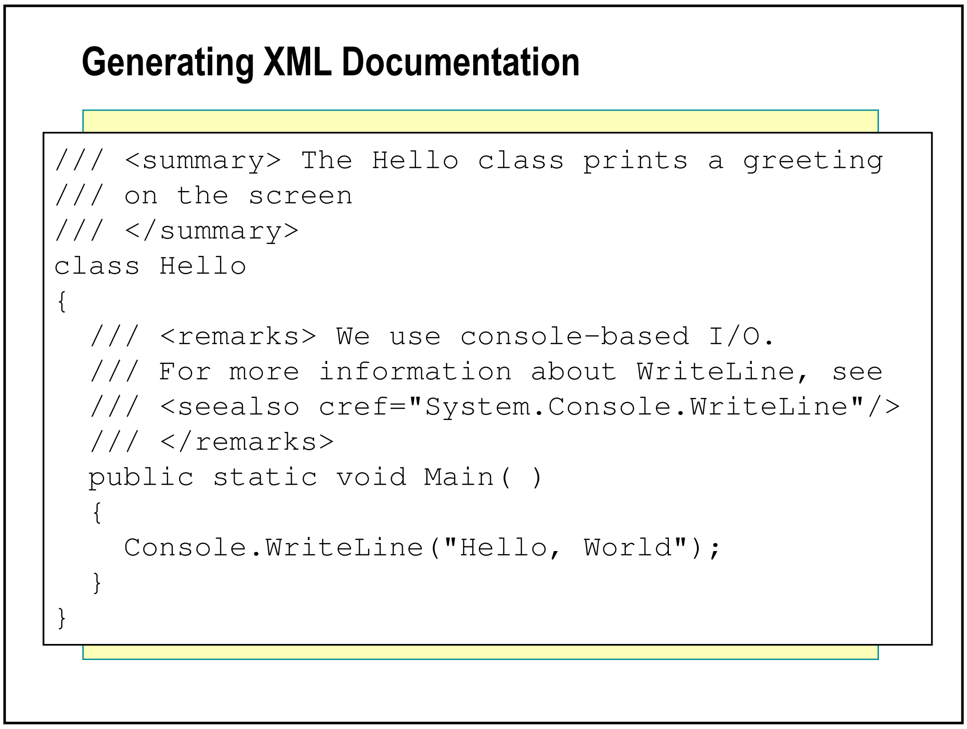
Commenting Applications

- **Comments are important**
  - A well-commented application permits a developer to fully understand the structure of the application
- **Single-line comments**

```
// Get the user’s name
Console.WriteLine("What is your name? ");
name = Console.ReadLine( );
```

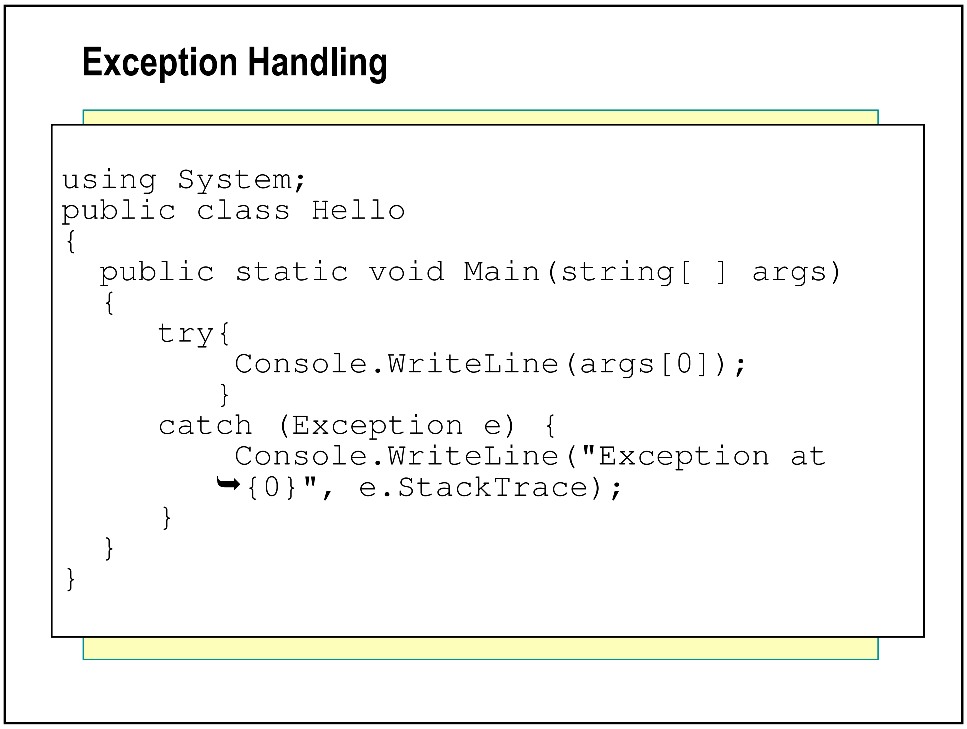
- **Multiple-line comments**

```
/* Find the higher root of the
   quadratic equation */
x = (…);
```



Generating XML Documentation

```
/// <summary> The Hello class prints a greeting
/// on the screen
/// </summary>
class Hello
{
  /// <remarks> We use console-based I/O.
  /// For more information about WriteLine, see
  /// <seealso cref="System.Console.WriteLine"/>
  /// </remarks>
  public static void Main( )
  {
    Console.WriteLine("Hello, World");
  }
}
```



Exception Handling

```
using System;
public class Hello
{
  public static void Main(string[ ] args)
  {
     try{
         Console.WriteLine(args[0]);
        }
     catch (Exception e) {
         Console.WriteLine("Exception at
        ➥{0}", e.StackTrace);
     }
  }
}
```

◆ Compiling, Running, and Debugging

- Invoking the Compiler
- Running the Application
- Demo: Compiling and Running a C# Program
- Debugging
- Demo: Using the Visual Studio Debugger
- The SDK Tools
- Demo: Using ILDASM

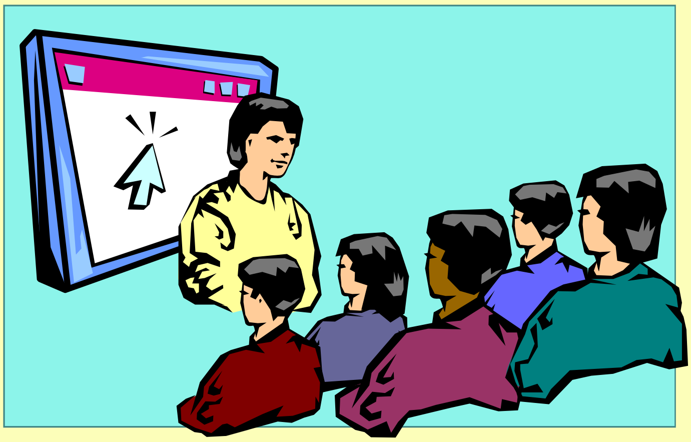
Invoking the Compiler

- Common Compiler Switches
- Compiling from the Command Line
- Compiling from Visual Studio
- Locating Errors

Running the Application

- **Running from the Command Line**
  - Type the name of the application
- **Running from Visual Studio**
  - Click **Start Without Debugging** on the **Debug** menu

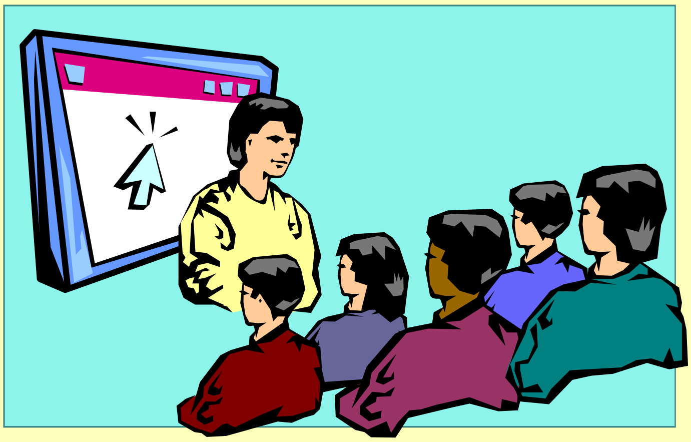
Demo: Compiling and Running a C# Program



Debugging

- **Exceptions and JIT Debugging**
- **The Visual Studio Debugger**
  - Setting breakpoints and watches
  - Stepping through code
  - Examining and modifying variables

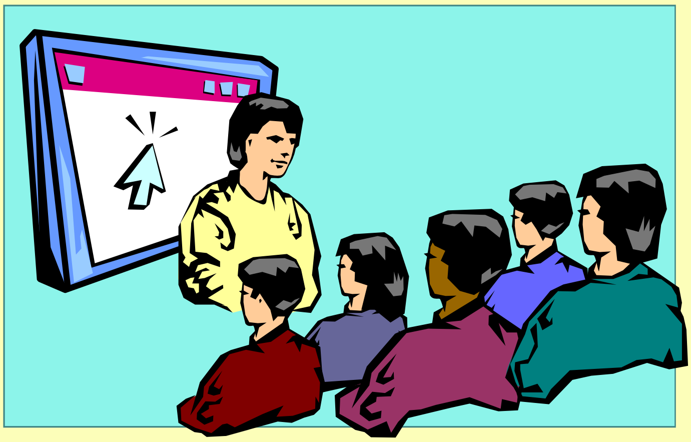
Demo: Using the Visual Studio Debugger

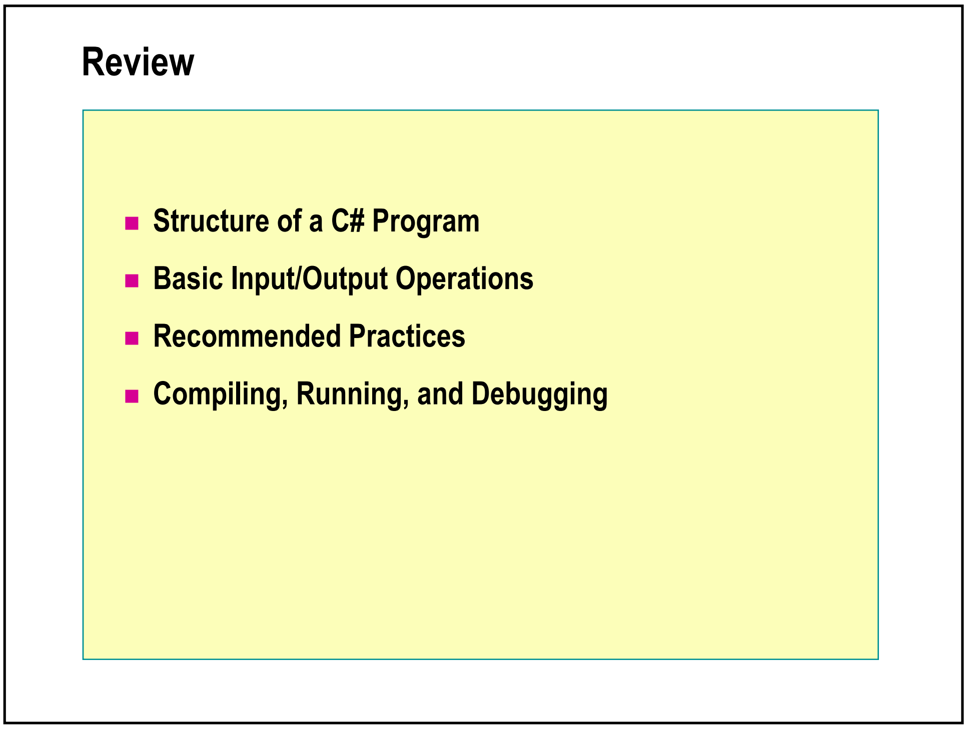


The SDK Tools

- General Tools and Utilities
- Windows Forms Design Tools and Utilities
- Security Tools and Utilities
- Configuration and Deployment Tools and Utilities

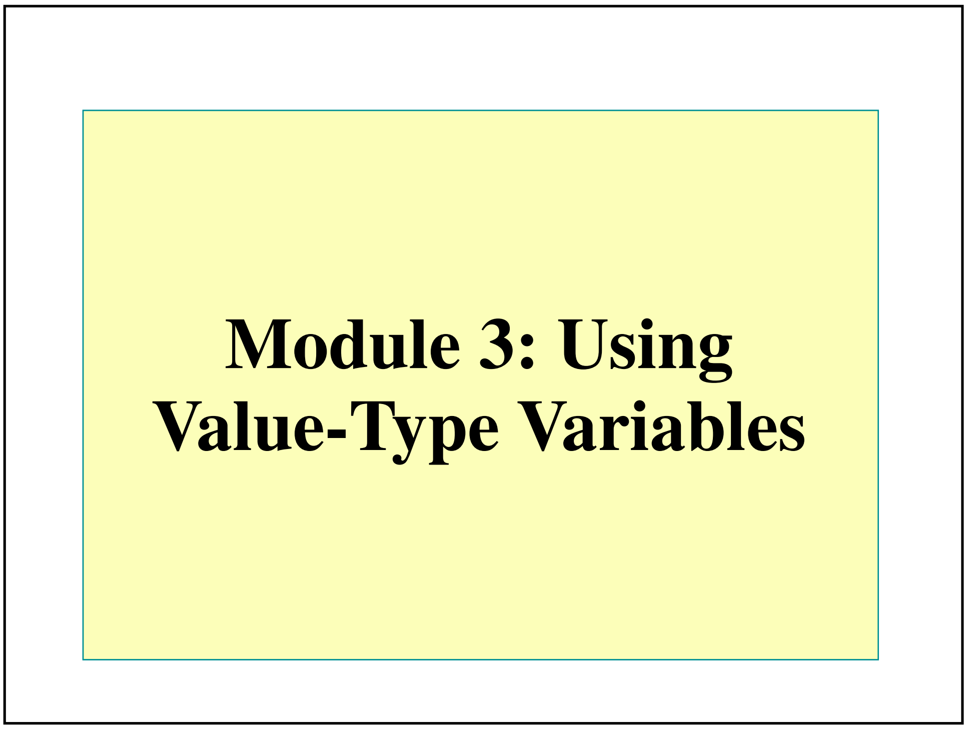
Demo: Using ILDASM



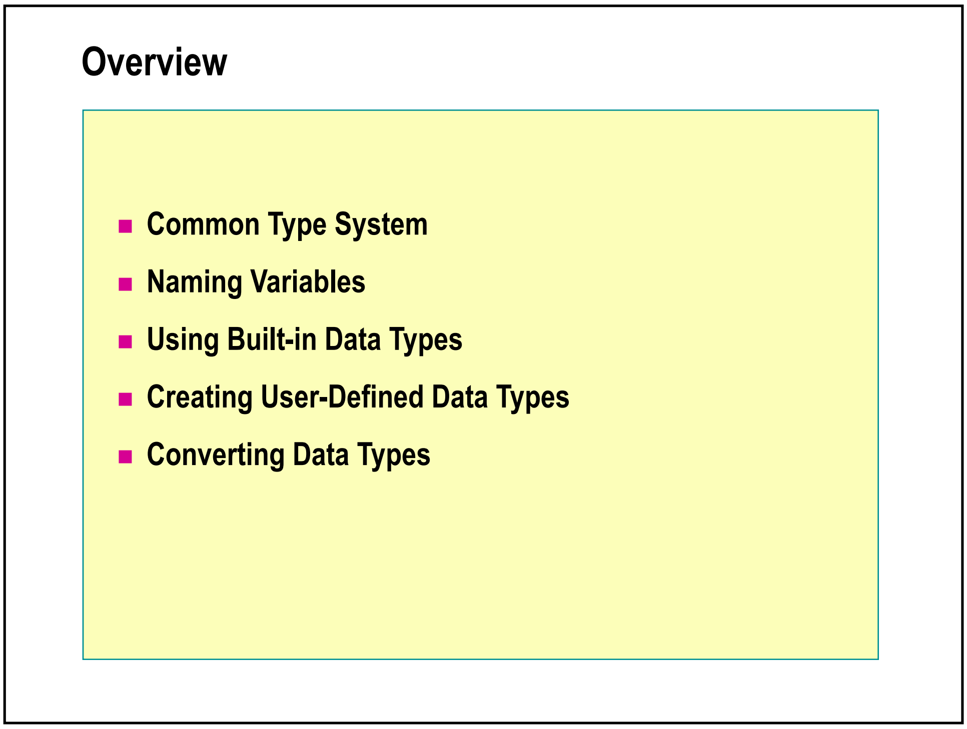


Review

- Structure of a C# Program
- Basic Input/Output Operations
- Recommended Practices
- Compiling, Running, and Debugging

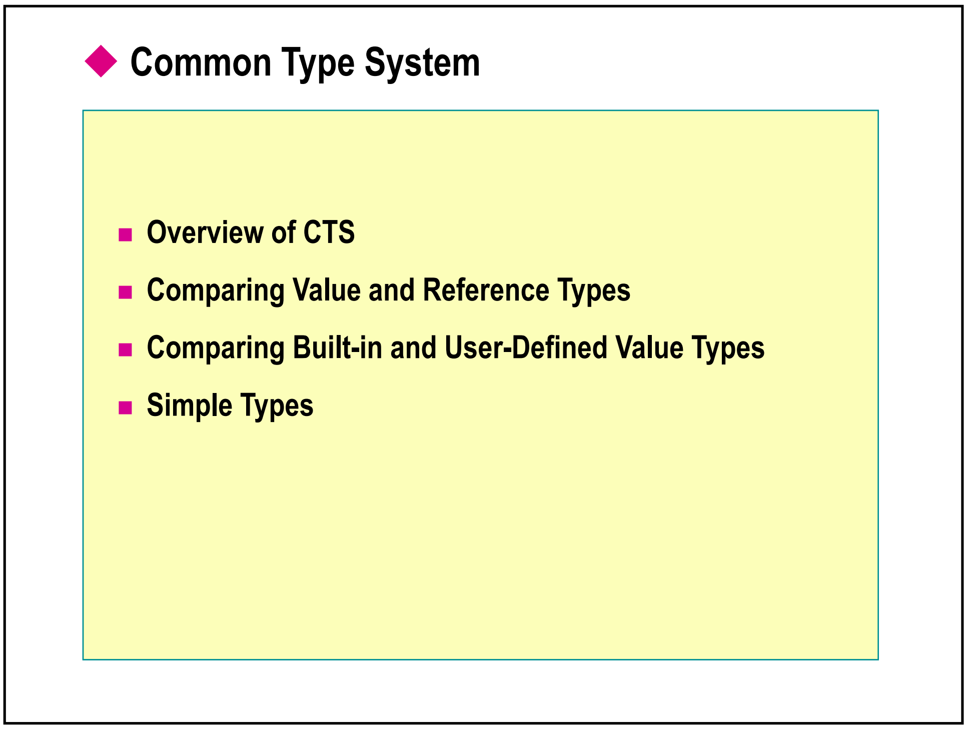


Module 3: Using Value-Type Variables



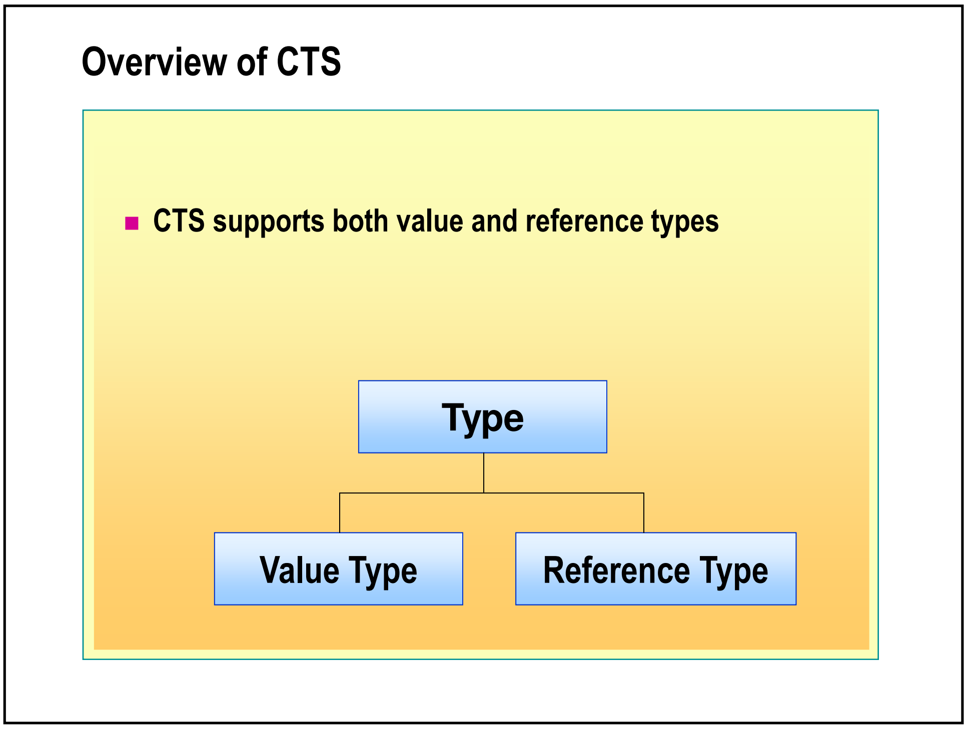
Overview

- Common Type System
- Naming Variables
- Using Built-in Data Types
- Creating User-Defined Data Types
- Converting Data Types



◆ Common Type System

- Overview of CTS
- Comparing Value and Reference Types
- Comparing Built-in and User-Defined Value Types
- Simple Types



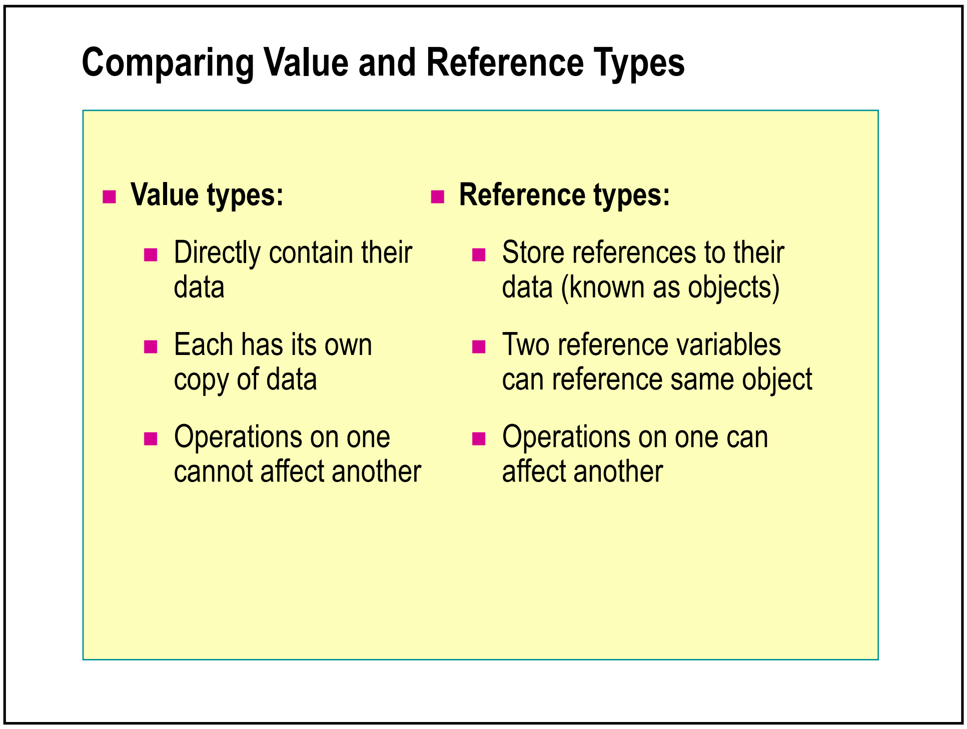
Overview of CTS

- CTS supports both value and reference types

Type

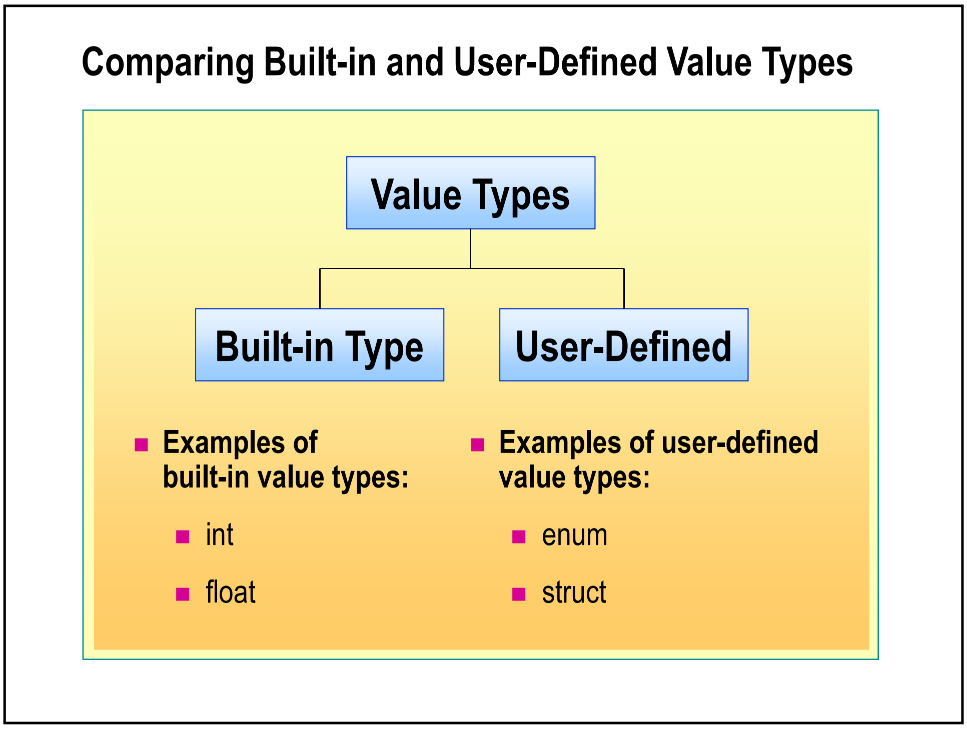
Value Type

Reference Type



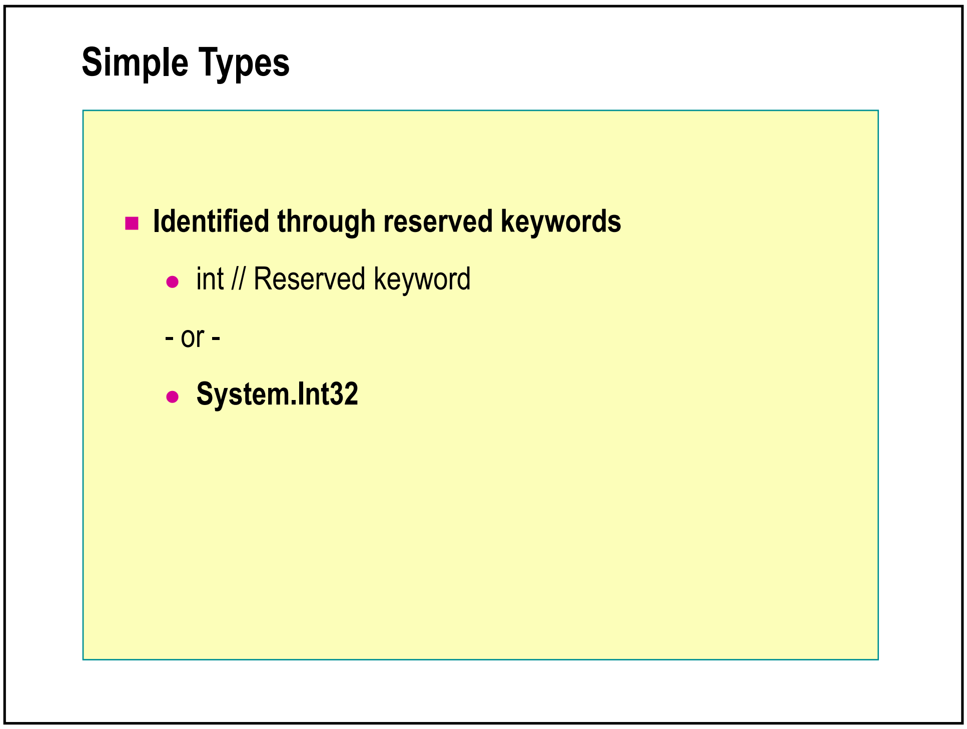
Comparing Value and Reference Types

- **Value types:**
  - Directly contain their data
  - Each has its own copy of data
  - Operations on one cannot affect another
- **Reference types:**
  - Store references to their data (known as objects)
  - Two reference variables can reference same object
  - Operations on one can affect another



Comparing Built-in and User-Defined Value Types

- **Examples of built-in value types:**
  - int
  - float
- **Examples of user-defined value types:**
  - enum
  - struct



Simple Types

- **Identified through reserved keywords**
  - int // Reserved keyword

  - or -

  - **System.Int32**

Naming Variables

- Rules and Recommendations for Naming Variables
- C# Keywords
- Quiz: Can You Spot Disallowed Variable Names?

Rules and Recommendations for Naming Variables

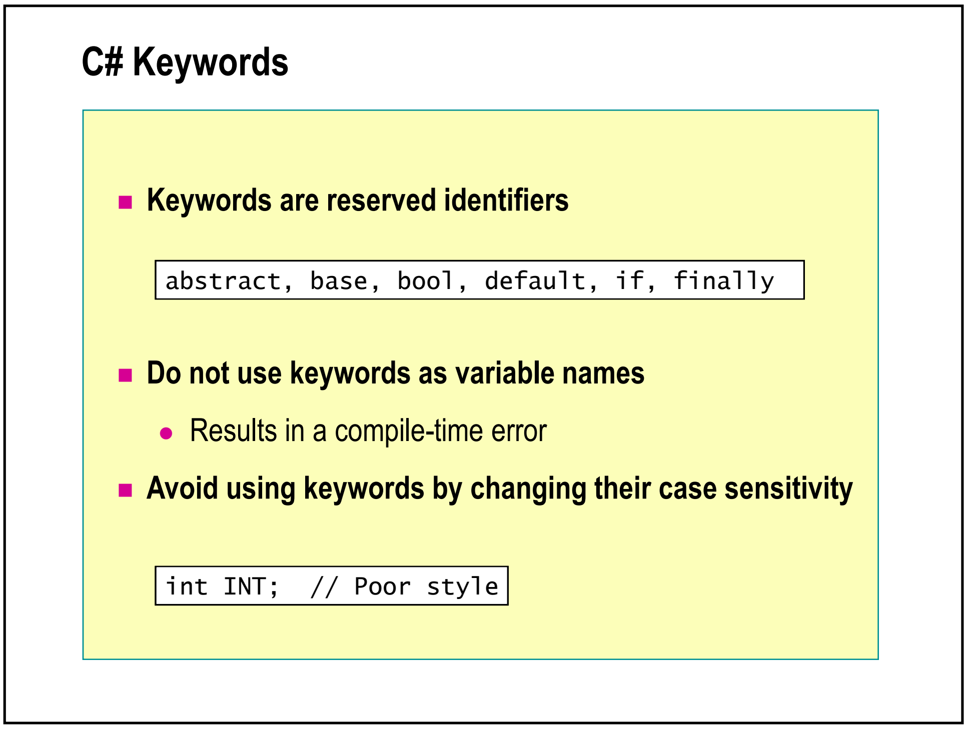
- **Rules**
  - Use letters, the underscore, and digits
- **Recommendations**
  - Avoid using all uppercase letters
  - Avoid starting with an underscore
  - Avoid using abbreviations
  - Use PascalCasing naming in multiple-word names

```
Answer42     ✓
42Answer     ✗
```

```
different    ✓
Different    ✓
```

```
BADSTYLE     ✗
_poorstyle   ✗
BestStyle    ✓
```

```
Msg          ✗
Message      ✓
```



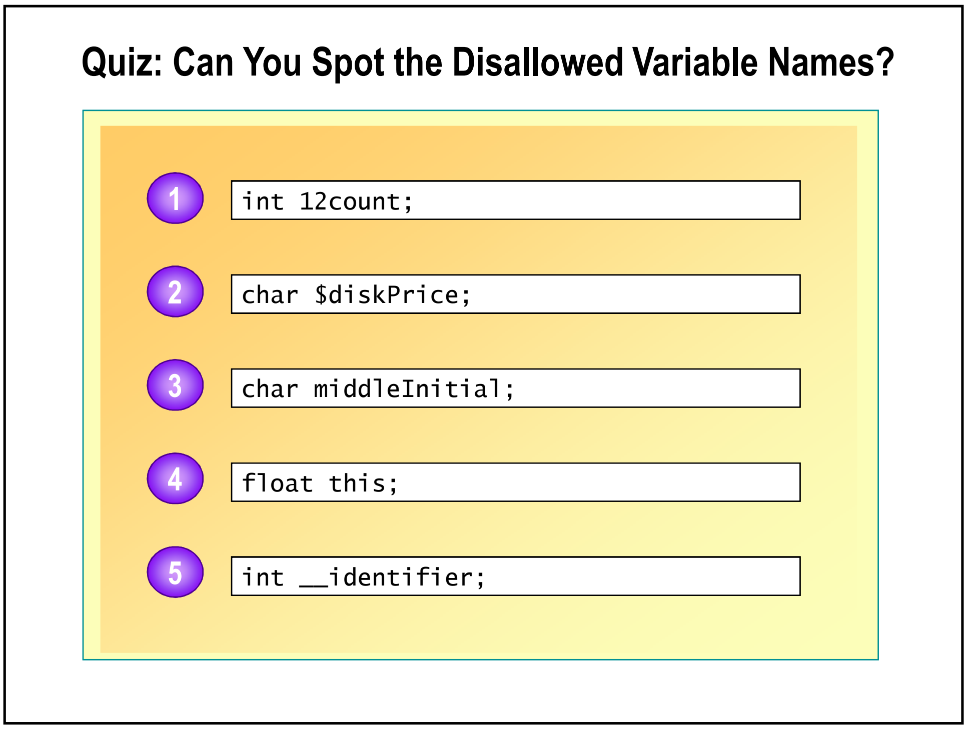
C# Keywords

- **Keywords are reserved identifiers**

```
abstract, base, bool, default, if, finally
```

- **Do not use keywords as variable names**
  - Results in a compile-time error
- **Avoid using keywords by changing their case sensitivity**

```
int INT;   // Poor style
```



Quiz: Can You Spot the Disallowed Variable Names?

1
```
int 12count;
```

2
```
char $diskPrice;
```

3
```
char middleInitial;
```

4
```
float this;
```

5
```
int __identifier;
```

◆ Using Built-in Data Types

- Declaring Local Variables
- Assigning Values to Variables
- Compound Assignment
- Common Operators
- Increment and Decrement
- Operator Precedence

Declaring Local Variables

- Usually declared by data type and variable name:

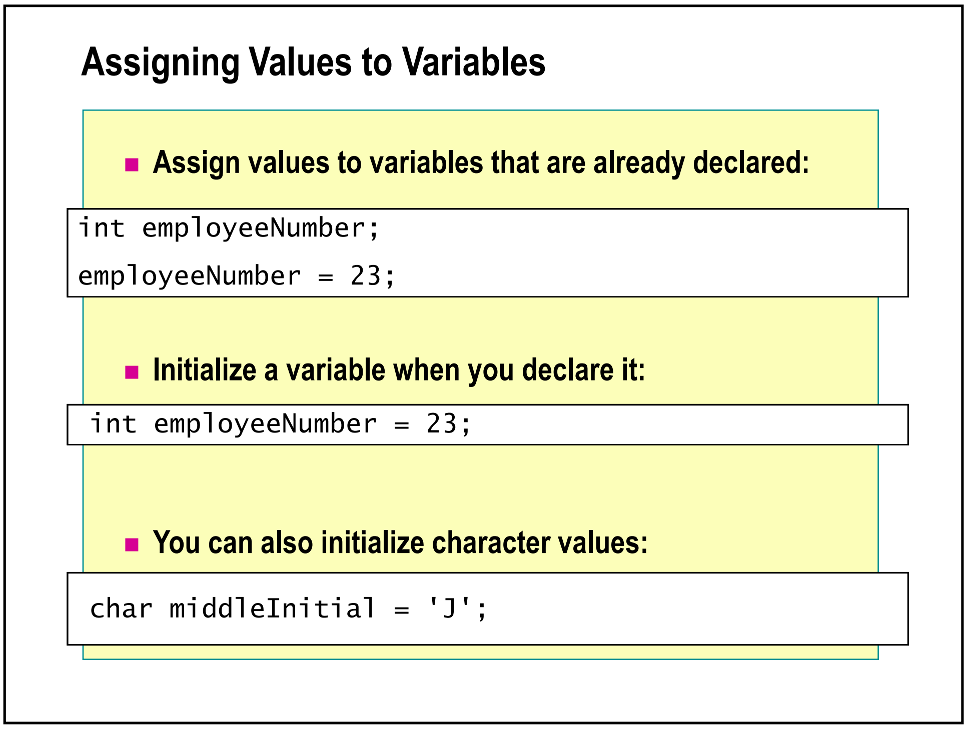
```
int itemCount;
```

- Possible to declare multiple variables in one declaration:

```
int itemCount, employeeNumber;
```

--or--

```
int itemCount,
    employeeNumber;
```



Assigning Values to Variables

- Assign values to variables that are already declared:

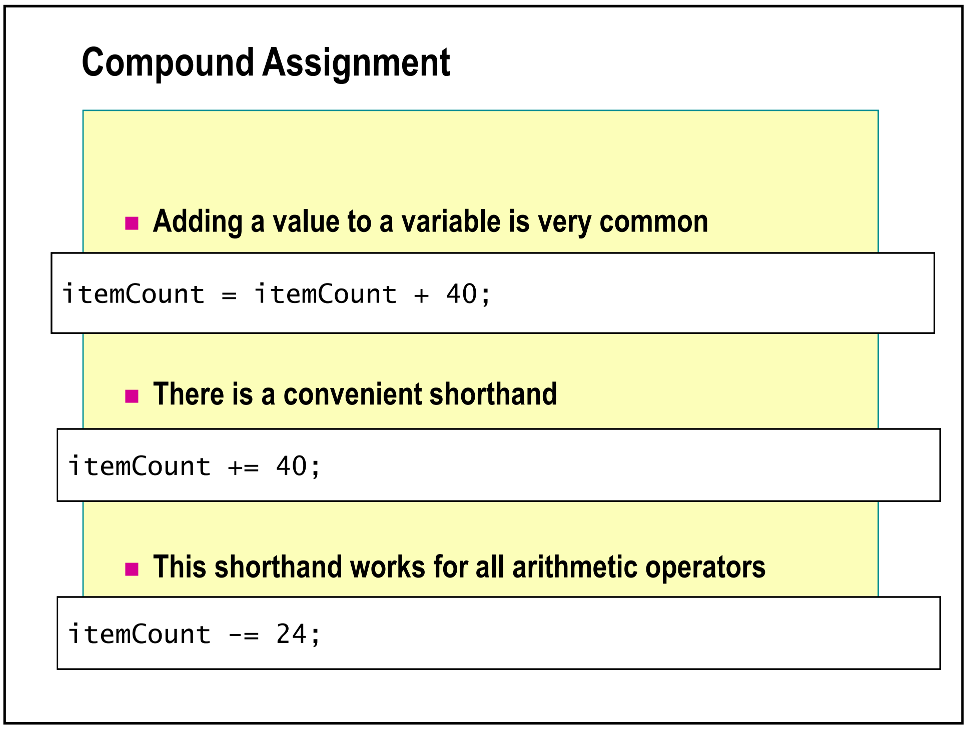
```
int employeeNumber;
employeeNumber = 23;
```

- Initialize a variable when you declare it:

```
int employeeNumber = 23;
```

- You can also initialize character values:

```
char middleInitial = 'J';
```



Compound Assignment

- Adding a value to a variable is very common

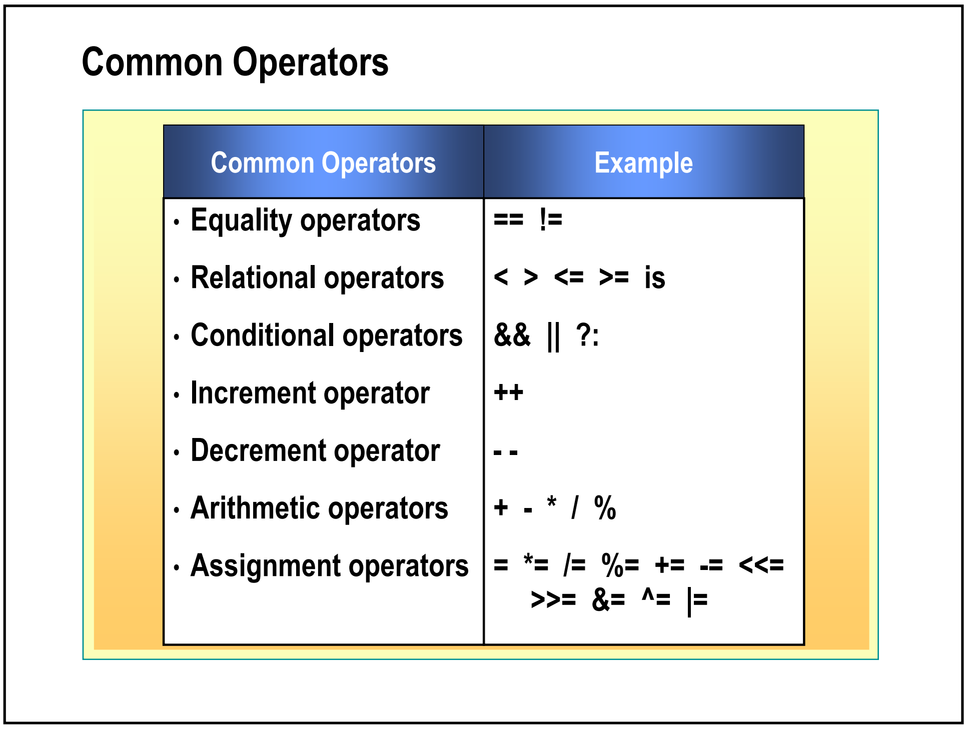
```
itemCount = itemCount + 40;
```

- There is a convenient shorthand

```
itemCount += 40;
```

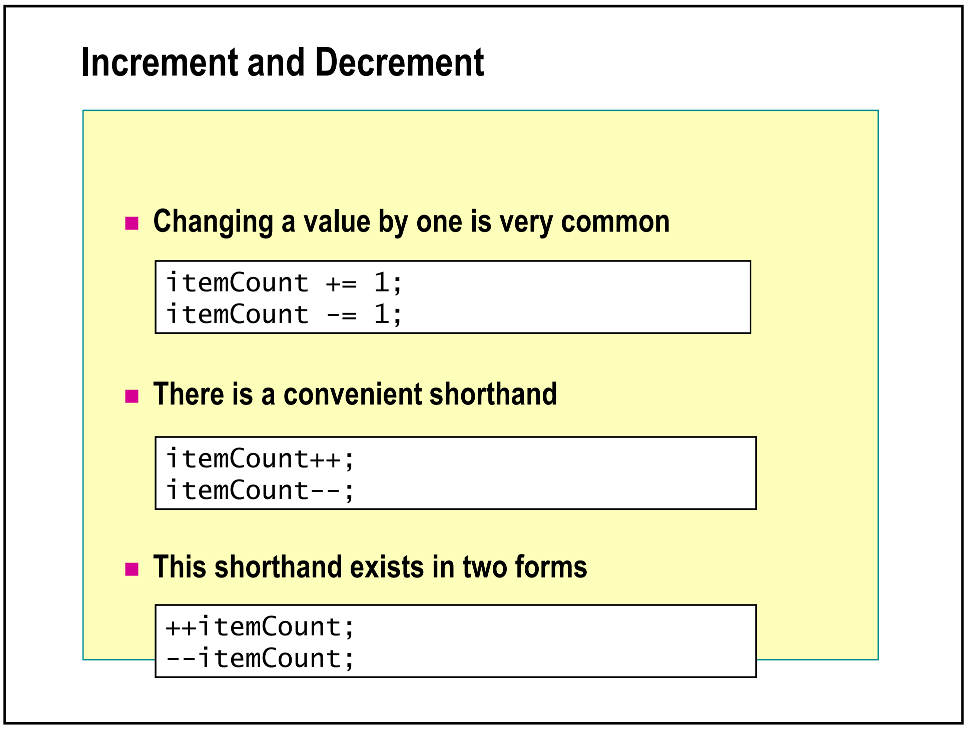
- This shorthand works for all arithmetic operators

```
itemCount -= 24;
```



Common Operators

Common Operators	Example
• Equality operators	== !=
• Relational operators	< > <= >= is
• Conditional operators	&& || ?:
• Increment operator	++
• Decrement operator	- -
• Arithmetic operators	+ - * / %
• Assignment operators	= *= /= %= += -= <<= >>= &= ^= |=



Increment and Decrement

- Changing a value by one is very common

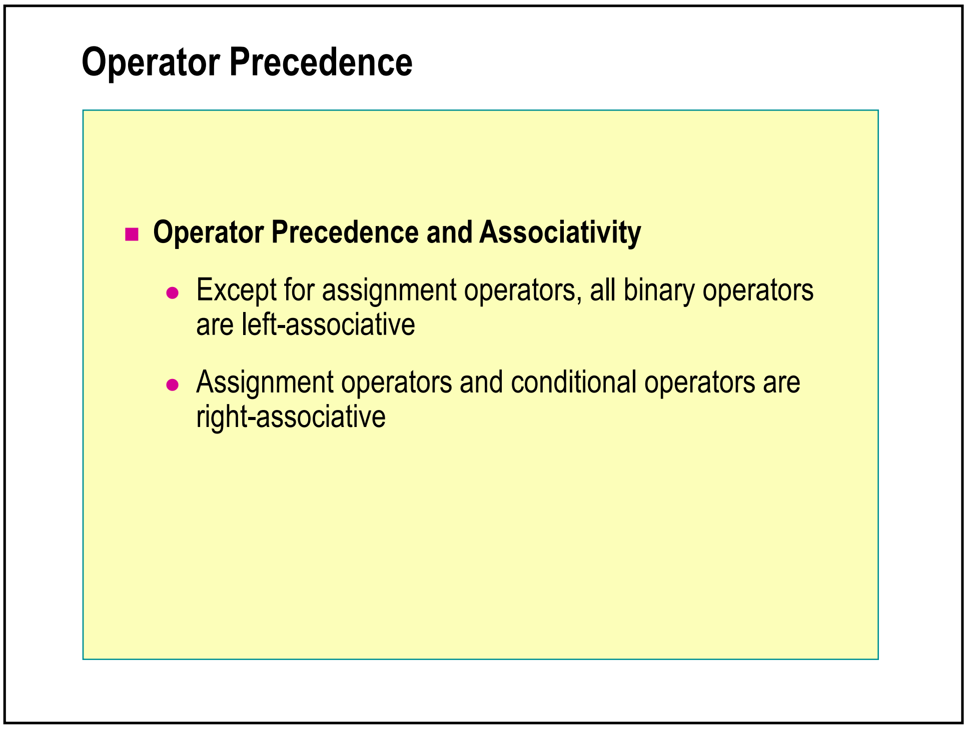
```
itemCount += 1;
itemCount -= 1;
```

- There is a convenient shorthand

```
itemCount++;
itemCount--;
```

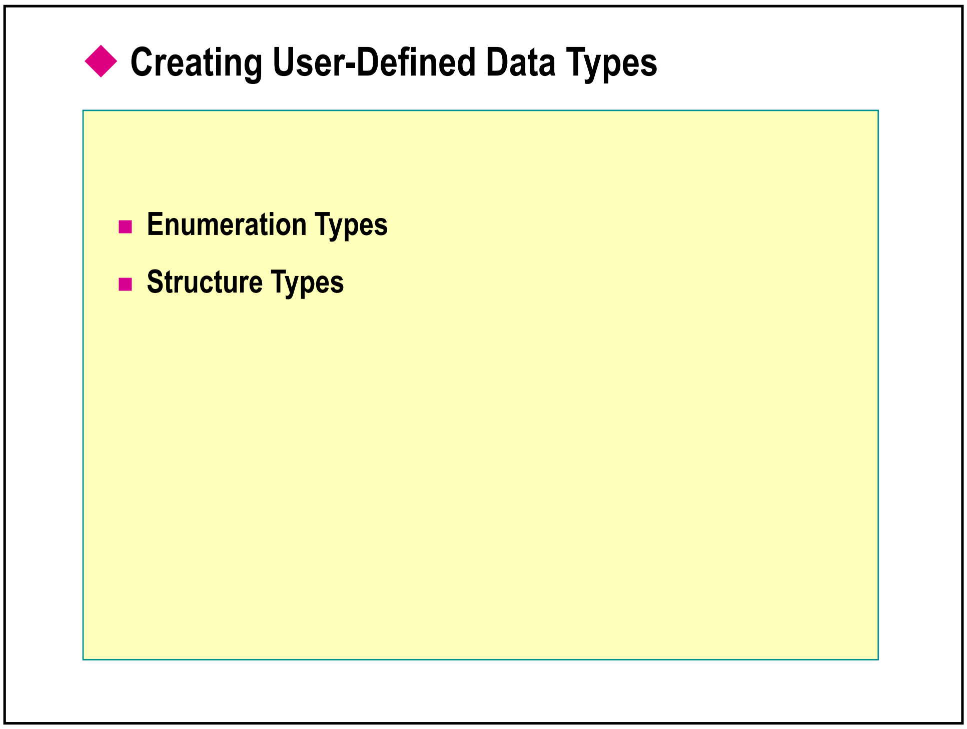
- This shorthand exists in two forms

```
++itemCount;
--itemCount;
```



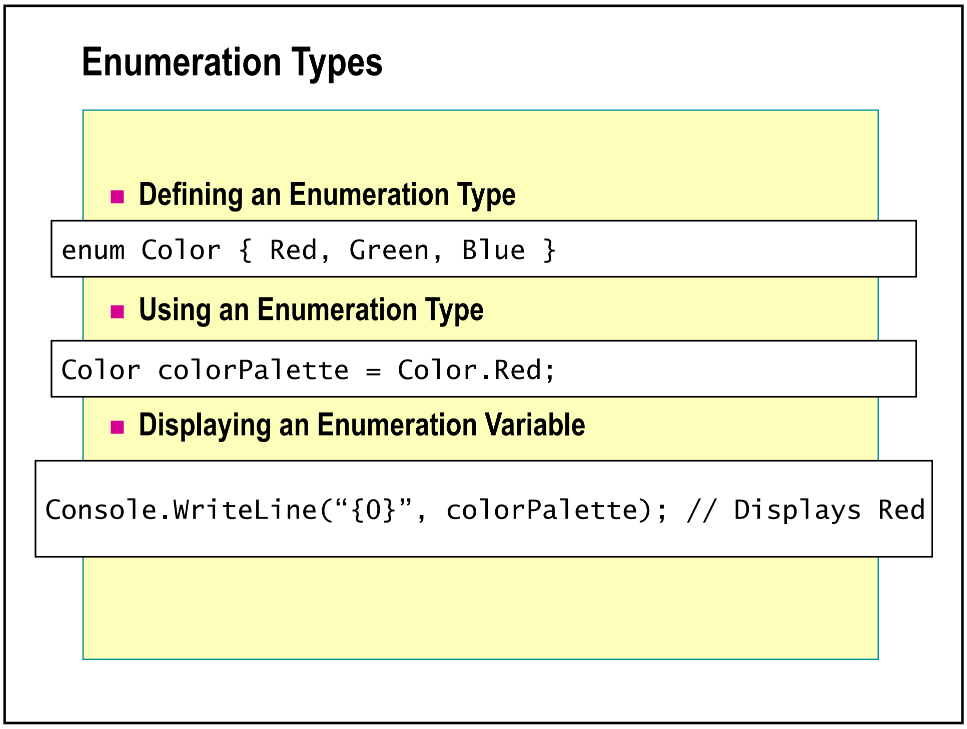
Operator Precedence

- **Operator Precedence and Associativity**
  - Except for assignment operators, all binary operators are left-associative
  - Assignment operators and conditional operators are right-associative



◆ Creating User-Defined Data Types

- **Enumeration Types**
- **Structure Types**



Enumeration Types

- Defining an Enumeration Type

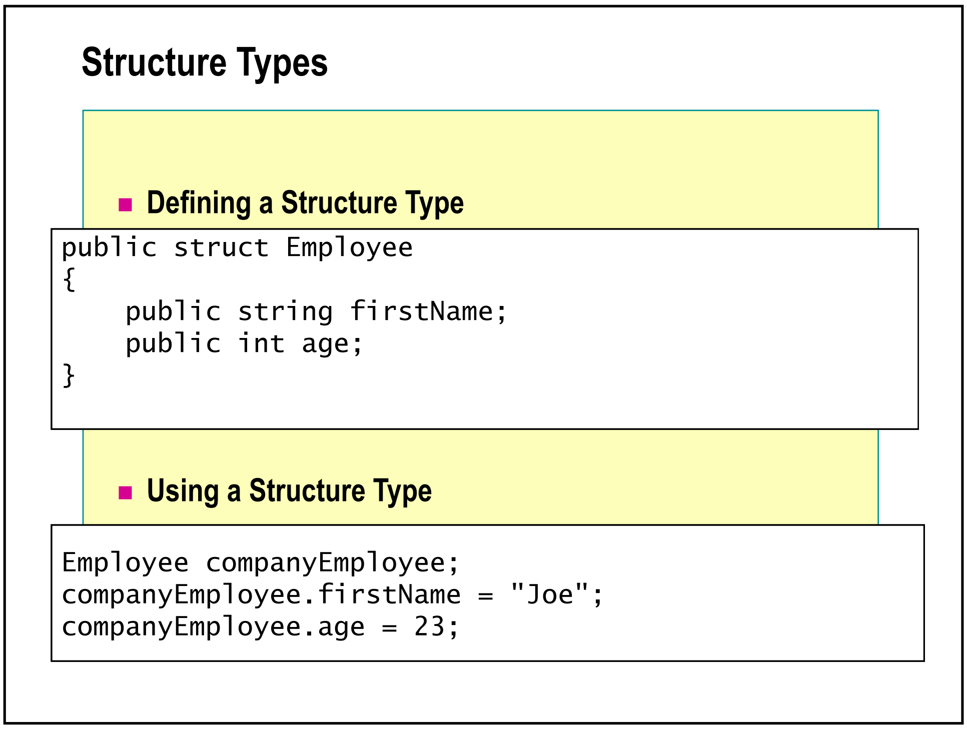
```
enum Color { Red, Green, Blue }
```

- Using an Enumeration Type

```
Color colorPalette = Color.Red;
```

- Displaying an Enumeration Variable

```
Console.WriteLine(“{0}”, colorPalette); // Displays Red
```



Structure Types

- Defining a Structure Type

```
public struct Employee
{
    public string firstName;
    public int age;
}
```

- Using a Structure Type

```
Employee companyEmployee;
companyEmployee.firstName = "Joe";
companyEmployee.age = 23;
```

◆ Converting Data Types

- Implicit Data Type Conversion
- Explicit Data Type Conversion

Implicit Data Type Conversion

- To Convert int to long:

```
using System;
class Test
{
    static void Main( )
    {
         int intValue = 123;
         long longValue = intValue;
         Console.WriteLine("(long) {0} = {1}", intValue,
    ➥longValue);
    }
}
```

- Implicit conversions cannot fail
  - May lose precision, but not magnitude

Explicit Data Type Conversion

- To do explicit conversions, use a cast expression:

```
using System;
class Test
{
   static void Main( )
   {
       long longValue = Int64.MaxValue;
       int intValue = (int) longValue;
       Console.WriteLine("(int) {0} = {1}", longValue,
   ➥intValue);
   }
}
```

Review

- Common Type System
- Naming Variables
- Using Built-in Data Types
- Creating User-Defined Data Types
- Converting Data Types

Module 4: Statements and Exceptions

Overview

- Introduction to Statements
- Using Selection Statements
- Using Iteration Statements
- Using Jump Statements
- Handling Basic Exceptions
- Raising Exceptions

◆ Introduction to Statements

- Statement Blocks
- Types of Statements

Statement Blocks

- Use braces As block delimiters

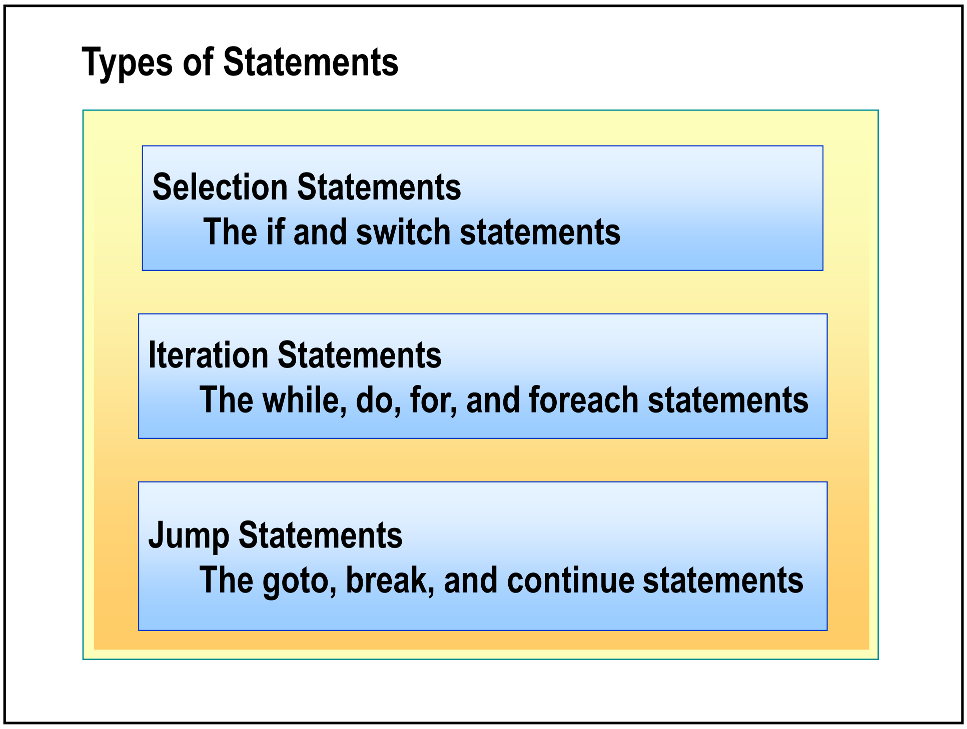
```
{
  // code
}
```

- A block and its parent block cannot have a variable with the same name

```
{
  int i;
  ...
  {
    int i;
    ...
  }
}
```

- Sibling blocks can have variables with the same name

```
{
  int i;
  ...
}
...
{
  int i;
  ...
}
```

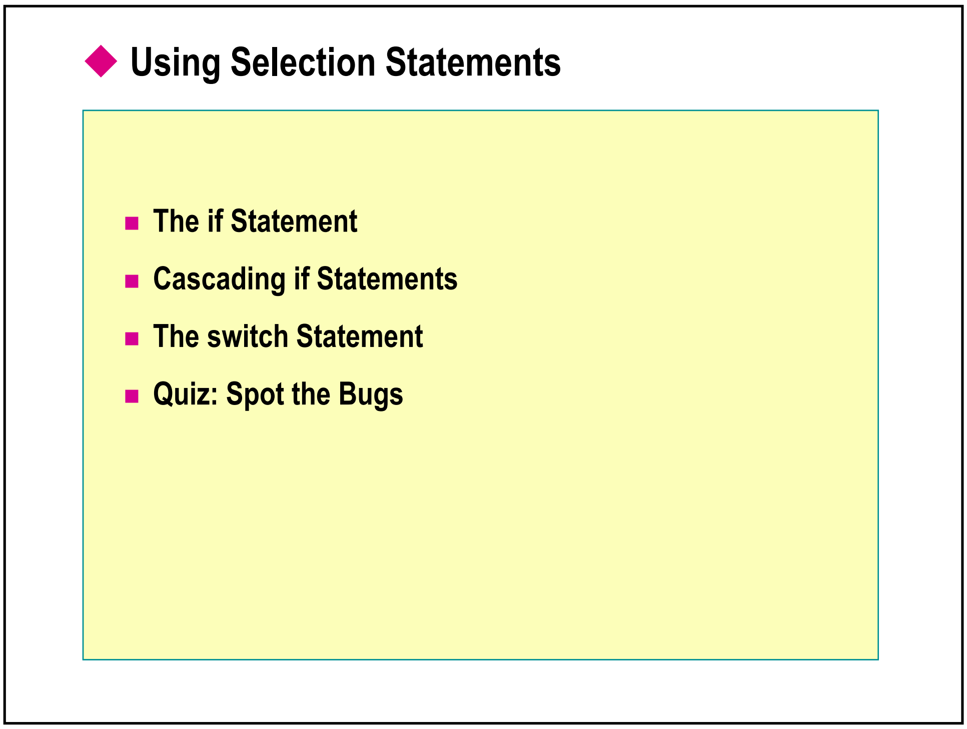


Types of Statements

Selection Statements
The if and switch statements

Iteration Statements
The while, do, for, and foreach statements

Jump Statements
The goto, break, and continue statements



◆ Using Selection Statements

- The if Statement
- Cascading if Statements
- The switch Statement
- Quiz: Spot the Bugs

The if Statement

- Syntax:

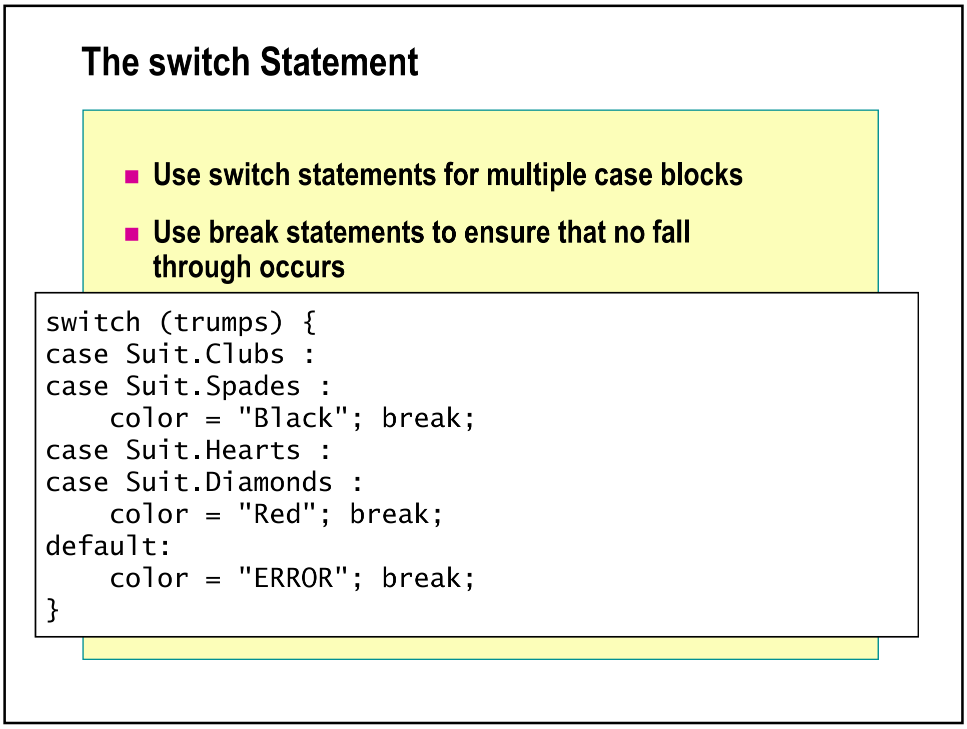
```
if ( Boolean-expression )
  first-embedded-statement
else
  second-embedded-statement
```

- No implicit conversion from int to bool

```
int x;
...
if (x) ...     // Must be if (x != 0) in C#
if (x = 0) ... // Must be if (x == 0) in C#
```

Cascading if Statements

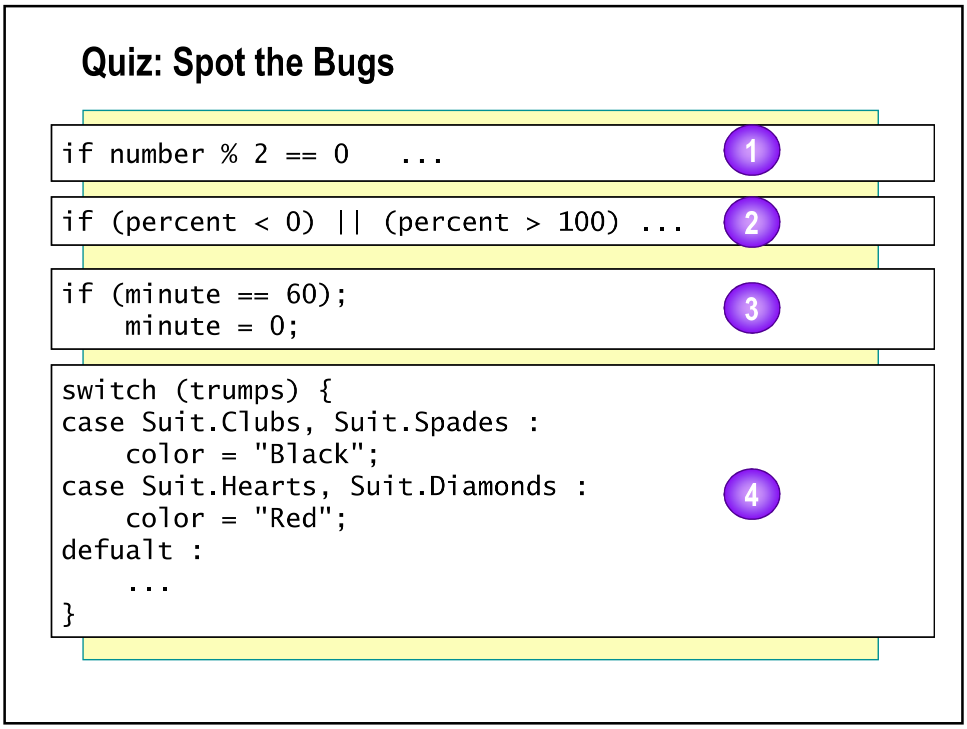
```
enum Suit { Clubs, Hearts, Diamonds, Spades }
Suit trumps = Suit.Hearts;
if (trumps == Suit.Clubs)
    color = "Black";
else if (trumps == Suit.Hearts)
    color = "Red";
else if (trumps == Suit.Diamonds)
    color = "Red";
else
    color = "Black";
```



The switch Statement

- Use switch statements for multiple case blocks
- Use break statements to ensure that no fall through occurs

```
switch (trumps) {
case Suit.Clubs :
case Suit.Spades :
    color = "Black"; break;
case Suit.Hearts :
case Suit.Diamonds :
    color = "Red"; break;
default:
    color = "ERROR"; break;
}
```



Quiz: Spot the Bugs

1

```
if number % 2 == 0   ...
```

2

```
if (percent < 0) || (percent > 100) ...
```

3

```
if (minute == 60);
    minute = 0;
```

4

```
switch (trumps) {
case Suit.Clubs, Suit.Spades :
    color = "Black";
case Suit.Hearts, Suit.Diamonds :
    color = "Red";
defualt :
    ...
}
```

◆ Using Iteration Statements

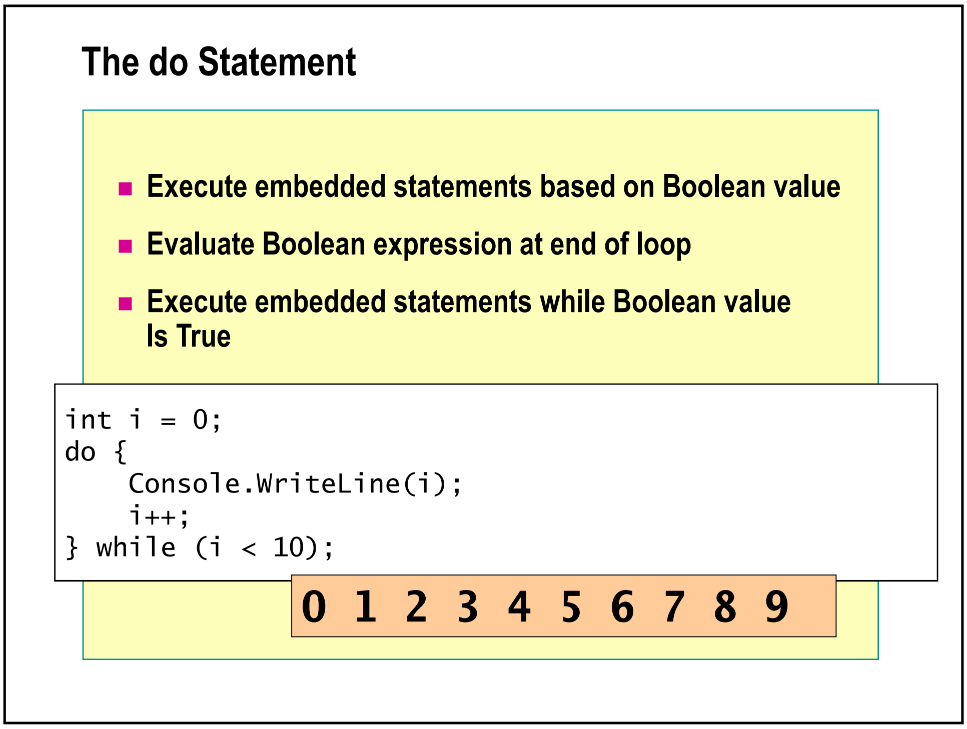
- The while Statement
- The do Statement
- The for Statement
- The foreach Statement
- Quiz: Spot the Bugs

The while Statement

- Execute embedded statements based on Boolean value
- Evaluate Boolean expression at beginning of loop
- Execute embedded statements while Boolean value Is True

```
int i = 0;
while (i < 10) {
    Console.WriteLine(i);
    i++;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

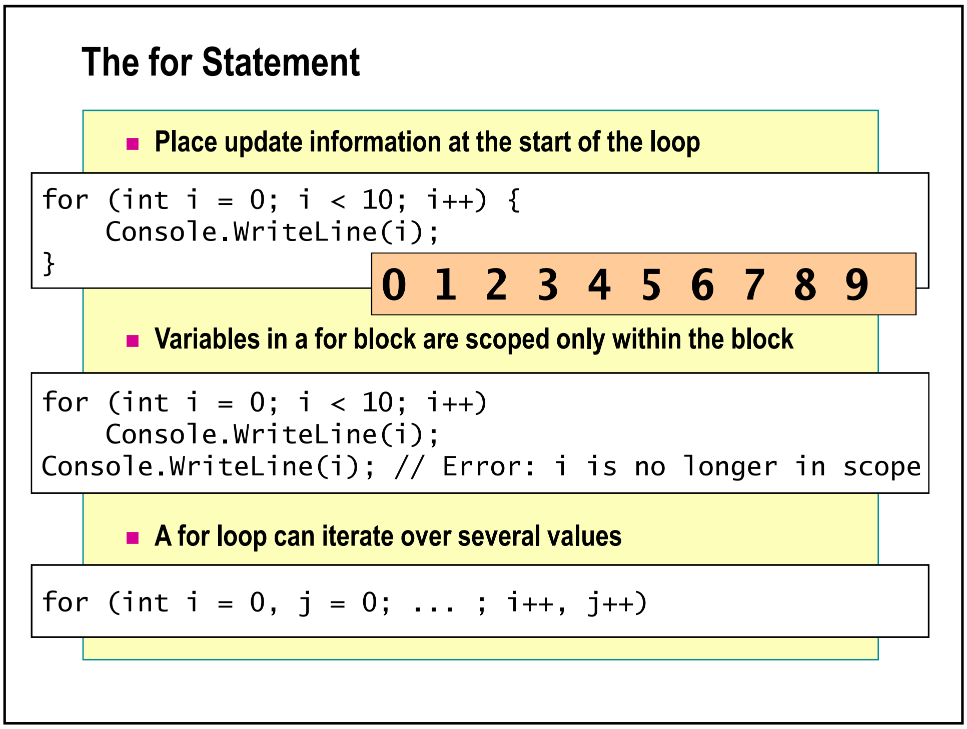


The do Statement

- Execute embedded statements based on Boolean value
- Evaluate Boolean expression at end of loop
- Execute embedded statements while Boolean value Is True

```
int i = 0;
do {
    Console.WriteLine(i);
    i++;
} while (i < 10);
```

```
0 1 2 3 4 5 6 7 8 9
```



The for Statement

- Place update information at the start of the loop

```
for (int i = 0; i < 10; i++) {
    Console.WriteLine(i);
}
```

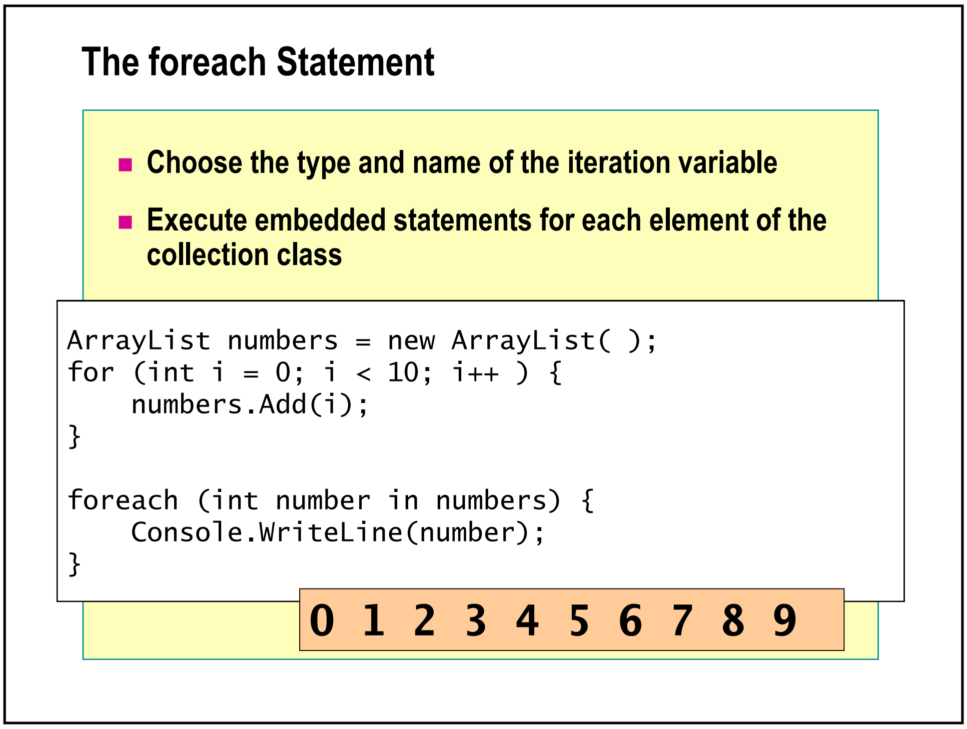
```
0 1 2 3 4 5 6 7 8 9
```

- Variables in a for block are scoped only within the block

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(i);
Console.WriteLine(i); // Error: i is no longer in scope
```

- A for loop can iterate over several values

```
for (int i = 0, j = 0; ... ; i++, j++)
```

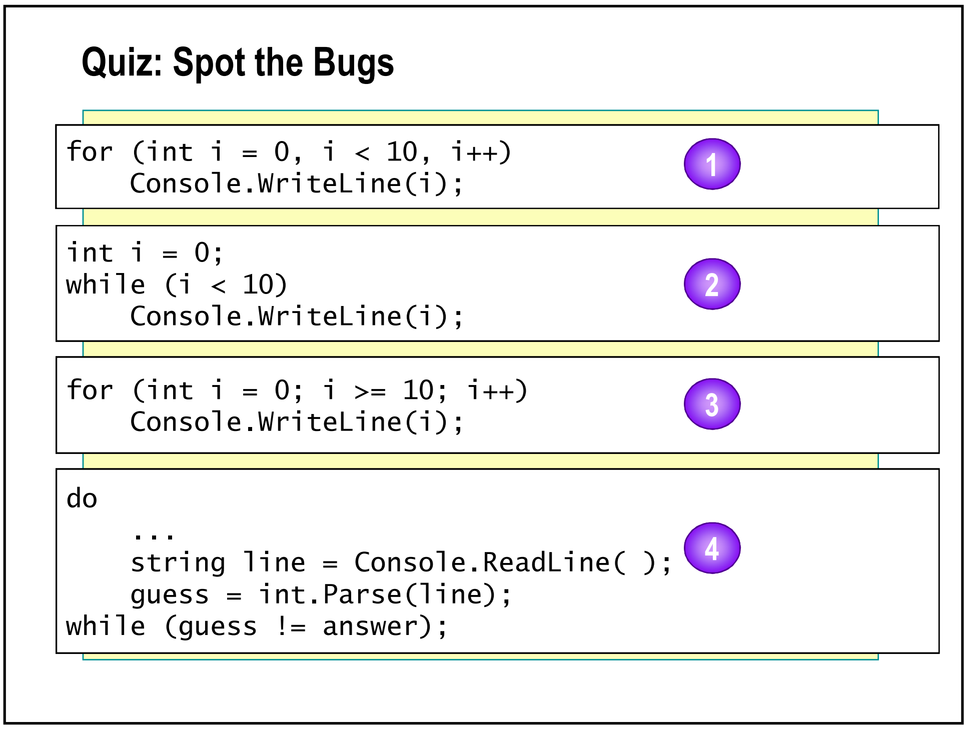


The foreach Statement

- Choose the type and name of the iteration variable
- Execute embedded statements for each element of the collection class

```
ArrayList numbers = new ArrayList( );
for (int i = 0; i < 10; i++ ) {
    numbers.Add(i);
}

foreach (int number in numbers) {
    Console.WriteLine(number);
}
```

```
0 1 2 3 4 5 6 7 8 9
```



Quiz: Spot the Bugs

```
for (int i = 0, i < 10, i++)
    Console.WriteLine(i);
```

1

```
int i = 0;
while (i < 10)
    Console.WriteLine(i);
```

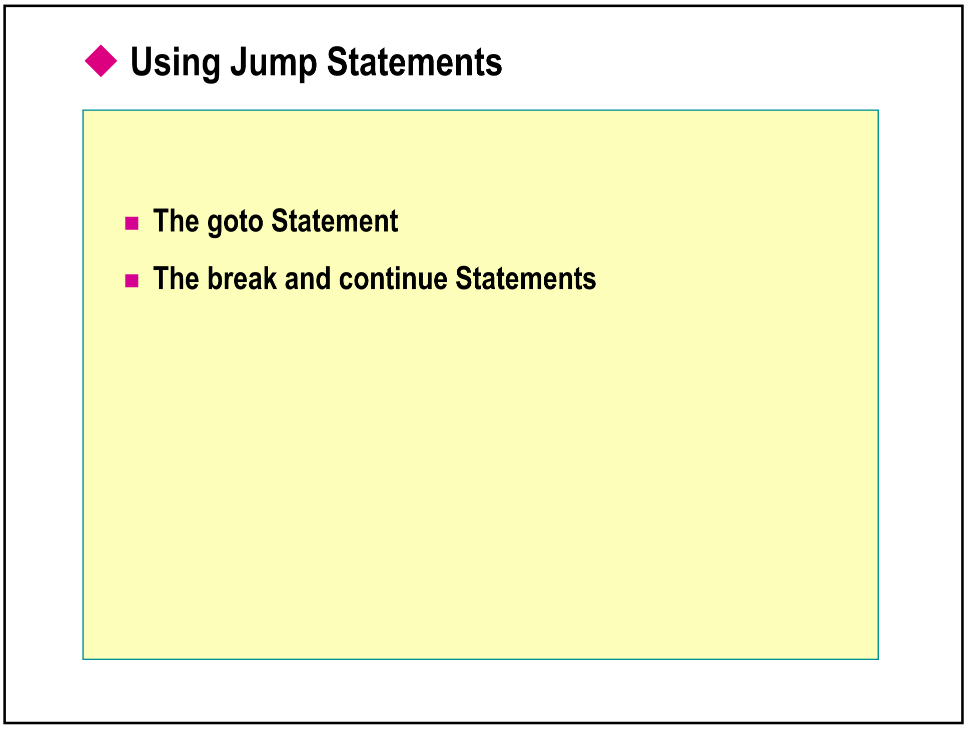
2

```
for (int i = 0; i >= 10; i++)
    Console.WriteLine(i);
```

3

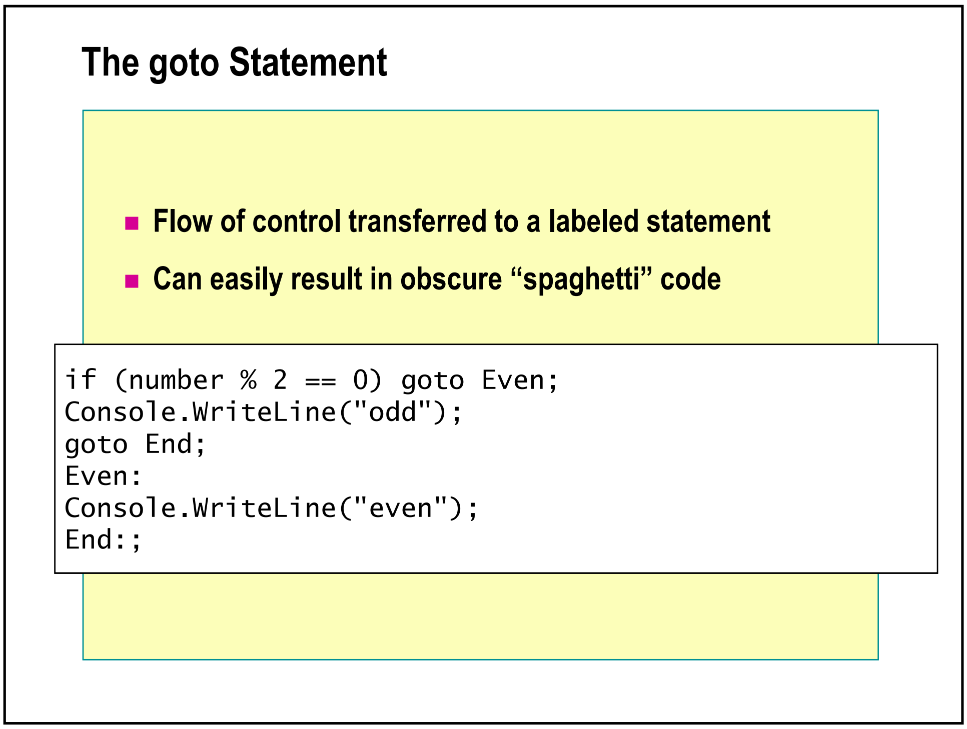
```
do
    ...
    string line = Console.ReadLine( );
    guess = int.Parse(line);
while (guess != answer);
```

4



◆ Using Jump Statements

- The goto Statement
- The break and continue Statements



The goto Statement

- Flow of control transferred to a labeled statement
- Can easily result in obscure “spaghetti” code

```
if (number % 2 == 0) goto Even;
Console.WriteLine("odd");
goto End;
Even:
Console.WriteLine("even");
End:;
```

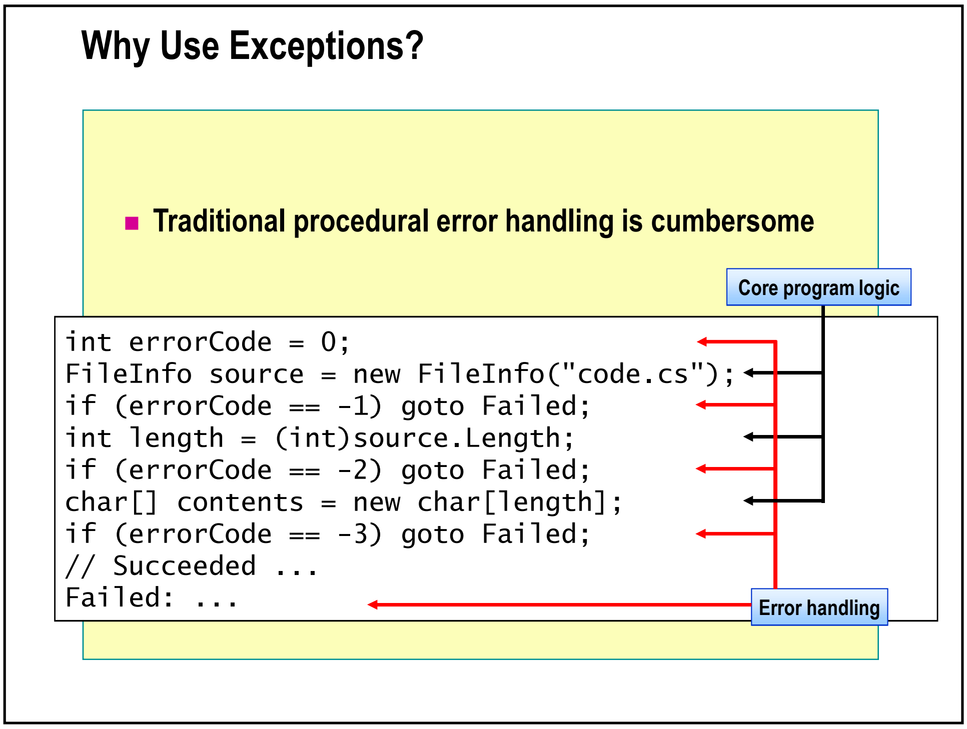
The break and continue Statements

- The break statement jumps out of an iteration
- The continue statement jumps to the next iteration

```
int i = 0;
while (true) {
    Console.WriteLine(i);
    i++;
    if (i < 10)
        continue;
    else
        break;
}
```

◆ Handling Basic Exceptions

- Why Use Exceptions?
- Exception Objects
- Using try and catch Blocks
- Multiple catch Blocks



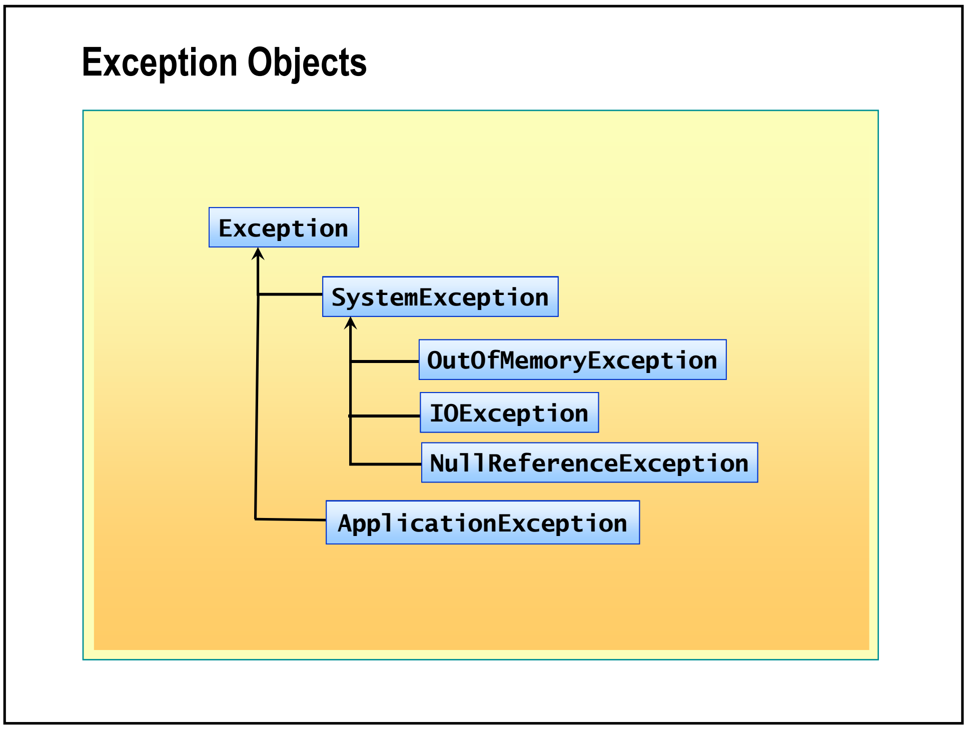
Why Use Exceptions?

- Traditional procedural error handling is cumbersome

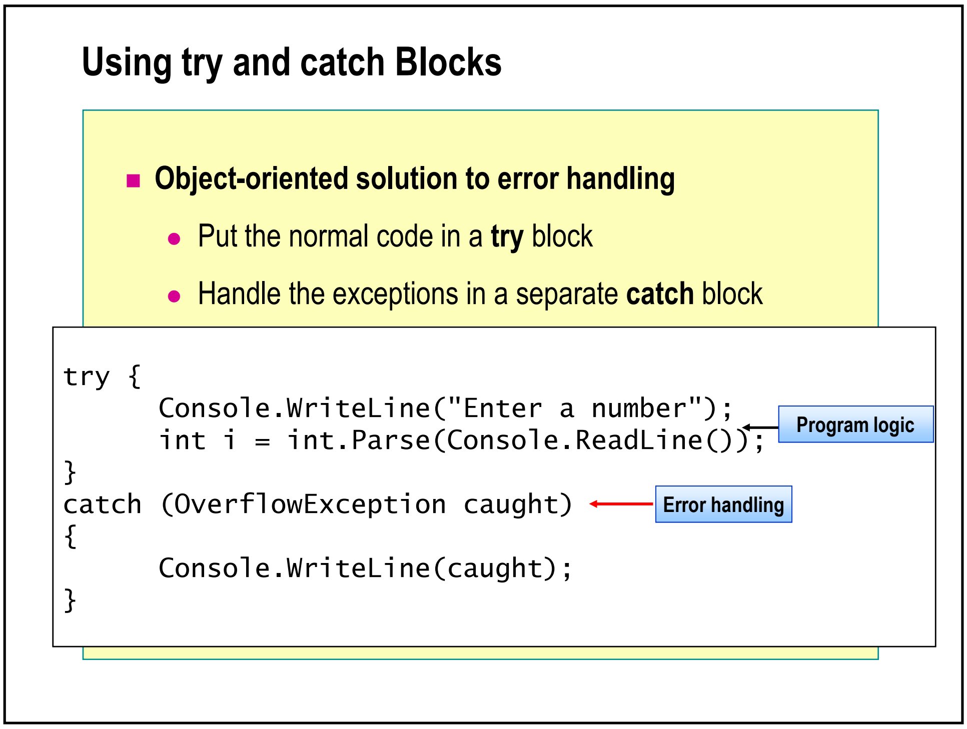
Core program logic

```
int errorCode = 0;
FileInfo source = new FileInfo("code.cs");
if (errorCode == -1) goto Failed;
int length = (int)source.Length;
if (errorCode == -2) goto Failed;
char[] contents = new char[length];
if (errorCode == -3) goto Failed;
// Succeeded ...
Failed: ...
```

Error handling



Exception Objects



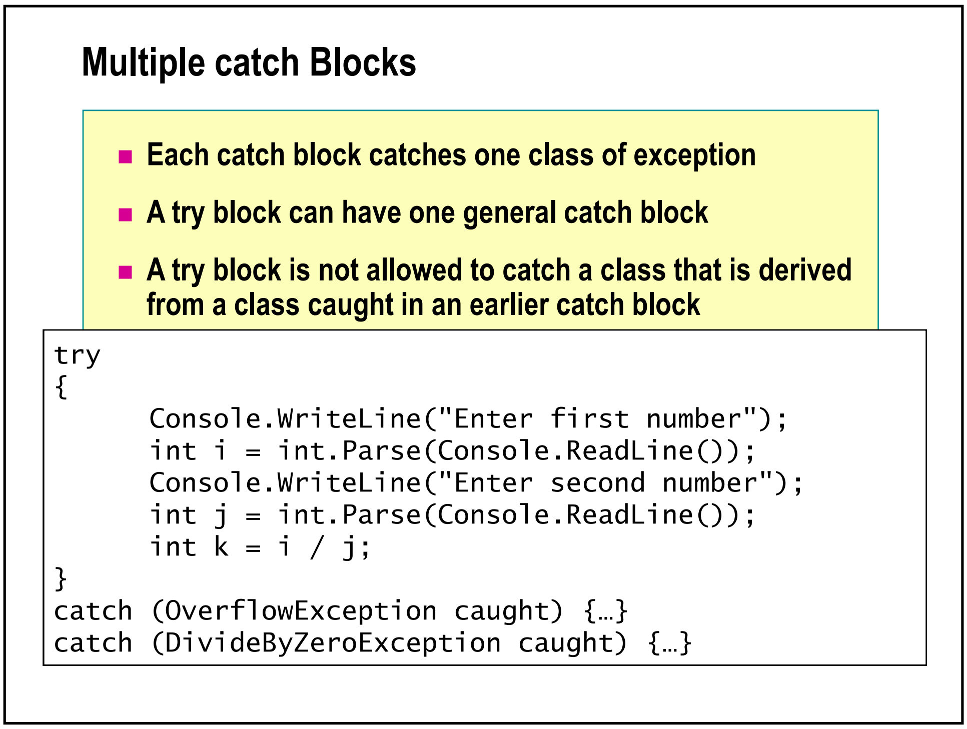
Using try and catch Blocks

- **Object-oriented solution to error handling**
  - Put the normal code in a **try** block
  - Handle the exceptions in a separate **catch** block

```
try {
      Console.WriteLine("Enter a number");
      int i = int.Parse(Console.ReadLine());
}
catch (OverflowException caught)
{
      Console.WriteLine(caught);
}
```

Program logic

Error handling



Multiple catch Blocks

- **Each catch block catches one class of exception**
- **A try block can have one general catch block**
- **A try block is not allowed to catch a class that is derived from a class caught in an earlier catch block**

```
try
{
      Console.WriteLine("Enter first number");
      int i = int.Parse(Console.ReadLine());
      Console.WriteLine("Enter second number");
      int j = int.Parse(Console.ReadLine());
      int k = i / j;
}
catch (OverflowException caught) {…}
catch (DivideByZeroException caught) {…}
```

◆ Raising Exceptions

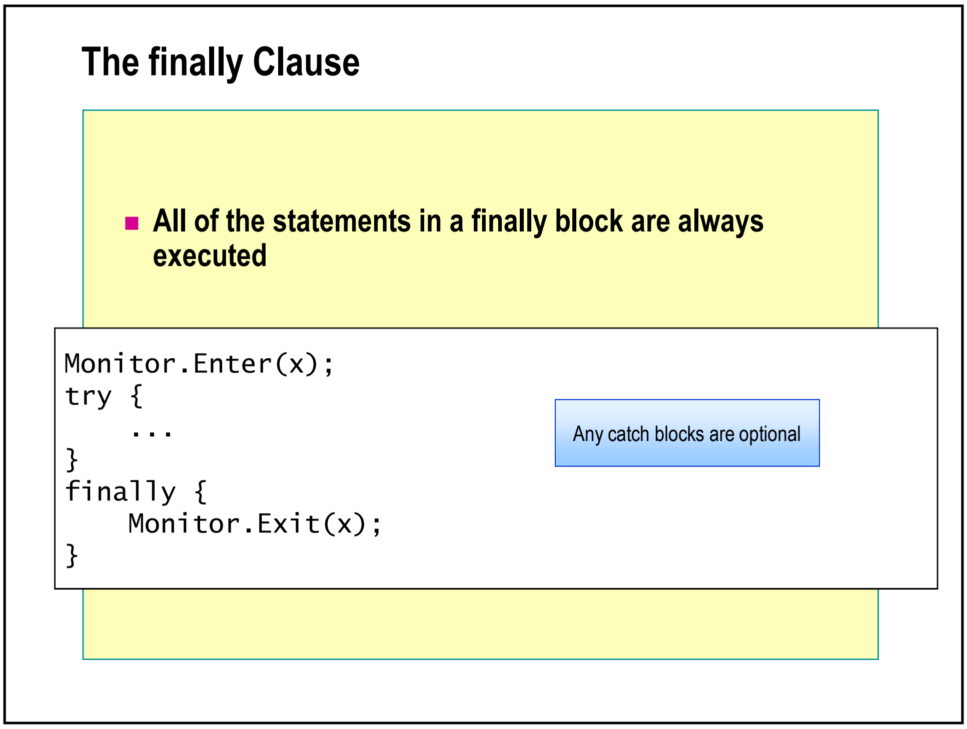
- The throw Statement
- The finally Clause
- Checking for Arithmetic Overflow
- Guidelines for Handling Exceptions

The throw Statement

- Throw an appropriate exception
- Give the exception a meaningful message

```
throw expression ;
```

```
if (minute < 1 || minute >= 60) {
  throw new InvalidTimeException(minute +
                                " is not a valid minute");
  // !! Not reached !!
}
```

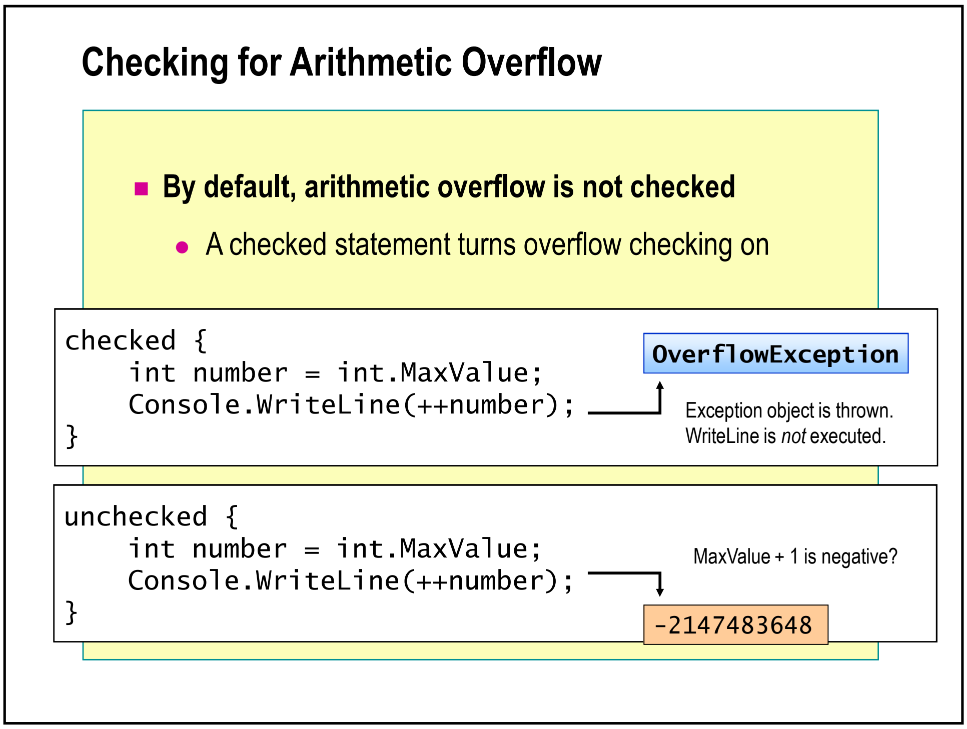


The finally Clause

- **All of the statements in a finally block are always executed**

```
Monitor.Enter(x);
try {
    ...
}
finally {
    Monitor.Exit(x);
}
```

Any catch blocks are optional



Checking for Arithmetic Overflow

- **By default, arithmetic overflow is not checked**
  - A checked statement turns overflow checking on

```
checked {
    int number = int.MaxValue;
    Console.WriteLine(++number);
}
```

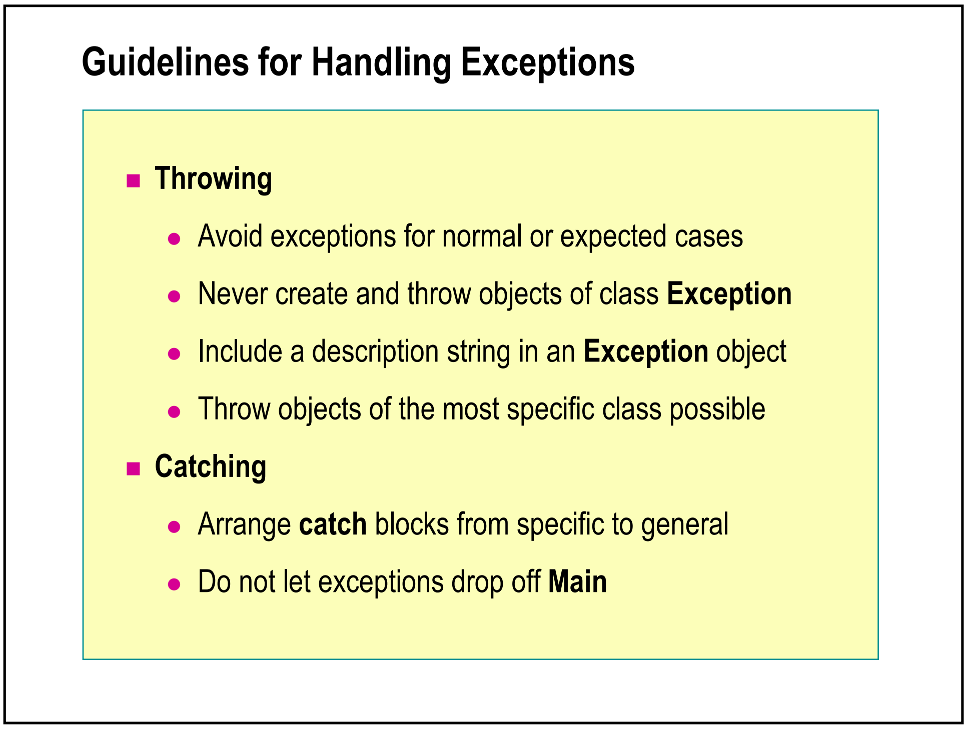
`OverflowException`

Exception object is thrown.
WriteLine is *not* executed.

```
unchecked {
    int number = int.MaxValue;
    Console.WriteLine(++number);
}
```

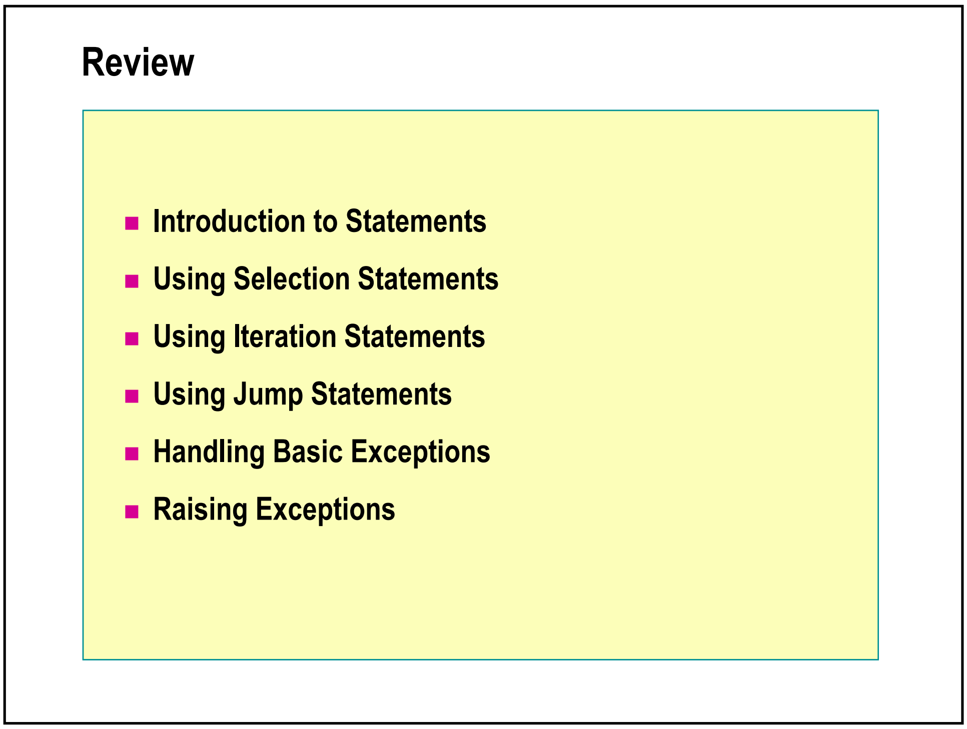
MaxValue + 1 is negative?

`-2147483648`



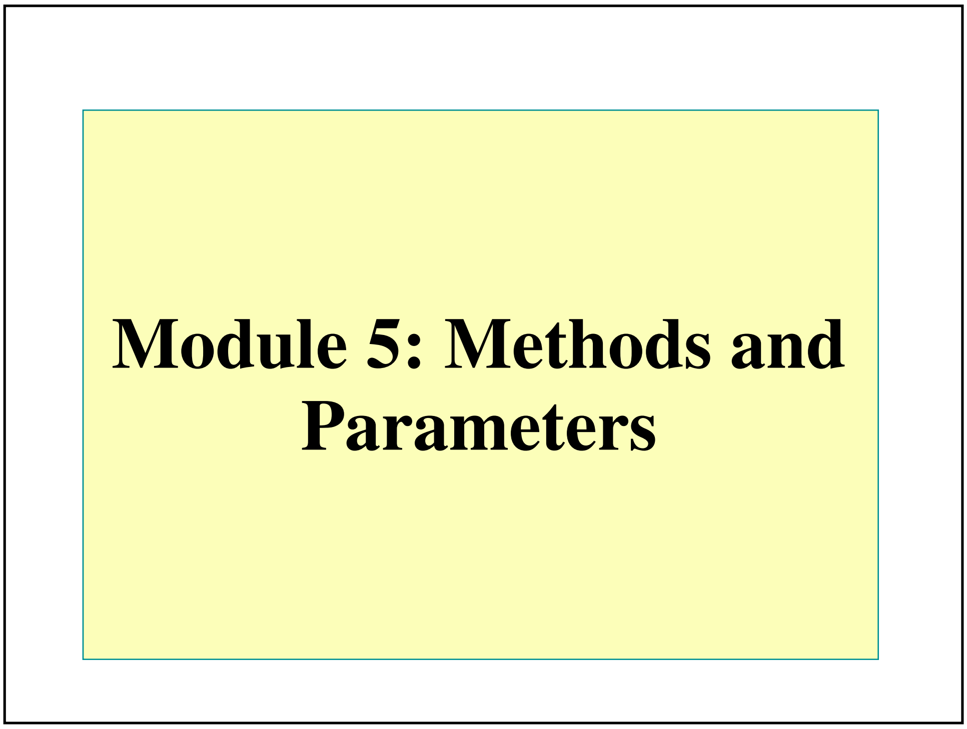
Guidelines for Handling Exceptions

- **Throwing**
  - Avoid exceptions for normal or expected cases
  - Never create and throw objects of class **Exception**
  - Include a description string in an **Exception** object
  - Throw objects of the most specific class possible
- **Catching**
  - Arrange **catch** blocks from specific to general
  - Do not let exceptions drop off **Main**

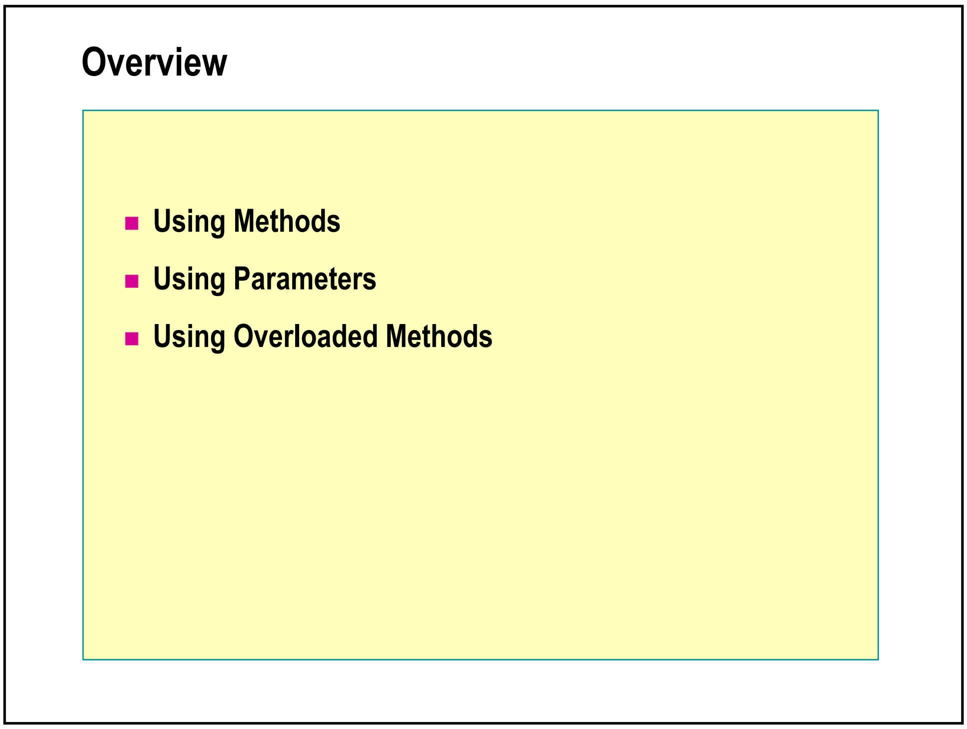


Review

- **Introduction to Statements**
- **Using Selection Statements**
- **Using Iteration Statements**
- **Using Jump Statements**
- **Handling Basic Exceptions**
- **Raising Exceptions**

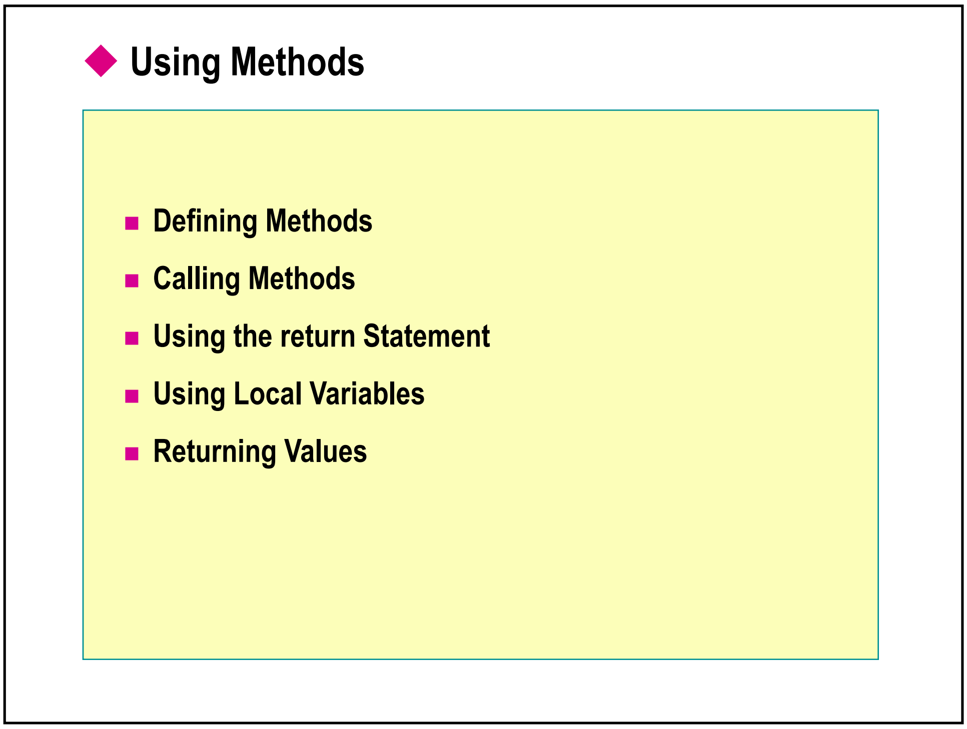


Module 5: Methods and Parameters



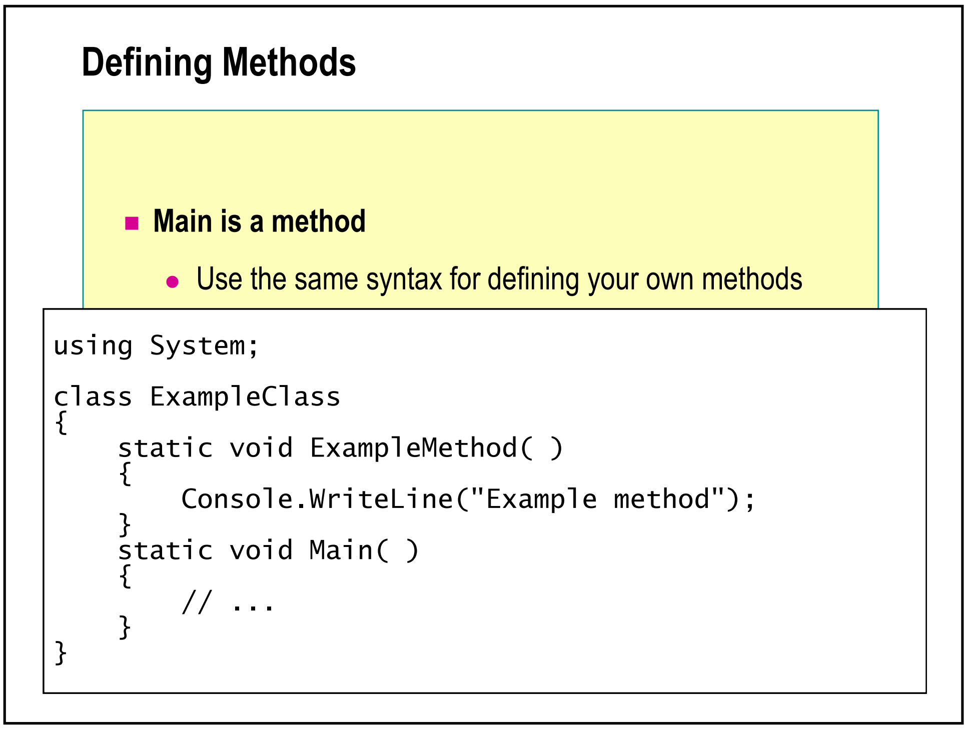
Overview

- Using Methods
- Using Parameters
- Using Overloaded Methods



◆ Using Methods

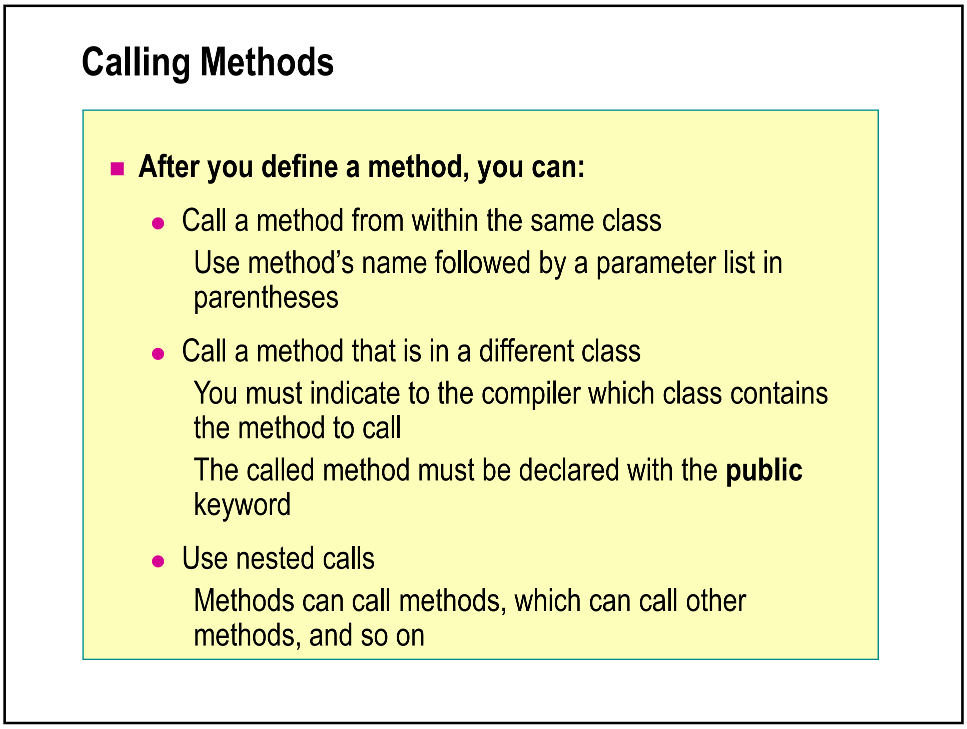
- Defining Methods
- Calling Methods
- Using the return Statement
- Using Local Variables
- Returning Values



Defining Methods

- Main is a method
  - Use the same syntax for defining your own methods

```
using System;
class ExampleClass
{
    static void ExampleMethod( )
    {
        Console.WriteLine("Example method");
    }
    static void Main( )
    {
        // ...
    }
}
```



Calling Methods

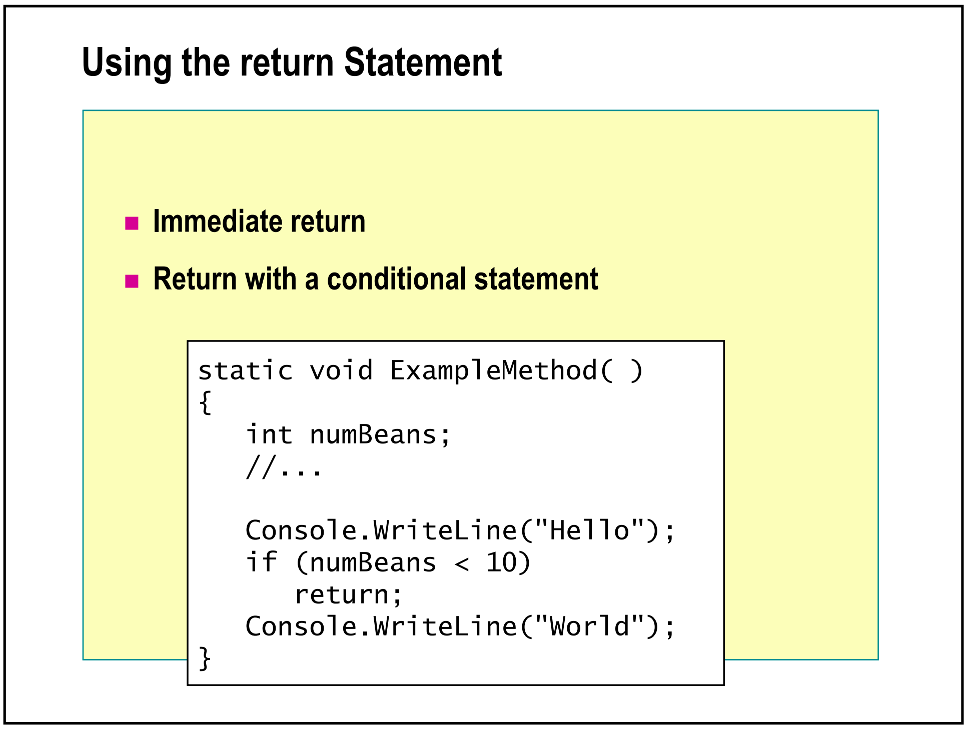
- **After you define a method, you can:**
  - Call a method from within the same class

    Use method’s name followed by a parameter list in parentheses
  - Call a method that is in a different class

    You must indicate to the compiler which class contains the method to call

    The called method must be declared with the **public** keyword
  - Use nested calls

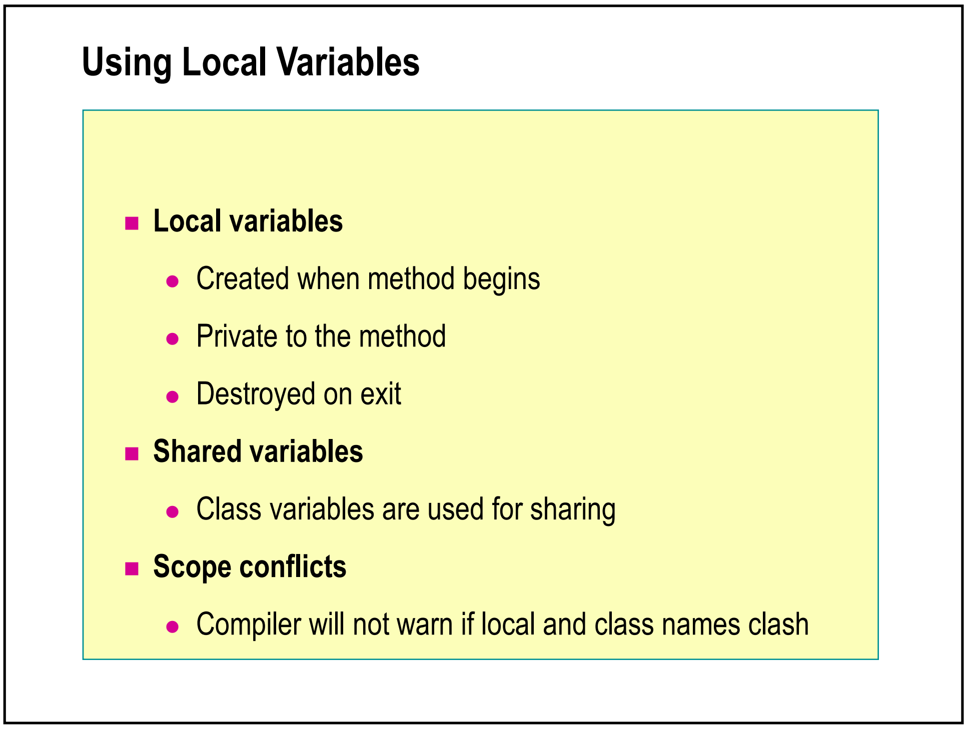
    Methods can call methods, which can call other methods, and so on



Using the return Statement

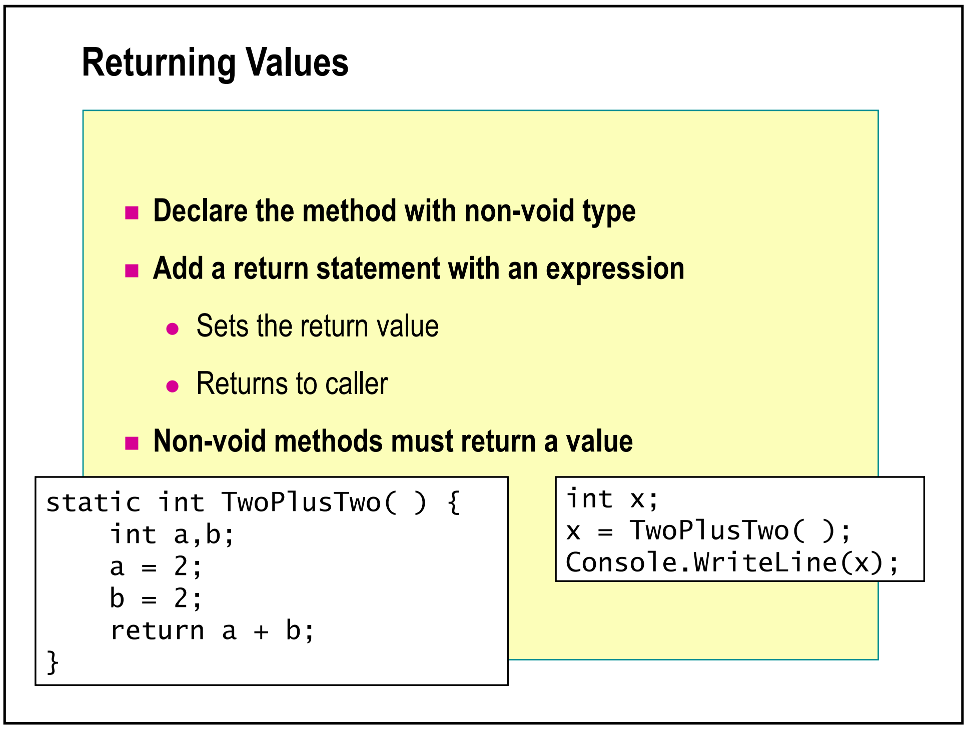
- **Immediate return**
- **Return with a conditional statement**

```
static void ExampleMethod( )
{
   int numBeans;
   //...

   Console.WriteLine("Hello");
   if (numBeans < 10)
      return;
   Console.WriteLine("World");
}
```



Using Local Variables

- **Local variables**
  - Created when method begins
  - Private to the method
  - Destroyed on exit
- **Shared variables**
  - Class variables are used for sharing
- **Scope conflicts**
  - Compiler will not warn if local and class names clash



Returning Values

- **Declare the method with non-void type**
- **Add a return statement with an expression**
  - Sets the return value
  - Returns to caller
- **Non-void methods must return a value**

```
static int TwoPlusTwo( ) {
    int a,b;
    a = 2;
    b = 2;
    return a + b;
}
```

```
int x;
x = TwoPlusTwo( );
Console.WriteLine(x);
```

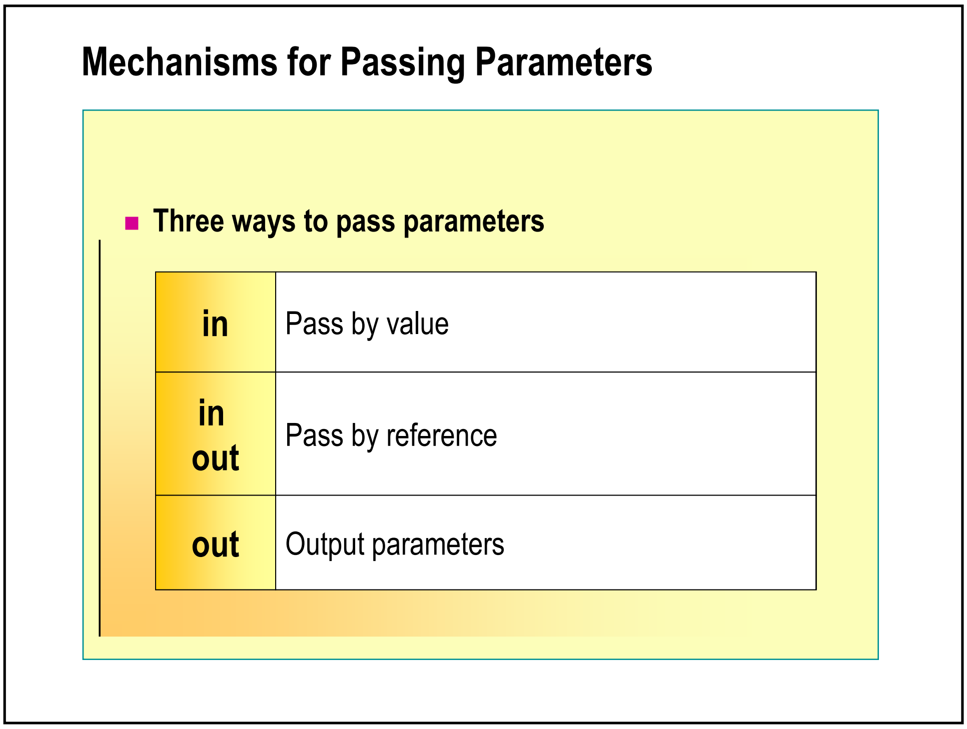
◆ Using Parameters

- **Declaring and Calling Parameters**
- **Mechanisms for Passing Parameters**
- **Pass by Value**
- **Pass by Reference**
- **Output Parameters**
- **Using Variable-Length Parameter Lists**
- **Guidelines for Passing Parameters**
- **Using Recursive Methods**

Declaring and Calling Parameters

- **Declaring parameters**
  - Place between parentheses after method name
  - Define type and name for each parameter
- **Calling methods with parameters**
  - Supply a value for each parameter

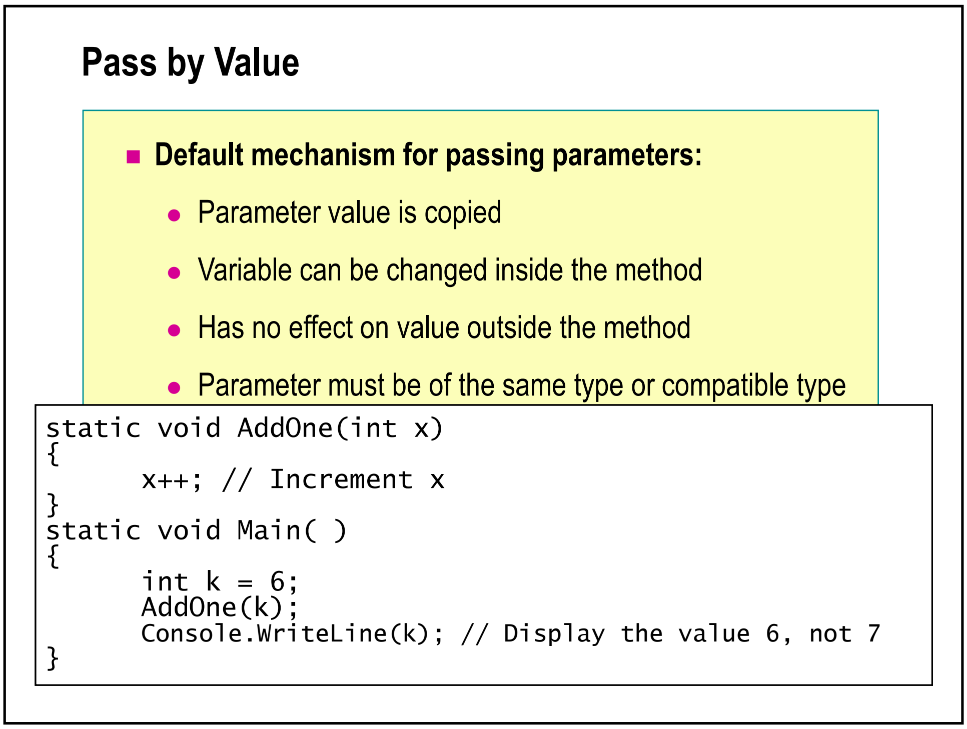
```
static void MethodWithParameters(int n, string y)
{ ... }

MethodWithParameters(2, "Hello, world");
```



Mechanisms for Passing Parameters

- Three ways to pass parameters

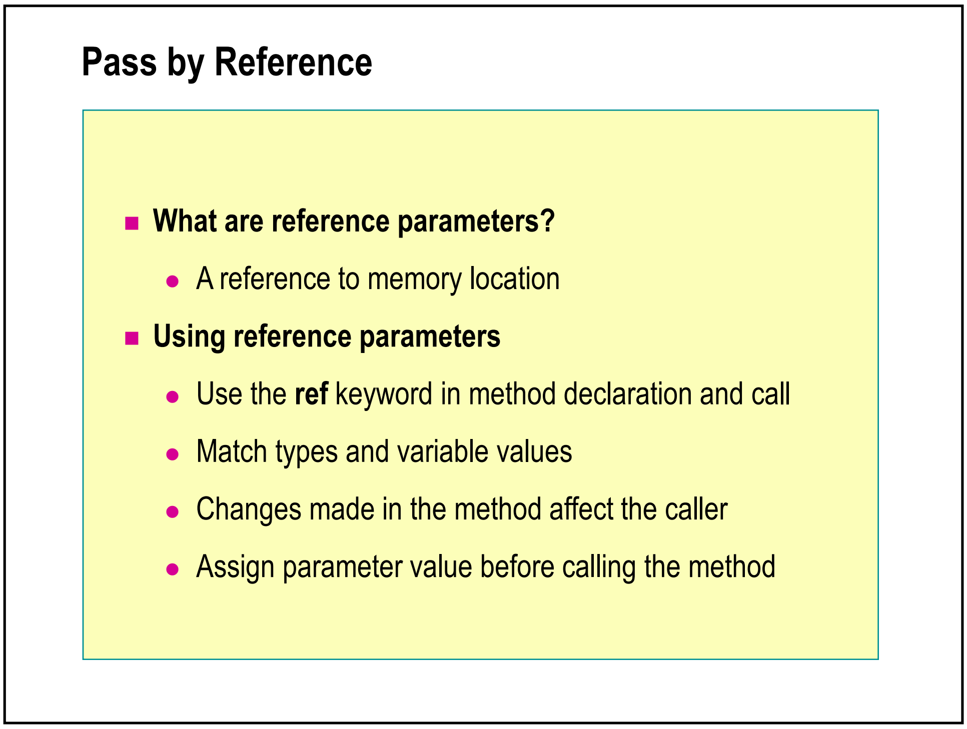
in	Pass by value
in out	Pass by reference
out	Output parameters



Pass by Value

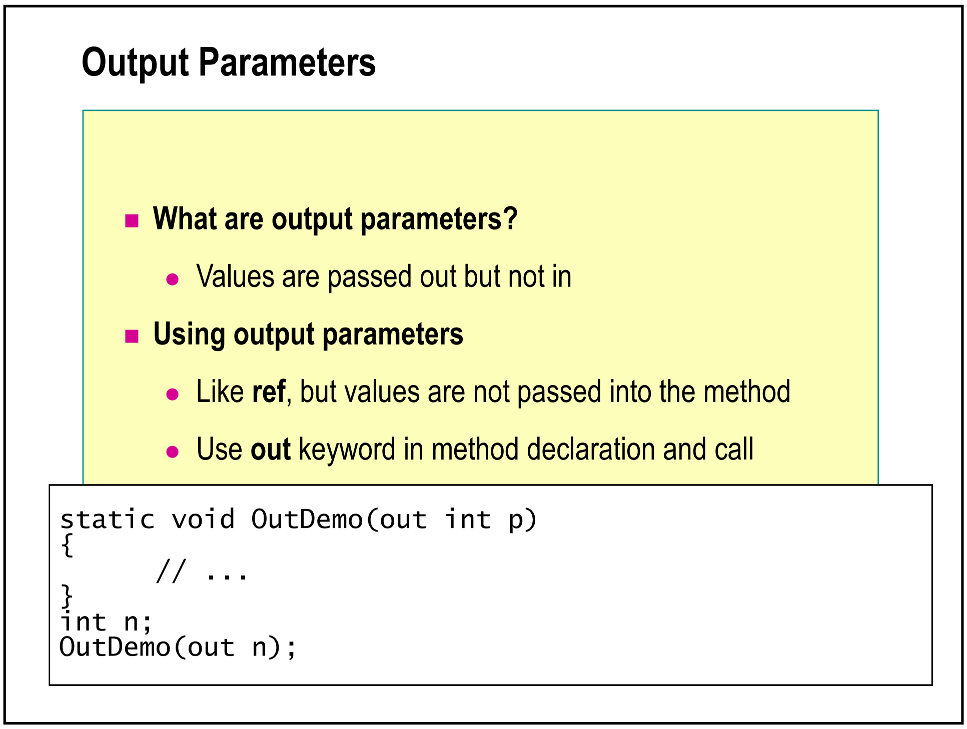
- **Default mechanism for passing parameters:**
  - Parameter value is copied
  - Variable can be changed inside the method
  - Has no effect on value outside the method
  - Parameter must be of the same type or compatible type

```
static void AddOne(int x)
{
      x++; // Increment x
}
static void Main( )
{
      int k = 6;
      AddOne(k);
      Console.WriteLine(k); // Display the value 6, not 7
}
```



Pass by Reference

- What are reference parameters?
  - A reference to memory location
- Using reference parameters
  - Use the **ref** keyword in method declaration and call
  - Match types and variable values
  - Changes made in the method affect the caller
  - Assign parameter value before calling the method



Output Parameters

- What are output parameters?
  - Values are passed out but not in
- Using output parameters
  - Like **ref**, but values are not passed into the method
  - Use **out** keyword in method declaration and call

```
static void OutDemo(out int p)
{
      // ...
}
int n;
OutDemo(out n);
```

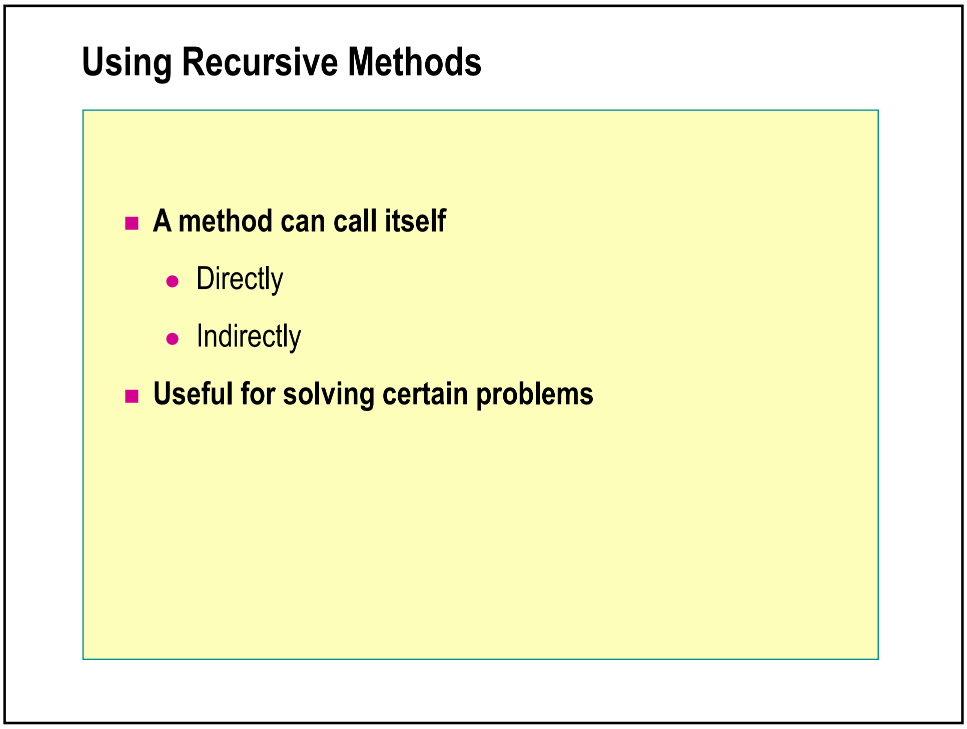
Using Variable-Length Parameter Lists

- **Use the params keyword**
- **Declare as an array at the end of the parameter list**
- **Always pass by value**

```
static long AddList(params long[ ] v)
{
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Main( )
{
    long x = AddList(63,21,84);
}
```

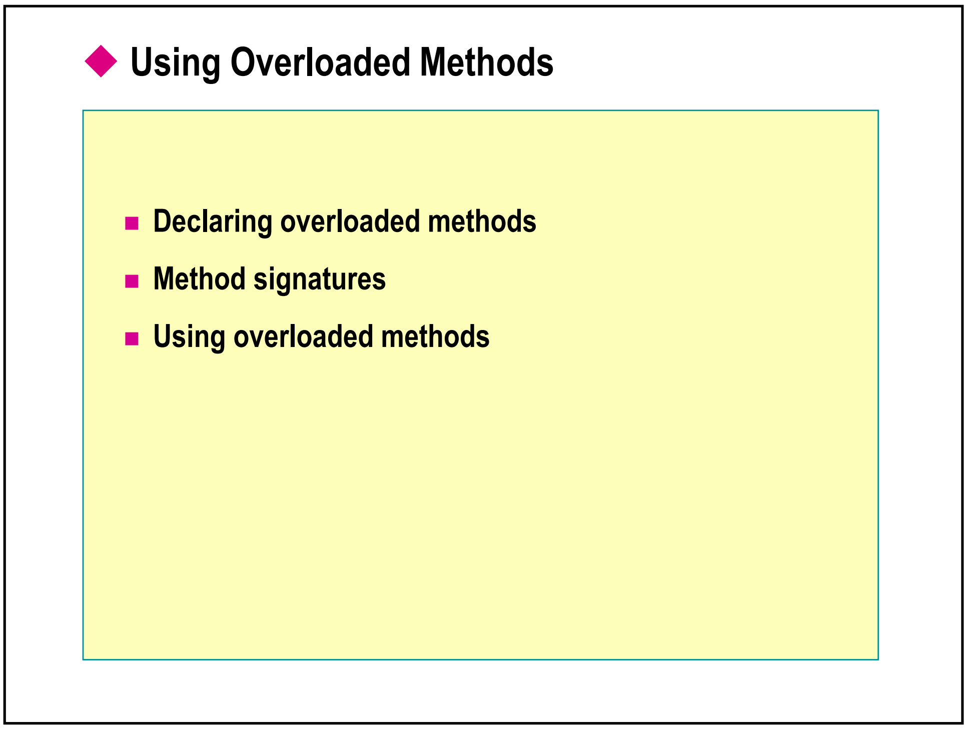
Guidelines for Passing Parameters

- **Mechanisms**
  - Pass by value is most common
  - Method return value is useful for single values
  - Use **ref** and/or **out** for multiple return values
  - Only use **ref** if data is transferred both ways
- **Efficiency**
  - Pass by value is generally the most efficient



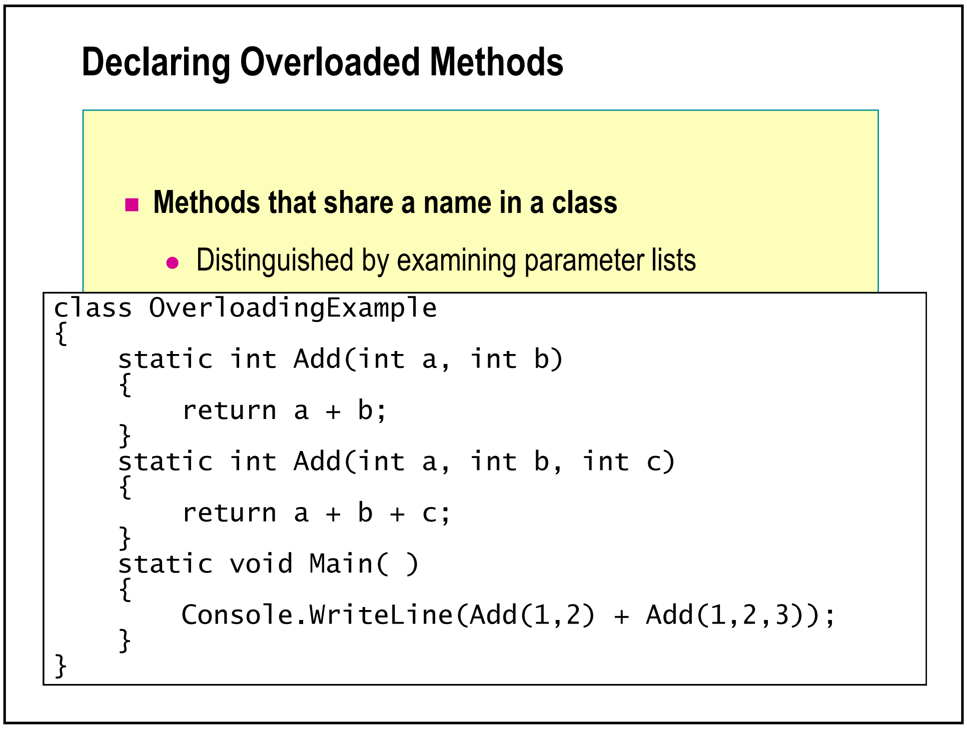
Using Recursive Methods

- A method can call itself
  - Directly
  - Indirectly
- Useful for solving certain problems



◆ Using Overloaded Methods

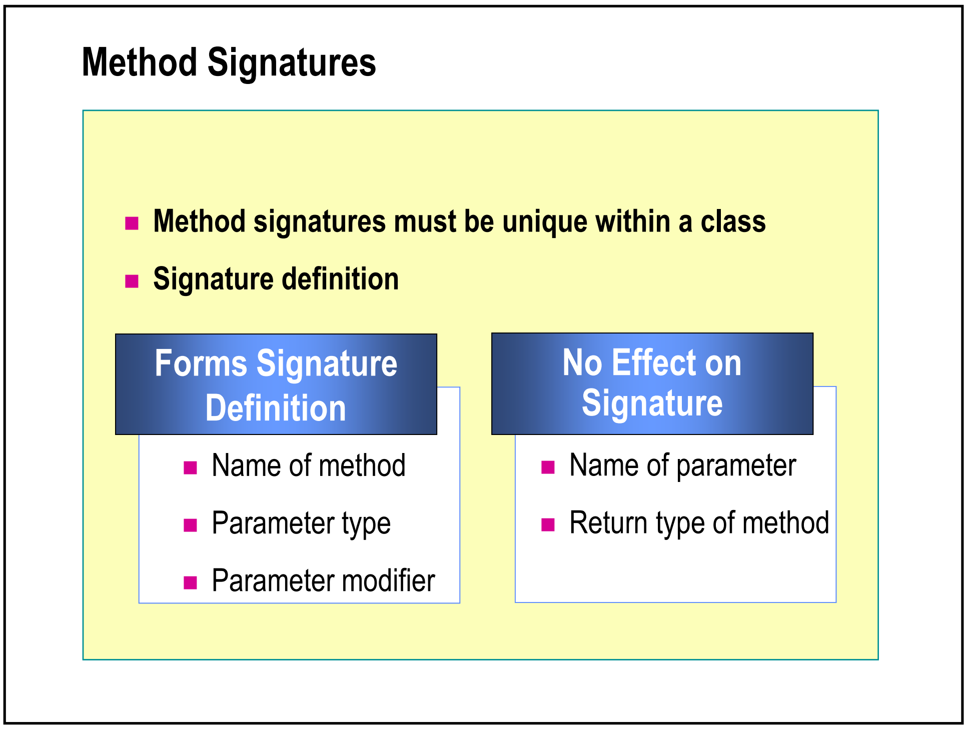
- Declaring overloaded methods
- Method signatures
- Using overloaded methods



Declaring Overloaded Methods

- **Methods that share a name in a class**
  - Distinguished by examining parameter lists

```
class OverloadingExample
{
    static int Add(int a, int b)
    {
        return a + b;
    }
    static int Add(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(Add(1,2) + Add(1,2,3));
    }
}
```



Method Signatures

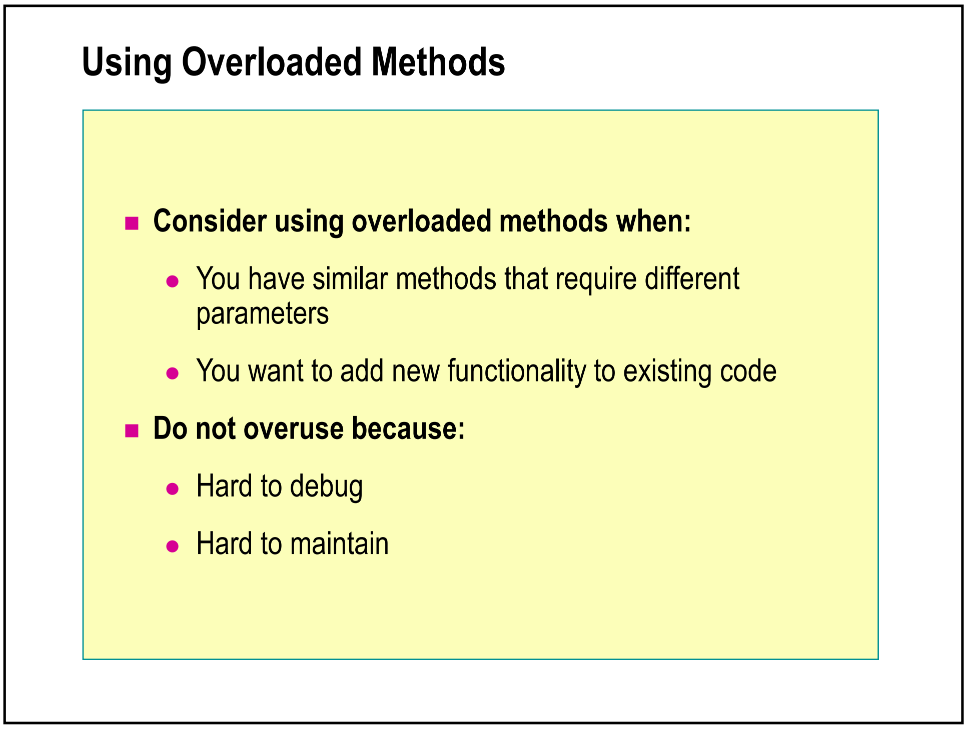
- **Method signatures must be unique within a class**
- **Signature definition**

Forms Signature Definition

- Name of method
- Parameter type
- Parameter modifier

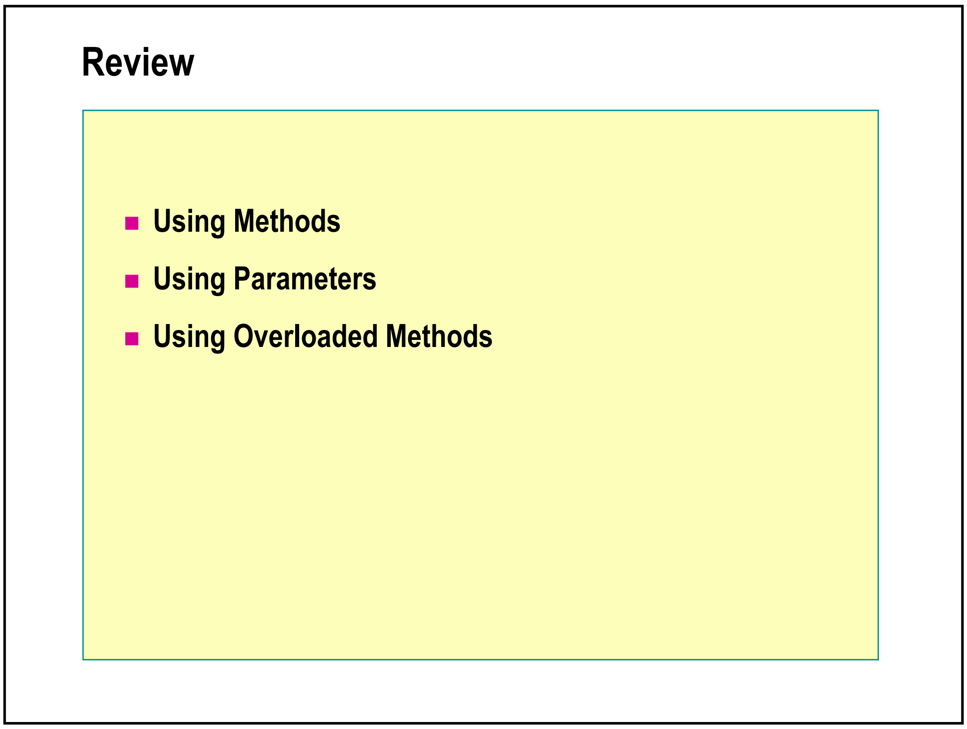
No Effect on Signature

- Name of parameter
- Return type of method



Using Overloaded Methods

- **Consider using overloaded methods when:**
  - You have similar methods that require different parameters
  - You want to add new functionality to existing code
- **Do not overuse because:**
  - Hard to debug
  - Hard to maintain



Review

- **Using Methods**
- **Using Parameters**
- **Using Overloaded Methods**

Module 6: Arrays

Overview

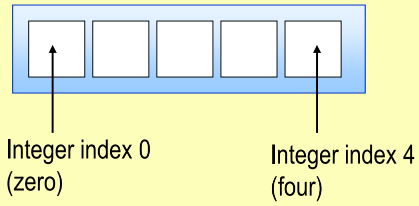
- Overview of Arrays
- Creating Arrays
- Using Arrays

◆ Overview of Arrays

- What Is an Array?
- Array Notation in C#
- Array Rank
- Accessing Array Elements
- Checking Array Bounds
- Comparing Arrays to Collections

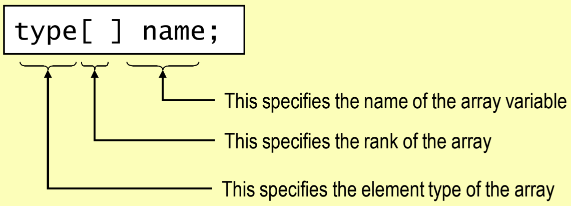
What Is an Array?

- **An array is a sequence of elements**
  - All elements in an array have the same type
  - Structs can have elements of different types
  - Individual elements are accessed using integer indexes



Array Notation in C#

- **You declare an array variable by specifying:**
  - The element type of the array
  - The rank of the array
  - The name of the variable



```
type[ ] name;
```

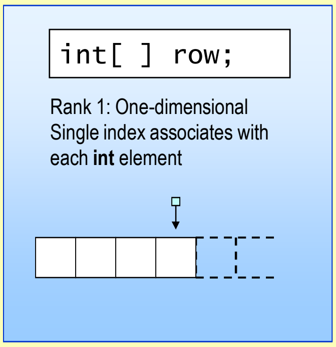
This specifies the name of the array variable

This specifies the rank of the array

This specifies the element type of the array

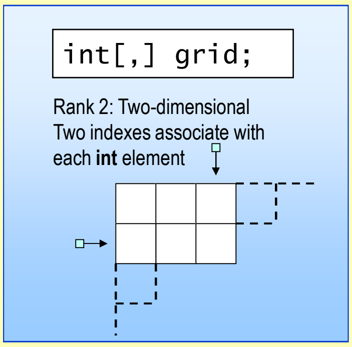
Array Rank

- **Rank is also known as the array dimension**
- **The number of indexes associated with each element**



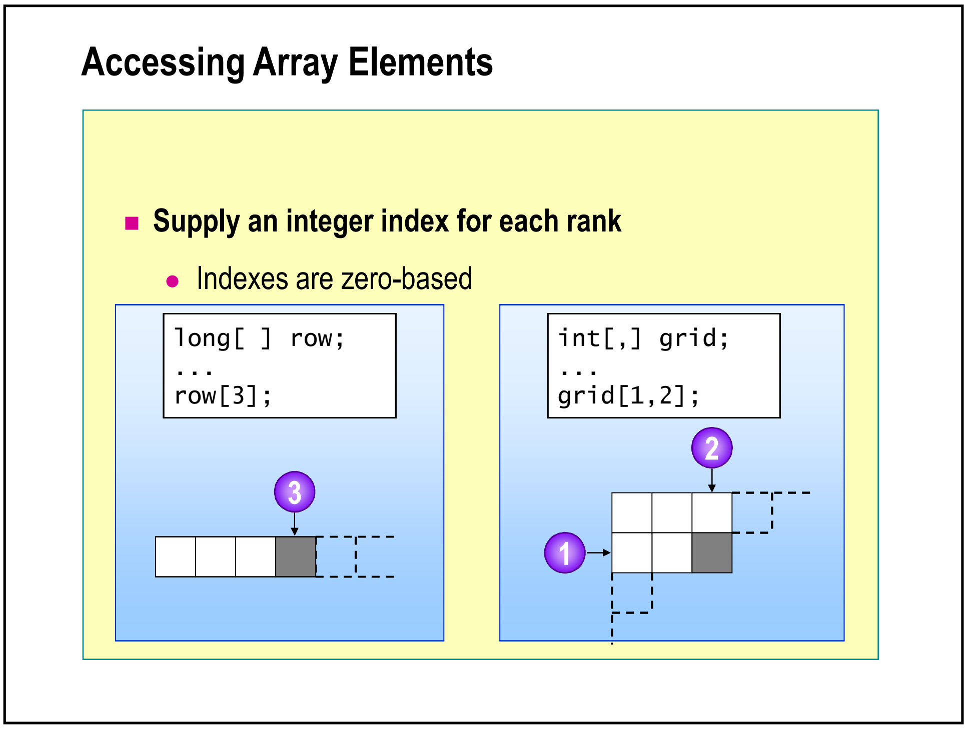
```
int[ ] row;
```

Rank 1: One-dimensional
Single index associates with
each **int** element



```
int[,] grid;
```

Rank 2: Two-dimensional
Two indexes associate with
each **int** element

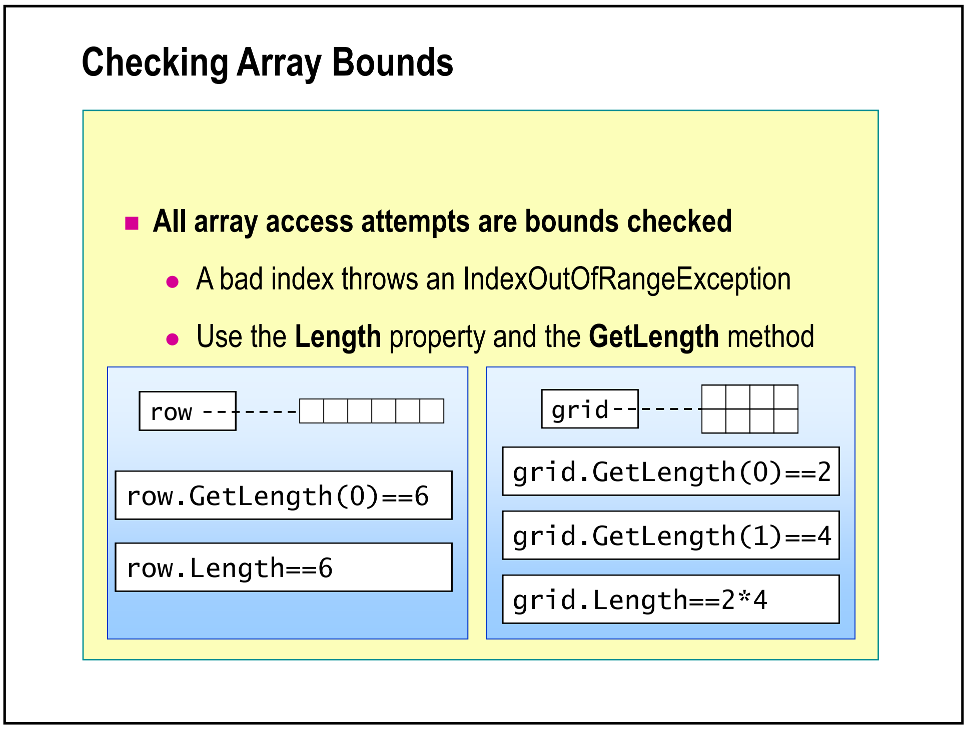


Accessing Array Elements

- **Supply an integer index for each rank**
  - Indexes are zero-based

```
long[ ] row;
...
row[3];
```

```
int[,] grid;
...
grid[1,2];
```



Checking Array Bounds

- **All array access attempts are bounds checked**
  - A bad index throws an IndexOutOfRangeException
  - Use the **Length** property and the **GetLength** method

```
row.GetLength(0)==6
```

```
row.Length==6
```

```
grid.GetLength(0)==2
```

```
grid.GetLength(1)==4
```

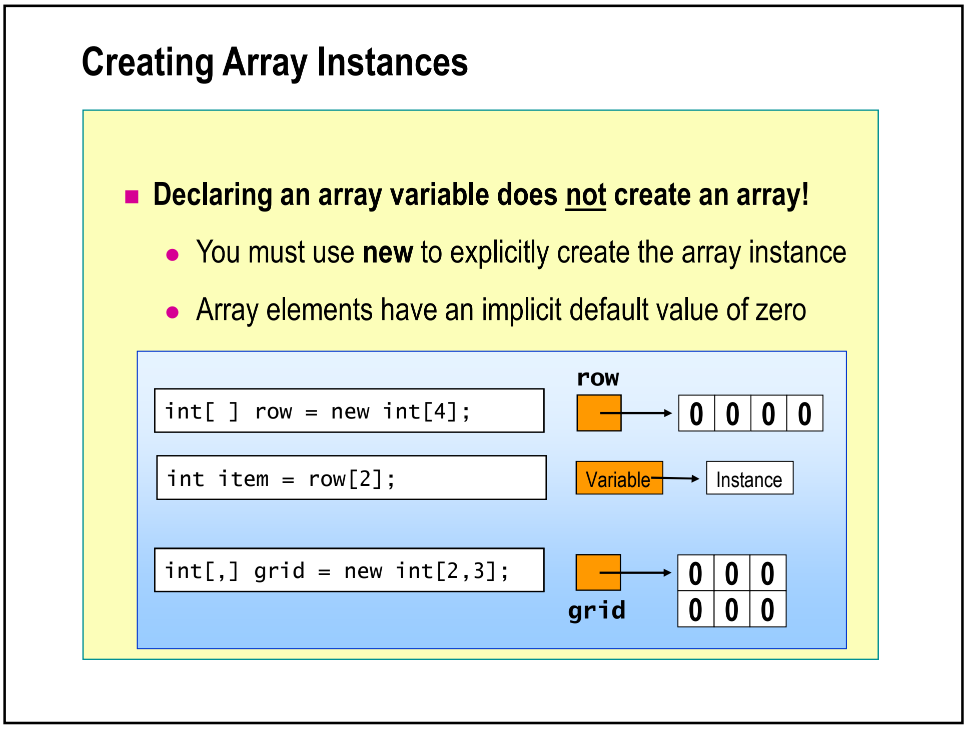
```
grid.Length==2*4
```

Comparing Arrays to Collections

- An array cannot resize itself when full
  - A collection class, such as ArrayList, can resize
- An array is intended to store elements of one type
  - A collection is designed to store elements of different types
- Elements of an array cannot have read-only access
  - A collection can have read-only access
- In general, arrays are faster but less flexible
  - Collections are slightly slower but more flexible

◆ Creating Arrays

- Creating Array Instances
- Initializing Array Elements
- Initializing Multidimensional Array Elements
- Creating a Computed Size Array
- Copying Array Variables



Creating Array Instances

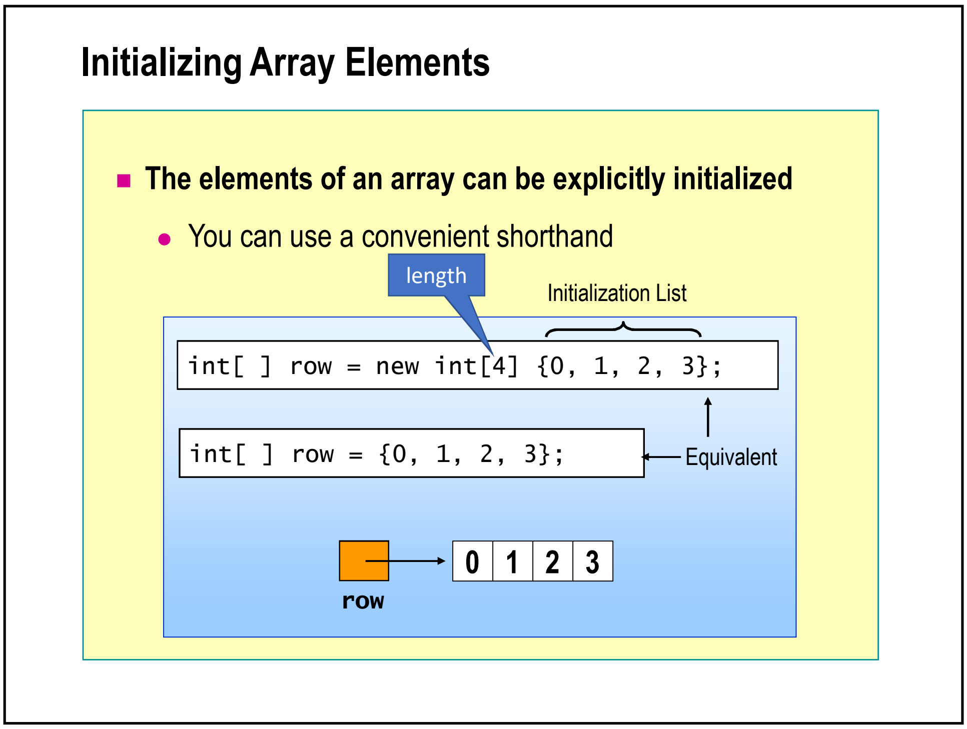
- **Declaring an array variable does <u>not</u> create an array!**
  - You must use **new** to explicitly create the array instance
  - Array elements have an implicit default value of zero

```
int[ ] row = new int[4];
```

```
int item = row[2];
```

```
int[,] grid = new int[2,3];
```

row
0 0 0 0
Variable
Instance
grid
0 0 0
0 0 0



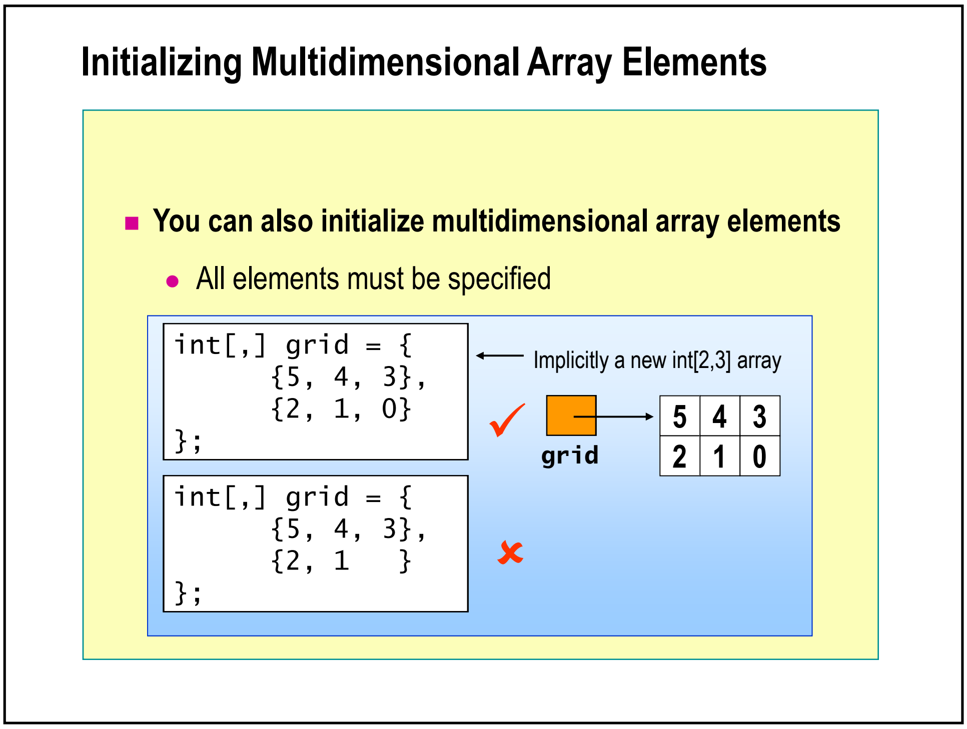
Initializing Array Elements

- **The elements of an array can be explicitly initialized**
  - You can use a convenient shorthand

```
int[ ] row = new int[4] {0, 1, 2, 3};
```

```
int[ ] row = {0, 1, 2, 3};
```

length
Initialization List
Equivalent
row
0 1 2 3



Initializing Multidimensional Array Elements

- You can also initialize multidimensional array elements
  - All elements must be specified

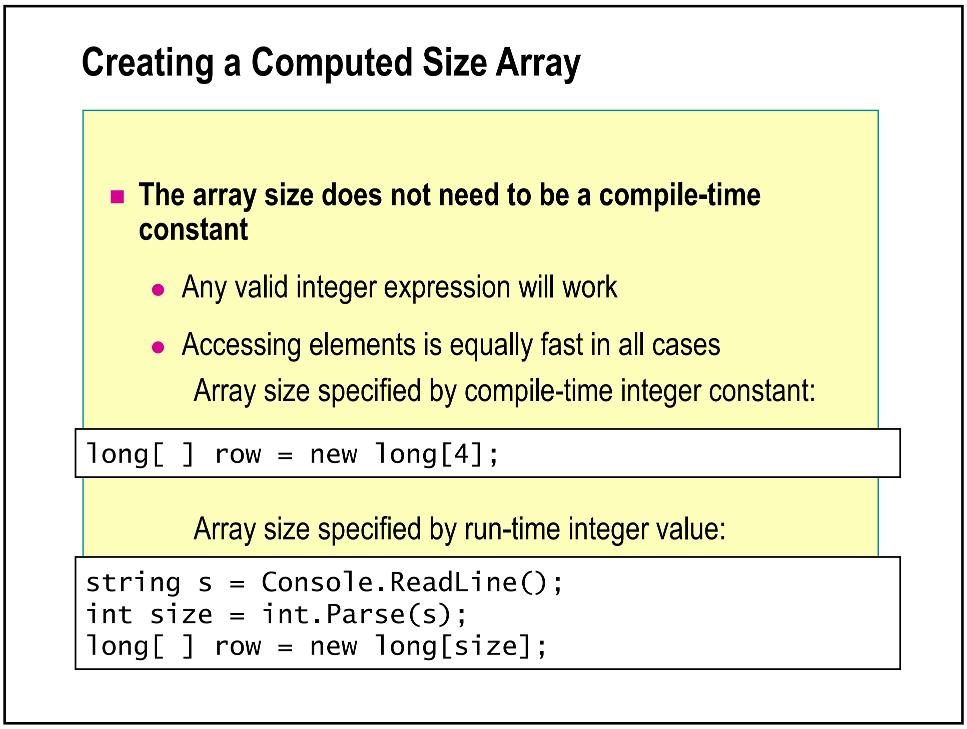
```
int[,] grid = {
       {5, 4, 3},
       {2, 1, 0}
};
```

✓ ← Implicitly a new int[2,3] array

5	4	3
2	1	0

```
int[,] grid = {
       {5, 4, 3},
       {2, 1   }
};
```

✗



Creating a Computed Size Array

- The array size does not need to be a compile-time constant
  - Any valid integer expression will work
  - Accessing elements is equally fast in all cases

Array size specified by compile-time integer constant:

```
long[ ] row = new long[4];
```

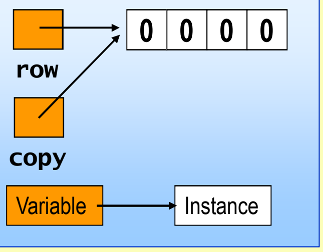
Array size specified by run-time integer value:

```
string s = Console.ReadLine();
int size = int.Parse(s);
long[ ] row = new long[size];
```

Copying Array Variables

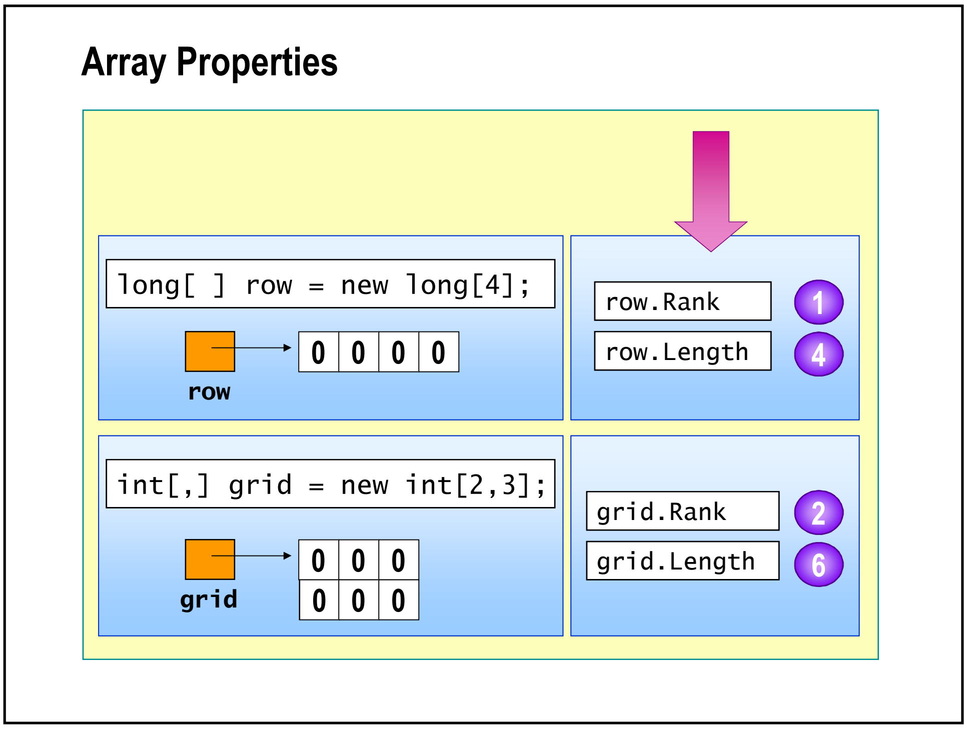
- Copying an array variable copies the array variable only
  - It does not copy the array instance
  - Two array variables can refer to the same array instance

```
long[ ] row = new long[4];
long[ ] copy = row;
...
row[0]++;
long value = copy[0];
Console.WriteLine(value);
```



Using Arrays

- Array Properties
- Array Methods
- Returning Arrays from Methods
- Passing Arrays as Parameters
- Command-Line Arguments
- Using Arrays with foreach
- Quiz: Spot the Bugs



Array Properties

```
long[ ] row = new long[4];
```

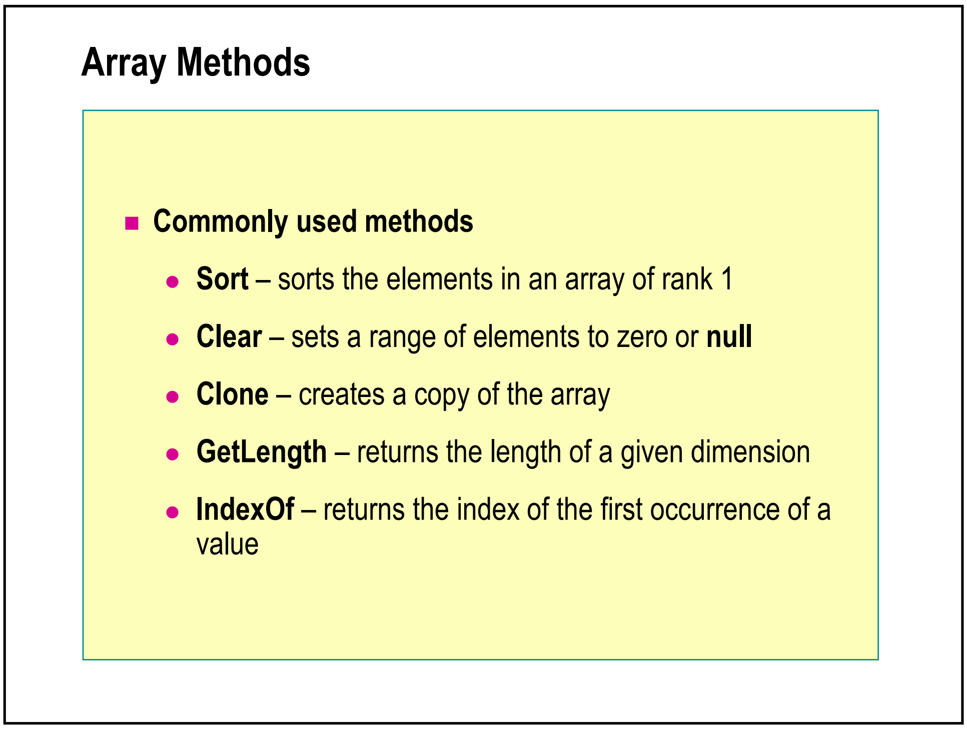
```
int[,] grid = new int[2,3];
```

`row.Rank` 1

`row.Length` 4

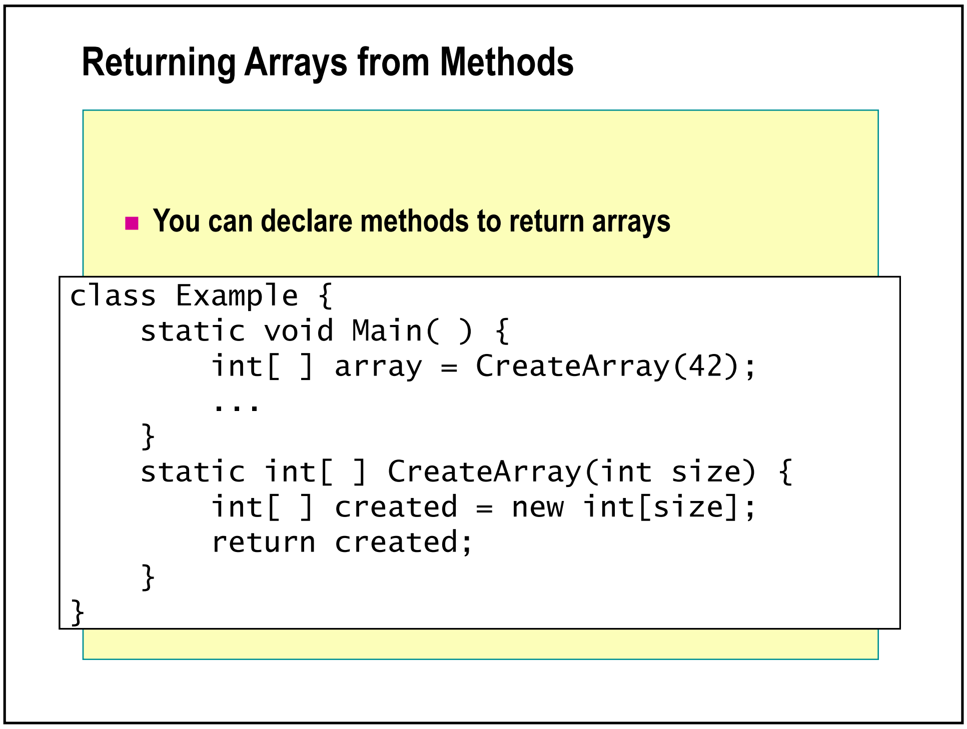
`grid.Rank` 2

`grid.Length` 6



Array Methods

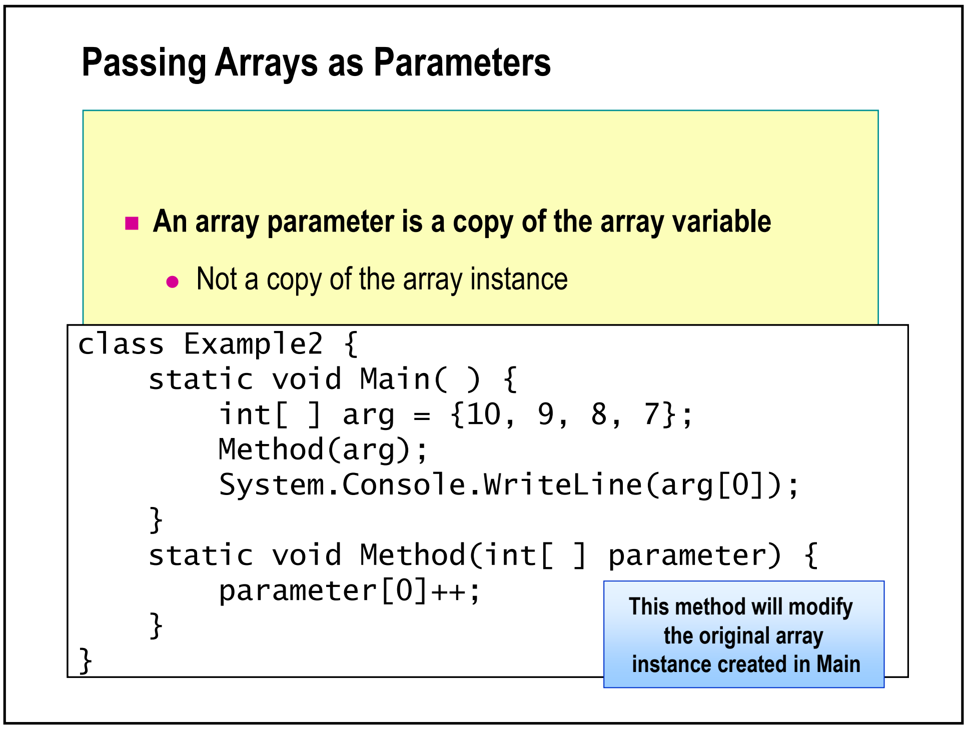
- **Commonly used methods**
  - **Sort** – sorts the elements in an array of rank 1
  - **Clear** – sets a range of elements to zero or **null**
  - **Clone** – creates a copy of the array
  - **GetLength** – returns the length of a given dimension
  - **IndexOf** – returns the index of the first occurrence of a value



Returning Arrays from Methods

- You can declare methods to return arrays

```
class Example {
    static void Main( ) {
        int[ ] array = CreateArray(42);
        ...
    }
    static int[ ] CreateArray(int size) {
        int[ ] created = new int[size];
        return created;
    }
}
```



Passing Arrays as Parameters

- An array parameter is a copy of the array variable
  - Not a copy of the array instance

```
class Example2 {
    static void Main( ) {
        int[ ] arg = {10, 9, 8, 7};
        Method(arg);
        System.Console.WriteLine(arg[0]);
    }
    static void Method(int[ ] parameter) {
        parameter[0]++;
    }
}
```

This method will modify the original array instance created in Main

Command-Line Arguments

- **The runtime passes command line arguments to Main**
  - **Main** can take an array of strings as a parameter
  - The name of the program is not a member of the array

```
class Example3 {
    static void Main(string[ ] args) {
        for (int i = 0; i < args.Length; i++) {
            System.Console.WriteLine(args[i]);
        }
    }
}
```

Using Arrays with foreach

- **The foreach statement abstracts away many details of array handling**

```
class Example4 {
    static void Main(string[ ] args) {
        foreach (string arg in args) {
            System.Console.WriteLine(arg);
        }
    }
}
```

Quiz: Spot the Bugs

1
```
int [ ] array;
array = {0, 2, 4, 6};
```

2
```
int [ ] array;
System.Console.WriteLine(array[0]);
```

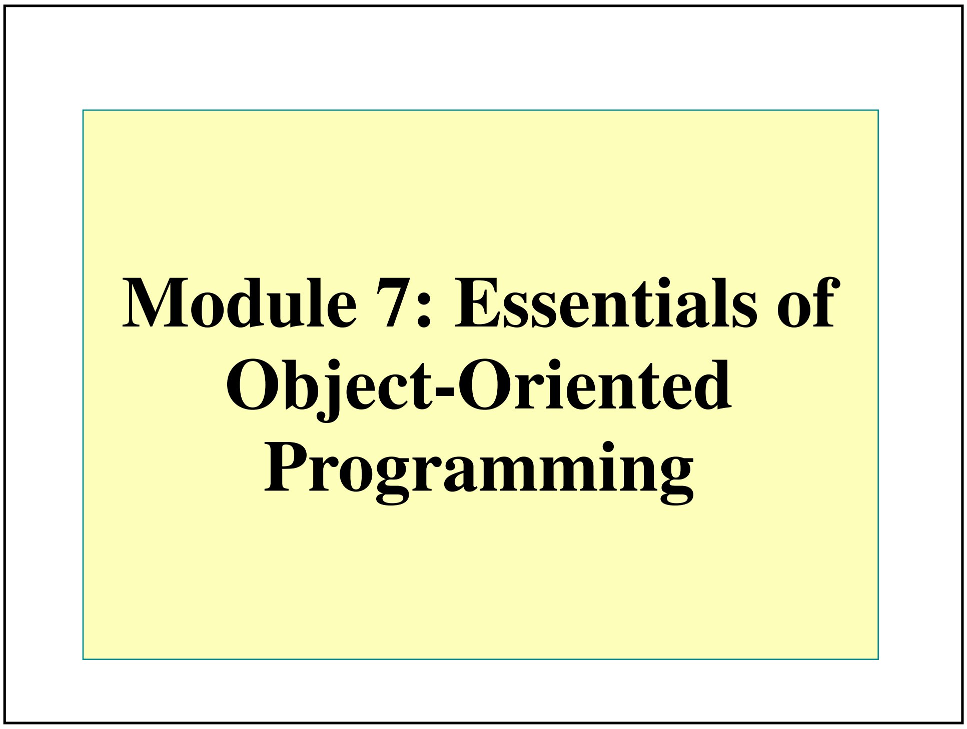
3
```
int [ ] array = new int[3];
System.Console.WriteLine(array[3]);
```

4
```
int [ ] array = new int[ ];
```

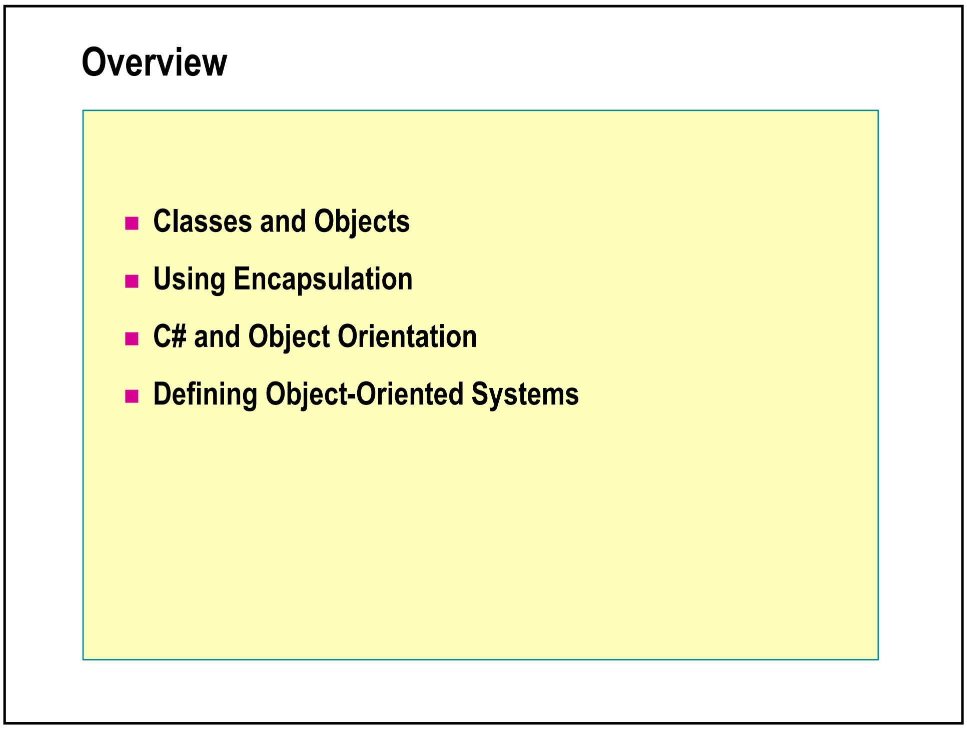
5
```
int [ ] array = new int[3]{0, 1, 2, 3};
```

Review

- **Overview of Arrays**
- **Creating Arrays**
- **Using Arrays**

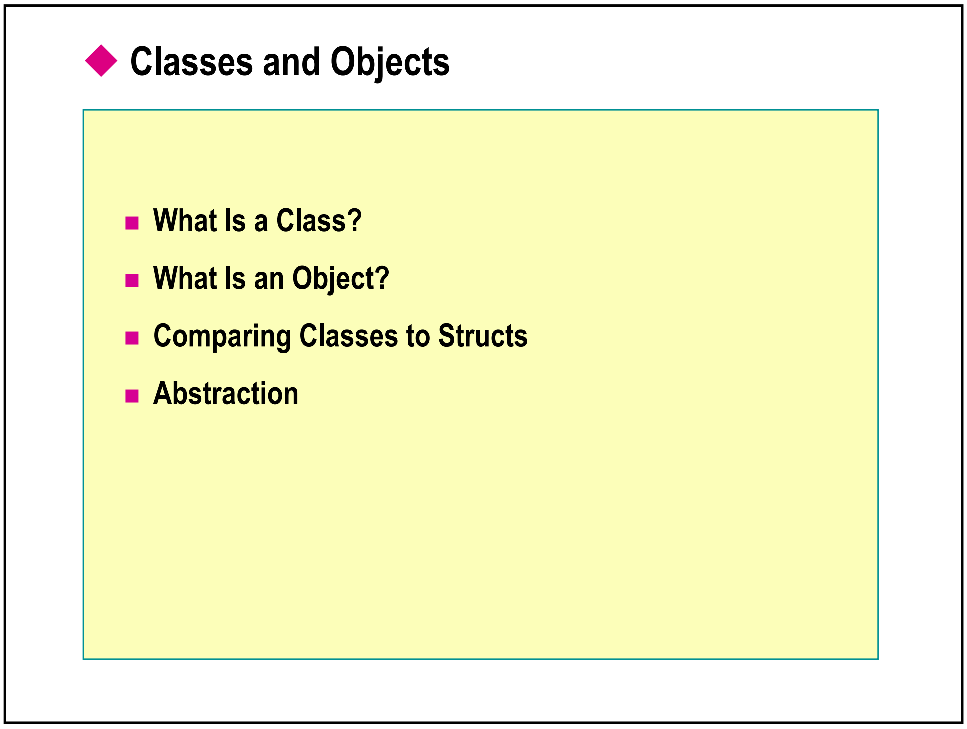


Module 7: Essentials of Object-Oriented Programming



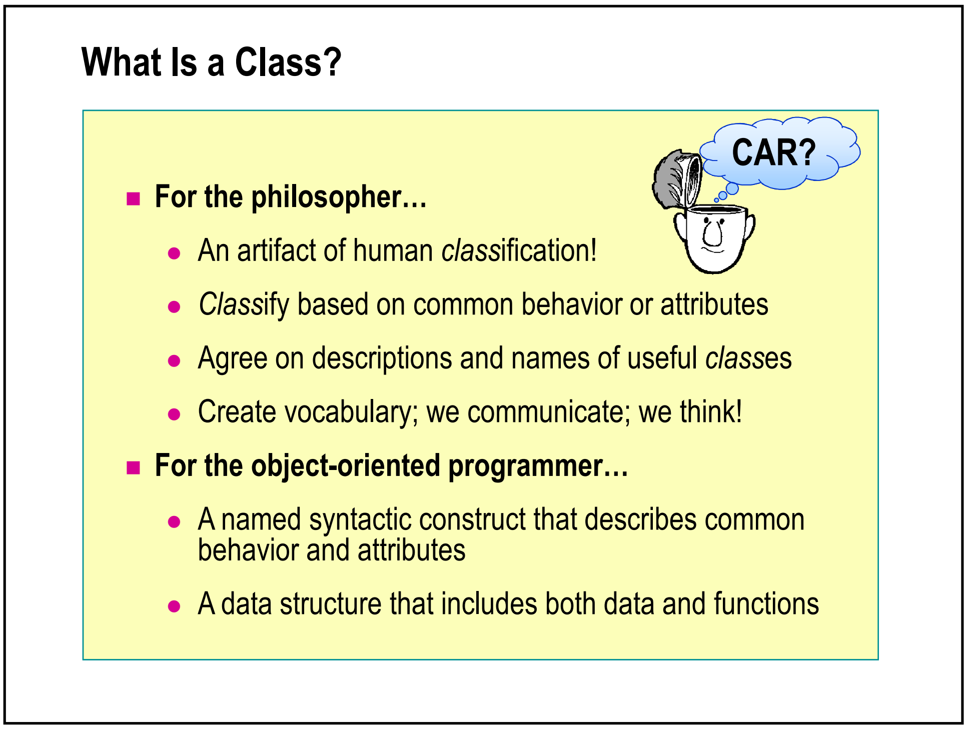
Overview

- Classes and Objects
- Using Encapsulation
- C# and Object Orientation
- Defining Object-Oriented Systems



◆ Classes and Objects

- What Is a Class?
- What Is an Object?
- Comparing Classes to Structs
- Abstraction

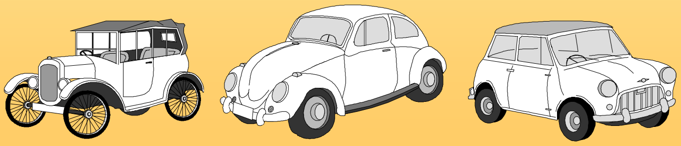


What Is a Class?

- **For the philosopher…**
  - An artifact of human *class*ification!
  - *Class*ify based on common behavior or attributes
  - Agree on descriptions and names of useful *class*es
  - Create vocabulary; we communicate; we think!
- **For the object-oriented programmer…**
  - A named syntactic construct that describes common behavior and attributes
  - A data structure that includes both data and functions

What Is an Object?

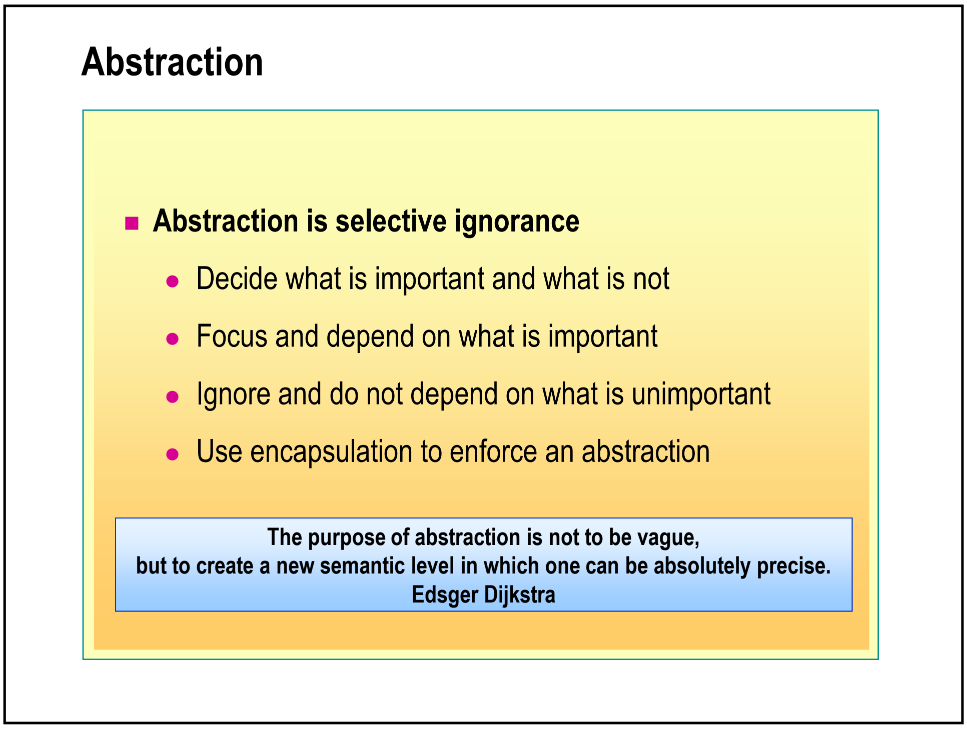
- **An object is an instance of a class**
- **Objects exhibit:**
  - Identity: Objects are distinguishable from one another
  - Behavior: Objects can perform tasks
  - State: Objects store information



Comparing Classes to Structs

- **A struct is a blueprint for a value**
  - No identity, accessible state, no added behavior
- **A class is a blueprint for an object**
  - Identity, inaccessible state, added behavior

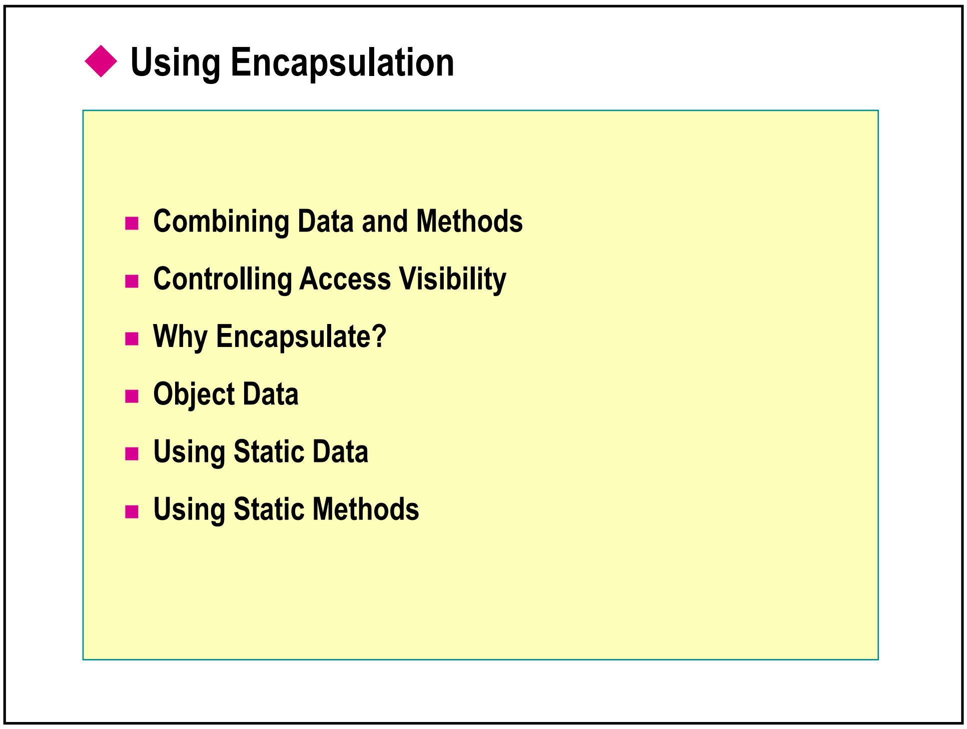
```
struct Time                         class BankAccount
{                                   {
    public int hour;                    ...
    public int minute;                  ...
}                                   }
```



Abstraction

- **Abstraction is selective ignorance**
  - Decide what is important and what is not
  - Focus and depend on what is important
  - Ignore and do not depend on what is unimportant
  - Use encapsulation to enforce an abstraction

> **The purpose of abstraction is not to be vague,**
> **but to create a new semantic level in which one can be absolutely precise.**
> **Edsger Dijkstra**

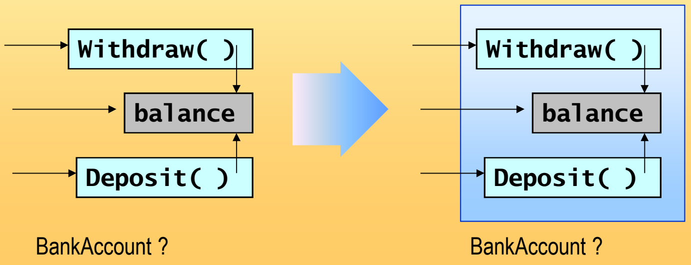


◆ Using Encapsulation

- **Combining Data and Methods**
- **Controlling Access Visibility**
- **Why Encapsulate?**
- **Object Data**
- **Using Static Data**
- **Using Static Methods**

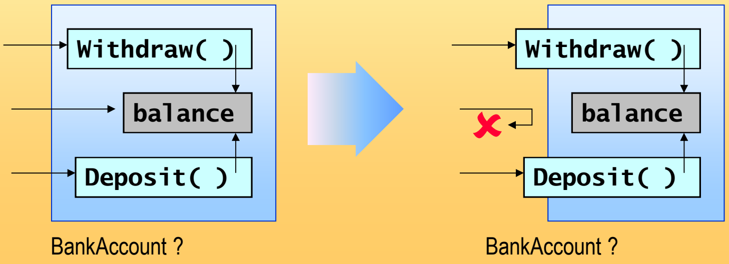
Combining Data and Methods

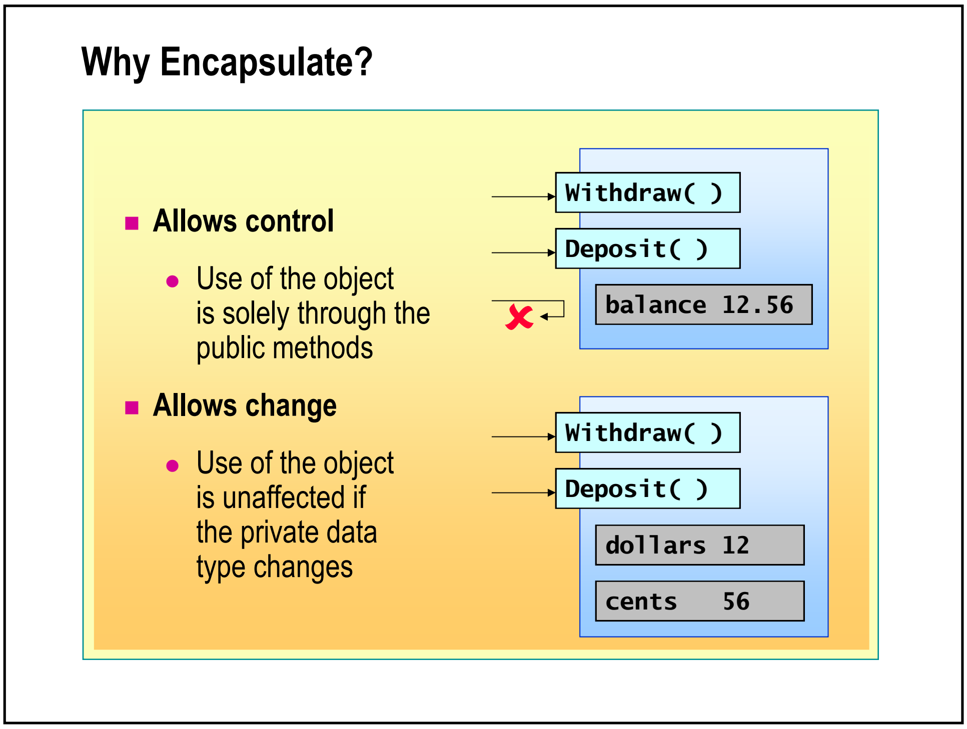
- Combine the data and methods in a single *capsule*
- The capsule boundary forms an inside and an outside



Controlling Access Visibility

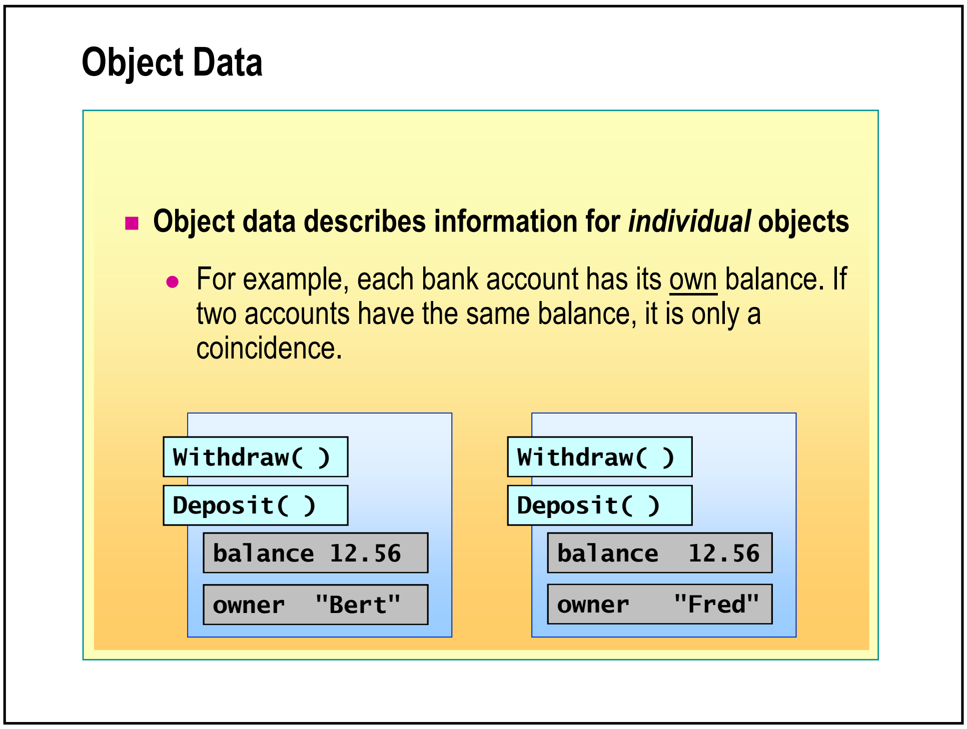
- Methods are *public*, accessible from the outside
- Data is *private*, accessible only from the inside





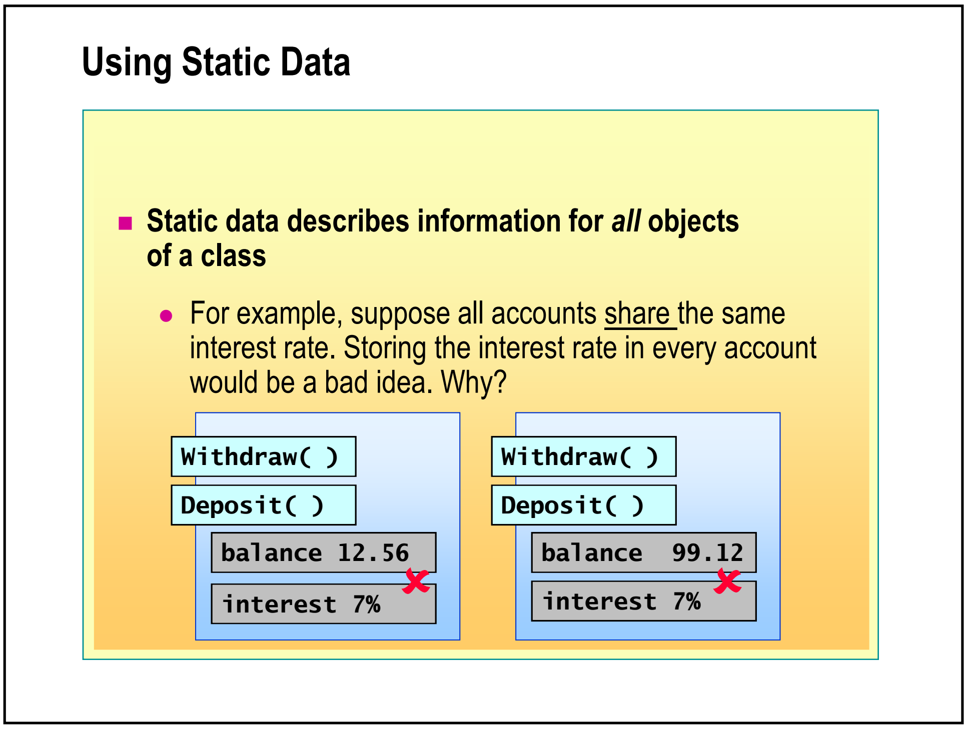
Why Encapsulate?

- **Allows control**
  - Use of the object is solely through the public methods
- **Allows change**
  - Use of the object is unaffected if the private data type changes



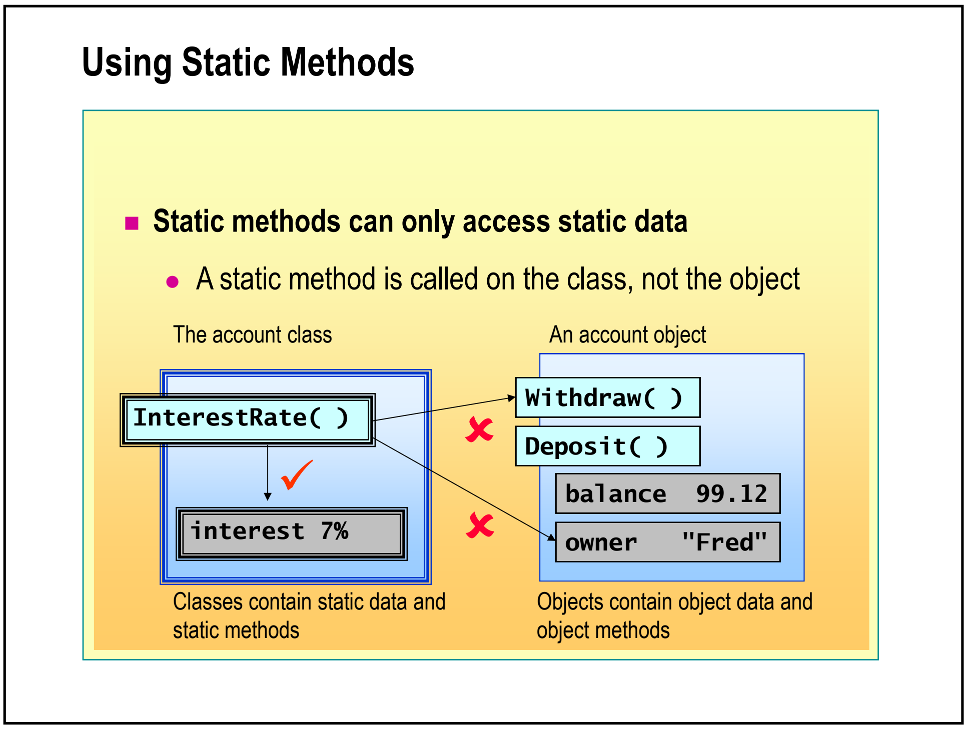
Object Data

- **Object data describes information for *individual* objects**
  - For example, each bank account has its own balance. If two accounts have the same balance, it is only a coincidence.



Using Static Data

- **Static data describes information for *all* objects of a class**
  - For example, suppose all accounts share the same interest rate. Storing the interest rate in every account would be a bad idea. Why?



Using Static Methods

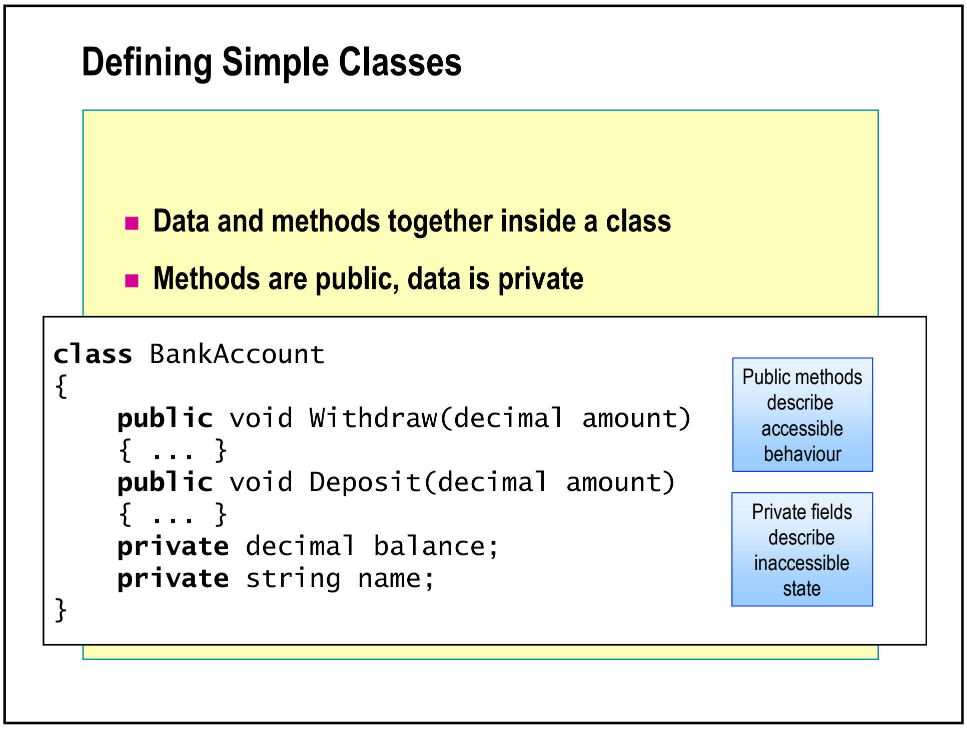
- **Static methods can only access static data**
  - A static method is called on the class, not the object

◆ C# and Object Orientation

- Hello, World Revisited
- Defining Simple Classes
- Instantiating New Objects
- Using the this Operator
- Creating Nested Classes
- Accessing Nested Classes

Hello, World Revisited

```
using System;

class Hello
{
	public static int Main( )
	{
		Console.WriteLine("Hello, World");
		return 0;
	}
}
```



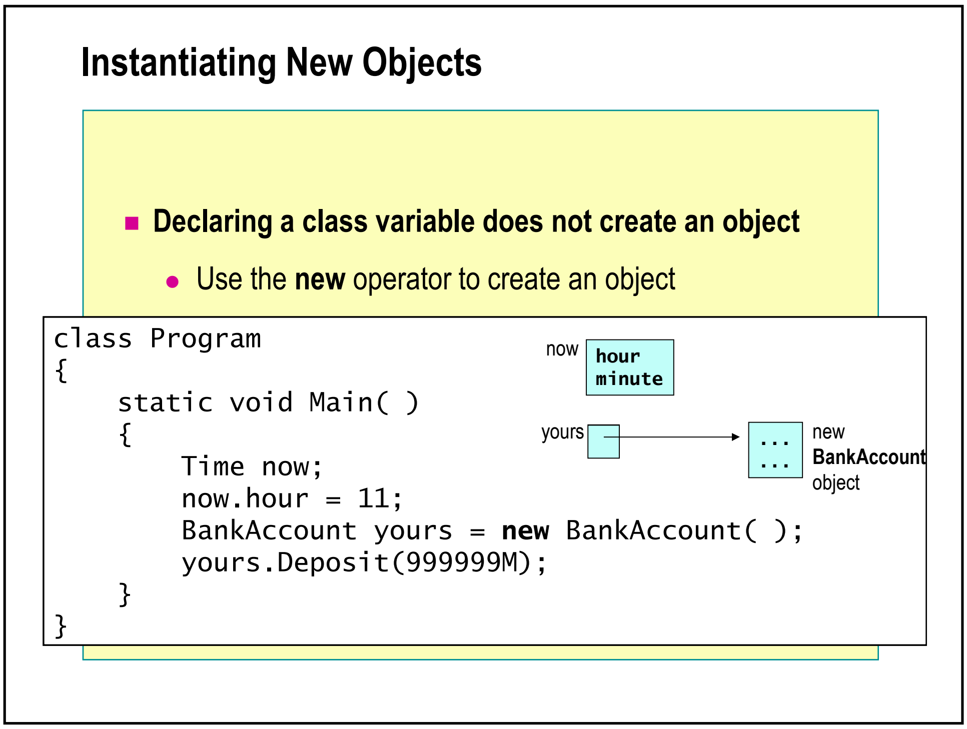
Defining Simple Classes

- Data and methods together inside a class
- Methods are public, data is private

```
class BankAccount
{
    public void Withdraw(decimal amount)
    { ... }
    public void Deposit(decimal amount)
    { ... }
    private decimal balance;
    private string name;
}
```

Public methods describe accessible behaviour

Private fields describe inaccessible state

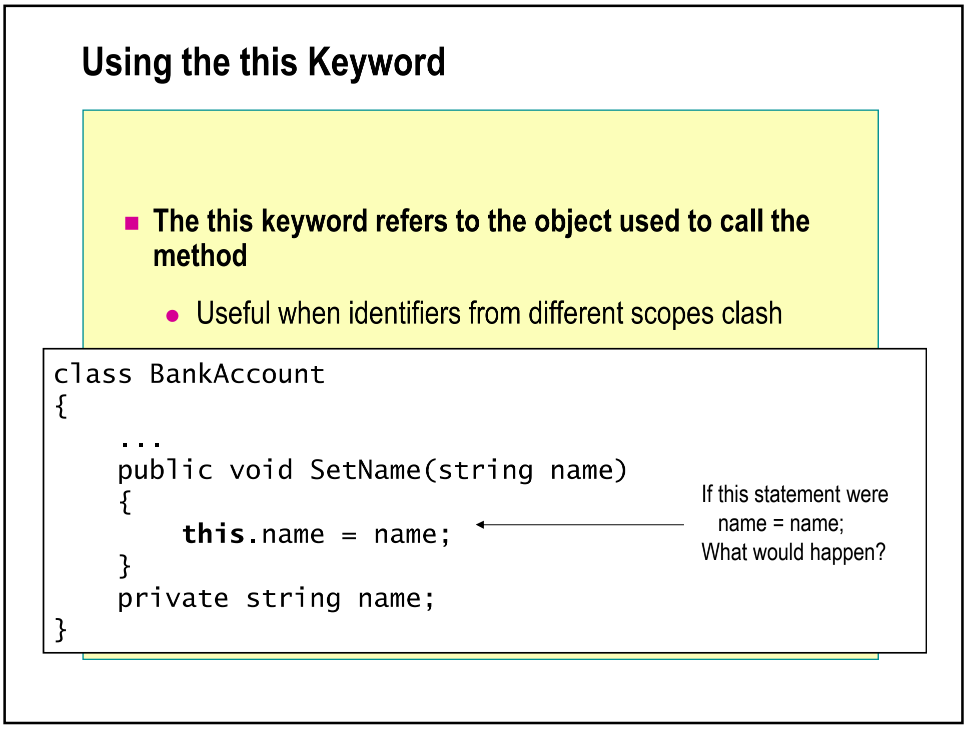


Instantiating New Objects

- Declaring a class variable does not create an object
  - Use the **new** operator to create an object

```
class Program
{
    static void Main( )
    {
        Time now;
        now.hour = 11;
        BankAccount yours = new BankAccount( );
        yours.Deposit(999999M);
    }
}
```

now
hour
minute
yours
...
...
new
BankAccount
object

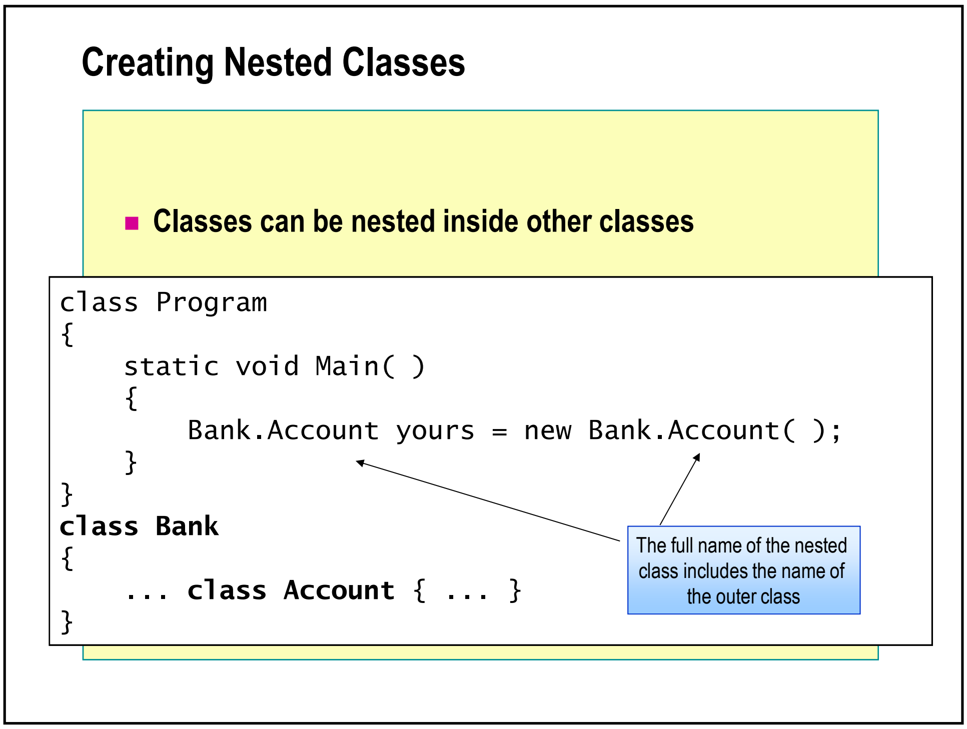


Using the this Keyword

- **The this keyword refers to the object used to call the method**
  - Useful when identifiers from different scopes clash

```
class BankAccount
{
    ...
    public void SetName(string name)
    {
        this.name = name;
    }
    private string name;
}
```

If this statement were
name = name;
What would happen?



Creating Nested Classes

- **Classes can be nested inside other classes**

```
class Program
{
    static void Main( )
    {
        Bank.Account yours = new Bank.Account( );
    }
}
class Bank
{
    ... class Account { ... }
}
```

The full name of the nested class includes the name of the outer class

Accessing Nested Classes

- Nested classes can also be declared as public or private

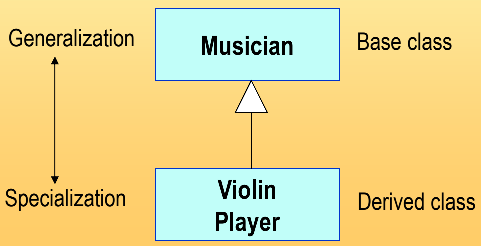
```
class Bank
{
    public  class Account { ... }
    private class AccountNumberGenerator { ... }
}
class Program
{
    static void Main( )
    {
        Bank.Account                      accessible; ✓
        Bank.AccountNumberGenerator inaccessible; ✗
    }
}
```

◆ Defining Object-Oriented Systems

- Inheritance
- Class Hierarchies
- Single and Multiple Inheritance
- Polymorphism
- Abstract Base Classes
- Interfaces
- Early and Late Binding

Inheritance

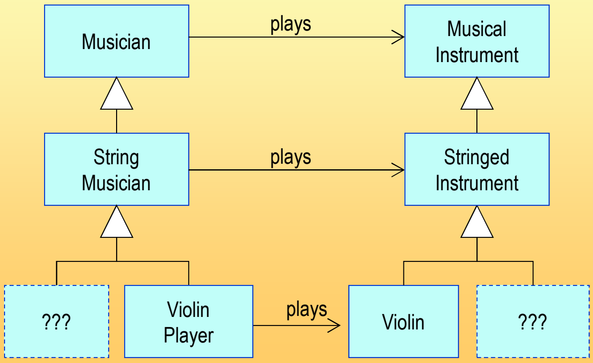
- **Inheritance specifies an “is a kind of" relationship**
  - Inheritance is a class relationship
  - New classes specialize existing classes



Is this a good example of inheritance ?

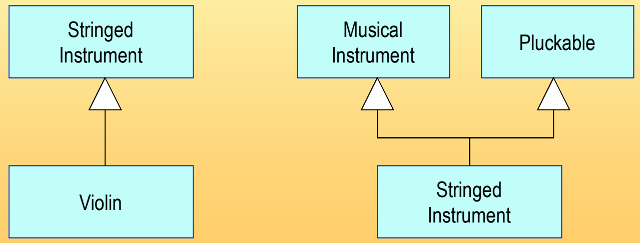
Class Hierarchies

- **Classes related by inheritance form class hierarchies**



Single and Multiple Inheritance

- **Single inheritance: deriving from one base class**
- **Multiple inheritance: deriving from two or more base classes**

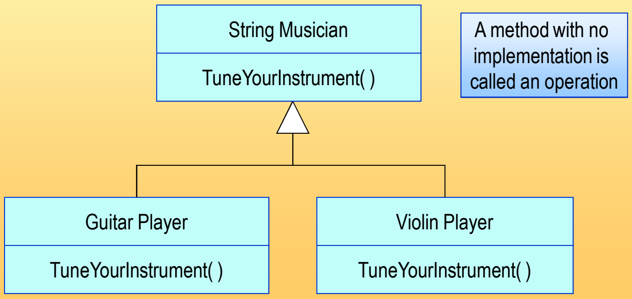


Violin has a single direct base class

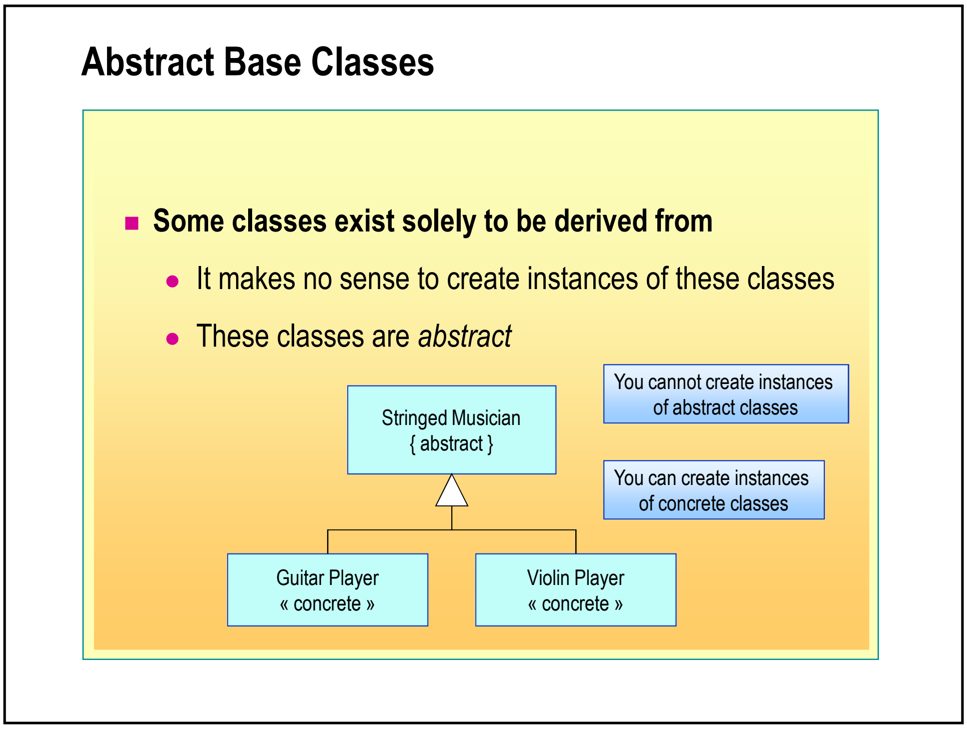
Stringed Instrument has two direct base classes

Polymorphism

- **The method name resides in the base class**
- **The method implementations reside in the derived classes**



A method with no implementation is called an operation



Abstract Base Classes

- **Some classes exist solely to be derived from**
  - It makes no sense to create instances of these classes
  - These classes are *abstract*

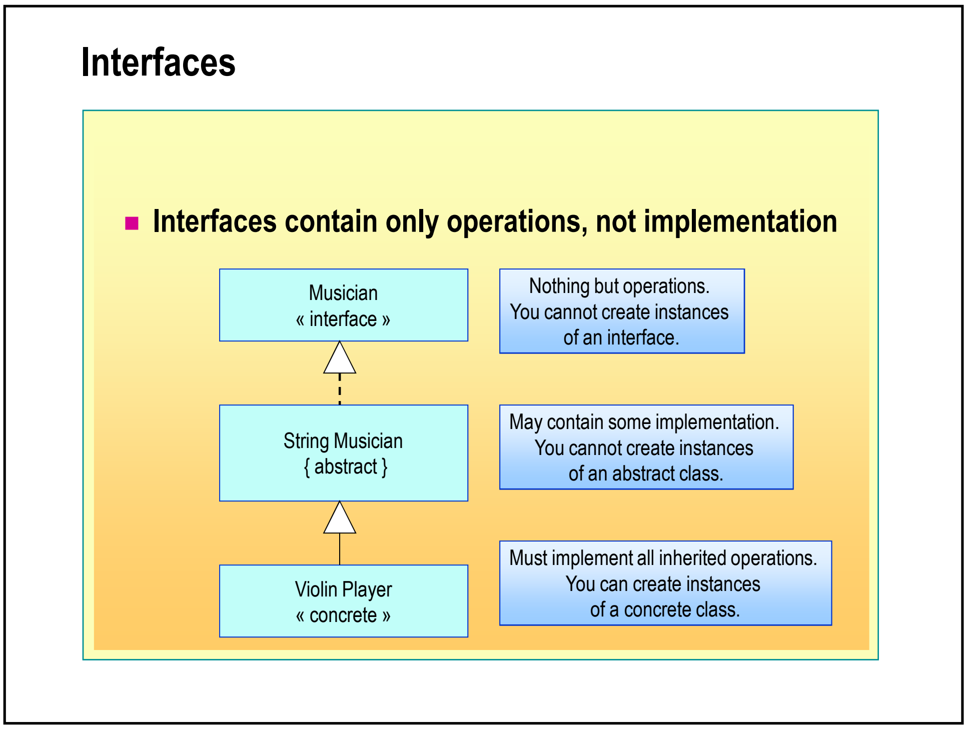
Stringed Musician
{ abstract }

Guitar Player
« concrete »

Violin Player
« concrete »

You cannot create instances of abstract classes

You can create instances of concrete classes



Interfaces

- **Interfaces contain only operations, not implementation**

Musician
« interface »

Nothing but operations.
You cannot create instances of an interface.

String Musician
{ abstract }

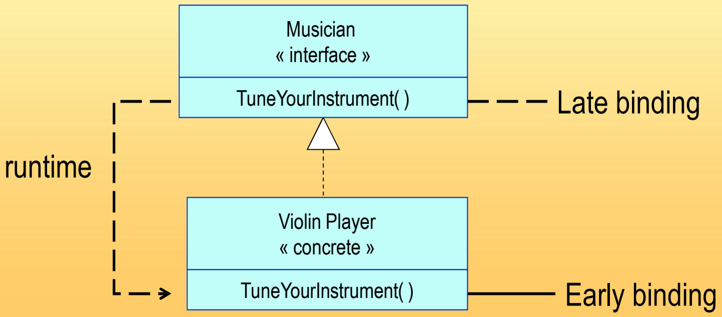
May contain some implementation.
You cannot create instances of an abstract class.

Violin Player
« concrete »

Must implement all inherited operations.
You can create instances of a concrete class.

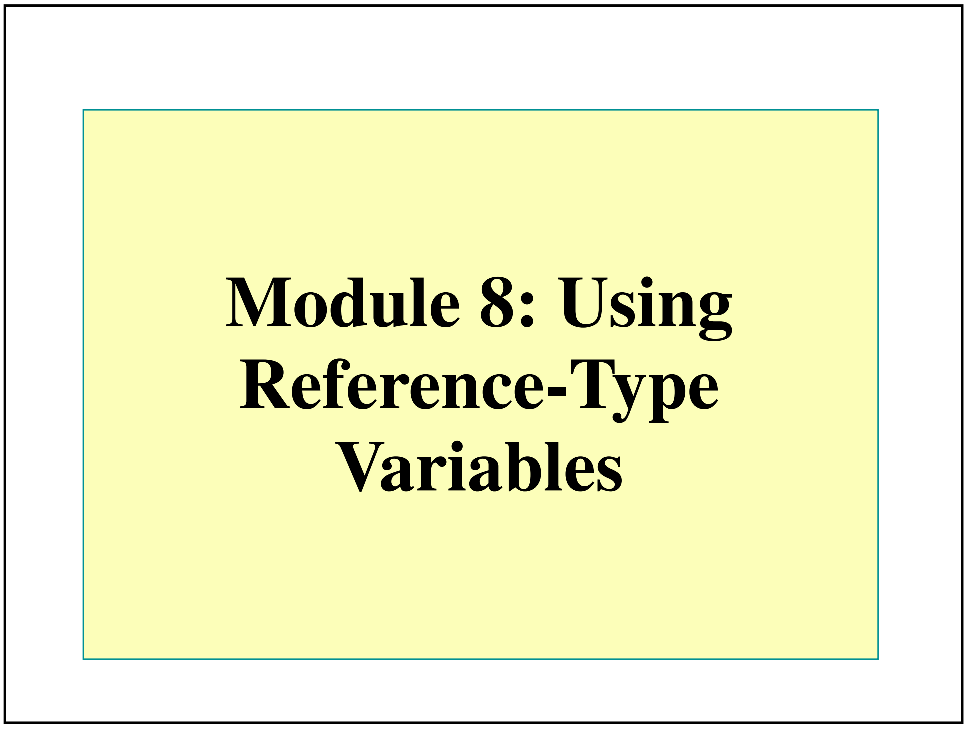
Early and Late Binding

- Normal method calls are resolved at compile time
- Polymorphic method calls are resolved at run time

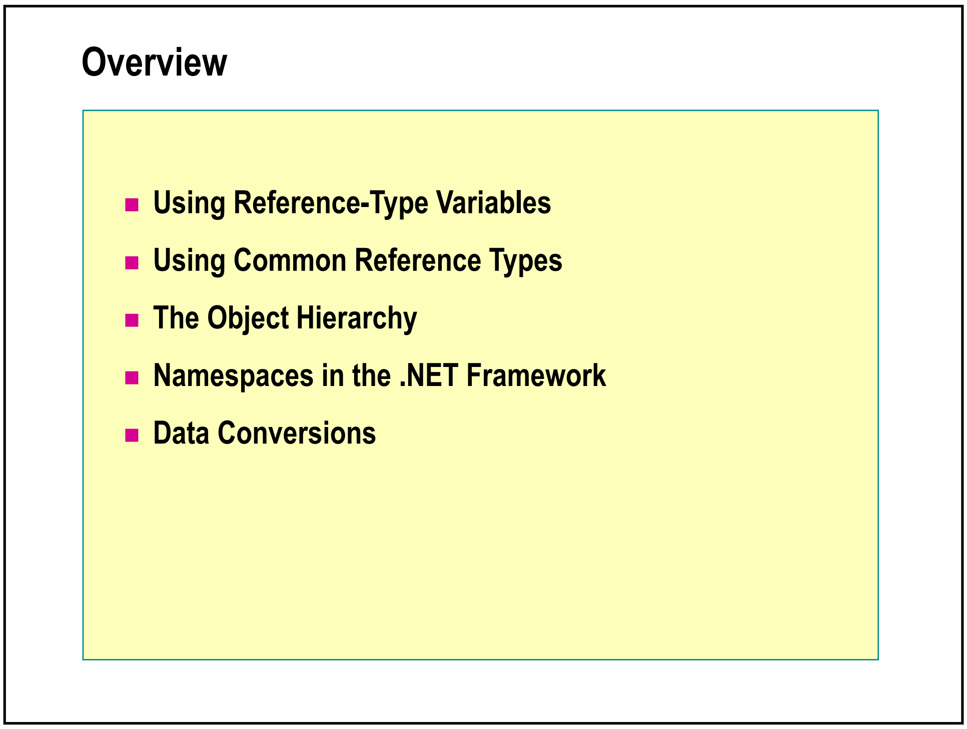


Review

- Classes and Objects
- Using Encapsulation
- C# and Object Orientation
- Defining Object-Oriented Systems



Module 8: Using Reference-Type Variables



Overview

- Using Reference-Type Variables
- Using Common Reference Types
- The Object Hierarchy
- Namespaces in the .NET Framework
- Data Conversions

◆ Using Reference-Type Variables

- Comparing Value Types to Reference Types
- Declaring and Releasing Reference Variables
- Invalid References
- Comparing Values and Comparing References
- Multiple References to the Same Object
- Using References as Method Parameters

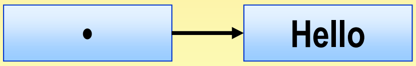
Comparing Value Types to Reference Types

- **Value types**
  - The variable contains the value directly
  - Examples: **char**, **int**

```
int mol;
mol = 42;
```

- **Reference types**
  - The variable contains a reference to the data
  - Data is stored in a separate memory area

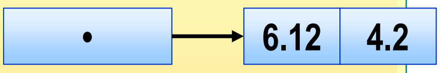
```
string mol;
mol = "Hello";
```



Declaring and Releasing Reference Variables

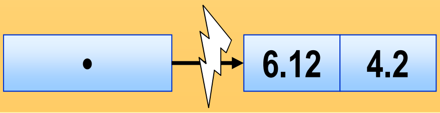
- Declaring reference variables

```
coordinate c1;
c1 = new coordinate();
c1.x = 6.12;
c1.y = 4.2;
```



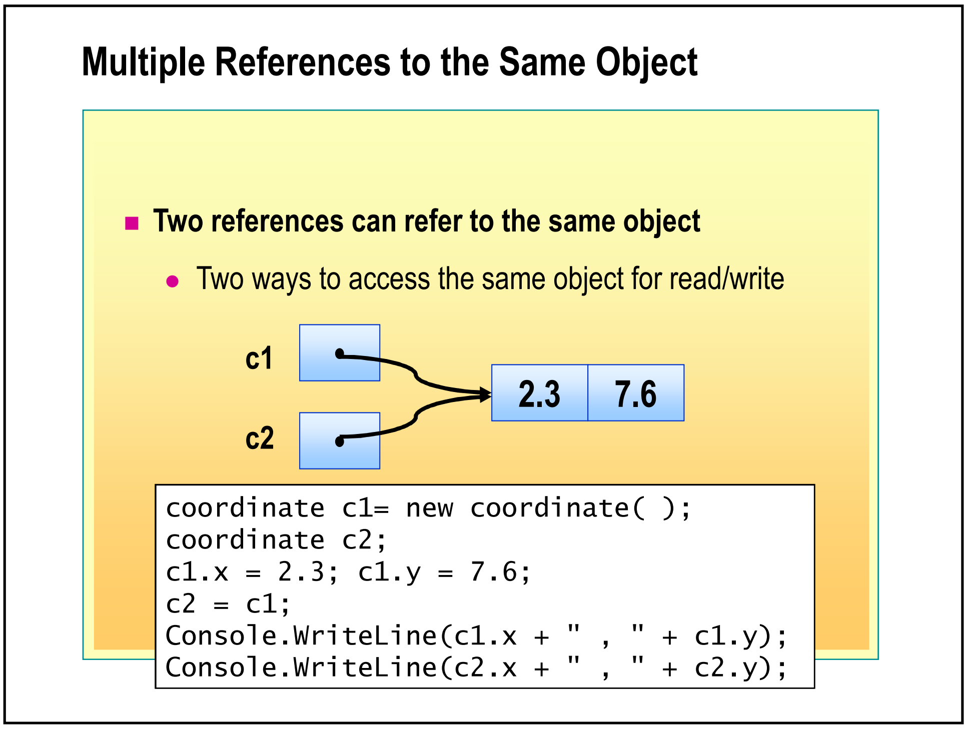
- Releasing reference variables

```
c1 = null;
```



Invalid References

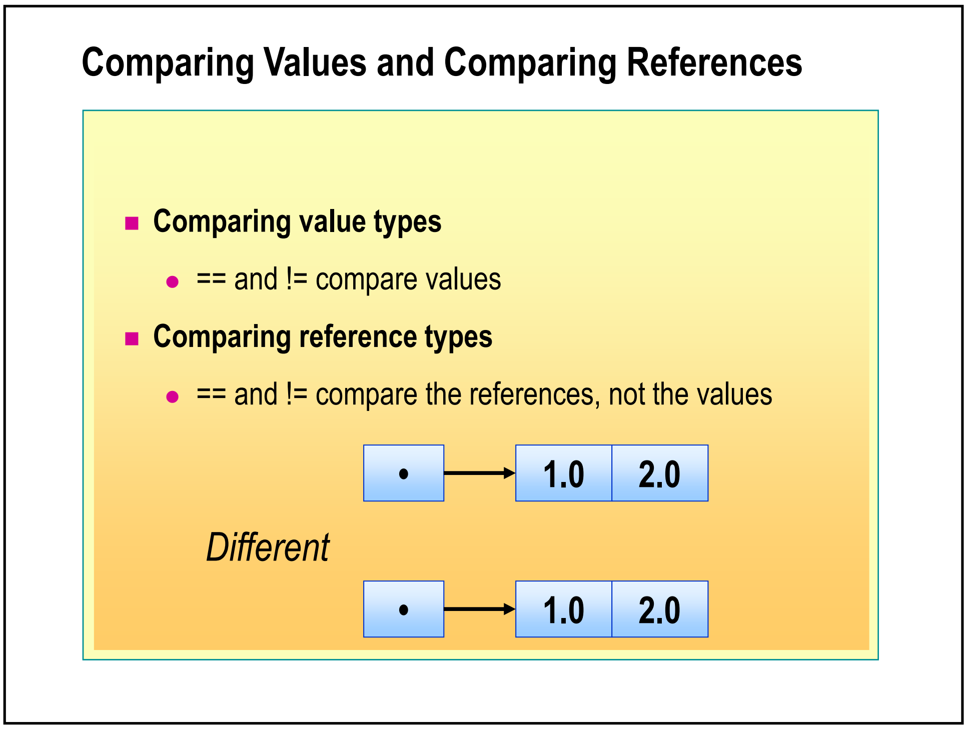
- **If you have invalid references**
  - You cannot access members or variables
- **Invalid references at compile time**
  - Compiler detects use of uninitialized references
- **Invalid references at run time**
  - System will generate an exception error



Multiple References to the Same Object

- **Two references can refer to the same object**
  - Two ways to access the same object for read/write

```
coordinate c1= new coordinate( );
coordinate c2;
c1.x = 2.3; c1.y = 7.6;
c2 = c1;
Console.WriteLine(c1.x + " , " + c1.y);
Console.WriteLine(c2.x + " , " + c2.y);
```

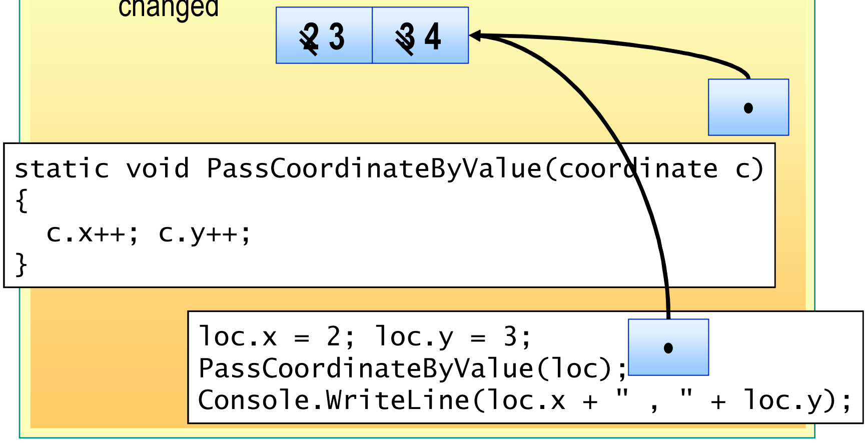


Comparing Values and Comparing References

- **Comparing value types**
  - == and != compare values
- **Comparing reference types**
  - == and != compare the references, not the values

Using References as Method Parameters

- **References can be used as parameters**
  - When passed by value, data being referenced may be changed

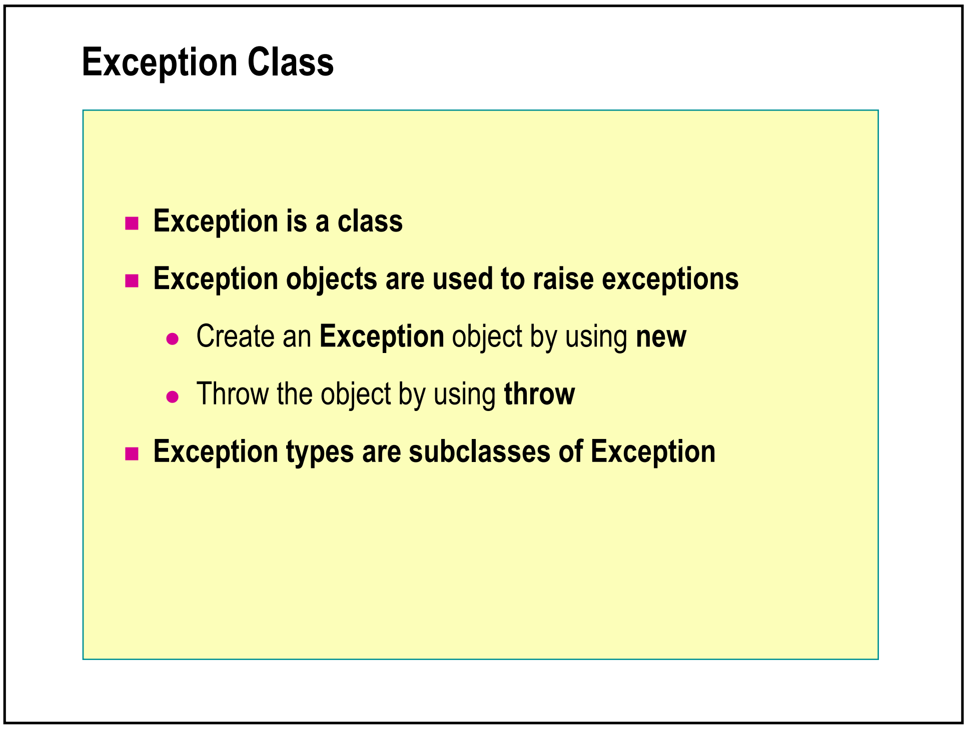


```
static void PassCoordinateByValue(coordinate c)
{
  c.x++; c.y++;
}
```

```
loc.x = 2; loc.y = 3;
PassCoordinateByValue(loc);
Console.WriteLine(loc.x + " , " + loc.y);
```

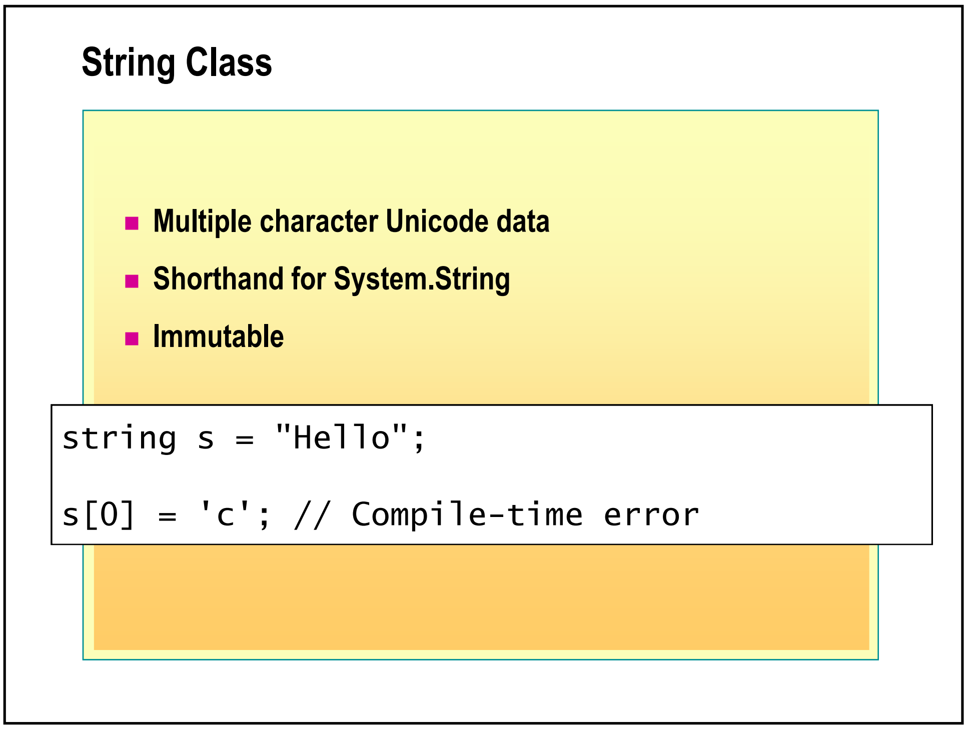
Using Common Reference Types

- **Exception Class**
- **String Class**
- **Common String Methods, Operators, and Properties**
- **String Comparisons**
- **String Comparison Operators**



Exception Class

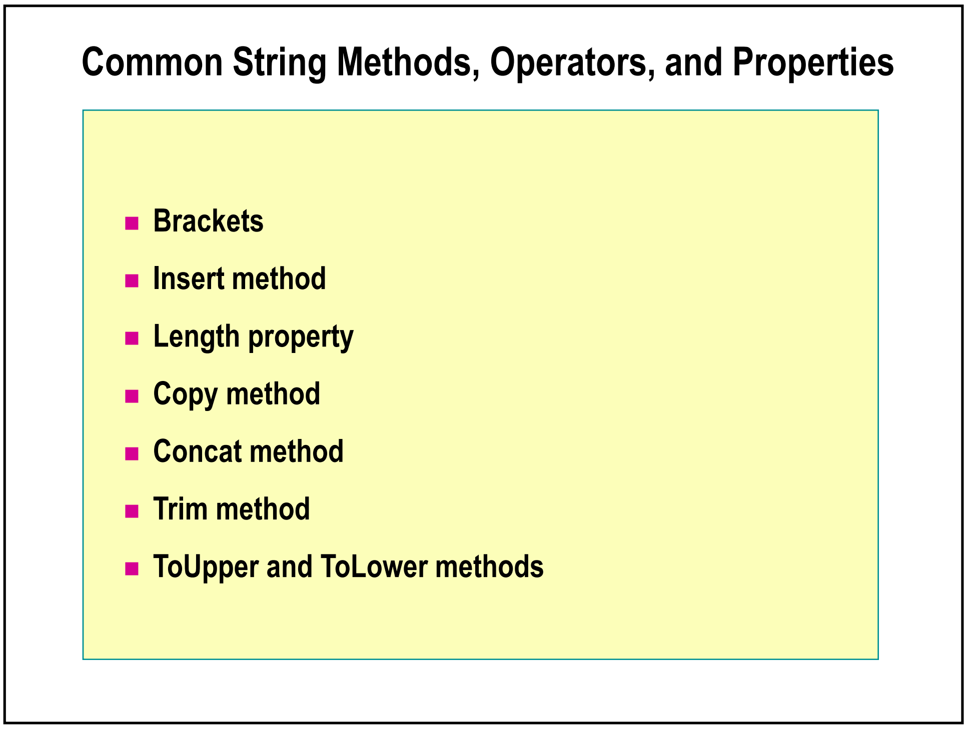
- Exception is a class
- Exception objects are used to raise exceptions
  - Create an **Exception** object by using **new**
  - Throw the object by using **throw**
- Exception types are subclasses of Exception



String Class

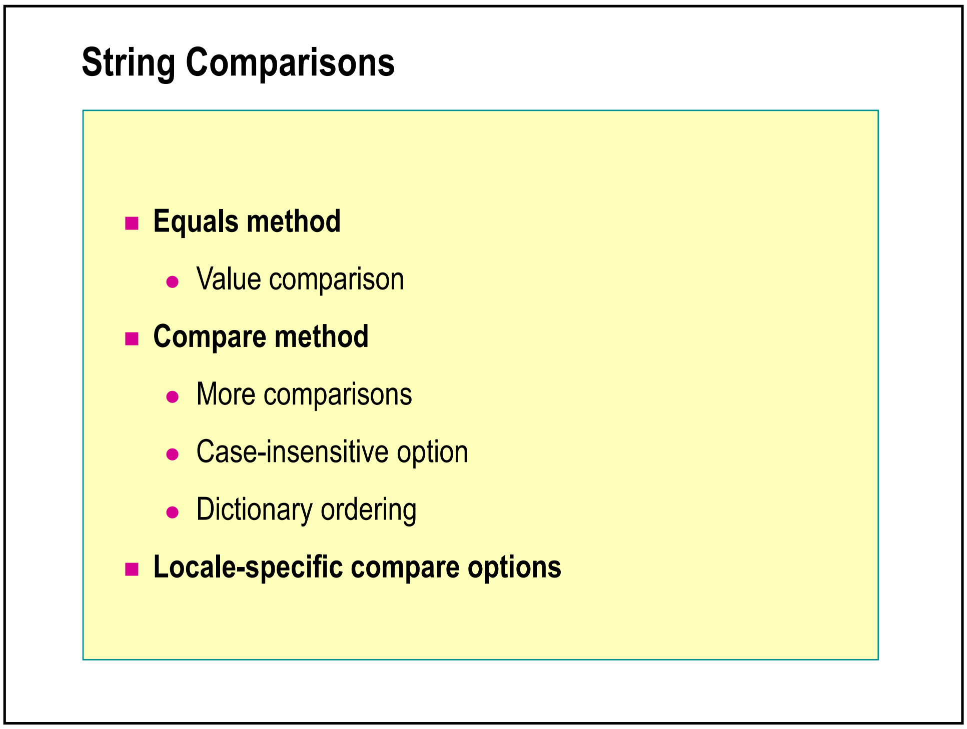
- Multiple character Unicode data
- Shorthand for System.String
- Immutable

```
string s = "Hello";

s[0] = 'c'; // Compile-time error
```



Common String Methods, Operators, and Properties

- Brackets
- Insert method
- Length property
- Copy method
- Concat method
- Trim method
- ToUpper and ToLower methods



String Comparisons

- Equals method
  - Value comparison
- Compare method
  - More comparisons
  - Case-insensitive option
  - Dictionary ordering
- Locale-specific compare options

String Comparison Operators

- The == and != operators are overloaded for strings
- They are equivalent to String.Equals and !String.Equals

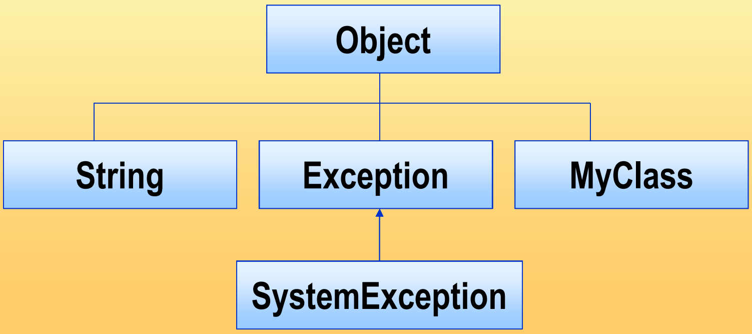
```
string a = "Test";
string b = "Test";
if (a == b) ...   // Returns true
```

◆ The Object Hierarchy

- The object Type
- Common Methods
- Reflection

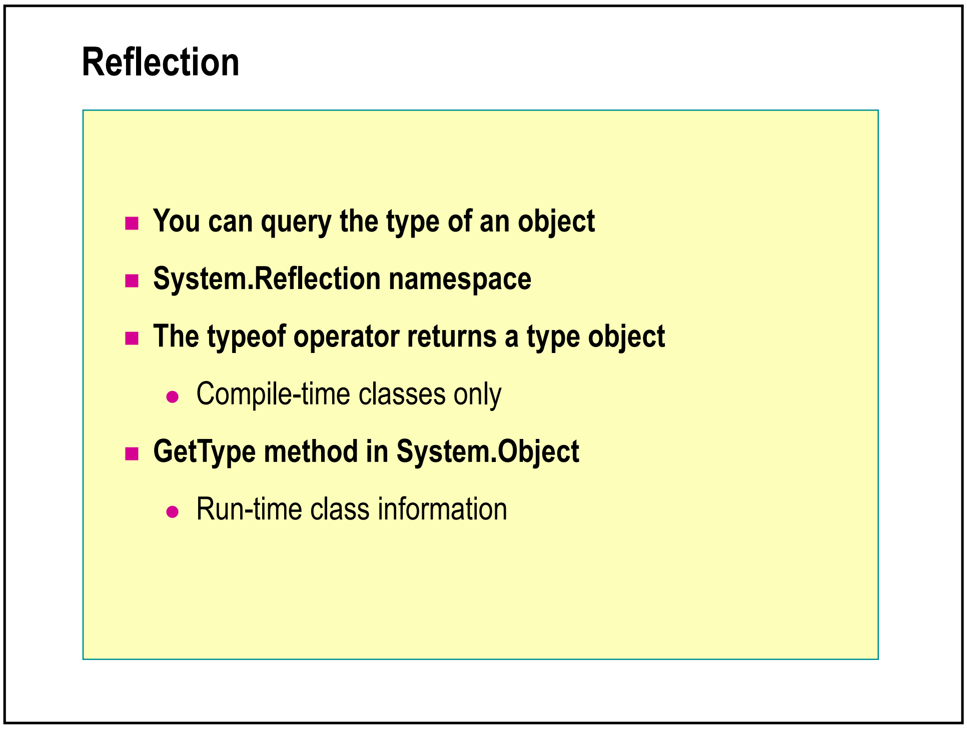
The object Type

- Synonym for System.Object
- Base class for all classes



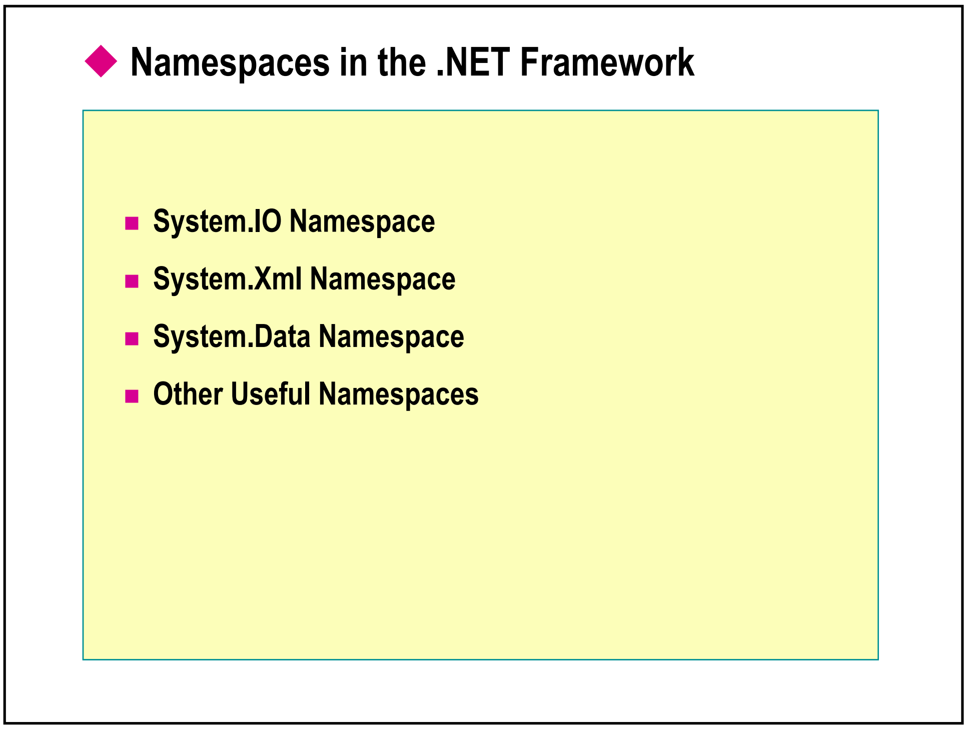
Common Methods

- Common methods for all reference types
  - **ToString** method
  - **Equals** method
  - **GetType** method
  - **Finalize** method



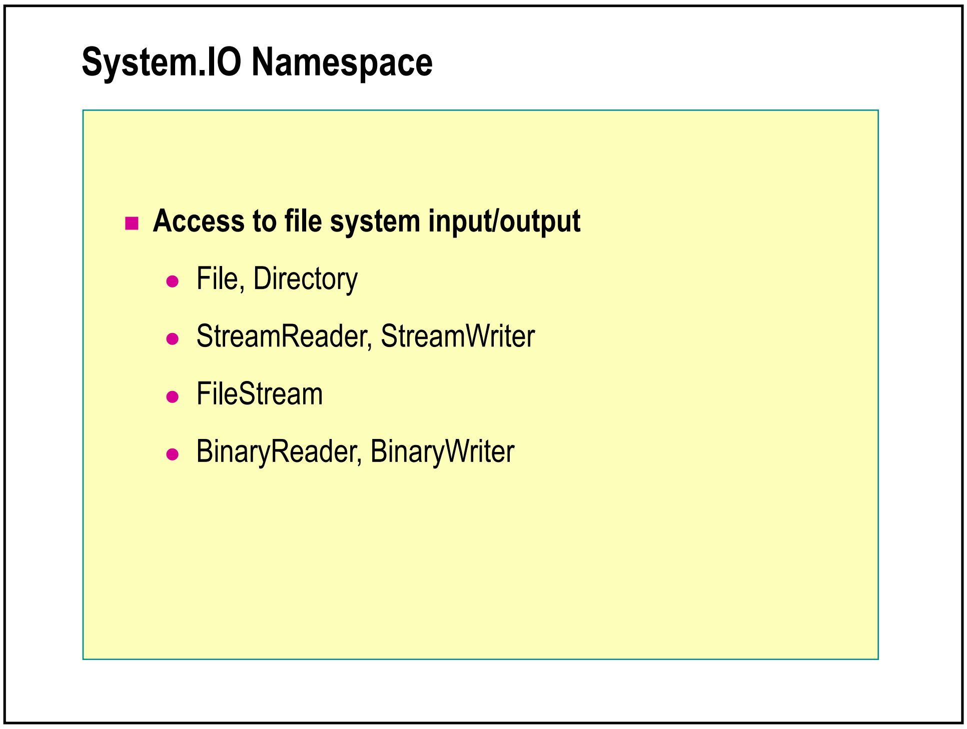
Reflection

- You can query the type of an object
- System.Reflection namespace
- The typeof operator returns a type object
  - Compile-time classes only
- GetType method in System.Object
  - Run-time class information



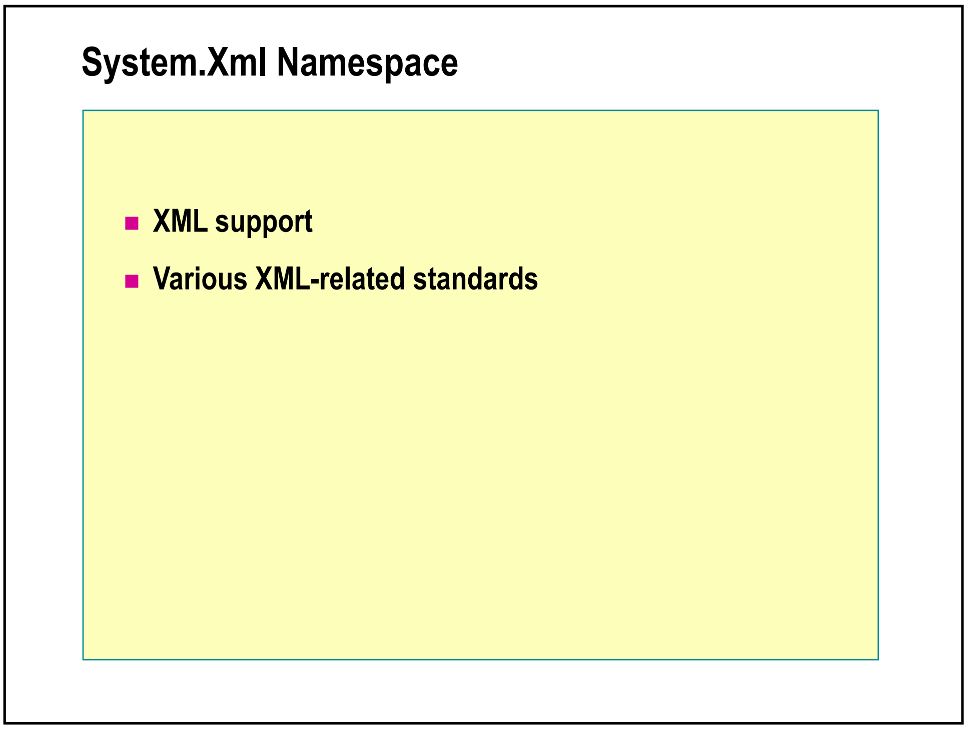
◆ Namespaces in the .NET Framework

- System.IO Namespace
- System.Xml Namespace
- System.Data Namespace
- Other Useful Namespaces



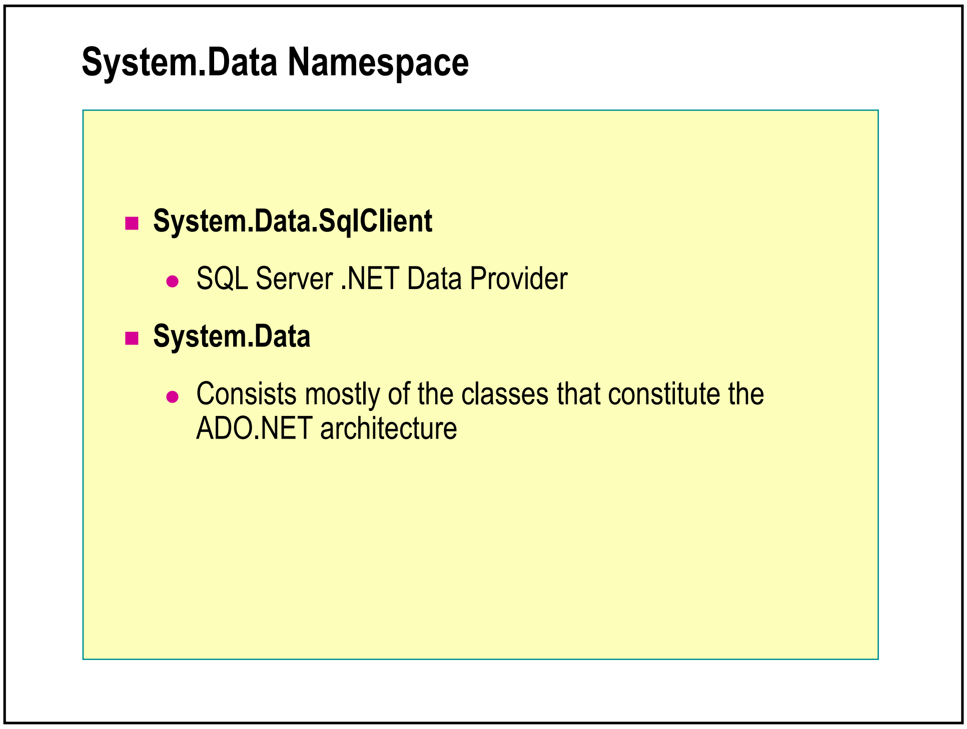
System.IO Namespace

- Access to file system input/output
  - File, Directory
  - StreamReader, StreamWriter
  - FileStream
  - BinaryReader, BinaryWriter



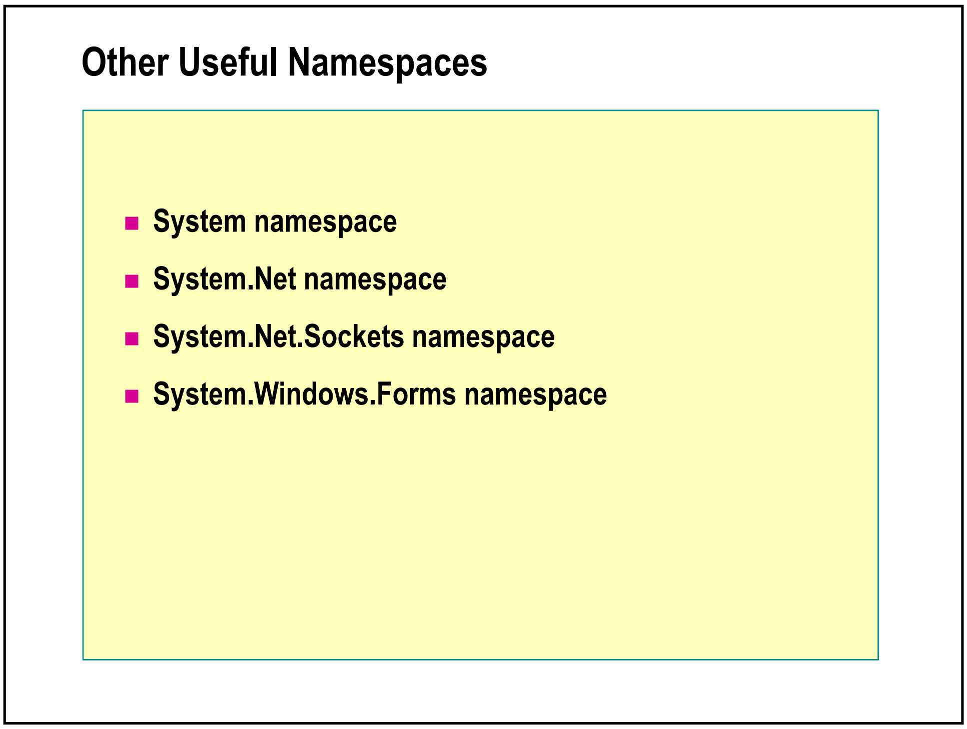
System.Xml Namespace

- XML support
- Various XML-related standards



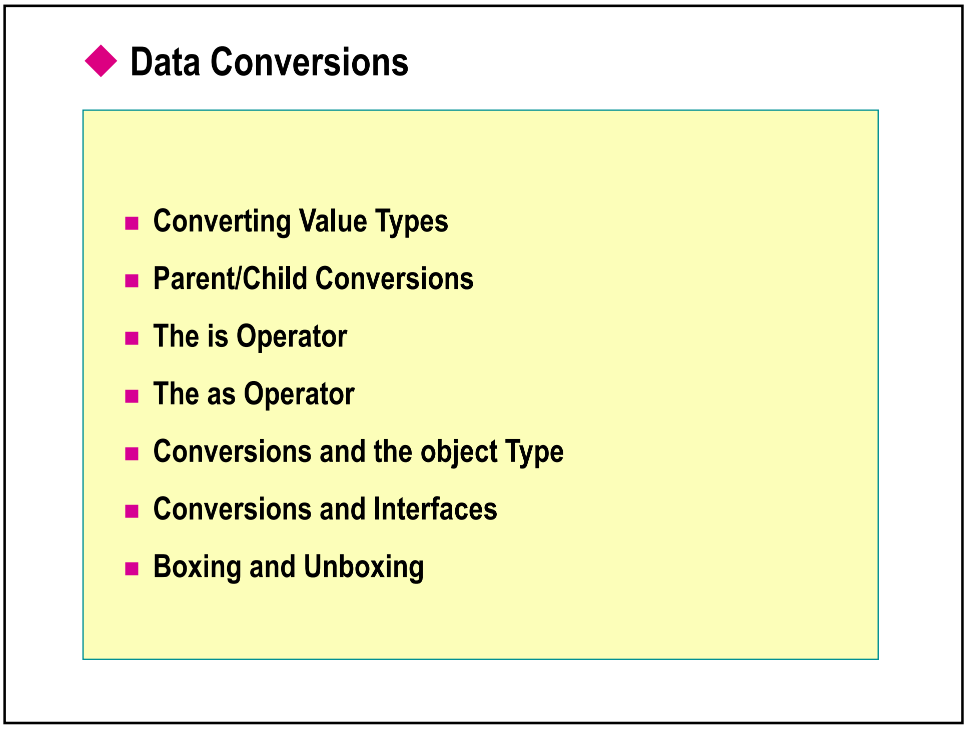
System.Data Namespace

- **System.Data.SqlClient**
  - SQL Server .NET Data Provider
- **System.Data**
  - Consists mostly of the classes that constitute the ADO.NET architecture



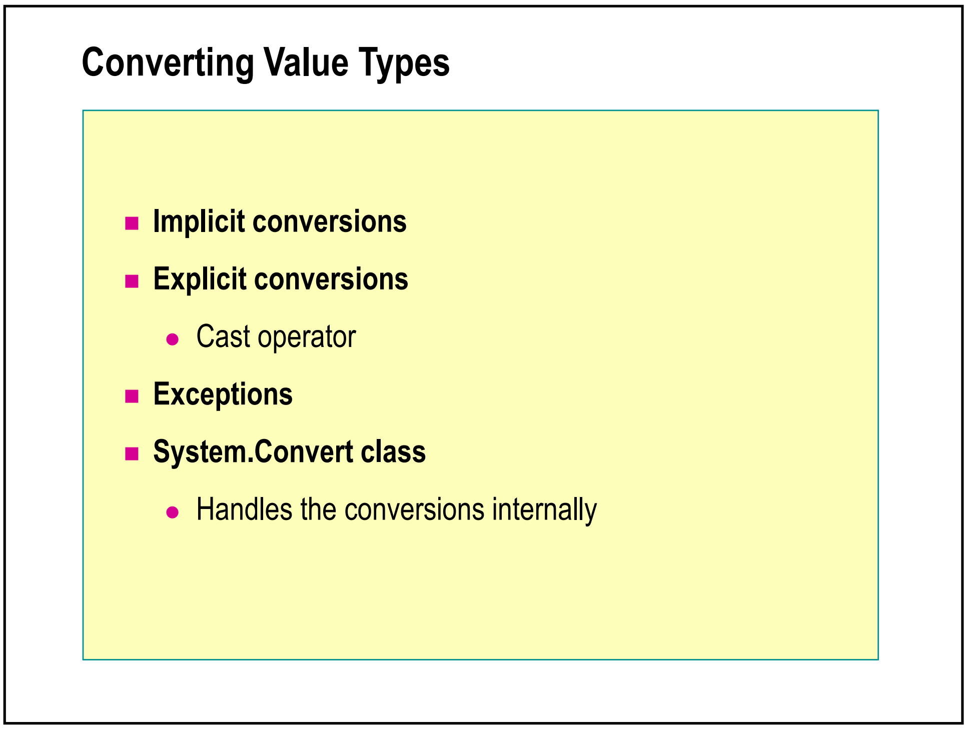
Other Useful Namespaces

- **System namespace**
- **System.Net namespace**
- **System.Net.Sockets namespace**
- **System.Windows.Forms namespace**



Data Conversions

- Converting Value Types
- Parent/Child Conversions
- The is Operator
- The as Operator
- Conversions and the object Type
- Conversions and Interfaces
- Boxing and Unboxing



Converting Value Types

- Implicit conversions
- Explicit conversions
  - Cast operator
- Exceptions
- System.Convert class
  - Handles the conversions internally

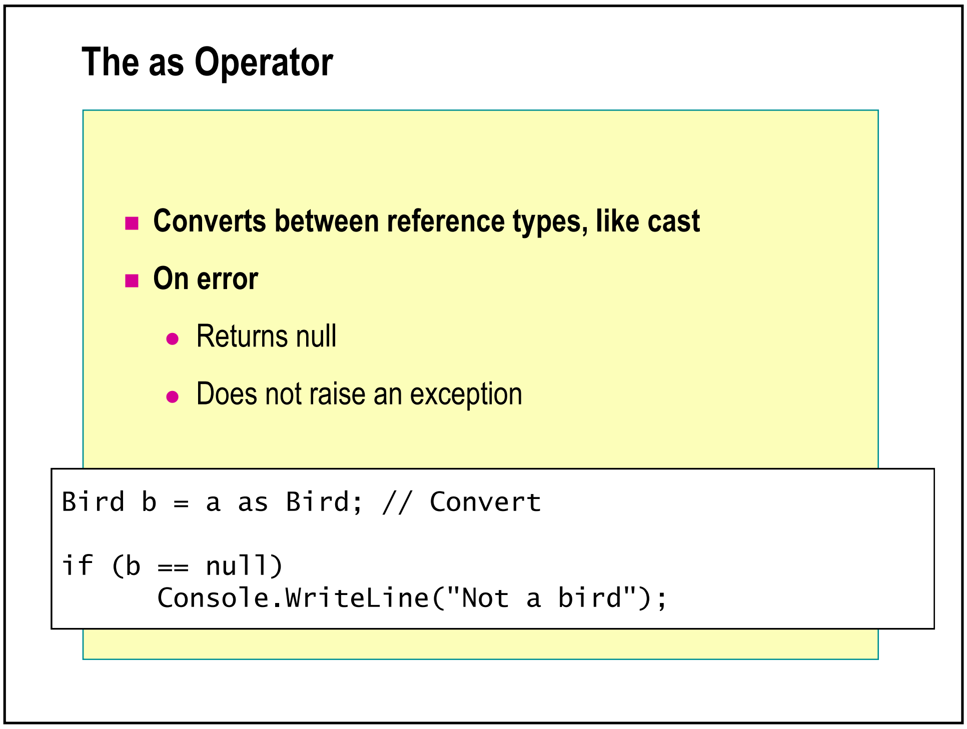
Parent/Child Conversions

- **Conversion to parent class reference**
  - Implicit or explicit
  - Always succeeds
  - Can always assign to object
- **Conversion to child class reference**
  - Explicit casting required
  - Will check that the reference is of the correct type
  - Will raise **InvalidCastException** if not

The is Operator

- **Returns true if a conversion can be made**

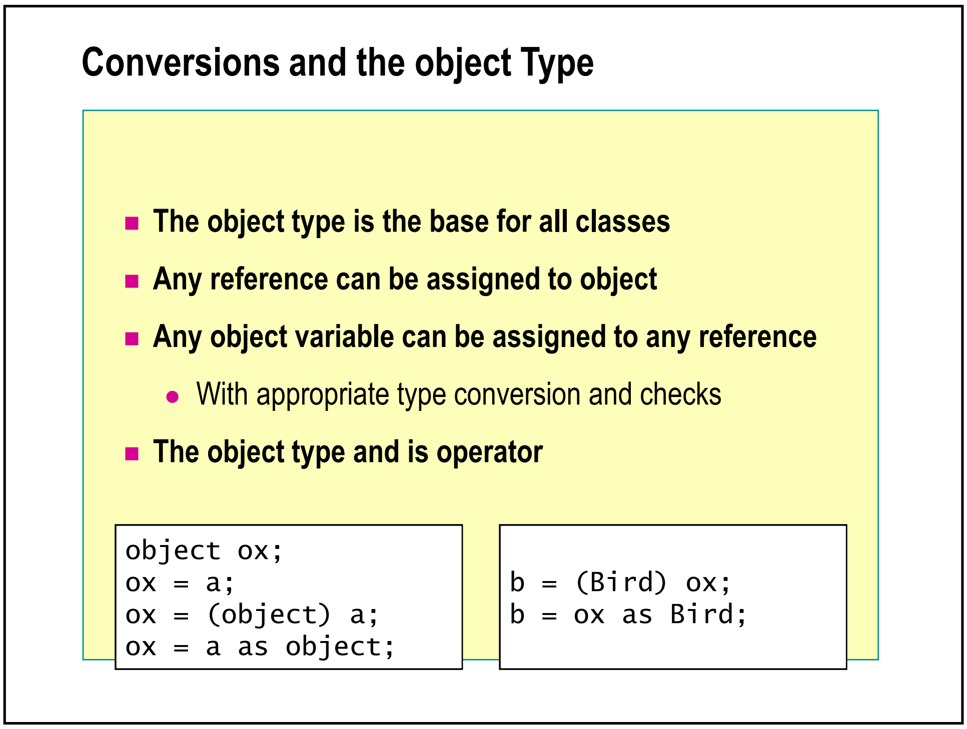
```
Bird b;
if (a is Bird)
    b = (Bird) a; // Safe
else
    Console.WriteLine("Not a Bird");
```



The as Operator

- Converts between reference types, like cast
- On error
  - Returns null
  - Does not raise an exception

```
Bird b = a as Bird; // Convert

if (b == null)
      Console.WriteLine("Not a bird");
```



Conversions and the object Type

- The object type is the base for all classes
- Any reference can be assigned to object
- Any object variable can be assigned to any reference
  - With appropriate type conversion and checks
- The object type and is operator

```
object ox;
ox = a;
ox = (object) a;
ox = a as object;
```

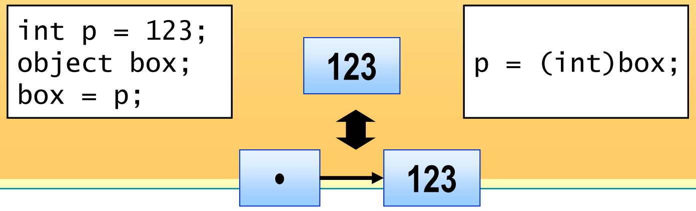
```
b = (Bird) ox;
b = ox as Bird;
```

Conversion and Interfaces

- An interface can only be used to access its own members
- Other methods and variables of the class are not accessible through the interface

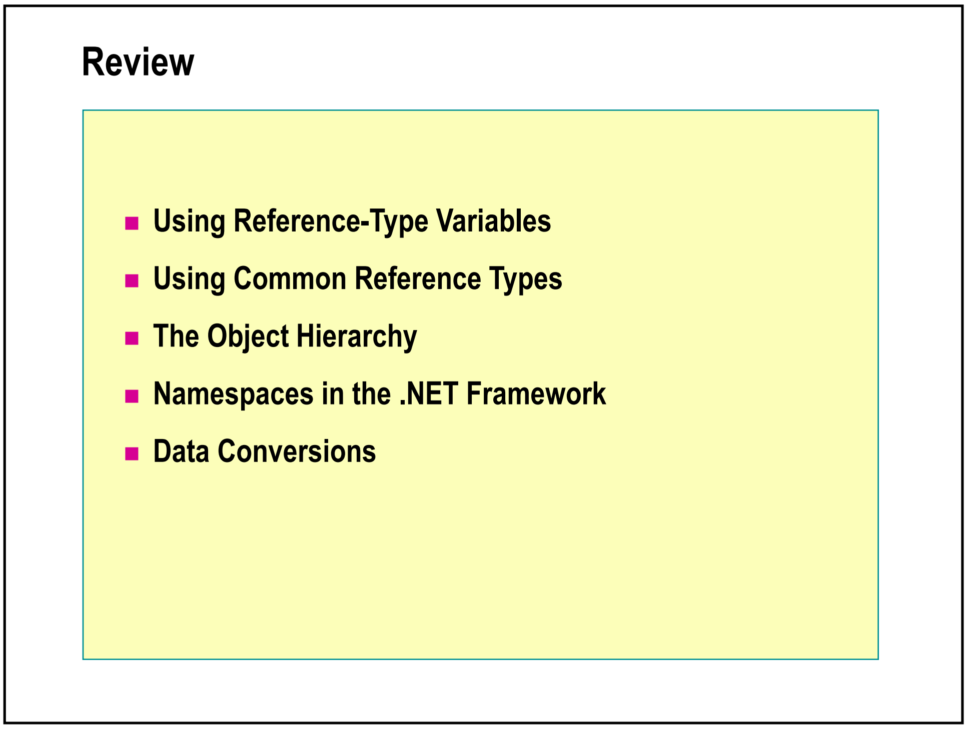
Boxing and Unboxing

- Unified type system
- Boxing
- Unboxing
- Calling object methods on value types



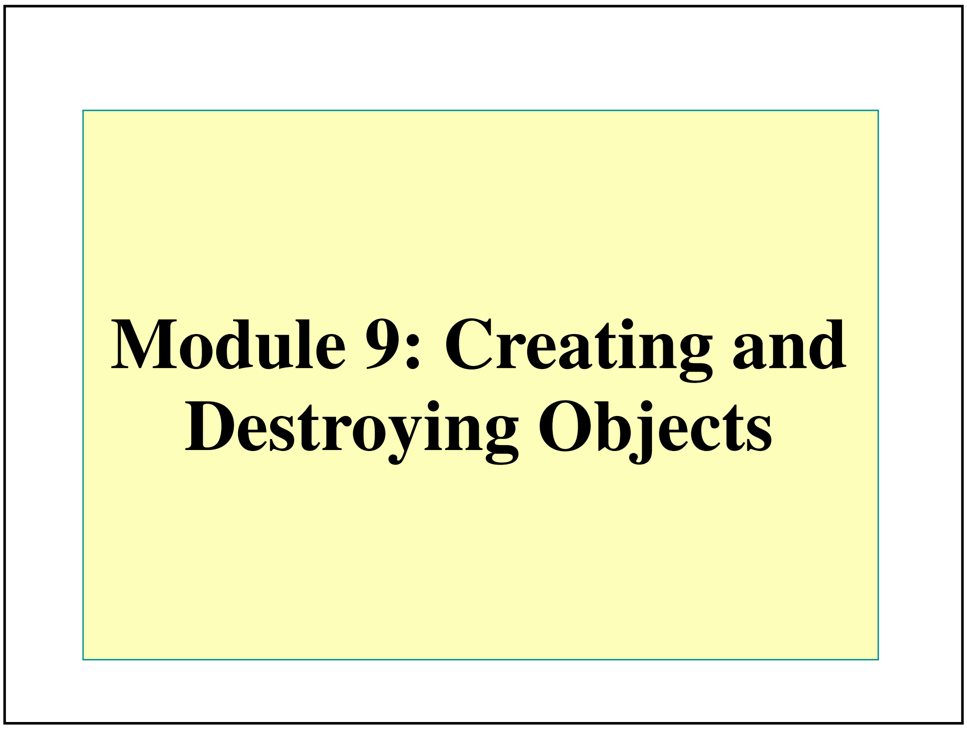
```
int p = 123;
object box;
box = p;
```

```
p = (int)box;
```



Review

- Using Reference-Type Variables
- Using Common Reference Types
- The Object Hierarchy
- Namespaces in the .NET Framework
- Data Conversions



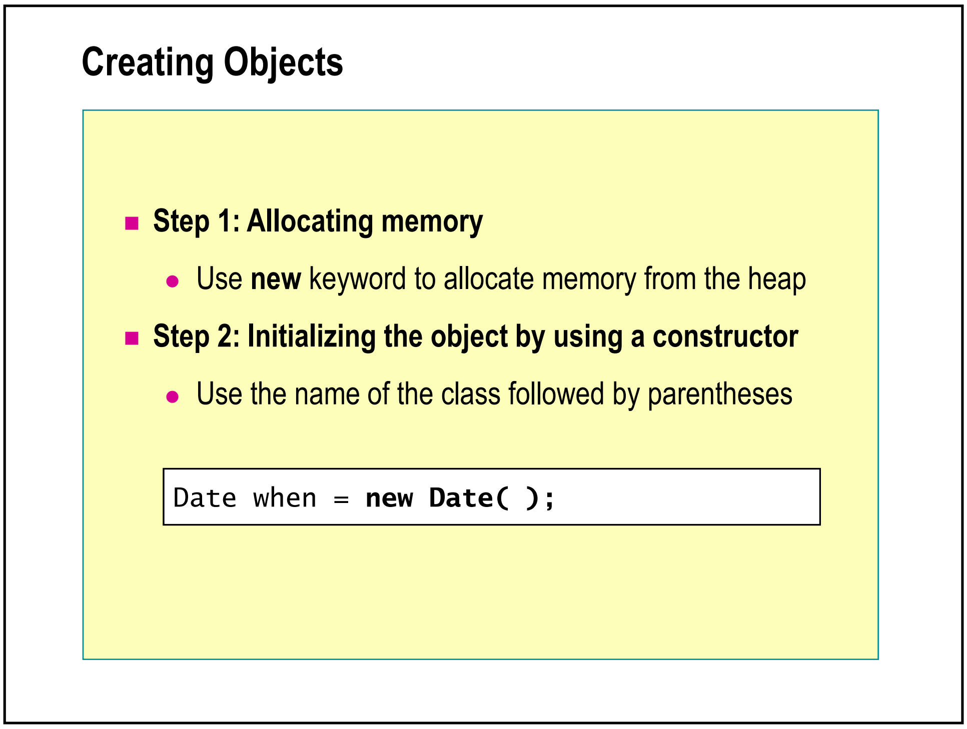
Module 9: Creating and Destroying Objects

Overview

- Using Constructors
- Initializing Data
- Objects and Memory
- Resource Management

◆ Using Constructors

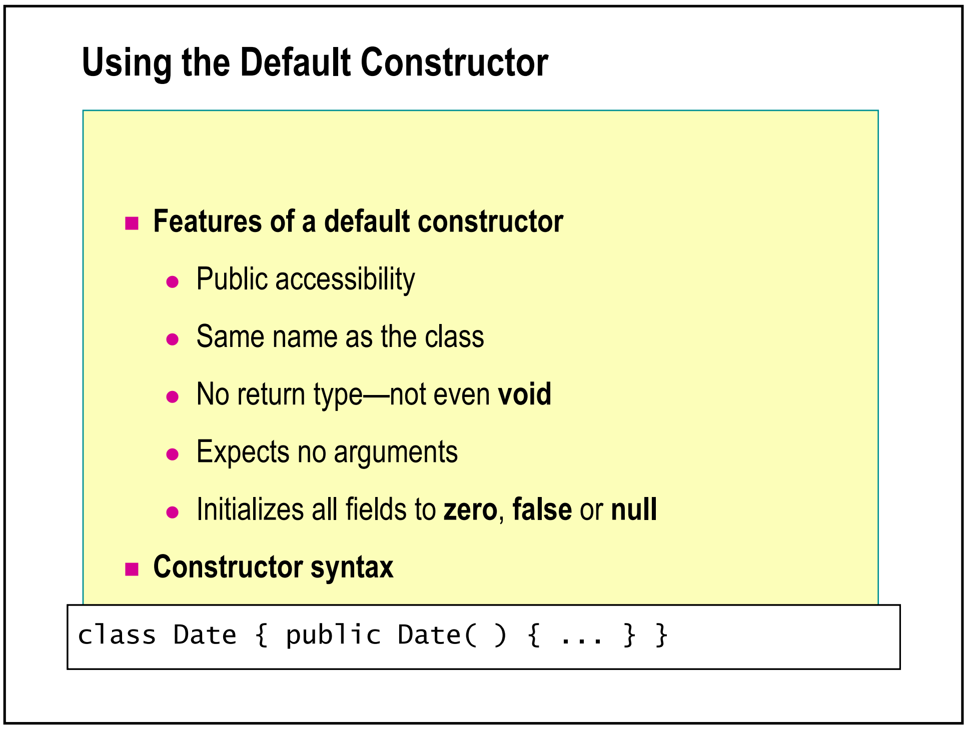
- Creating Objects
- Using the Default Constructor
- Overriding the Default Constructor
- Overloading Constructors



Creating Objects

- **Step 1: Allocating memory**
  - Use **new** keyword to allocate memory from the heap
- **Step 2: Initializing the object by using a constructor**
  - Use the name of the class followed by parentheses

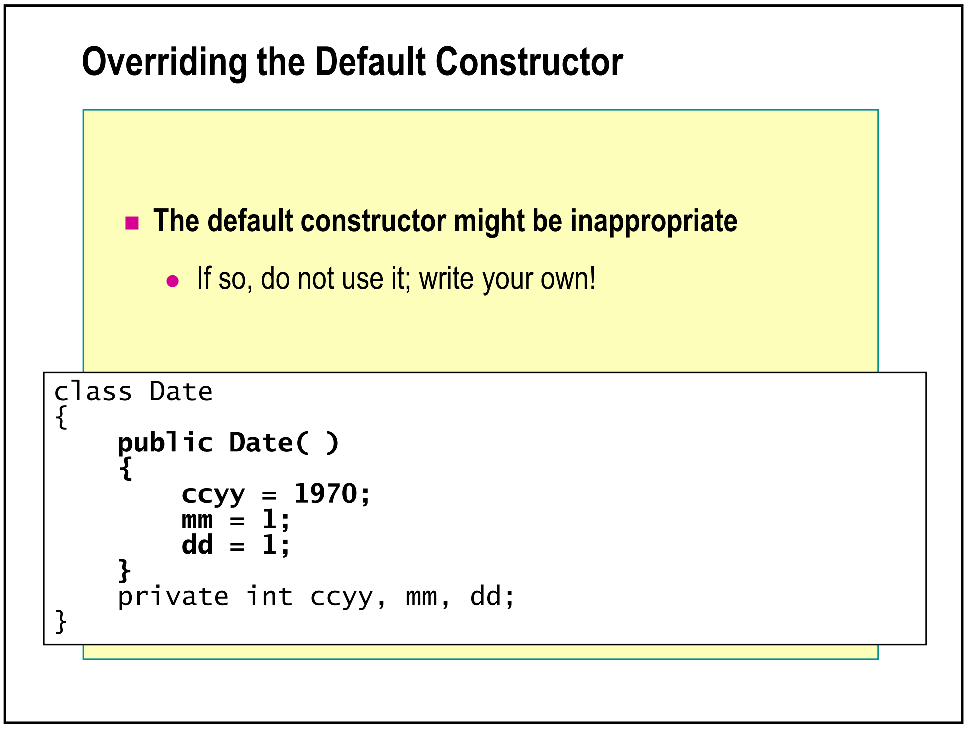
```
Date when = new Date( );
```



Using the Default Constructor

- **Features of a default constructor**
  - Public accessibility
  - Same name as the class
  - No return type—not even **void**
  - Expects no arguments
  - Initializes all fields to **zero**, **false** or **null**
- **Constructor syntax**

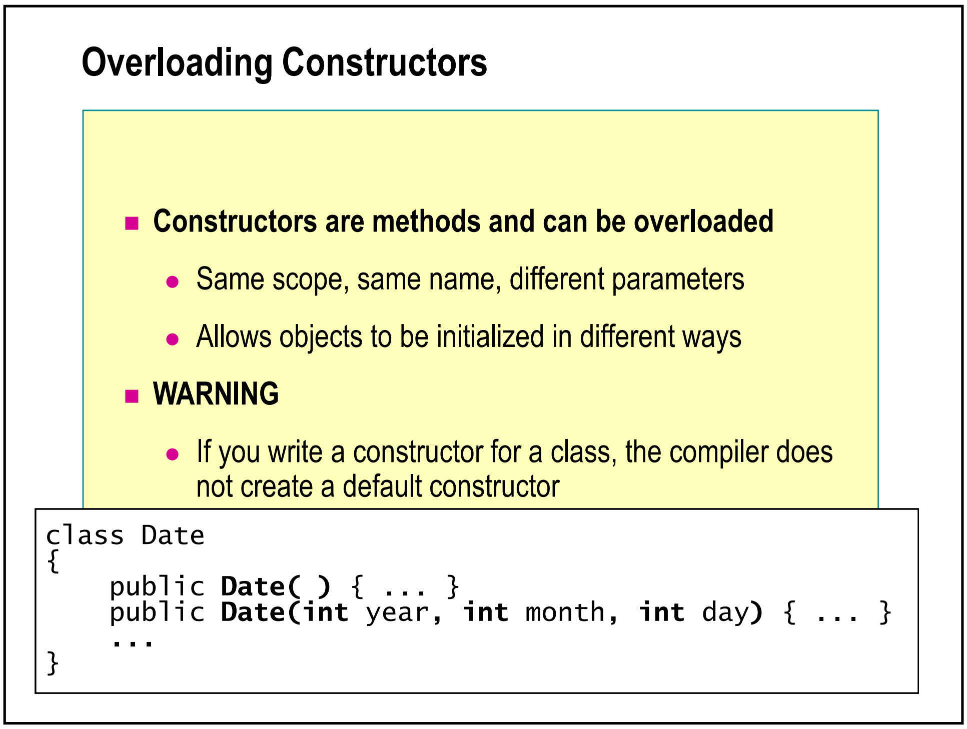
```
class Date { public Date( ) { ... } }
```



Overriding the Default Constructor

- **The default constructor might be inappropriate**
  - If so, do not use it; write your own!

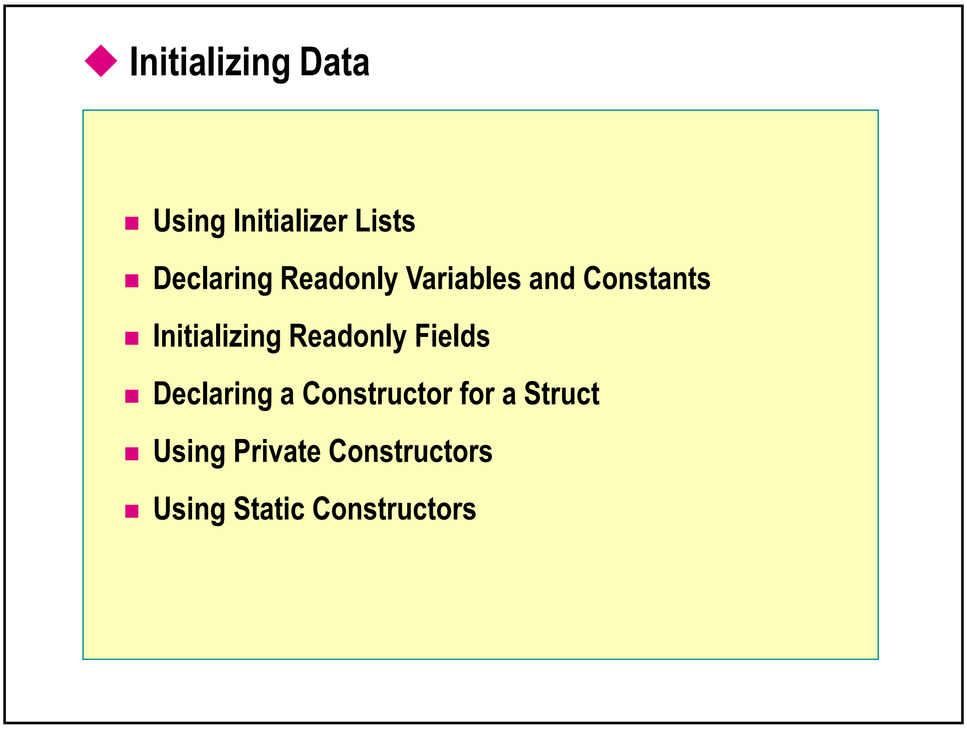
```
class Date
{
    public Date( )
    {
        ccyy = 1970;
        mm = 1;
        dd = 1;
    }
    private int ccyy, mm, dd;
}
```



Overloading Constructors

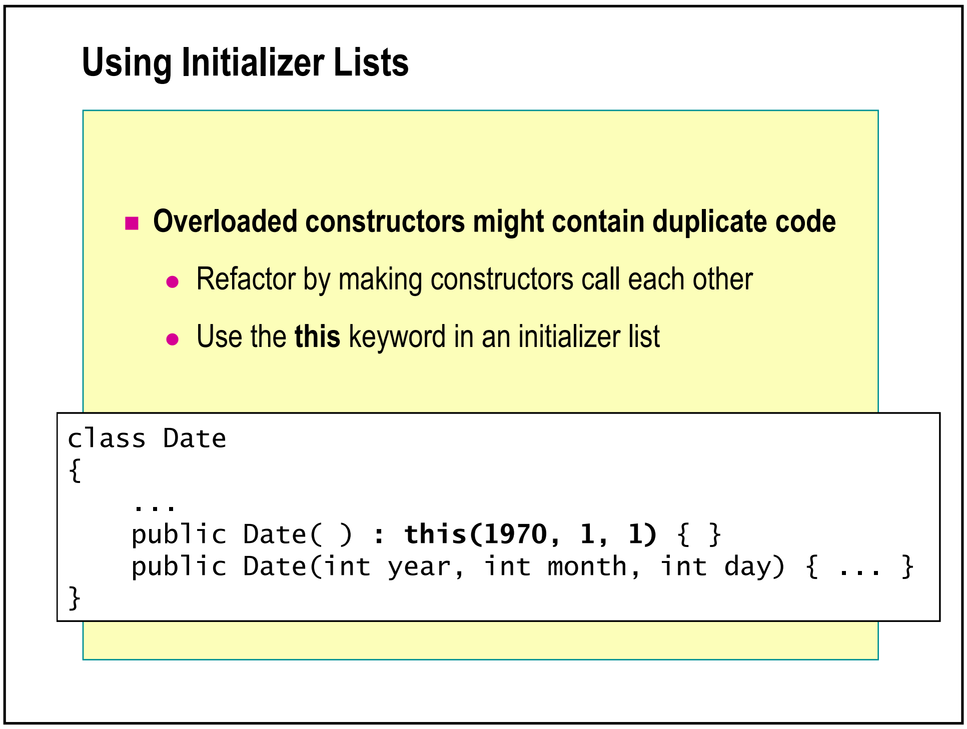
- **Constructors are methods and can be overloaded**
  - Same scope, same name, different parameters
  - Allows objects to be initialized in different ways
- **WARNING**
  - If you write a constructor for a class, the compiler does not create a default constructor

```
class Date
{
    public Date( ) { ... }
    public Date(int year, int month, int day) { ... }
    ...
}
```



◆ Initializing Data

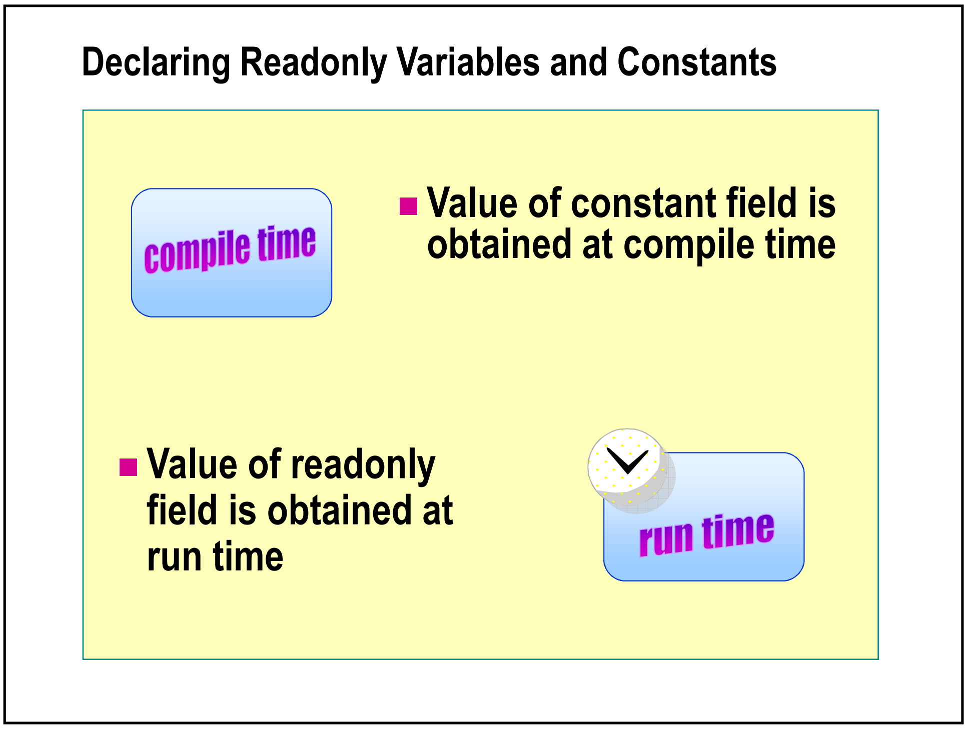
- Using Initializer Lists
- Declaring Readonly Variables and Constants
- Initializing Readonly Fields
- Declaring a Constructor for a Struct
- Using Private Constructors
- Using Static Constructors



Using Initializer Lists

- **Overloaded constructors might contain duplicate code**
  - Refactor by making constructors call each other
  - Use the **this** keyword in an initializer list

```
class Date
{
    ...
    public Date( ) : this(1970, 1, 1) { }
    public Date(int year, int month, int day) { ... }
}
```



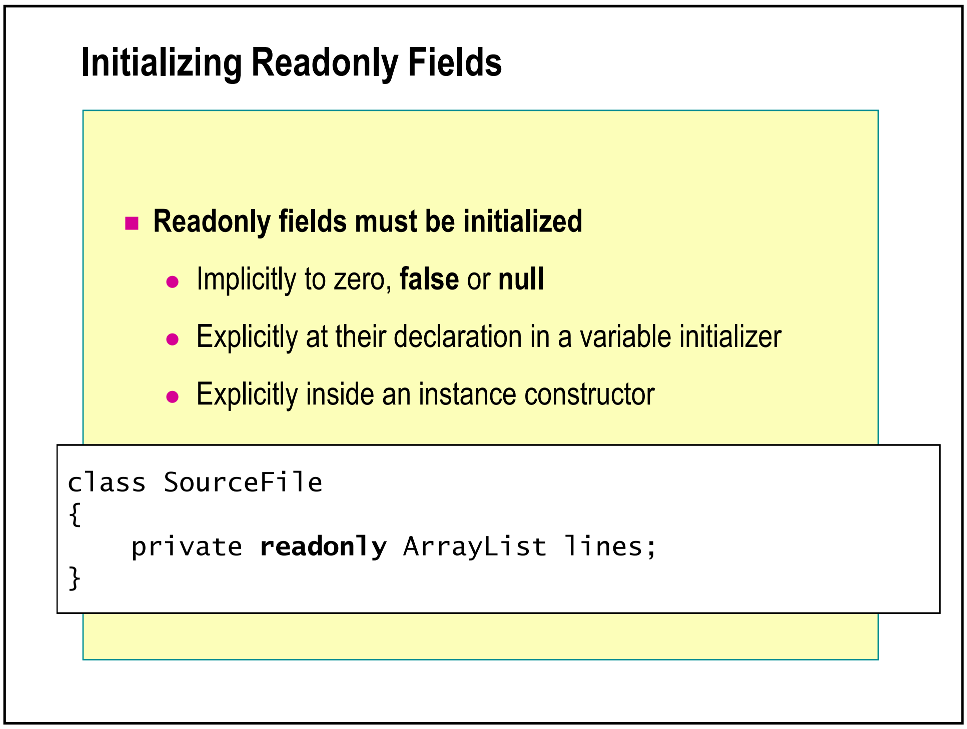
Declaring Readonly Variables and Constants

compile time

- Value of constant field is obtained at compile time

- Value of readonly field is obtained at run time

run time



Initializing Readonly Fields

- **Readonly fields must be initialized**
  - Implicitly to zero, **false** or **null**
  - Explicitly at their declaration in a variable initializer
  - Explicitly inside an instance constructor

```
class SourceFile
{
    private readonly ArrayList lines;
}
```

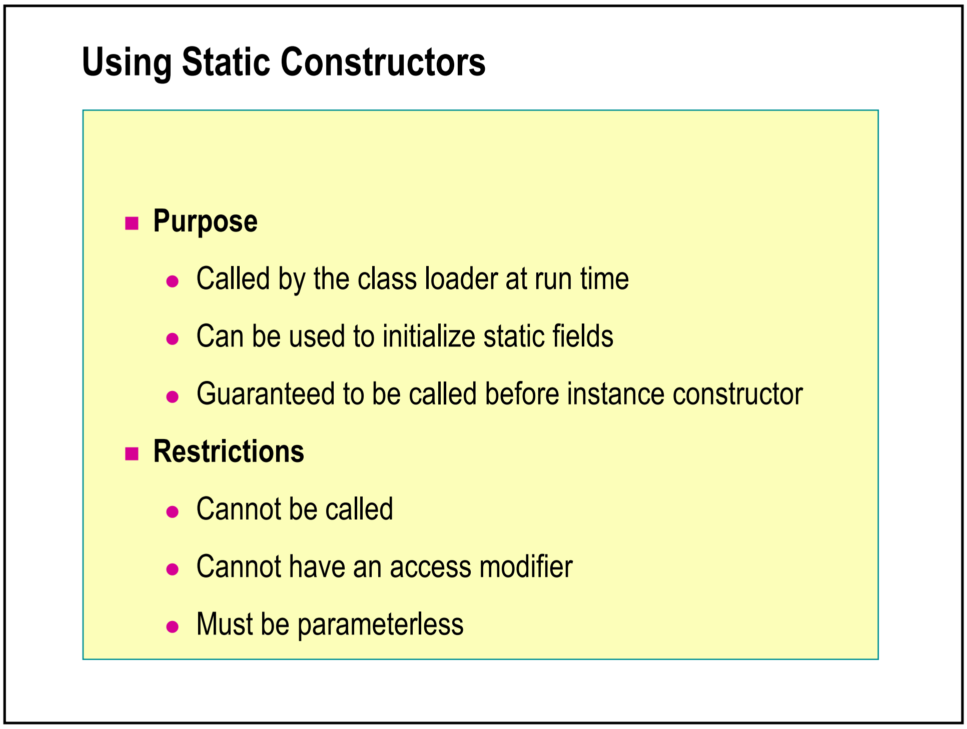
Declaring a Constructor for a Struct

- **The compiler**
  - Always generates a default constructor. Default constructors automatically initialize all fields to zero.
- **The programmer**
  - Can declare constructors with one or more arguments. Declared constructors do not automatically initialize fields to zero.
  - Can never declare a default constructor.
  - Can never declare a protected constructor.

Using Private Constructors

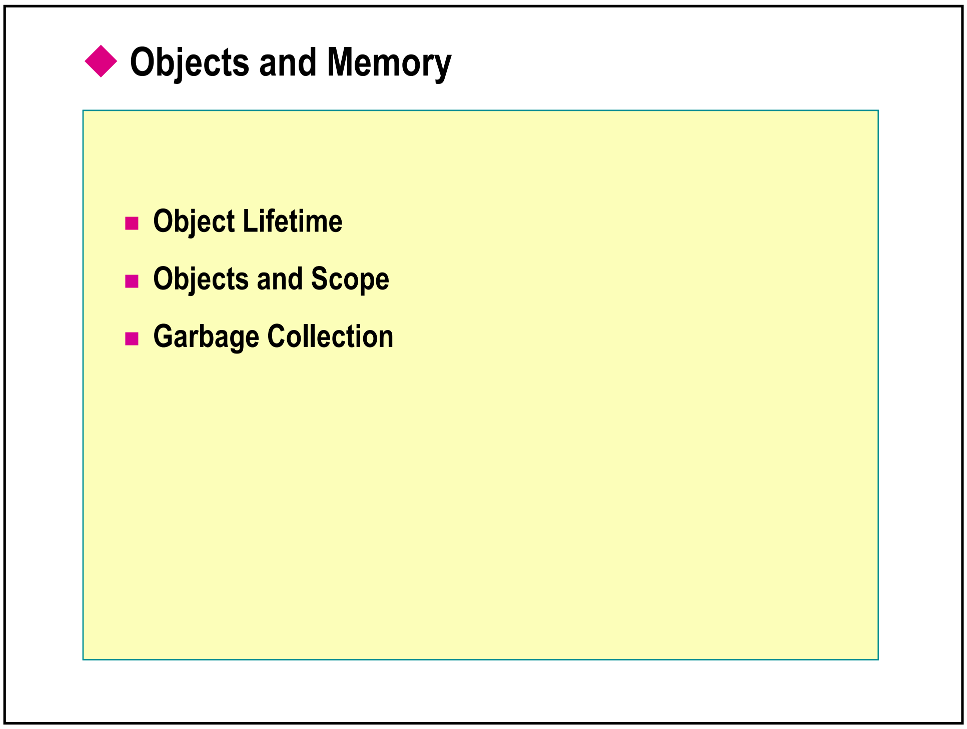
- **A private constructor prevents unwanted objects from being created**
  - Instance methods cannot be called
  - Static methods can be called
  - A useful way of implementing procedural functions

```
public class Math
{
    public static double Cos(double x) { ... }
    public static double Sin(double x) { ... }
    private Math( ) { }
}
```



Using Static Constructors

- **Purpose**
  - Called by the class loader at run time
  - Can be used to initialize static fields
  - Guaranteed to be called before instance constructor
- **Restrictions**
  - Cannot be called
  - Cannot have an access modifier
  - Must be parameterless



◆ Objects and Memory

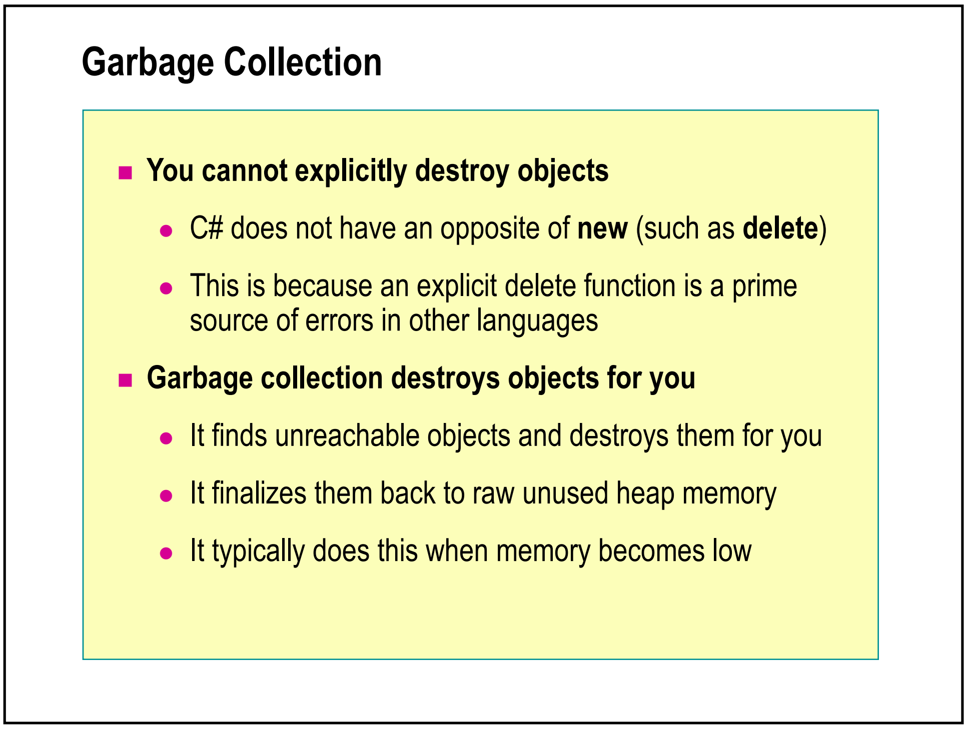
- **Object Lifetime**
- **Objects and Scope**
- **Garbage Collection**

Object Lifetime

- **Creating objects**
  - You allocate memory by using **new**
  - You initialize an object in that memory by using a constructor
- **Using objects**
  - You call methods
- **Destroying objects**
  - The object is converted back into memory
  - The memory is de-allocated

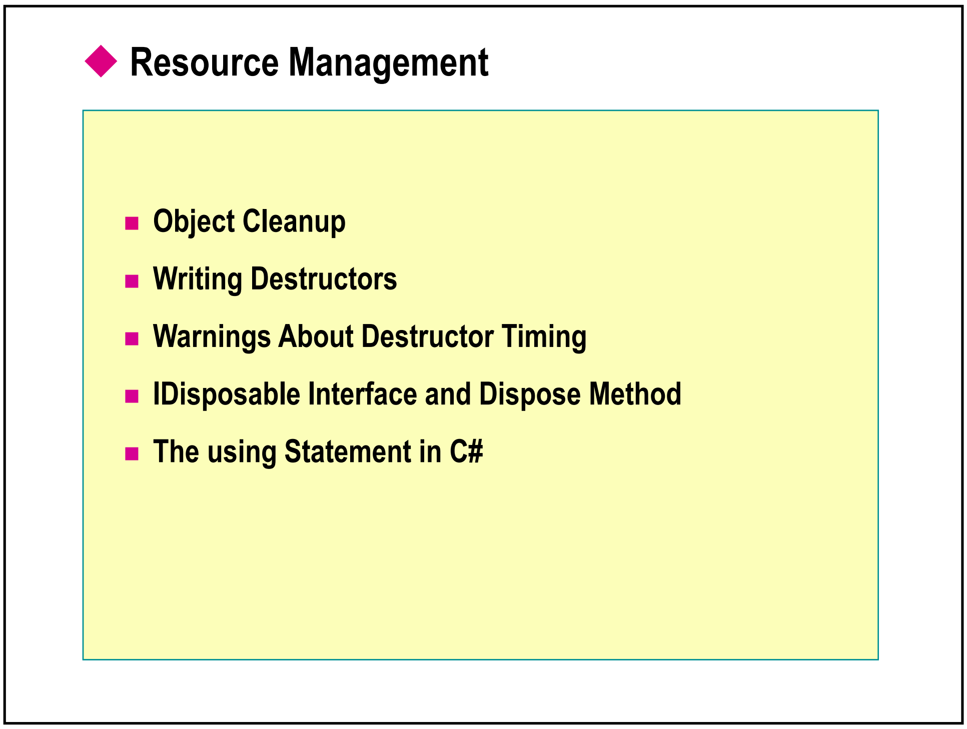
Objects and Scope

- **The lifetime of a local value is tied to the scope in which it is declared**
  - Short lifetime (typically)
  - Deterministic creation and destruction
- **The lifetime of a dynamic object is not tied to its scope**
  - A longer lifetime
  - A non-deterministic destruction



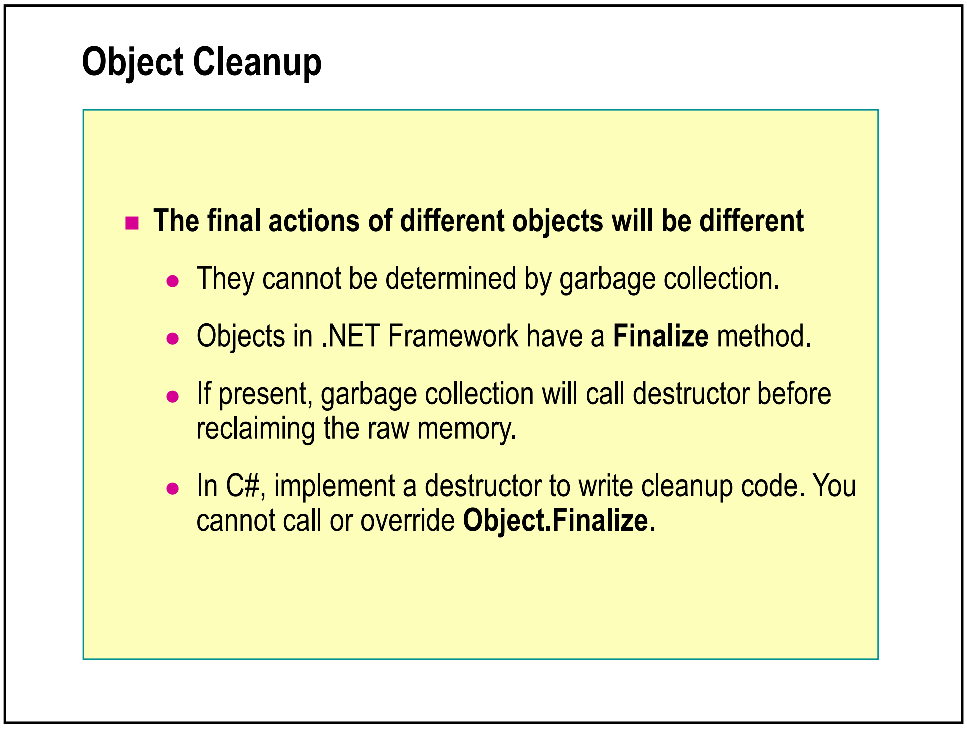
Garbage Collection

- **You cannot explicitly destroy objects**
  - C# does not have an opposite of **new** (such as **delete**)
  - This is because an explicit delete function is a prime source of errors in other languages
- **Garbage collection destroys objects for you**
  - It finds unreachable objects and destroys them for you
  - It finalizes them back to raw unused heap memory
  - It typically does this when memory becomes low



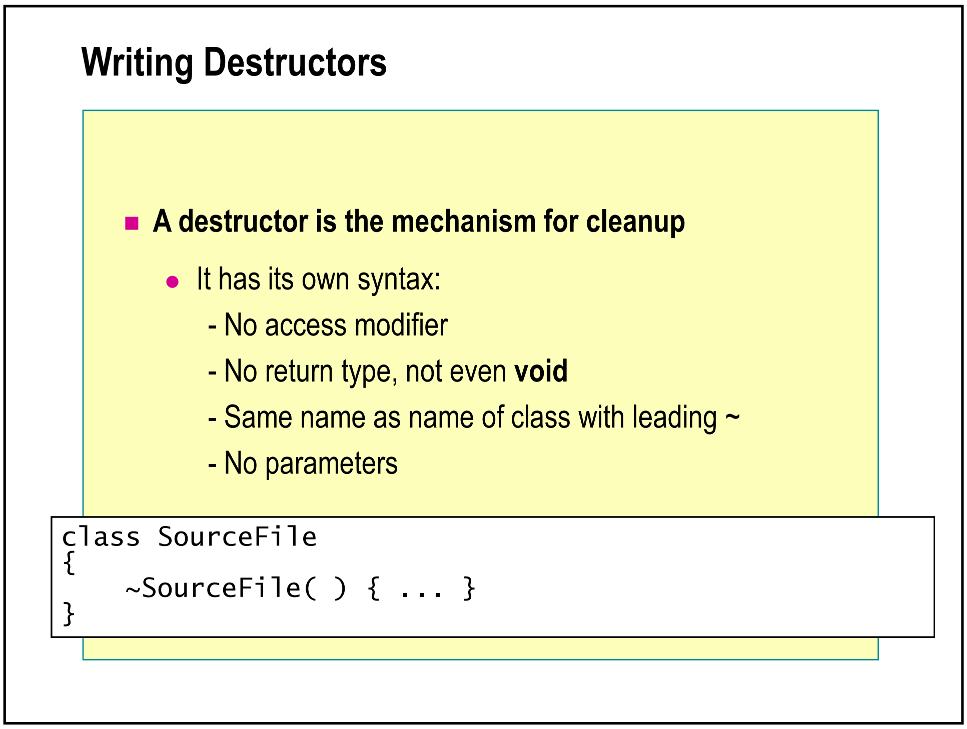
◆ Resource Management

- **Object Cleanup**
- **Writing Destructors**
- **Warnings About Destructor Timing**
- **IDisposable Interface and Dispose Method**
- **The using Statement in C#**



Object Cleanup

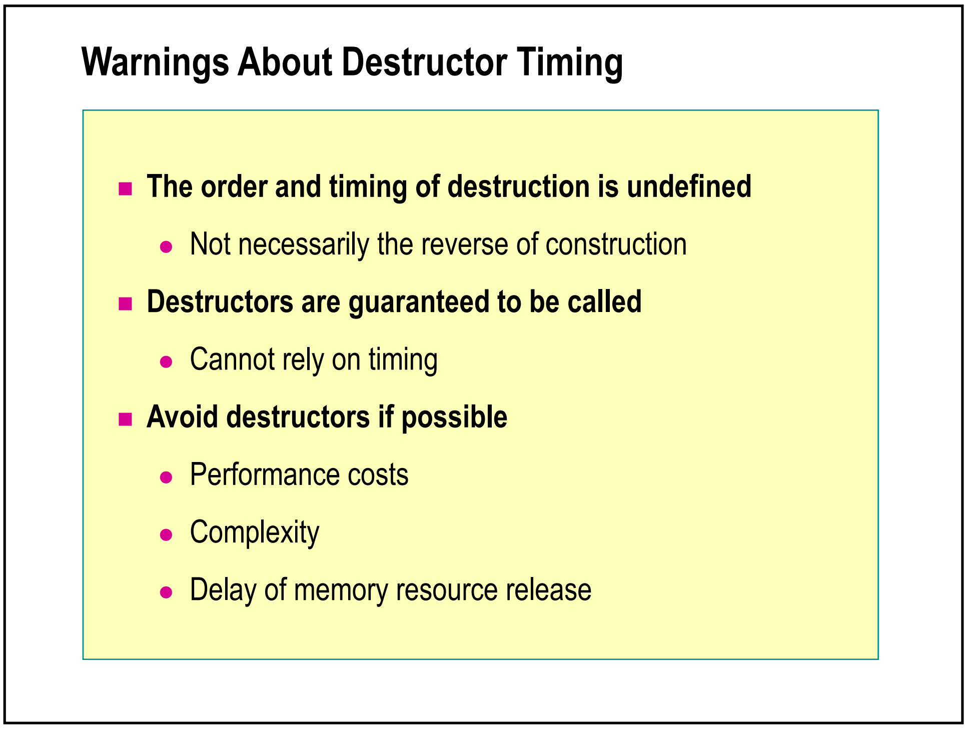
- **The final actions of different objects will be different**
  - They cannot be determined by garbage collection.
  - Objects in .NET Framework have a **Finalize** method.
  - If present, garbage collection will call destructor before reclaiming the raw memory.
  - In C#, implement a destructor to write cleanup code. You cannot call or override **Object.Finalize**.



Writing Destructors

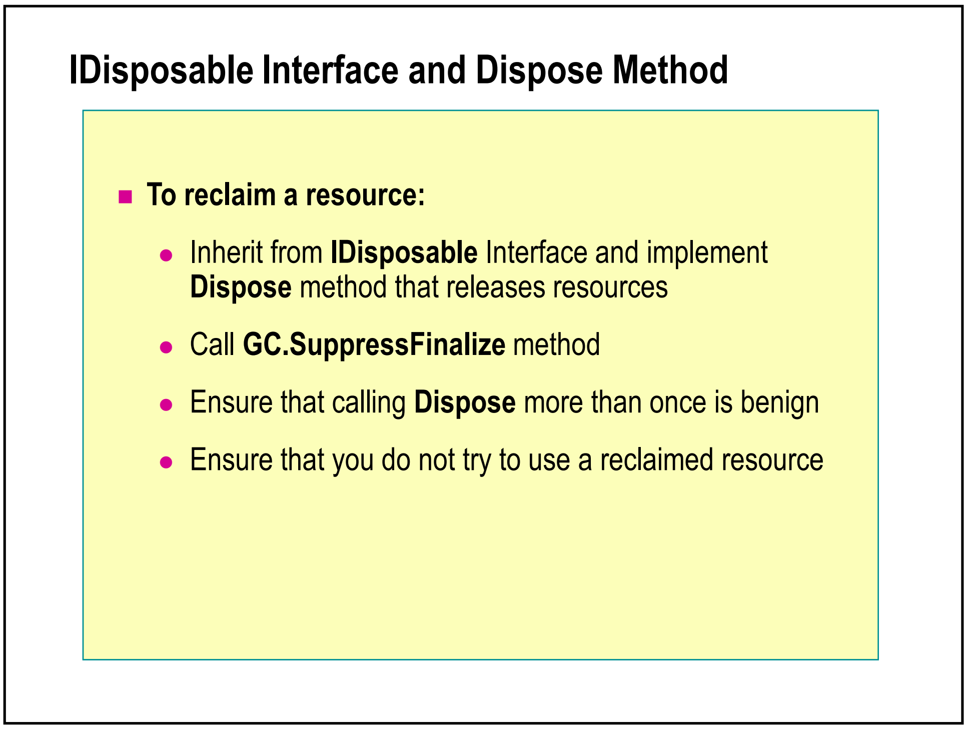
- **A destructor is the mechanism for cleanup**
  - It has its own syntax:
    - No access modifier
    - No return type, not even **void**
    - Same name as name of class with leading ~
    - No parameters

```
class SourceFile
{
    ~SourceFile( ) { ... }
}
```



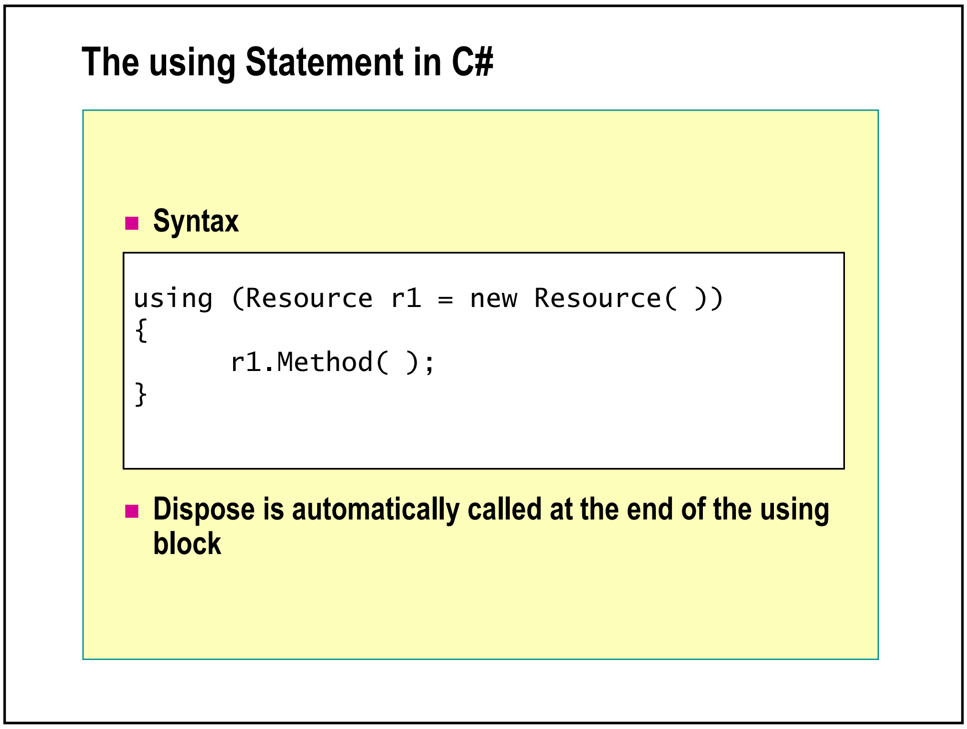
Warnings About Destructor Timing

- **The order and timing of destruction is undefined**
  - Not necessarily the reverse of construction
- **Destructors are guaranteed to be called**
  - Cannot rely on timing
- **Avoid destructors if possible**
  - Performance costs
  - Complexity
  - Delay of memory resource release



IDisposable Interface and Dispose Method

- **To reclaim a resource:**
  - Inherit from **IDisposable** Interface and implement **Dispose** method that releases resources
  - Call **GC.SuppressFinalize** method
  - Ensure that calling **Dispose** more than once is benign
  - Ensure that you do not try to use a reclaimed resource

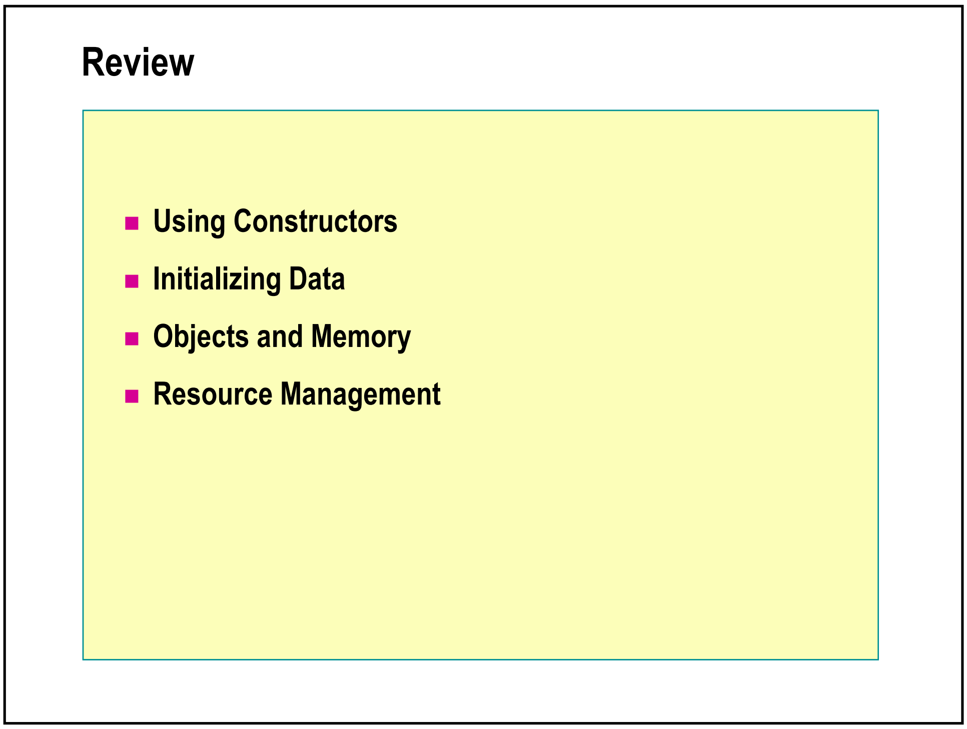


The using Statement in C#

- Syntax

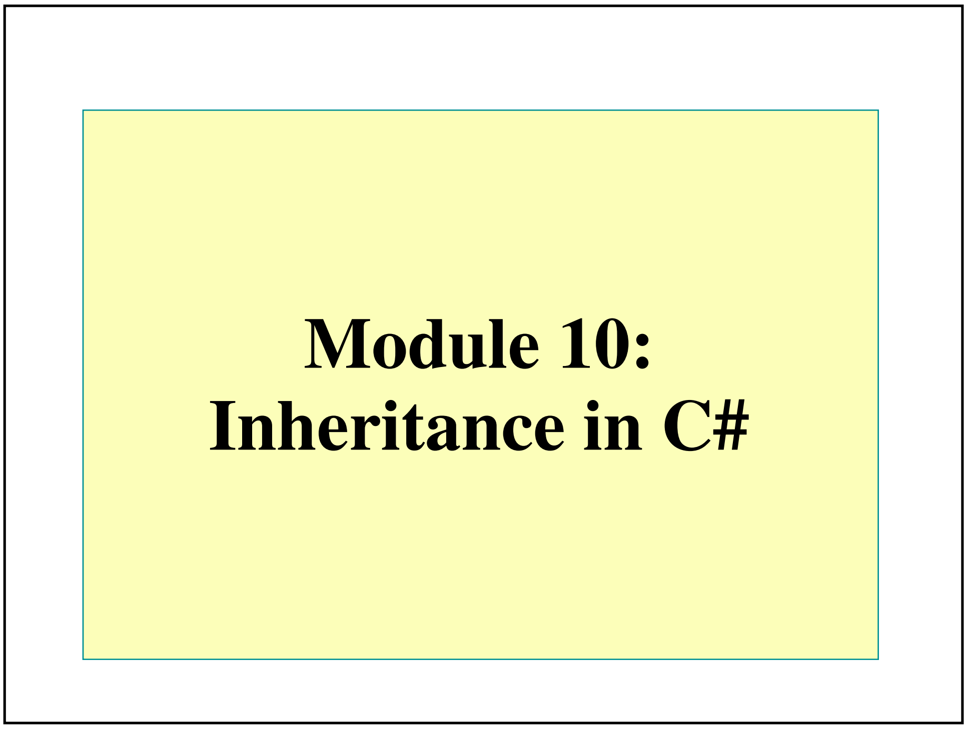
```
using (Resource r1 = new Resource( ))
{
        r1.Method( );
}
```

- Dispose is automatically called at the end of the using block

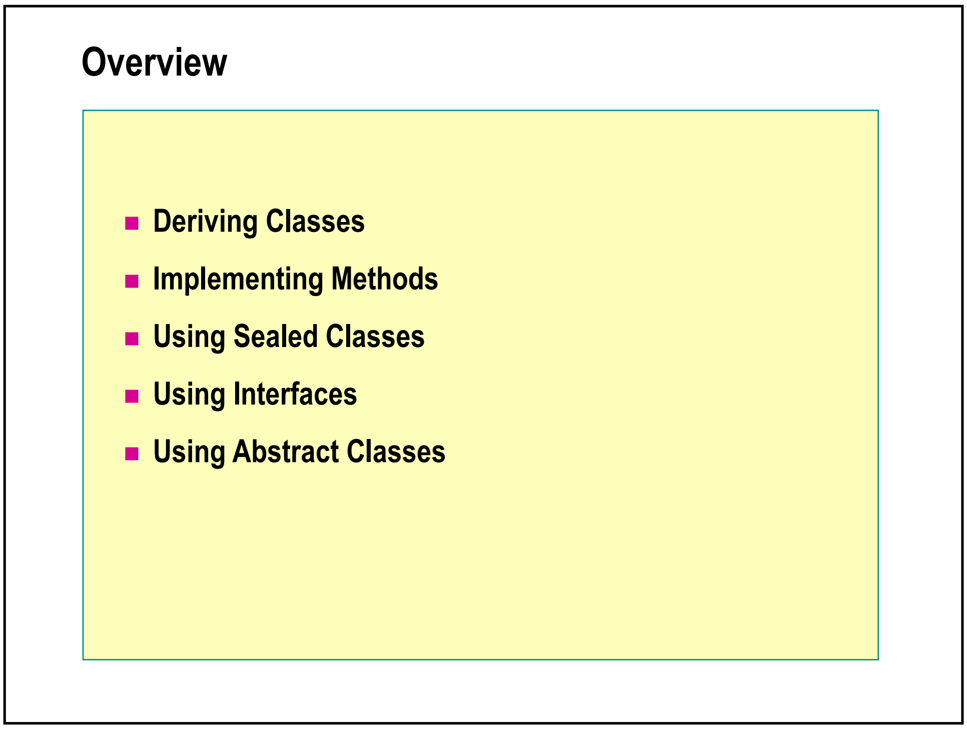


Review

- Using Constructors
- Initializing Data
- Objects and Memory
- Resource Management

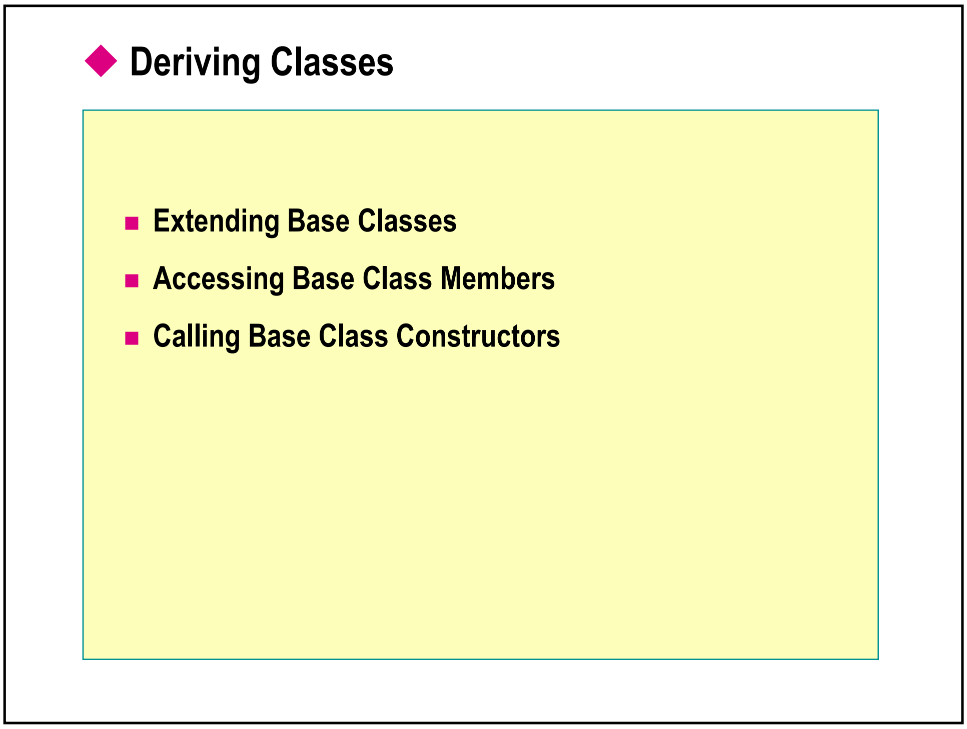


Module 10: Inheritance in C#



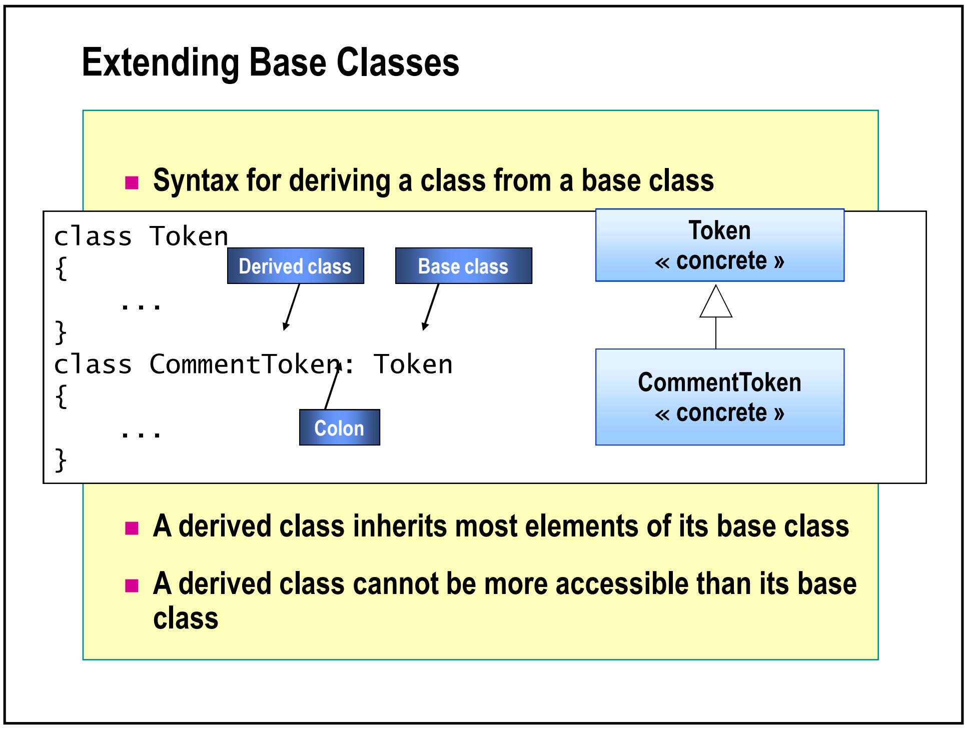
Overview

- Deriving Classes
- Implementing Methods
- Using Sealed Classes
- Using Interfaces
- Using Abstract Classes



◆ Deriving Classes

- Extending Base Classes
- Accessing Base Class Members
- Calling Base Class Constructors



Extending Base Classes

- Syntax for deriving a class from a base class

```
class Token
{
    ...
}
class CommentToken: Token
{
    ...
}
```

Derived class
Base class
Colon
Token
« concrete »
CommentToken
« concrete »

- A derived class inherits most elements of its base class
- A derived class cannot be more accessible than its base class

Accessing Base Class Members

```
class Token
{   ...
    protected string name;
}
class CommentToken: Token
{   ...
    public string Name( )
    {
        return name; ✓
    }
}
```

```
class Outside
{
    void Fails(Token t)
    {
       ...
        t.name ✗
       ...
    }
}
```

- Inherited protected members are implicitly protected in the derived class
- Methods of a derived class can access only their inherited protected members
- Protected access modifiers cannot be used in a struct

Calling Base Class Constructors

- Constructor declarations must use the base keyword

```
class Token
{
    protected Token(string name) { ... }
    ...
}
class CommentToken: Token
{
    public CommentToken(string name) : base(name) { }
    ...
}
```

- A private base class constructor cannot be accessed by a derived class
- Use the base keyword to qualify identifier scope

◆ Implementing Methods

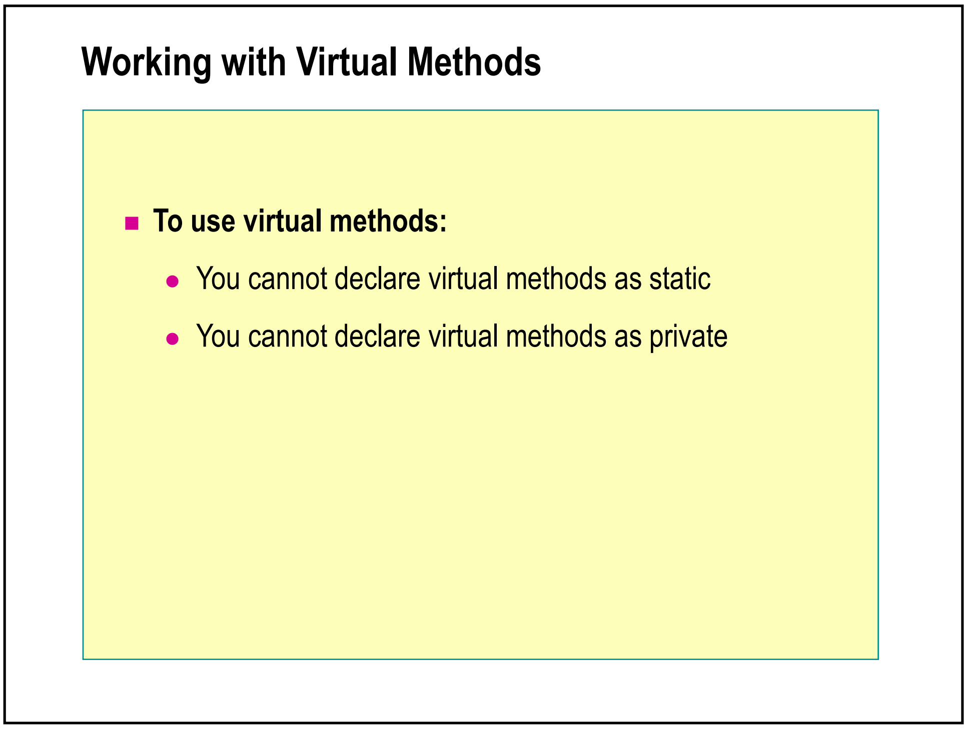
- Defining Virtual Methods
- Working with Virtual Methods
- Overriding Methods
- Working with Override Methods
- Using new to Hide Methods
- Working with the new Keyword
- Implementing Methods
- Quiz: Spot the Bugs

Defining Virtual Methods

- Syntax: Declare as virtual

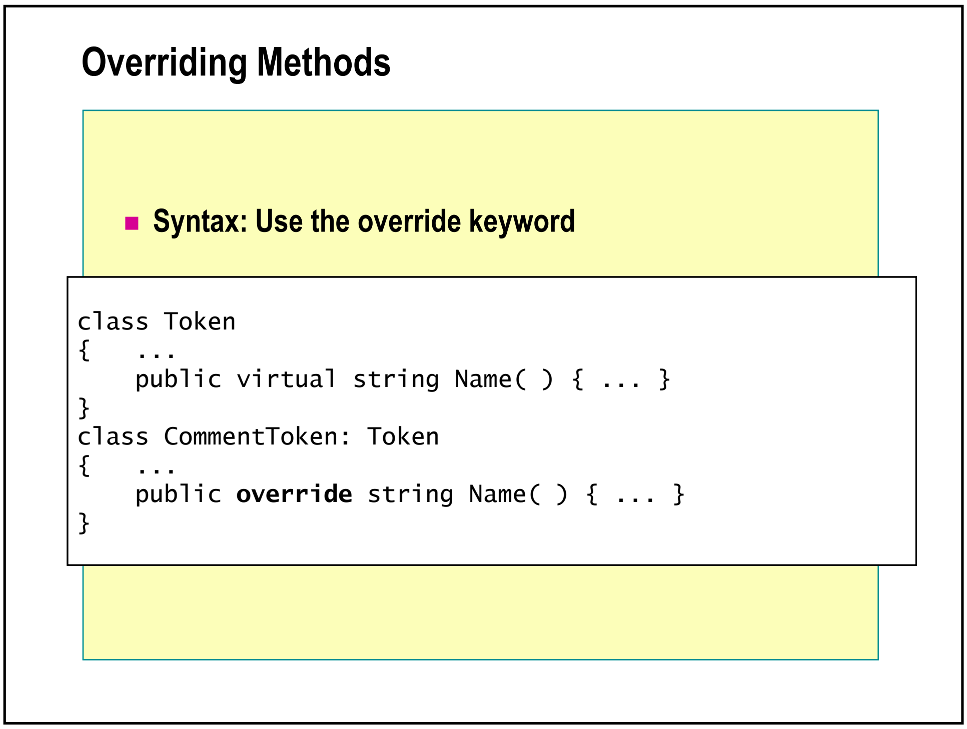
```
class Token
{
    ...
    public int LineNumber( )
    { ...
    }
    public virtual string Name( )
    { ...
    }
}
```

- Virtual methods are polymorphic



Working with Virtual Methods

- **To use virtual methods:**
  - You cannot declare virtual methods as static
  - You cannot declare virtual methods as private



Overriding Methods

- **Syntax: Use the override keyword**

```
class Token
{   ...
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{   ...
    public override string Name( ) { ... }
}
```

Working with Override Methods

- You can only override identical inherited virtual methods

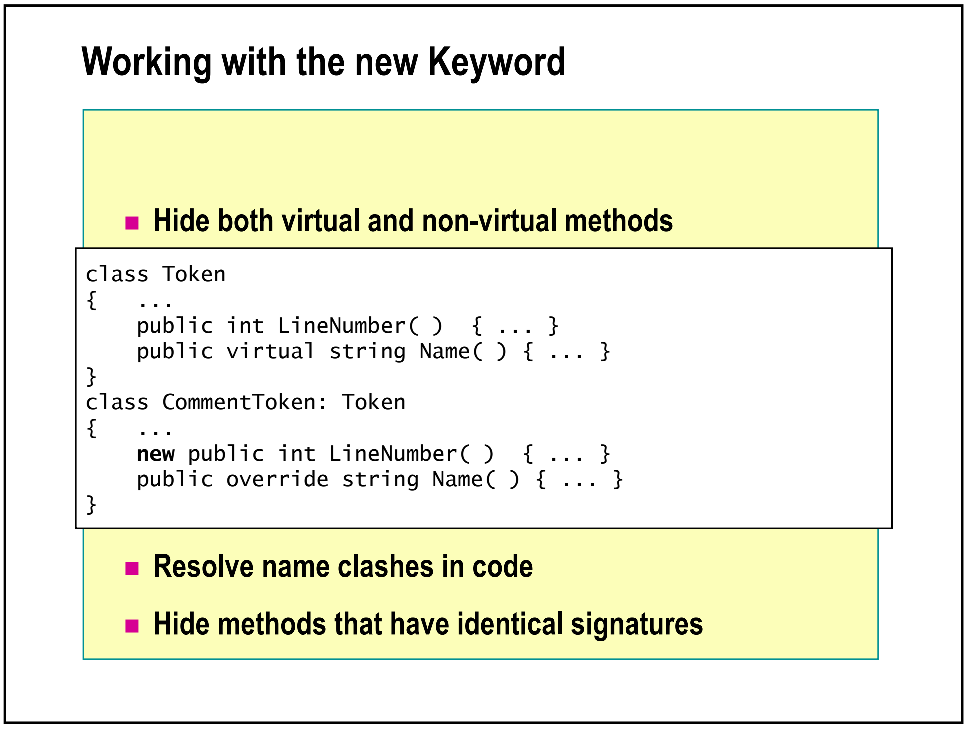
```
class Token
{   ...
    public int LineNumber( )  { ... }
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{   ...
    public override int LineNumber( ) { ... }  ✗
    public override string Name( ) { ... }     ✓
}
```

- You must match an override method with its associated virtual method
- You can override an override method
- You cannot explicitly declare an override method as virtual
- You cannot declare an override method as static or private

Using new to Hide Methods

- Syntax: Use the new keyword to hide a method

```
class Token
{   ...
    public int LineNumber( )  { ... }
}
class CommentToken: Token
{   ...
    new public int LineNumber( )  { ... }

}
```

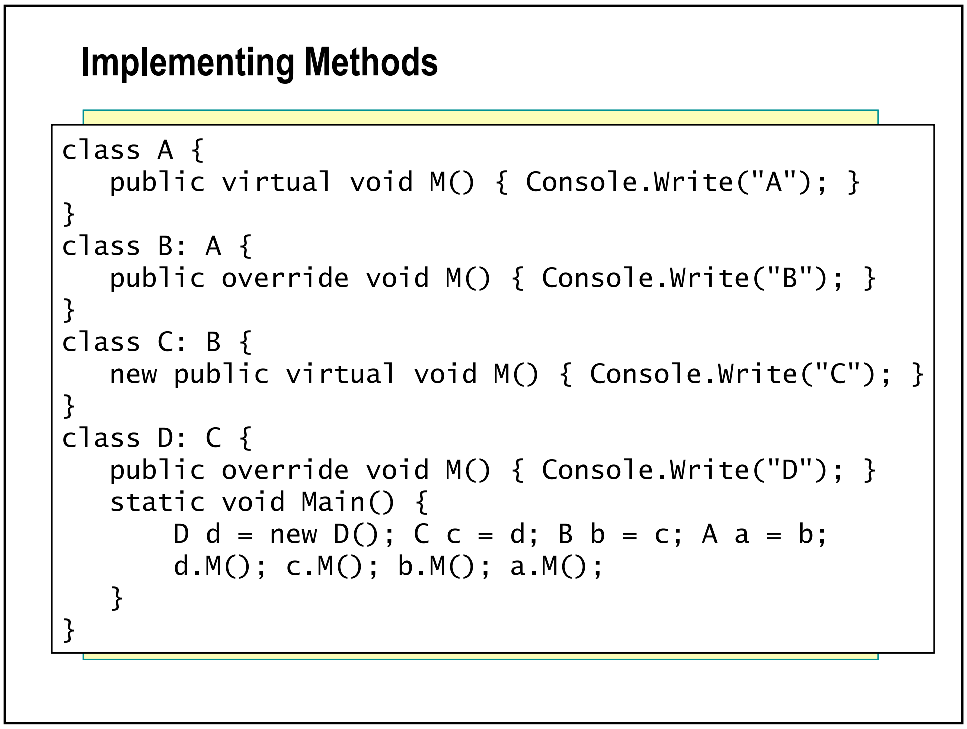


Working with the new Keyword

- Hide both virtual and non-virtual methods

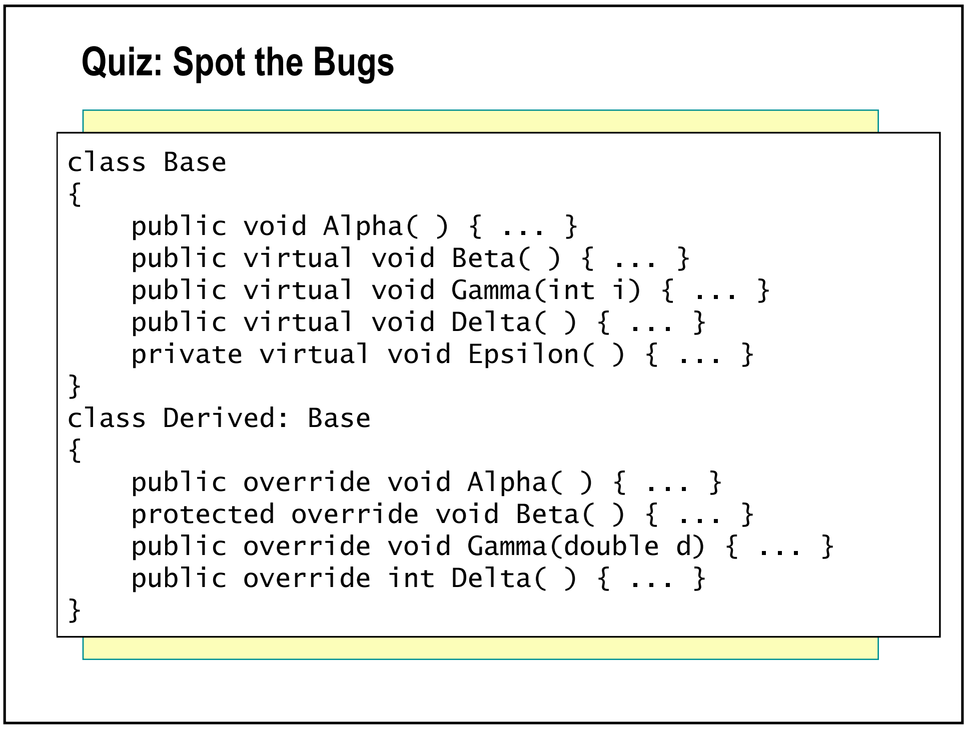
```
class Token
{   ...
    public int LineNumber( )  { ... }
    public virtual string Name( ) { ... }
}
class CommentToken: Token
{   ...
    new public int LineNumber( )  { ... }
    public override string Name( ) { ... }
}
```

- Resolve name clashes in code
- Hide methods that have identical signatures



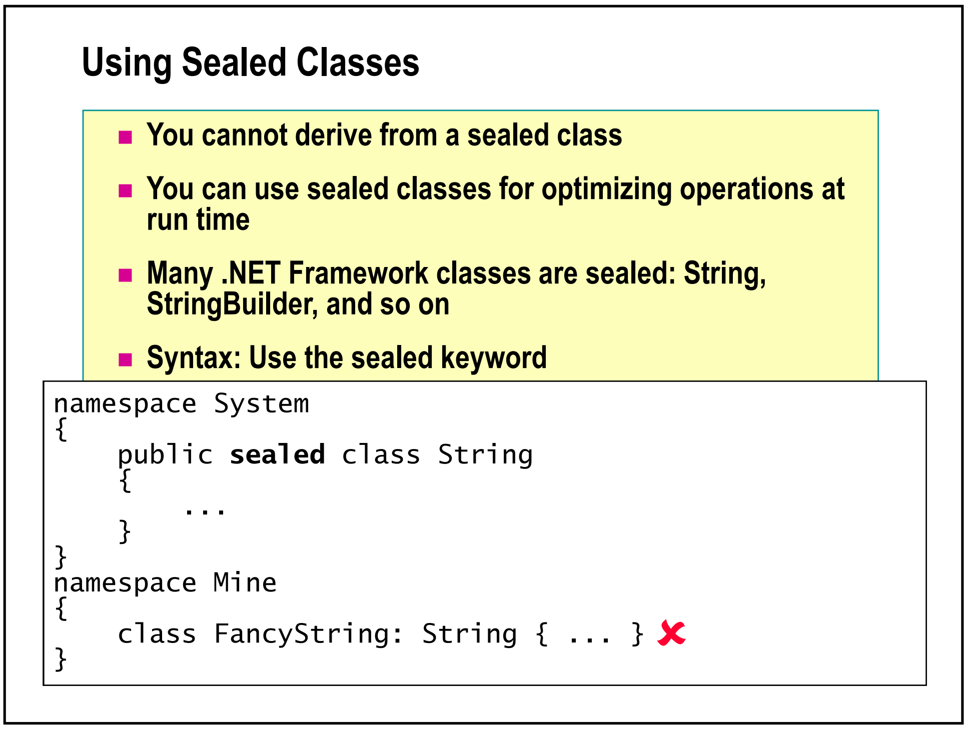
Implementing Methods

```
class A {
   public virtual void M() { Console.Write("A"); }
}
class B: A {
   public override void M() { Console.Write("B"); }
}
class C: B {
   new public virtual void M() { Console.Write("C"); }
}
class D: C {
   public override void M() { Console.Write("D"); }
   static void Main() {
       D d = new D(); C c = d; B b = c; A a = b;
       d.M(); c.M(); b.M(); a.M();
   }
}
```



Quiz: Spot the Bugs

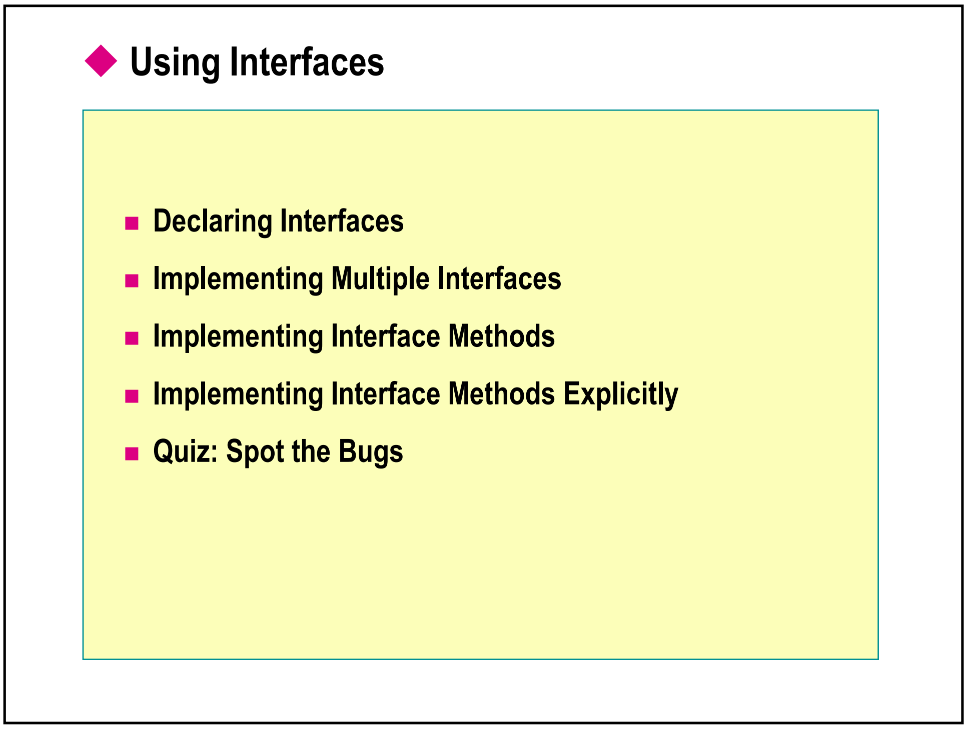
```
class Base
{
    public void Alpha( ) { ... }
    public virtual void Beta( ) { ... }
    public virtual void Gamma(int i) { ... }
    public virtual void Delta( ) { ... }
    private virtual void Epsilon( ) { ... }
}
class Derived: Base
{
    public override void Alpha( ) { ... }
    protected override void Beta( ) { ... }
    public override void Gamma(double d) { ... }
    public override int Delta( ) { ... }
}
```



Using Sealed Classes

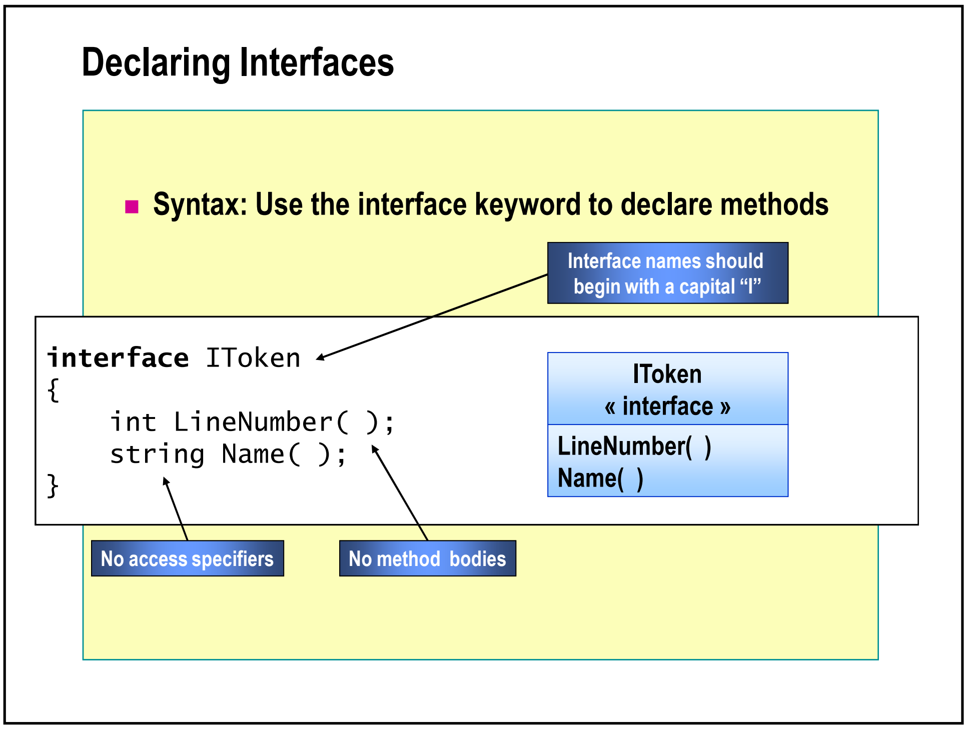
- **You cannot derive from a sealed class**
- **You can use sealed classes for optimizing operations at run time**
- **Many .NET Framework classes are sealed: String, StringBuilder, and so on**
- **Syntax: Use the sealed keyword**

```
namespace System
{
    public sealed class String
    {
        ...
    }
}
namespace Mine
{
    class FancyString: String { ... } ✗
}
```



Using Interfaces

- Declaring Interfaces
- Implementing Multiple Interfaces
- Implementing Interface Methods
- Implementing Interface Methods Explicitly
- Quiz: Spot the Bugs



Declaring Interfaces

- Syntax: Use the interface keyword to declare methods

Interface names should begin with a capital “I”

```
interface IToken
{
    int LineNumber( );
    string Name( );
}
```

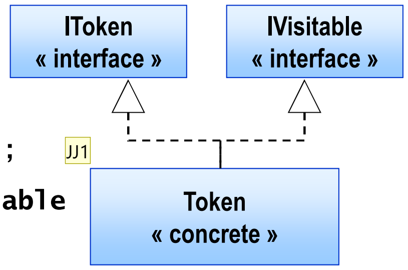
No access specifiers

No method bodies

Implementing Multiple Interfaces

- A class can implement zero or more interfaces

```
interface IToken
{
    string Name( );
}
interface IVisitable
{
    void Accept(IVisitor v);
}
class Token: IToken, IVisitable
{ ...
}
```



- An interface can extend zero or more interfaces
- A class can be more accessible than its base interfaces
- An interface cannot be more accessible than its base interfaces
- A class must implement all inherited interface methods

Implementing Interface Methods

- The implementing method must be the same as the interface method
- The implementing method can be virtual or non-virtual

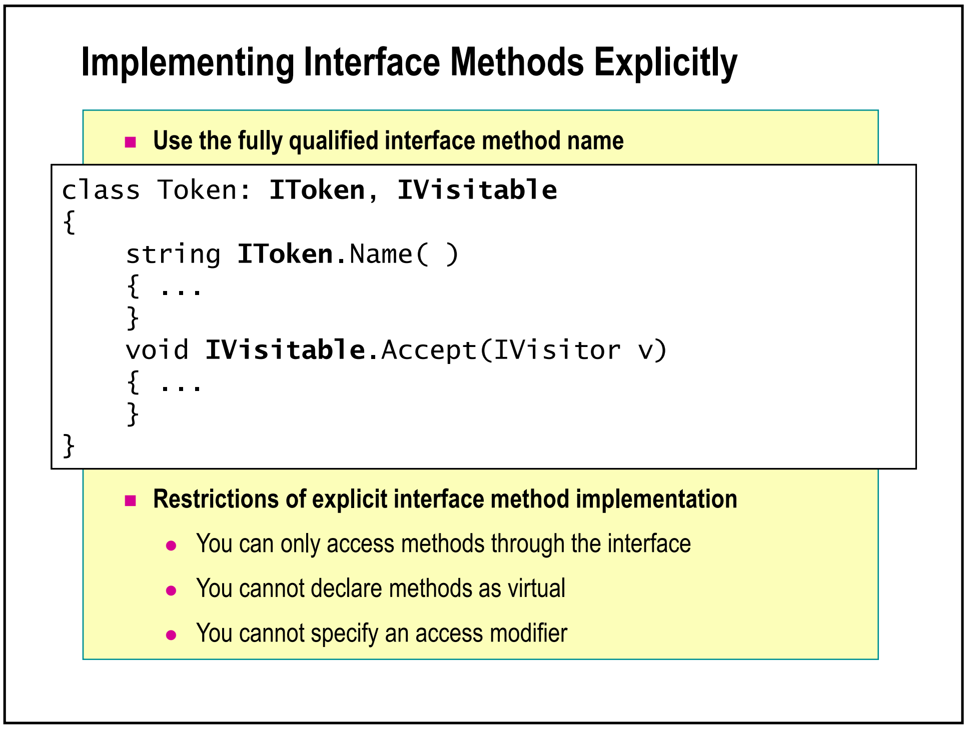
```
class Token: IToken, IVisitable
{
    public virtual string Name( )
    { ...
    }
    public void Accept(IVisitor v)
    { ...
    }
}
```

Same access
Same return type
Same name
Same parameters

Slide 251

JJ1 All method declarations must name the parameters

Jon Jagger, 05-12-2001

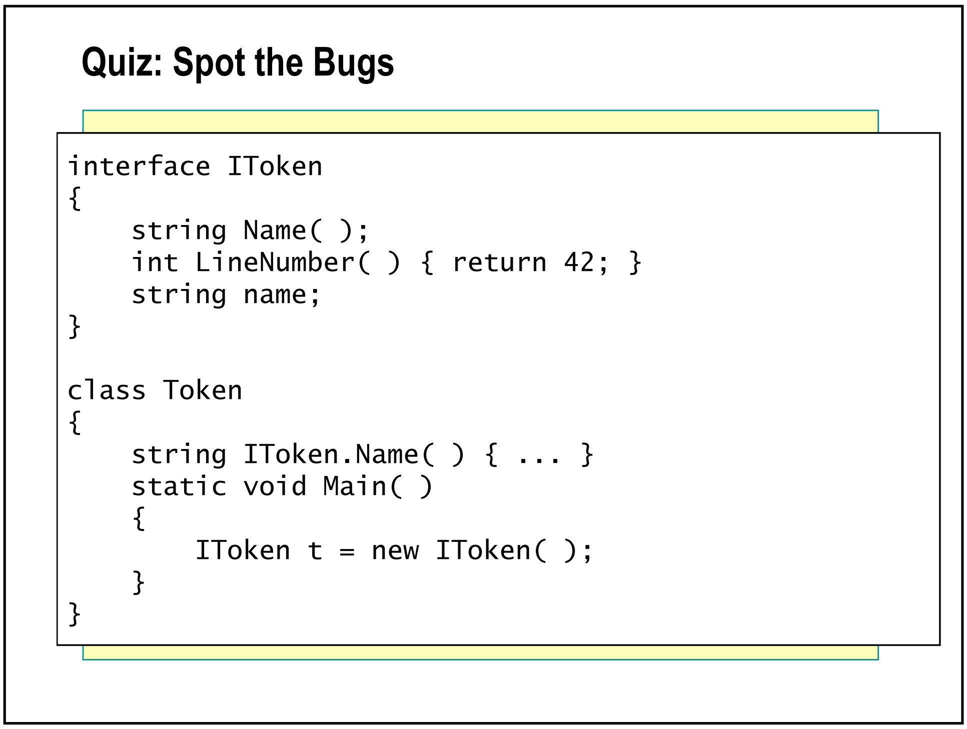


Implementing Interface Methods Explicitly

- Use the fully qualified interface method name

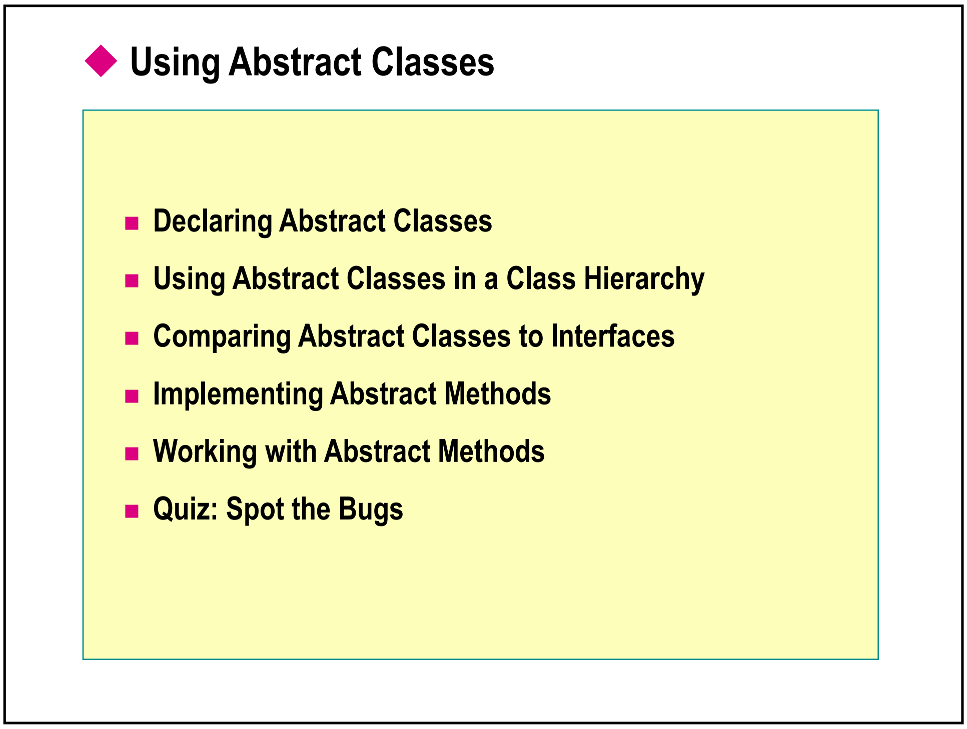
```
class Token: IToken, IVisitable
{
    string IToken.Name( )
    { ...
    }
    void IVisitable.Accept(IVisitor v)
    { ...
    }
}
```

- Restrictions of explicit interface method implementation
  - You can only access methods through the interface
  - You cannot declare methods as virtual
  - You cannot specify an access modifier



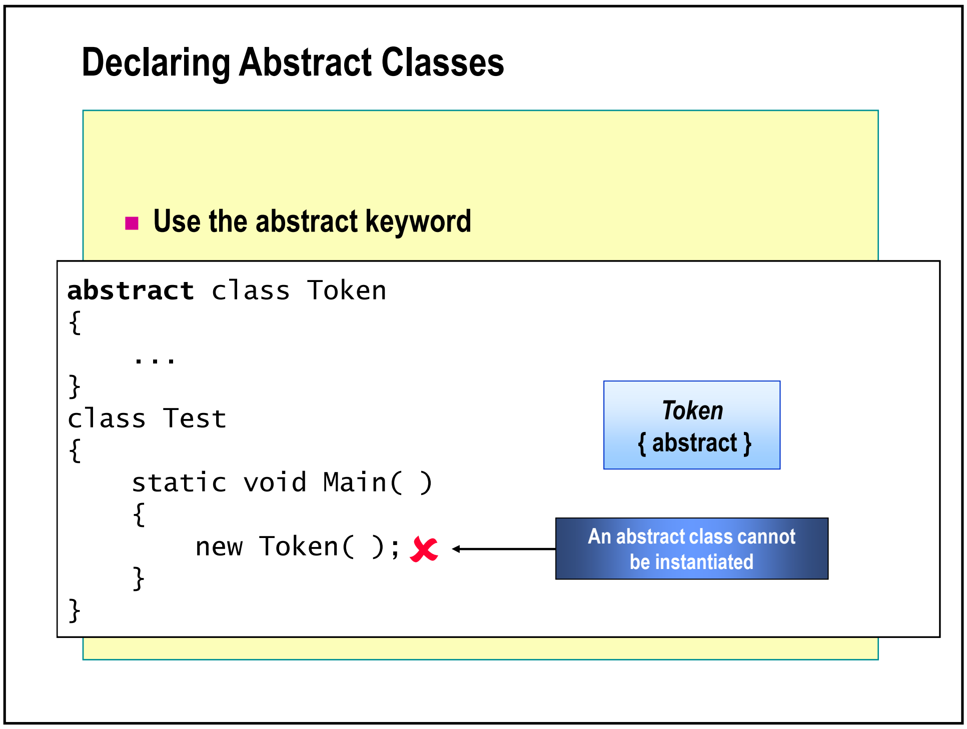
Quiz: Spot the Bugs

```
interface IToken
{
    string Name( );
    int LineNumber( ) { return 42; }
    string name;
}

class Token
{
    string IToken.Name( ) { ... }
    static void Main( )
    {
        IToken t = new IToken( );
    }
}
```



◆ Using Abstract Classes

- Declaring Abstract Classes
- Using Abstract Classes in a Class Hierarchy
- Comparing Abstract Classes to Interfaces
- Implementing Abstract Methods
- Working with Abstract Methods
- Quiz: Spot the Bugs



Declaring Abstract Classes

- Use the abstract keyword

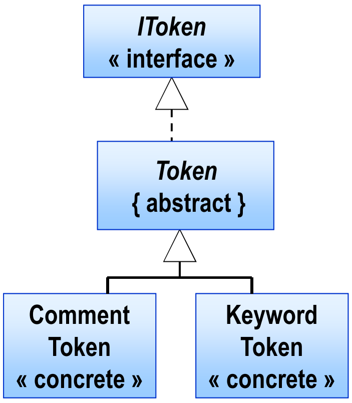
```
abstract class Token
{
    ...
}
class Test
{
    static void Main( )
    {
        new Token( ); ✘
    }
}
```

An abstract class cannot be instantiated

Using Abstract Classes in a Class Hierarchy

- Example 1

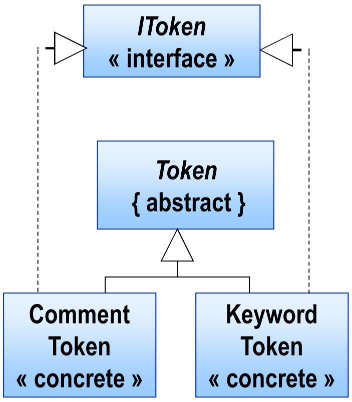
```
interface IToken
{
    string Name( );
}

abstract class Token: IToken
{
    string IToken.Name( )
    {   ...
    }
    ...
}

class CommentToken: Token
{   ...
}
class KeywordToken: Token
{   ...
}
```



Using Abstract Classes in a Class Hierarchy *(continued)*

- Example 2

```
interface IToken
{
    string Name( );
}

abstract class Token
{
    public virtual string Name( )
    {   ...
    }
    ...
}

class CommentToken: Token, IToken
{   ...
}
class KeywordToken: Token, IToken
{   ...
}
```



Comparing Abstract Classes to Interfaces

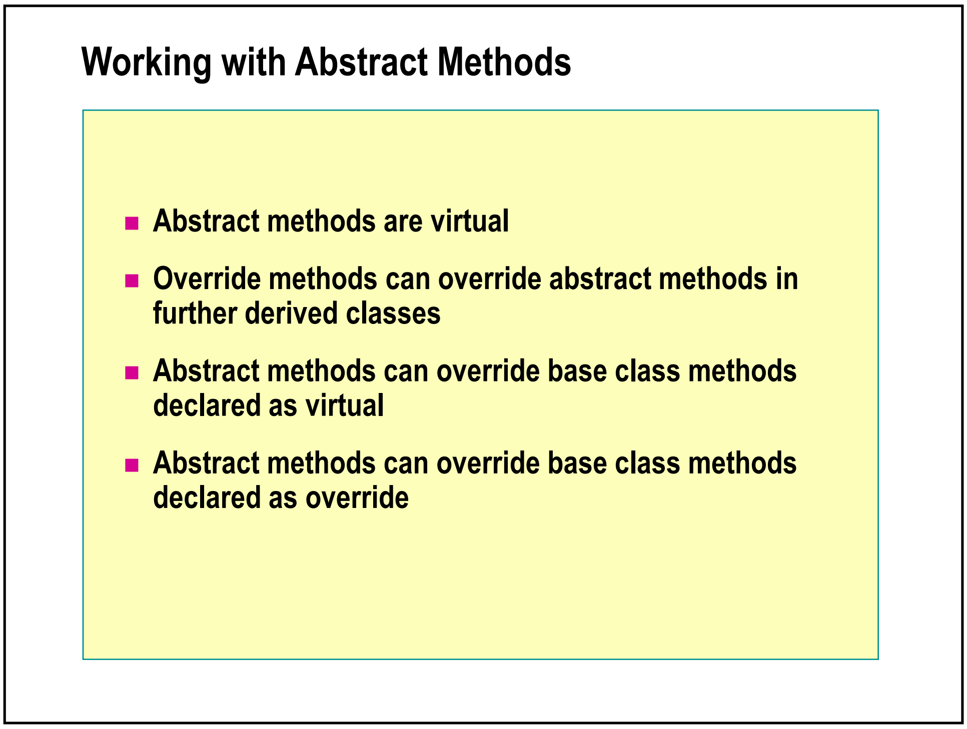
- **Similarities**
  - Neither can be instantiated
  - Neither can be sealed
- **Differences**
  - Interfaces cannot contain any implementation
  - Interfaces cannot declare non-public members
  - Interfaces cannot extend non-interfaces

Implementing Abstract Methods

- **Syntax: Use the abstract keyword**

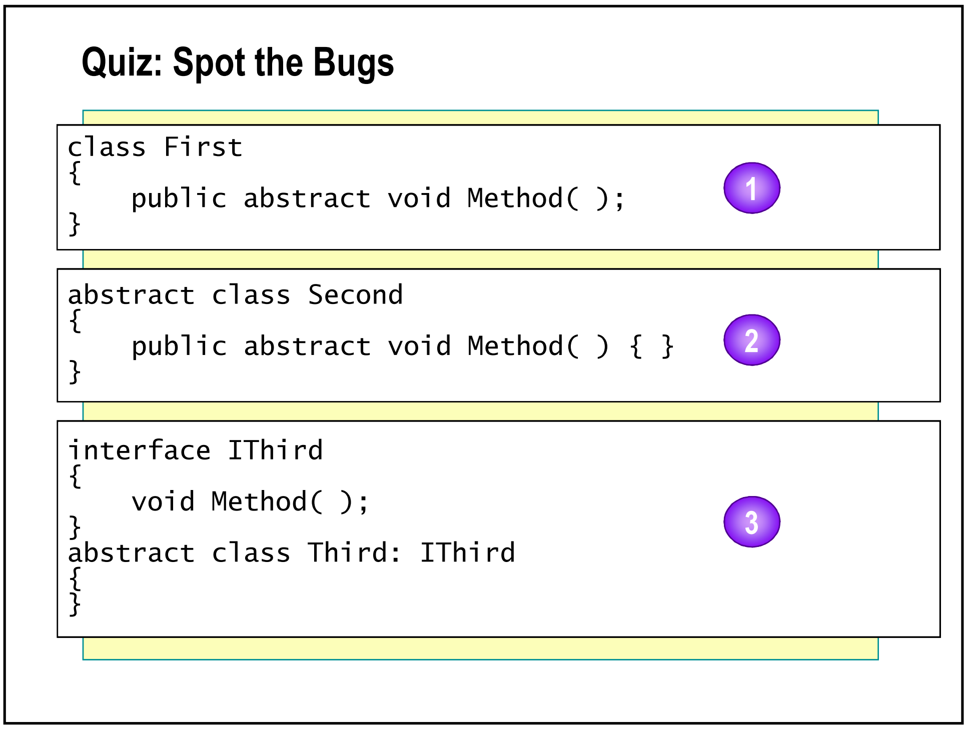
```
abstract class Token
{
    public virtual string Name( ) { ... }
    public abstract int Length( );
}
class CommentToken: Token
{
    public override string Name( ) { ... }
    public override int Length( ) { ... }
}
```

- **Only abstract classes can declare abstract methods**
- **Abstract methods cannot contain a method body**



Working with Abstract Methods

- Abstract methods are virtual
- Override methods can override abstract methods in further derived classes
- Abstract methods can override base class methods declared as virtual
- Abstract methods can override base class methods declared as override



Quiz: Spot the Bugs

```
class First
{
    public abstract void Method( );
}
```

1

```
abstract class Second
{
    public abstract void Method( ) { }
}
```

2

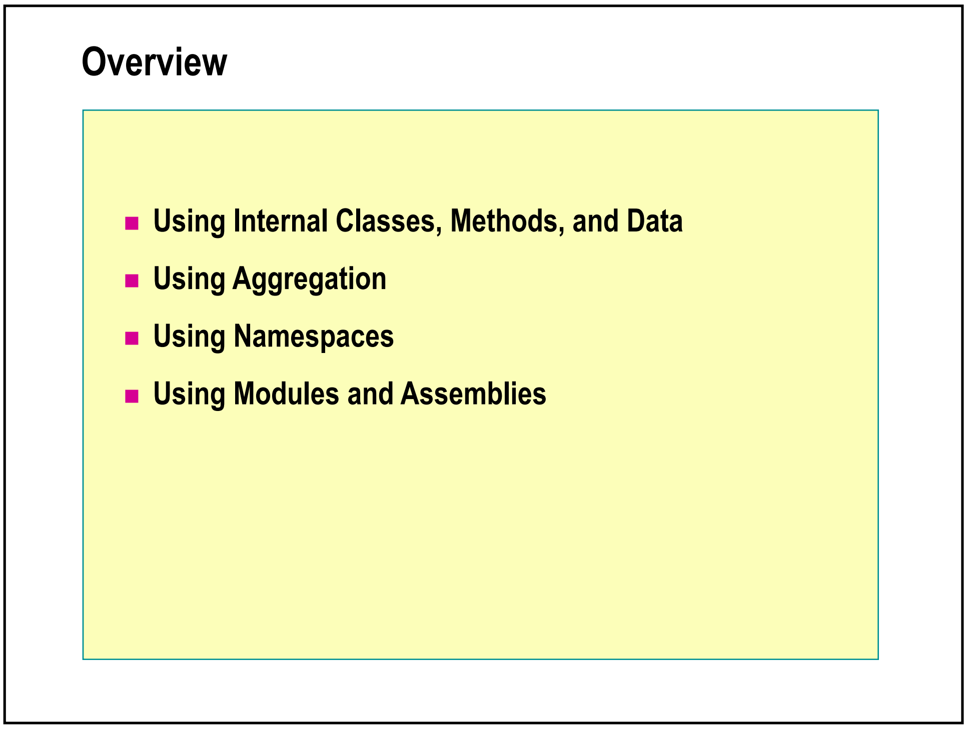
```
interface IThird
{
    void Method( );
}
abstract class Third: IThird
{
}
```

3

Review

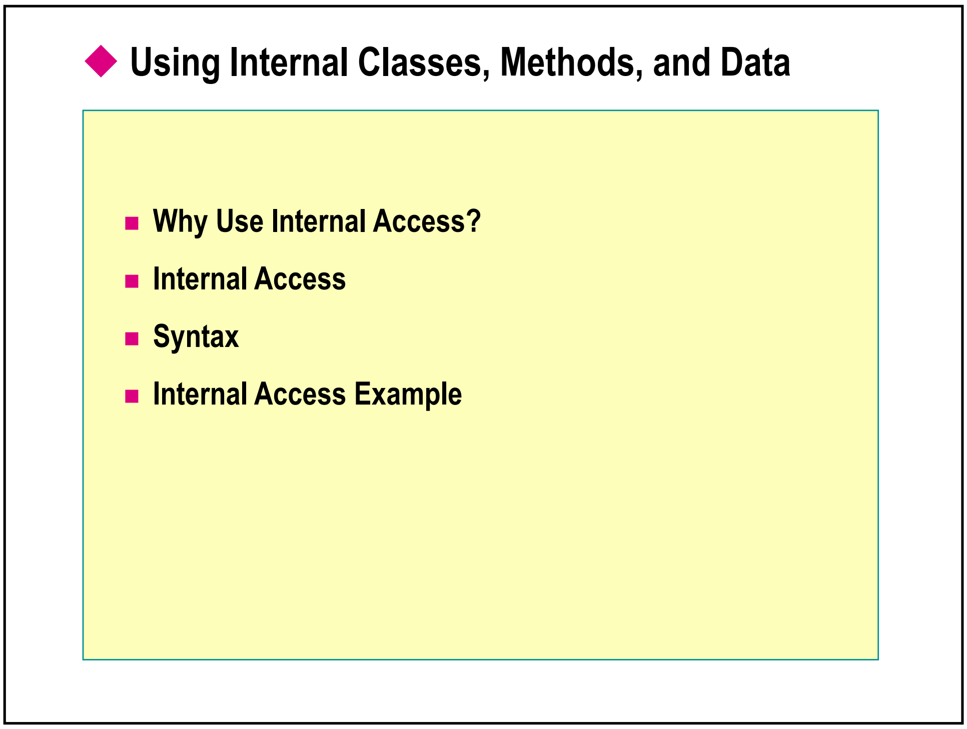
- Deriving Classes
- Implementing Methods
- Using Sealed Classes
- Using Interfaces
- Using Abstract Classes

Module 11: Aggregation, Namespaces, and Advanced Scope



Overview

- Using Internal Classes, Methods, and Data
- Using Aggregation
- Using Namespaces
- Using Modules and Assemblies

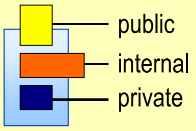


◆ Using Internal Classes, Methods, and Data

- Why Use Internal Access?
- Internal Access
- Syntax
- Internal Access Example

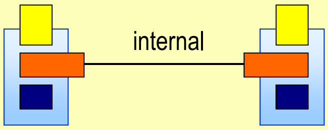
Why Use Internal Access?

- **Small objects are not very useful on their own**
- **Objects need to collaborate to form larger objects**
- **Access beyond the individual object is required**



Internal Access

- **Comparing access levels**
  - Public access is logical
  - Private access is logical
  - Internal access is physical



Syntax

```
internal class <outername>
{
    internal class <nestedname> { ... }
    internal <type> field;
    internal <type> Method( ) { ... }

    protected internal class <nestedname> { ... }
    protected internal <type> field;
    protected internal <type> Method( ) { ... }
}
```

protected internal means **protected** *or* **internal**

Internal Access Example

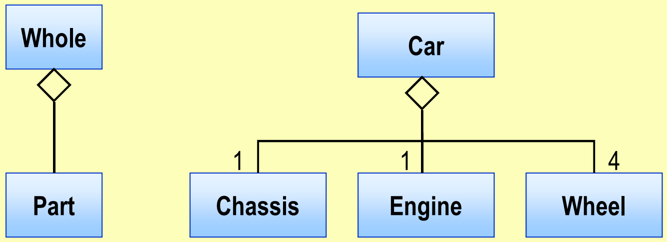
```
public interface IBankAccount { ... }

internal abstract class CommonBankAccount { ... }

internal class DepositAccount: CommonBankAccount,
                               IBankAccount { ... }
public class Bank
{
    public IBankAccount OpenAccount( )
    {
        return new DepositAccount( );
    }
}
```

◆ Using Aggregation

- Objects Within Objects
- Comparing Aggregation to Inheritance
- Factories
- Example Factory

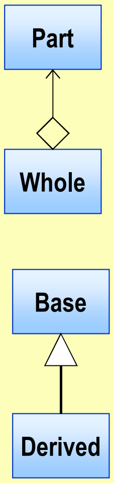
Objects Within Objects

- Complex objects are built from simpler objects
- Simpler objects are parts of complex whole objects
- This is called aggregation



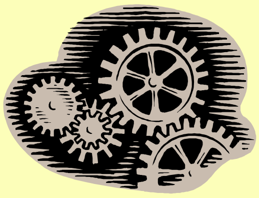
Comparing Aggregation to Inheritance

- **Aggregation**
  - Specifies an object relationship
  - A weak whole-to-part dependency
  - Dynamically flexible
- **Inheritance**
  - Specifies a class relationship
  - A strong derived-to-base dependency
  - Statically inflexible



Factories

- **Creation is often complex and restricted**
- **Many objects are made only in specialist factories**
- **The factory encapsulates the complex creation**
- **Factories are useful patterns when modelling software**



Factory Example

```
public class Bank
{
    public BankAccount OpenAccount( )
    {
        BankAccount opened = new BankAccount( );
        accounts[opened.Number( )] = opened;
        return opened;
    }
    private Hashtable accounts = new Hashtable( );
}
public class BankAccount
{
    internal BankAccount( ) { ... }
    public long Number( ) { ... }
    public void Deposit(decimal amount) { ... }
}
```

◆ Using Namespaces

- Scope Revisited
- Resolving Name Clashes
- Declaring Namespaces
- Fully Qualified Names
- Declaring using-namespace-directives
- Declaring using-alias-directives
- Guidelines for Naming Namespaces

Scope Revisited

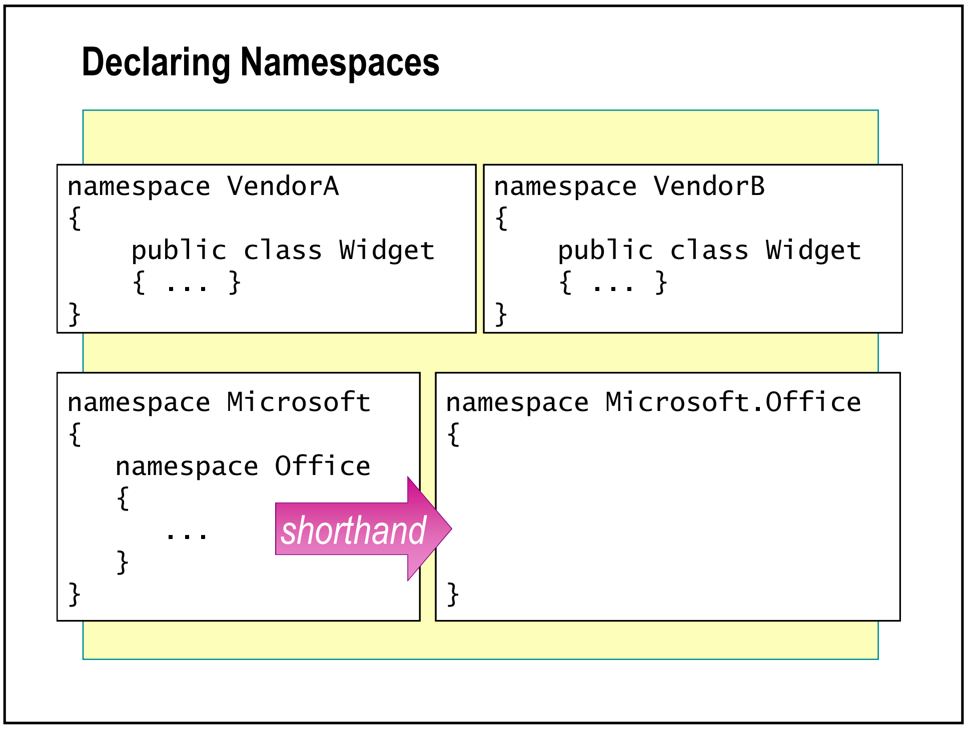
- The scope of a name is the region of program text in which you can refer to the name without qualification

```
public class Bank
{
    public class Account
    {
        public void Deposit(decimal amount)
        {
            balance += amount;
        }
        private decimal balance;
    }
    public Account OpenAccount( ) { ... }
}
```

Resolving Name Clashes

- Consider a large project that uses thousands of classes
- What if two classes have the same name?
- Do not add prefixes to all class names

```
// From Vendor A                     public class VendorAWidget
public class Widget                  { ... }
{ ... }                  ✘
// From Vendor B                     public class VendorBWidget
public class Widget                  { ... }
{ ... }
```



Declaring Namespaces

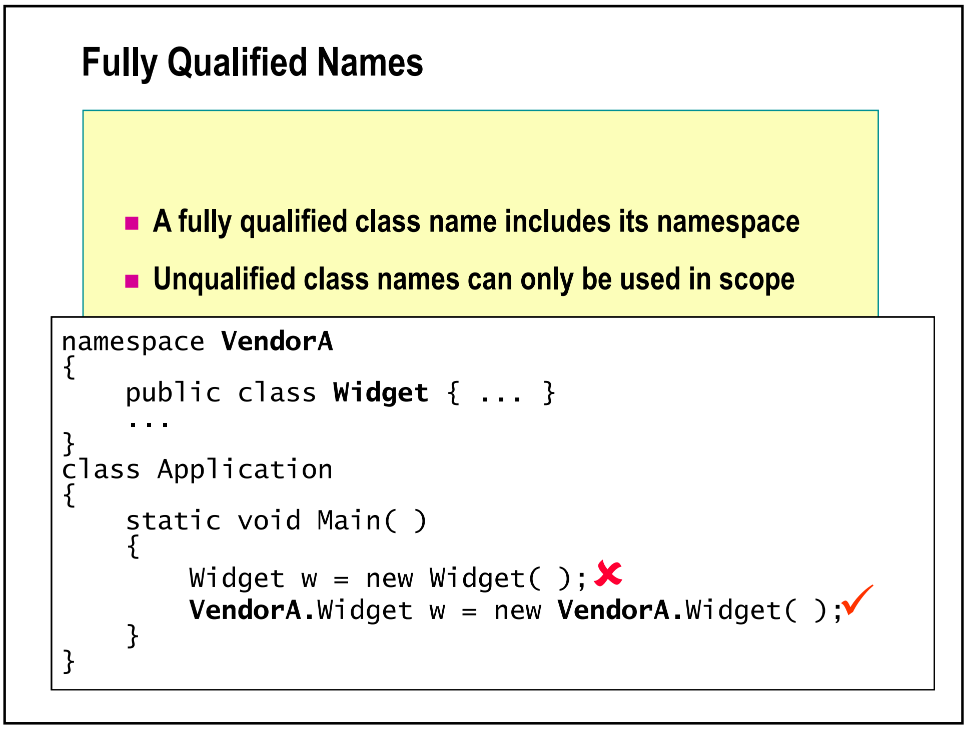
```
namespace VendorA
{
    public class Widget
    { ... }
}
```

```
namespace VendorB
{
    public class Widget
    { ... }
}
```

```
namespace Microsoft
{
   namespace Office
   {
      ...
   }
}
```

shorthand

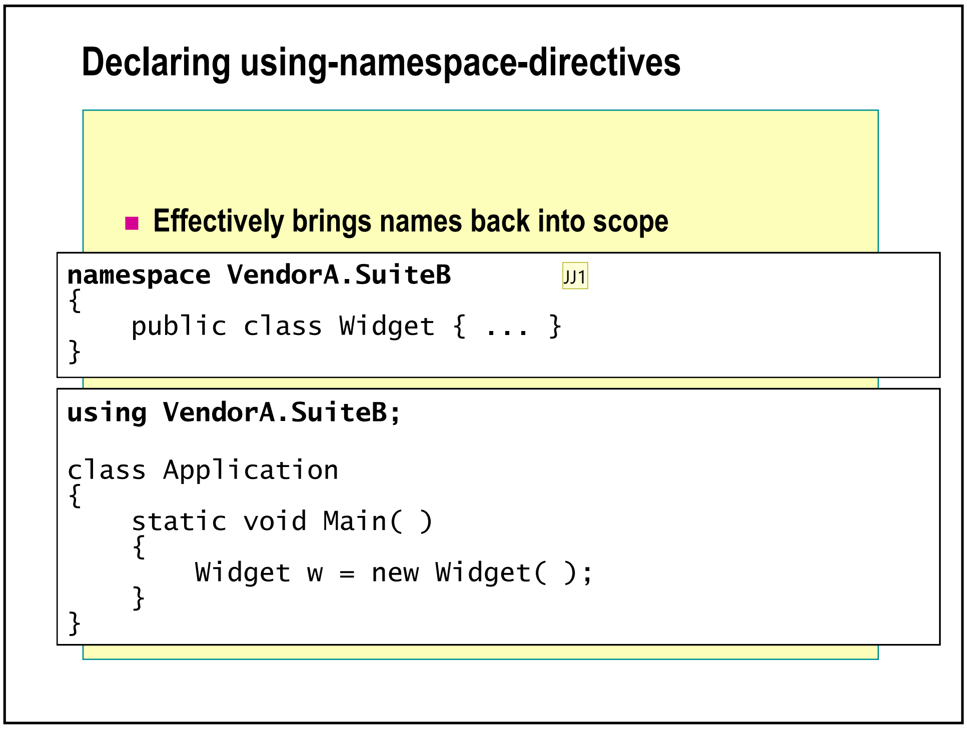
```
namespace Microsoft.Office
{



}
```



Fully Qualified Names

- A fully qualified class name includes its namespace
- Unqualified class names can only be used in scope

```
namespace VendorA
{
    public class Widget { ... }
    ...
}
class Application
{
    static void Main( )
    {
        Widget w = new Widget( );✗
        VendorA.Widget w = new VendorA.Widget( );✓
    }
}
```

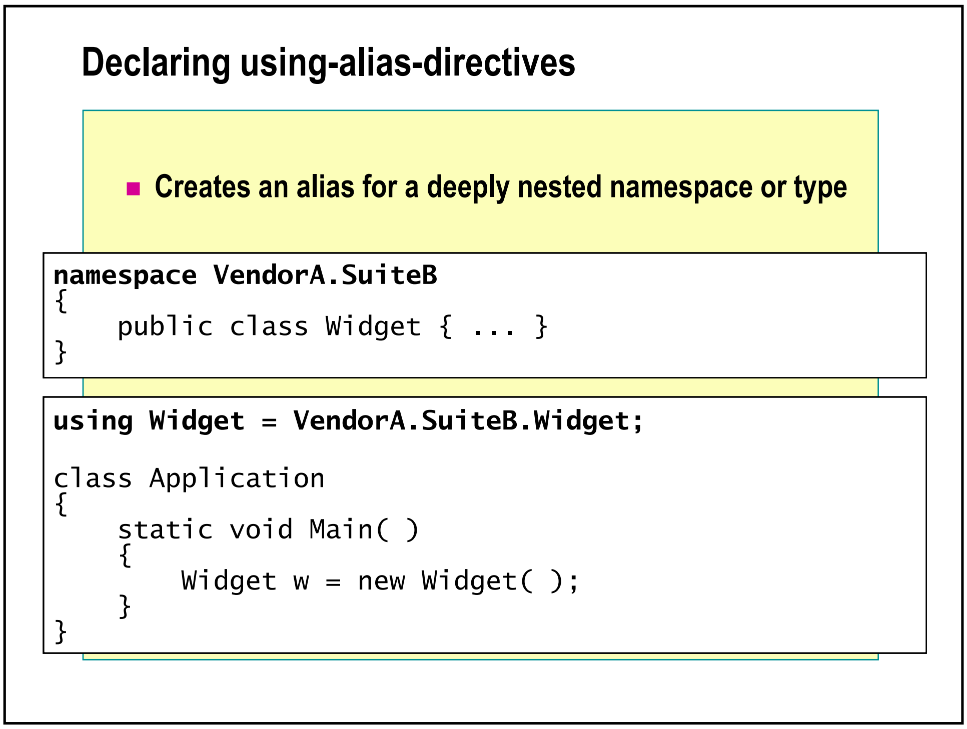


Declaring using-namespace-directives

- Effectively brings names back into scope

```
namespace VendorA.SuiteB
{
    public class Widget { ... }
}
```

```
using VendorA.SuiteB;

class Application
{
    static void Main( )
    {
        Widget w = new Widget( );
    }
}
```



Declaring using-alias-directives

- Creates an alias for a deeply nested namespace or type

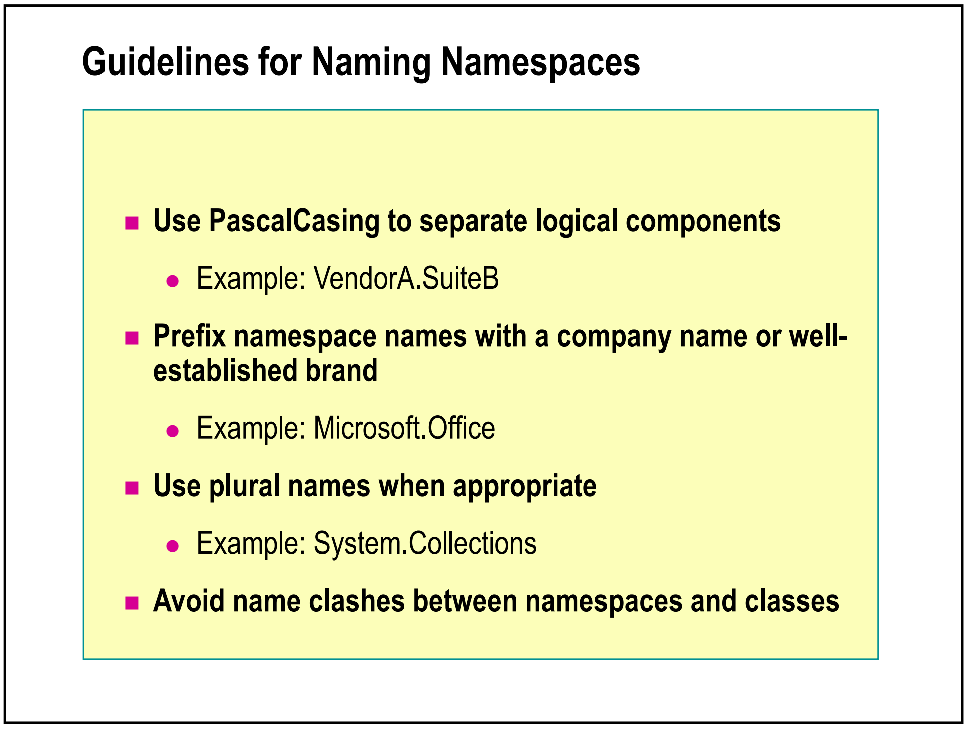
```
namespace VendorA.SuiteB
{
    public class Widget { ... }
}
```

```
using Widget = VendorA.SuiteB.Widget;

class Application
{
    static void Main( )
    {
        Widget w = new Widget( );
    }
}
```

Slide 281

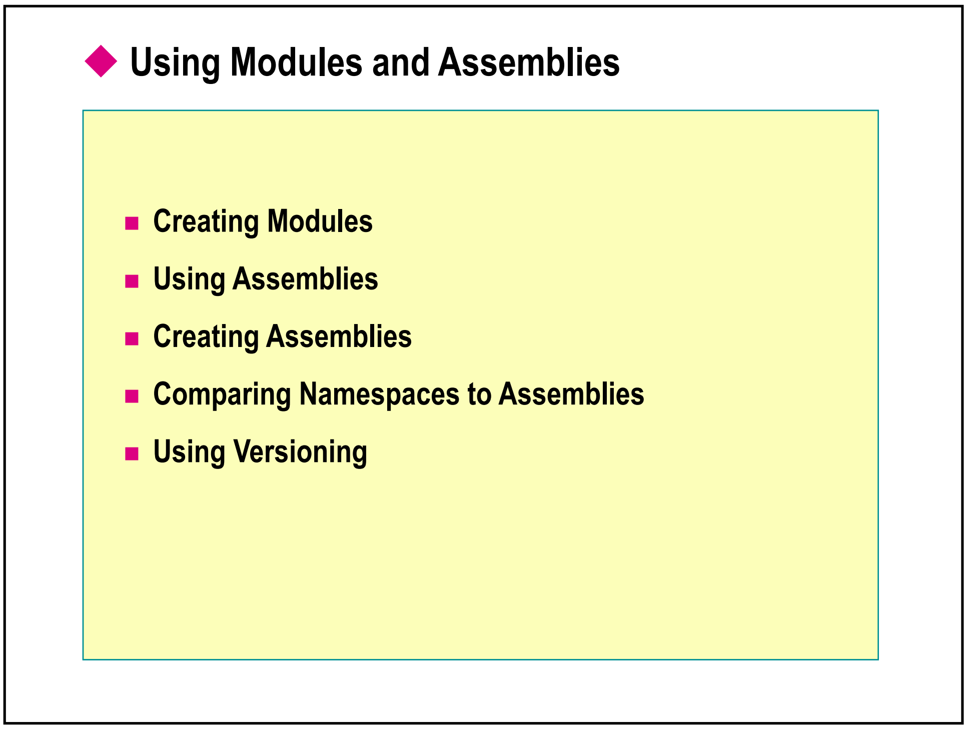
JJ1 Removed spurious semi colon

Jon Jagger, 18-12-2001



Guidelines for Naming Namespaces

- **Use PascalCasing to separate logical components**
  - Example: VendorA.SuiteB
- **Prefix namespace names with a company name or well-established brand**
  - Example: Microsoft.Office
- **Use plural names when appropriate**
  - Example: System.Collections
- **Avoid name clashes between namespaces and classes**



◆ Using Modules and Assemblies

- **Creating Modules**
- **Using Assemblies**
- **Creating Assemblies**
- **Comparing Namespaces to Assemblies**
- **Using Versioning**

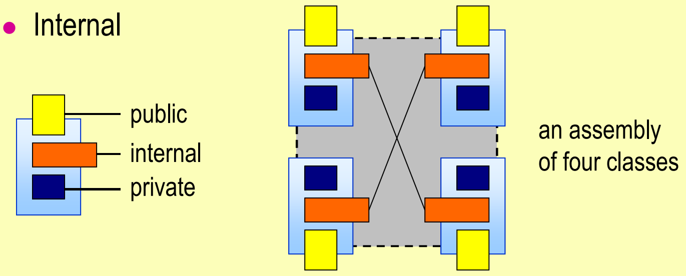
Using Modules

- .cs files can be compiled into a (.netmodule) managed module

```
csc /target:module Bank.cs
```

Using Assemblies

- **Group of collaborating classes**
  - Reusable, versionable, and secure deployment unit
- **Physical access control at assembly level**
  - Internal



Creating Assemblies

- Creating a single-file assembly

```
csc /target:library /out:Bank.dll     JJ2
                   Bank.cs Account.cs
```

- Creating multifile assembly

```
csc /t:library /addmodule:Account.netmJJ3dule
              /out:Bank.dll Bank.cs
```

Comparing Namespaces to Assemblies

- **Namespace: logical naming mechanism**
  - Classes from one namespace can reside in many assemblies
  - Classes from many namespaces can reside in one assembly
- **Assembly: physical grouping mechanism**
  - Assembly MSIL and manifest are contained directly
  - Assembly modules and resources can be external links

JJ3 This was originally...

```
csc /t:exe /addmodule:Bank.netmodule /out:Bank.exe Account.cs
```

This is wrong because
/1/ /t:exe needs to be /t:library
/2/ /out:Bank.exe needs to be /out:Bank.dll
/3/ Account needs to be a .netmodule (and not Bank) because there is an implicit assumption that Bank depends on Account

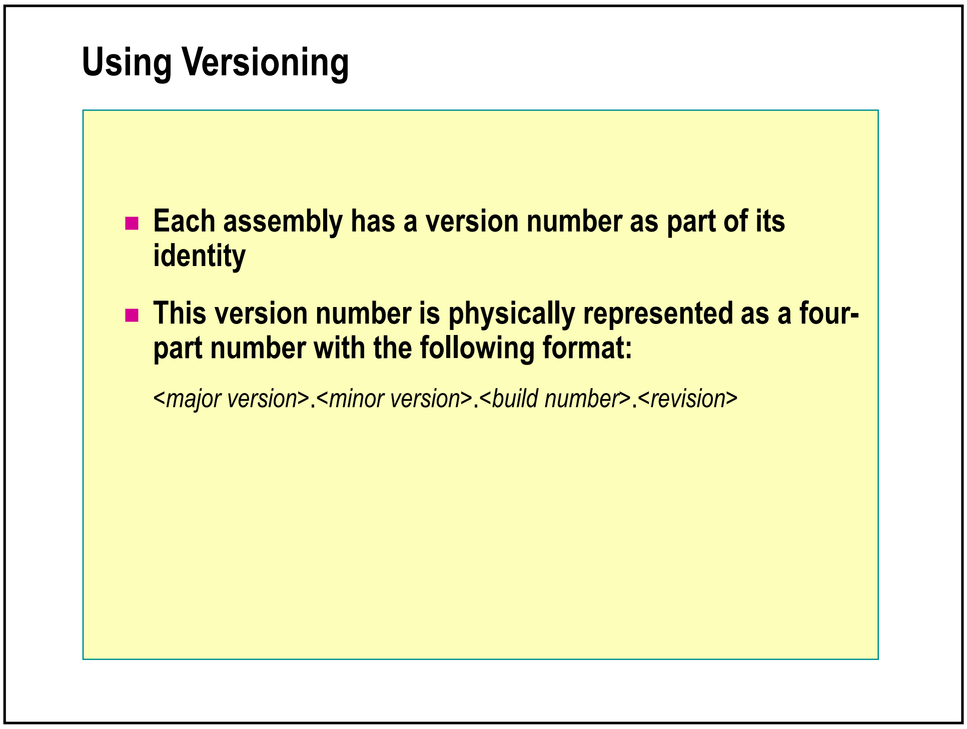
Jon Jagger, 18-12-2001

JJ2 This was originally...

```
csc /target:exe /out:Bank.exe Bank.cs Account.cs
```

This is wrong because
/1/ /target:exe needs to be /target:library
/2/ /out:bank.exe needs to be /out:Bank.dll

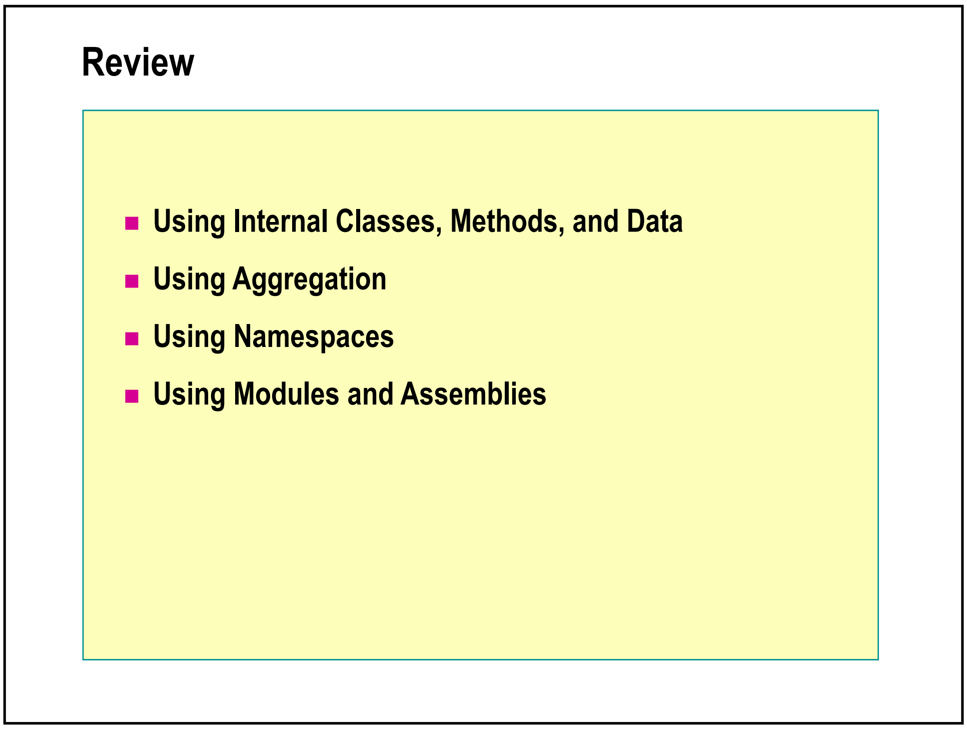
Jon Jagger, 18-12-2001



Using Versioning

- Each assembly has a version number as part of its identity
- This version number is physically represented as a four-part number with the following format:

  <major version>.*<minor version>*.*<build number>*.*<revision>*



Review

- Using Internal Classes, Methods, and Data
- Using Aggregation
- Using Namespaces
- Using Modules and Assemblies

Module 12: Operators, Delegates, and Events

Overview

- Introduction to Operators
- Operator Overloading
- Creating and Using Delegates
- Defining and Using Events

◆ Introduction to Operators

- **Operators and Methods**
- **Predefined C# Operators**

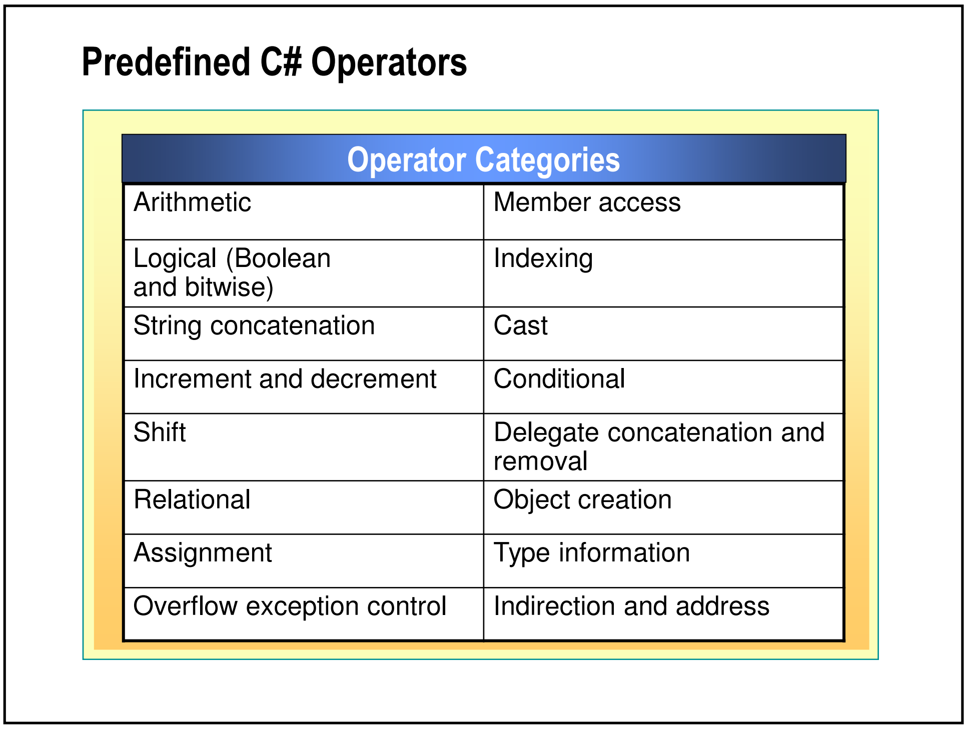
Operators and Methods

- **Using methods**
  - Reduces clarity
  - Increases risk of errors, both syntactic and semantic

```
myIntVar1 = Int.Add(myIntVar2,
            Int.Add(Int.Add(myIntVar3,
                            myIntVar4), 33));
```

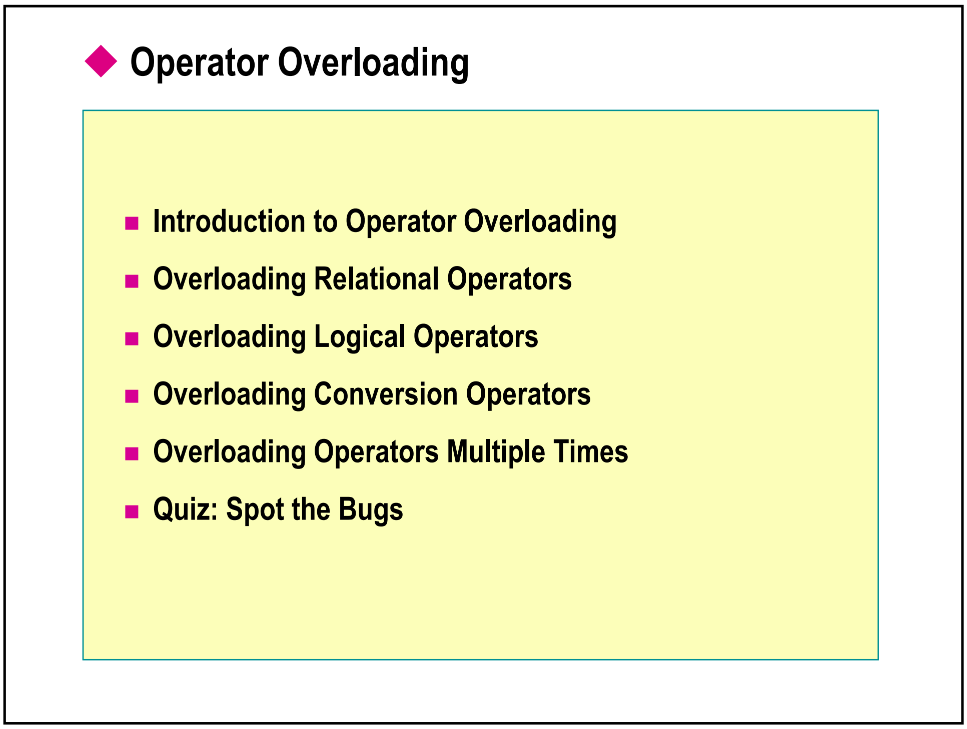
- **Using operators**
  - Makes expressions clear

```
myIntVar1 = myIntVar2 + myIntVar3 + myIntVar4 + 33;
```



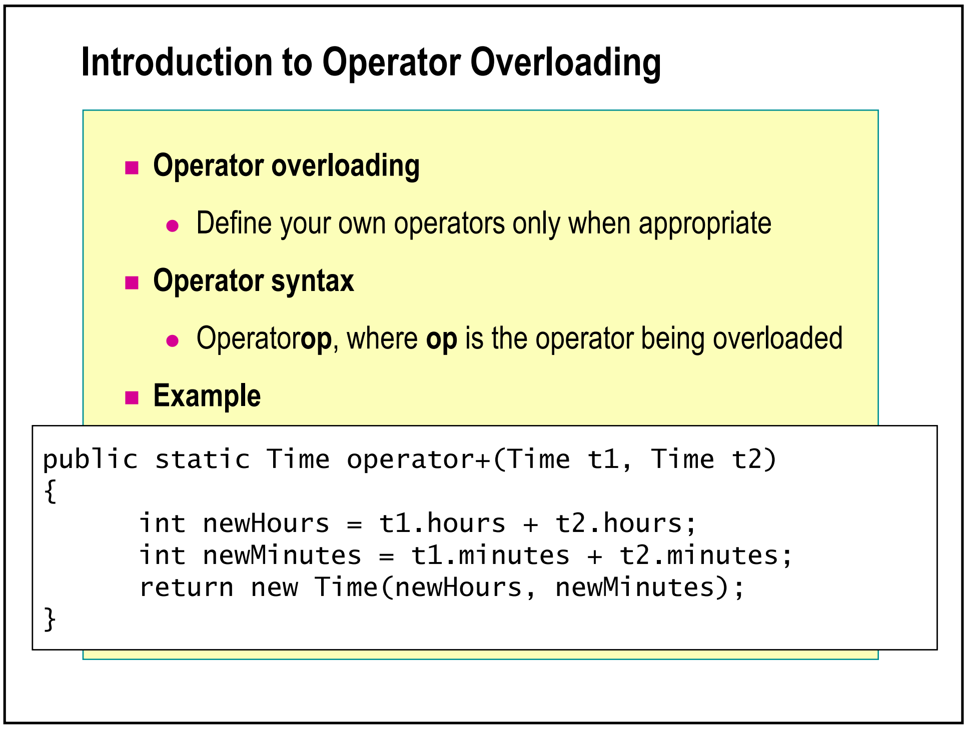
Predefined C# Operators

Operator Categories	
Arithmetic	Member access
Logical (Boolean and bitwise)	Indexing
String concatenation	Cast
Increment and decrement	Conditional
Shift	Delegate concatenation and removal
Relational	Object creation
Assignment	Type information
Overflow exception control	Indirection and address



◆ Operator Overloading

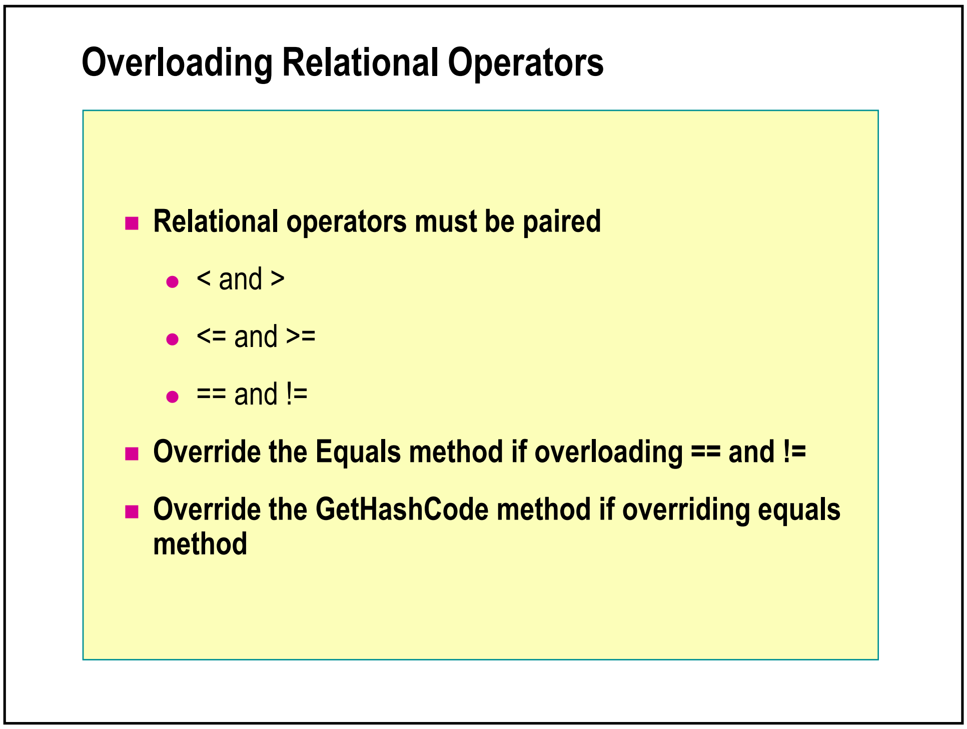
- **Introduction to Operator Overloading**
- **Overloading Relational Operators**
- **Overloading Logical Operators**
- **Overloading Conversion Operators**
- **Overloading Operators Multiple Times**
- **Quiz: Spot the Bugs**



Introduction to Operator Overloading

- **Operator overloading**
  - Define your own operators only when appropriate
- **Operator syntax**
  - Operator**op**, where **op** is the operator being overloaded
- **Example**

```
public static Time operator+(Time t1, Time t2)
{
      int newHours = t1.hours + t2.hours;
      int newMinutes = t1.minutes + t2.minutes;
      return new Time(newHours, newMinutes);
}
```



Overloading Relational Operators

- **Relational operators must be paired**
  - < and >
  - <= and >=
  - == and !=
- **Override the Equals method if overloading == and !=**
- **Override the GetHashCode method if overriding equals method**

Overloading Logical Operators

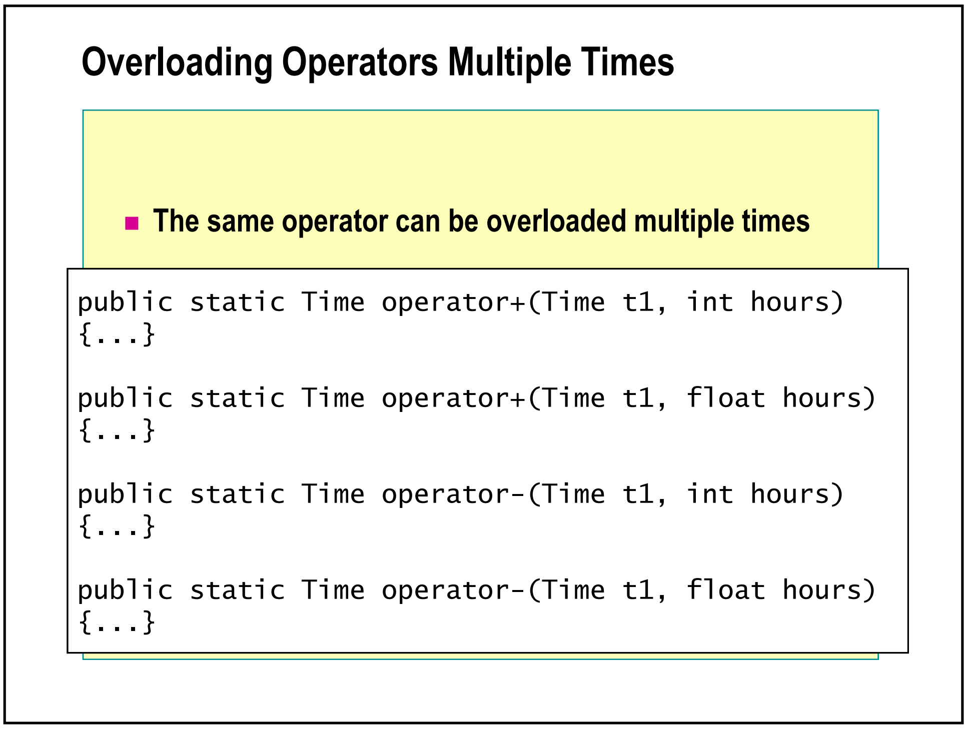
- **Operators && and || cannot be overloaded directly**
  - They are evaluated in terms of &, |, **true**, and **false**, which can be overloaded
  - **x && y** is evaluated as **T.false(x) ? x : T.&(x, y)**
  - **x || y** is evaluated as **T.true(x) ? x : T.|(x, y)**

Overloading Conversion Operators

- **Overloaded conversion operators**

```
public static explicit operator Time (float hours)
{ ... }
public static explicit operator float (Time t1)
{ ... }
public static implicit operator string (Time t1)
{ ... }
```

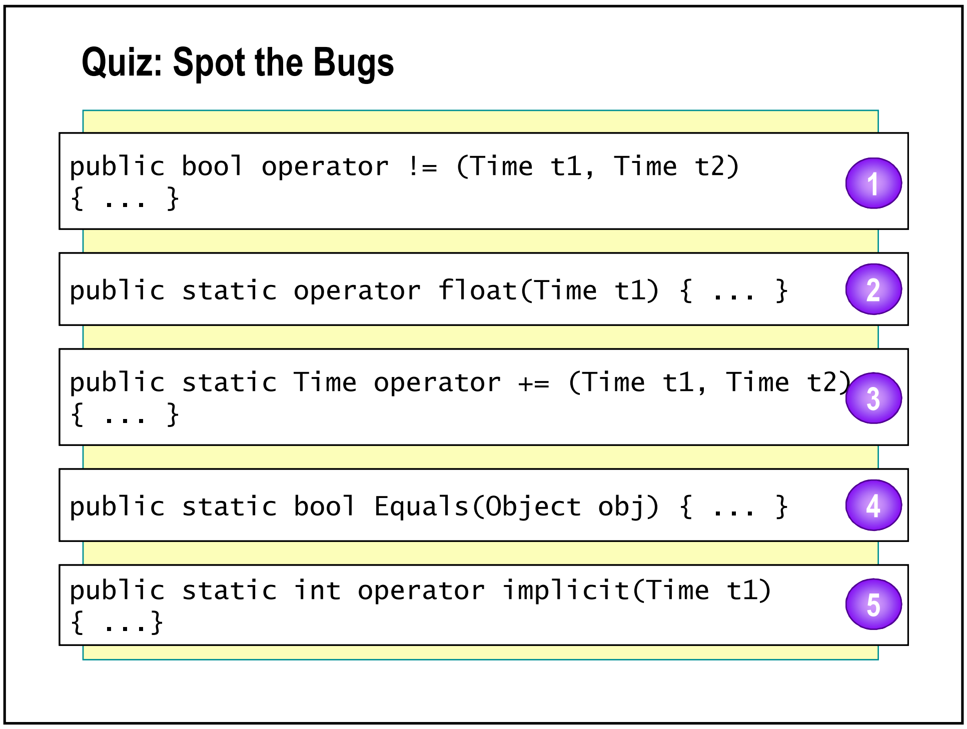
- **If a class defines a string conversion operator**
  - The class should override ToString



Overloading Operators Multiple Times

- The same operator can be overloaded multiple times

```
public static Time operator+(Time t1, int hours)
{...}

public static Time operator+(Time t1, float hours)
{...}

public static Time operator-(Time t1, int hours)
{...}

public static Time operator-(Time t1, float hours)
{...}
```



Quiz: Spot the Bugs

```
public bool operator != (Time t1, Time t2)
{ ... }
```
1

```
public static operator float(Time t1) { ... }
```
2

```
public static Time operator += (Time t1, Time t2)
{ ... }
```
3

```
public static bool Equals(Object obj) { ... }
```
4

```
public static int operator implicit(Time t1)
{ ...}
```
5

◆ Creating and Using Delegates

- **Scenario: Power Station**
- **Analyzing the Problem**
- **Creating Delegates**
- **Using Delegates**

Scenario: Power Station

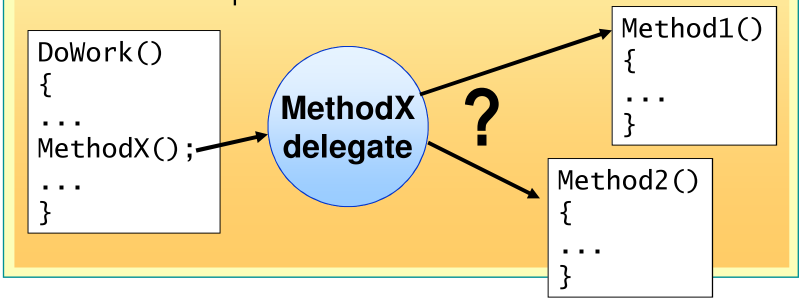
- **The problem**
  - How to respond to temperature events in a power station
  - Specifically, if the temperature of the reactor core rises above a certain temperature, coolant pumps need to be alerted and switched on
- **Possible solutions**
  - Should all coolant pumps monitor the core temperature?
  - Should a component that monitors the core turn on the appropriate pumps when the temperature changes?

Analyzing the Problem

- **Existing concerns**
  - There may be several types of pumps, supplied by different manufacturers
  - Each pump could have its own method for activation
- **Future concerns**
  - To add a new pump, the entire code will need to change
  - A high overhead cost will result with every such addition
- **A solution**
  - Use delegates in your code

Creating Delegates

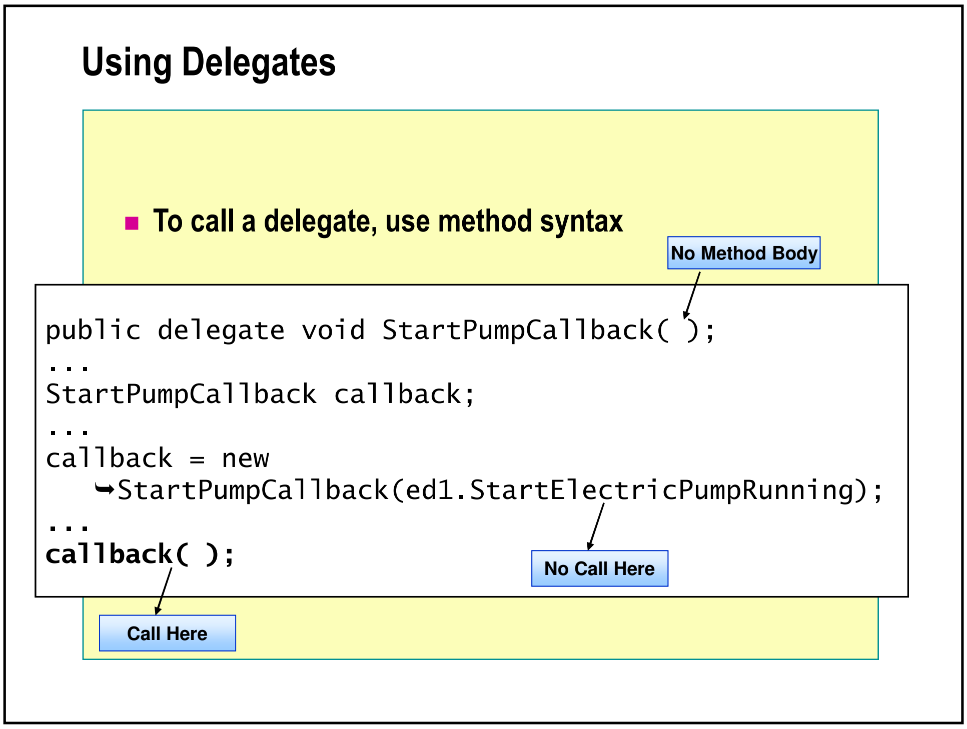
- **A delegate allows a method to be called indirectly**
  - It contains a reference to a method
  - All methods invoked by the same delegate must have the same parameters and return value



```
DoWork()
{
...
MethodX();
...
}
```

```
Method1()
{
...
}
```

```
Method2()
{
...
}
```



Using Delegates

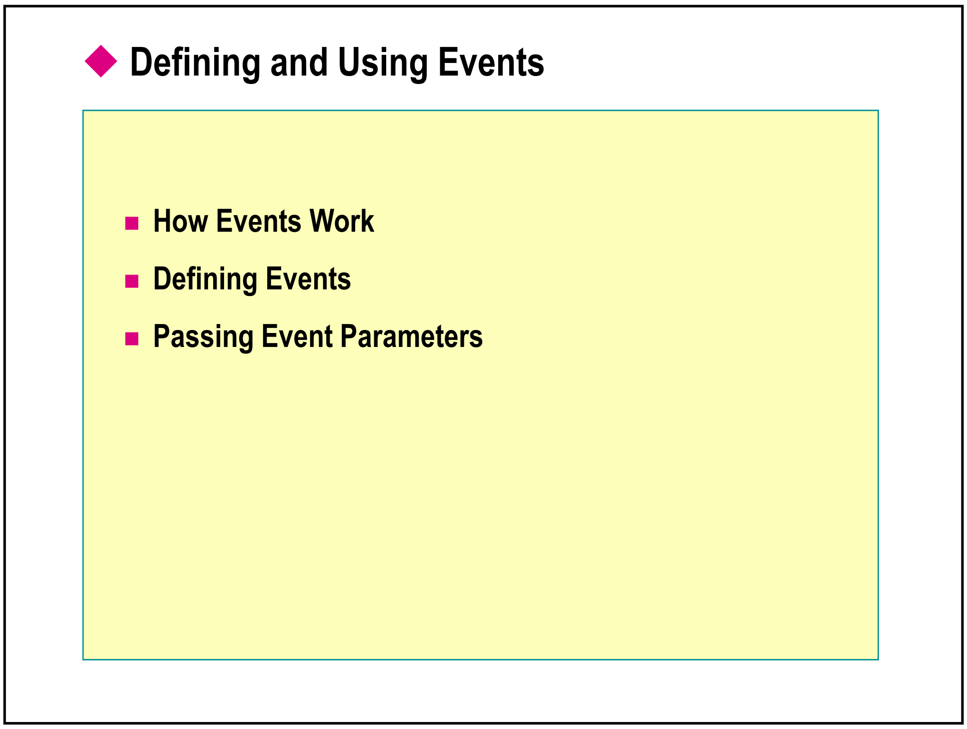
- To call a delegate, use method syntax

```
public delegate void StartPumpCallback( );
...
StartPumpCallback callback;
...
callback = new
   ➥StartPumpCallback(ed1.StartElectricPumpRunning);
...
callback( );
```

No Method Body

No Call Here

Call Here



Defining and Using Events

- How Events Work
- Defining Events
- Passing Event Parameters

How Events Work

- **Publisher**
  - Raises an event to alert all interested objects (subscribers)
- **Subscriber**
  - Provides a method to be called when the event is raised

Defining Events

- **Defining an event**

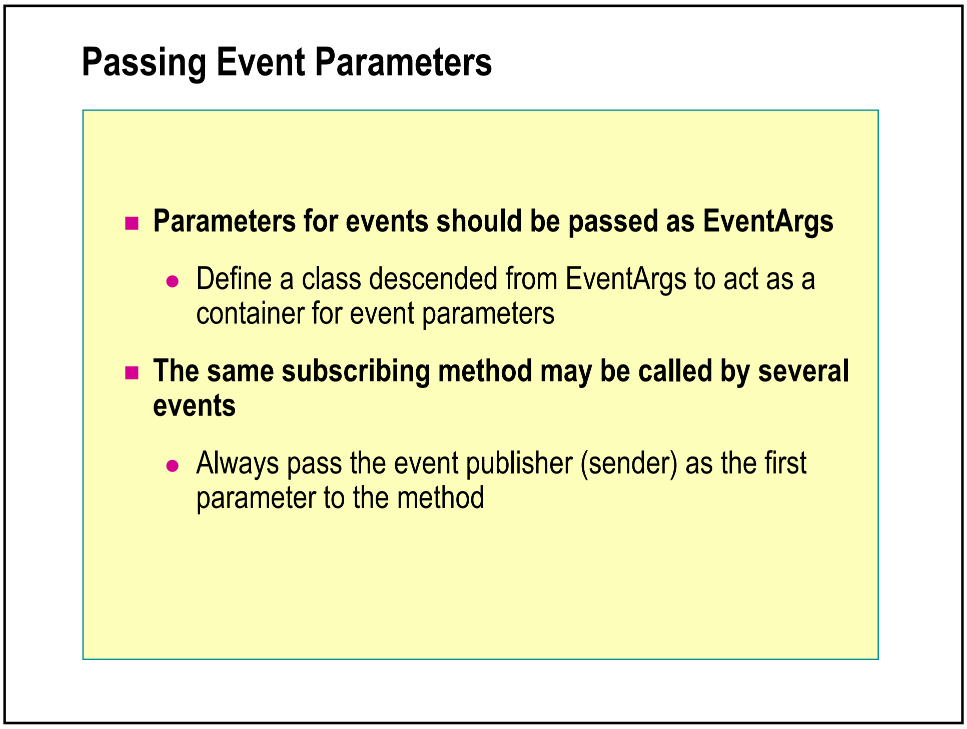
```
public delegate void StartPumpCallback( );
private event StartPumpCallback CoreOverheating;
```

- **Subscribing to an event**

```
PneumaticPumpDriver pd1 = new PneumaticPumpDriver( );
...
CoreOverheating += new StartPumpCallback(pd1.SwitchOn);
```

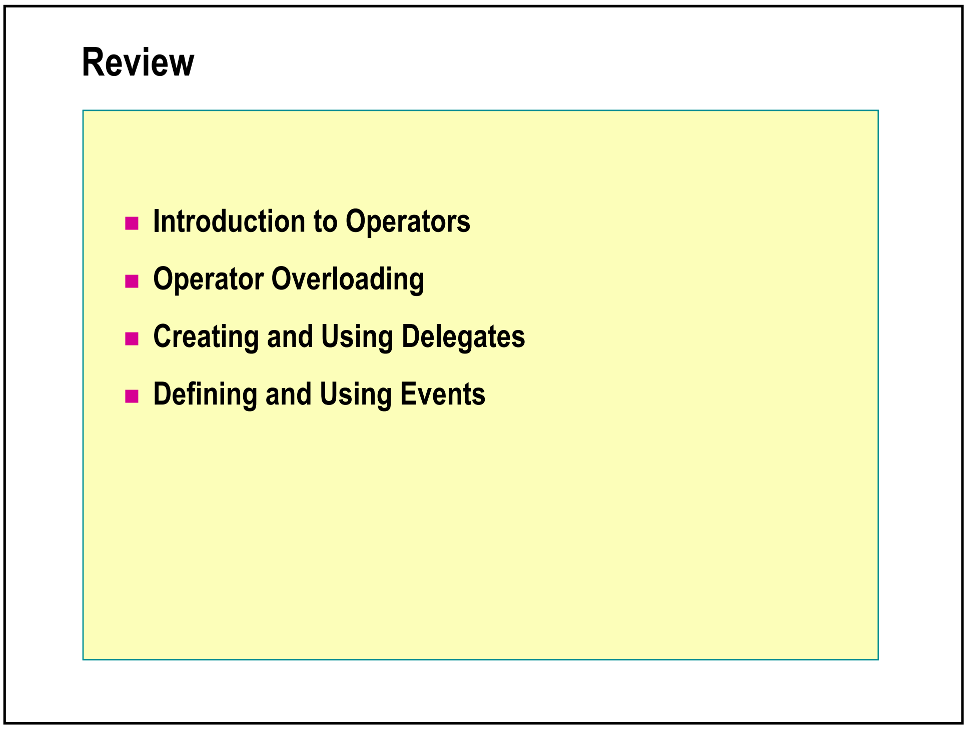
- **Notifying subscribers to an event**

```
public void SwitchOnAllPumps( ) {
  if (CoreOverheating != null) {
      CoreOverheating( );
  }
}
```



Passing Event Parameters

- **Parameters for events should be passed as EventArgs**
  - Define a class descended from EventArgs to act as a container for event parameters
- **The same subscribing method may be called by several events**
  - Always pass the event publisher (sender) as the first parameter to the method



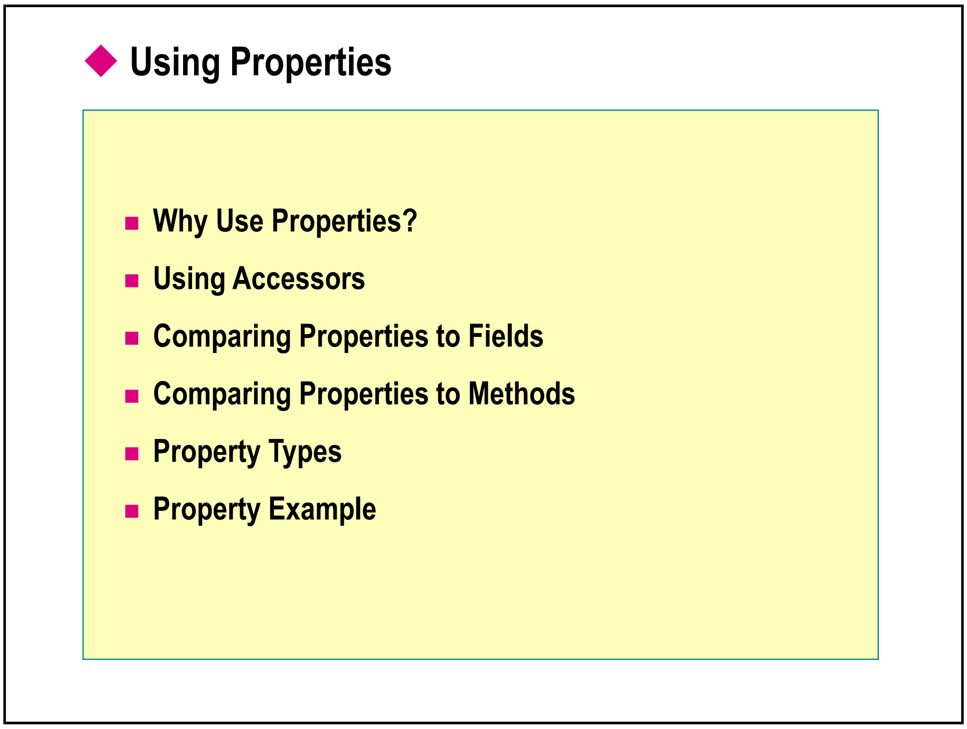
Review

- **Introduction to Operators**
- **Operator Overloading**
- **Creating and Using Delegates**
- **Defining and Using Events**

Module 13: Properties and Indexers

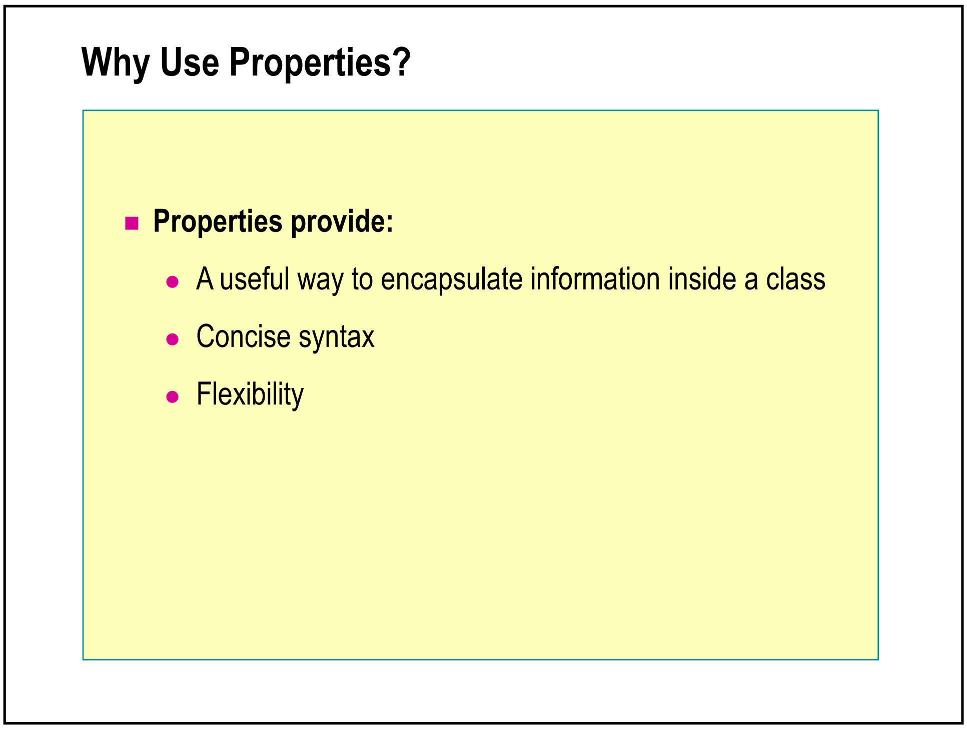
Overview

- Using Properties
- Using Indexers



◆ Using Properties

- Why Use Properties?
- Using Accessors
- Comparing Properties to Fields
- Comparing Properties to Methods
- Property Types
- Property Example



Why Use Properties?

- **Properties provide:**
  - A useful way to encapsulate information inside a class
  - Concise syntax
  - Flexibility

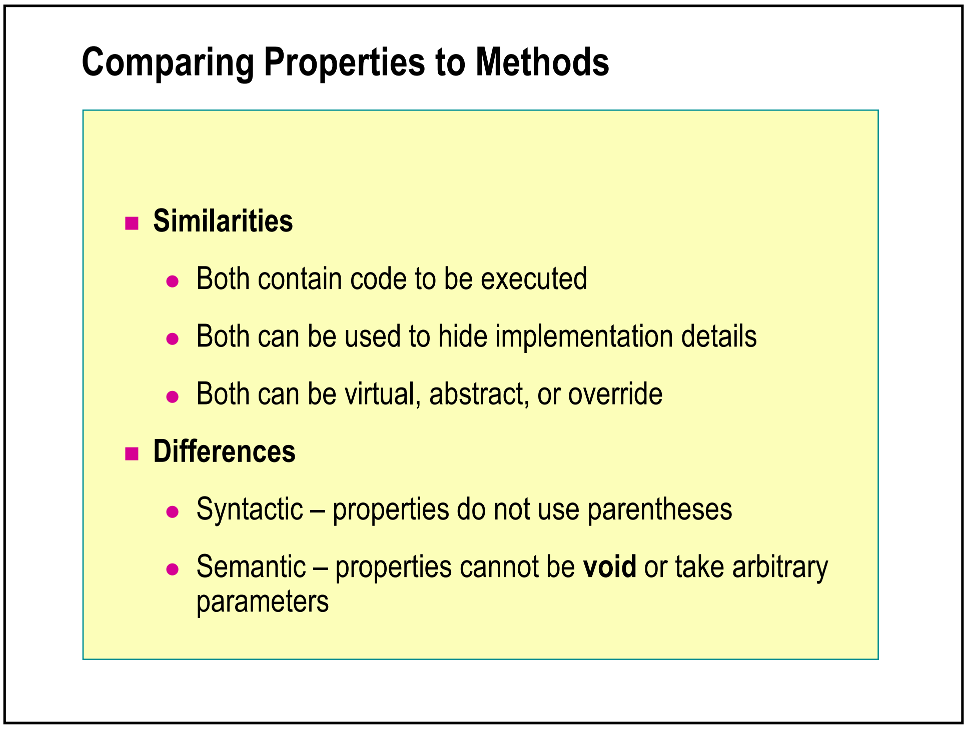
Using Accessors

- **Properties provide field-like access**
  - Use **get** accessor statements to provide read access
  - Use **set** accessor statements to provide write access

```
class Button
{
    public string Caption // Property
    {
        get { return caption; }
        set { caption = value; }
    }
    private string caption; // Field
}
```

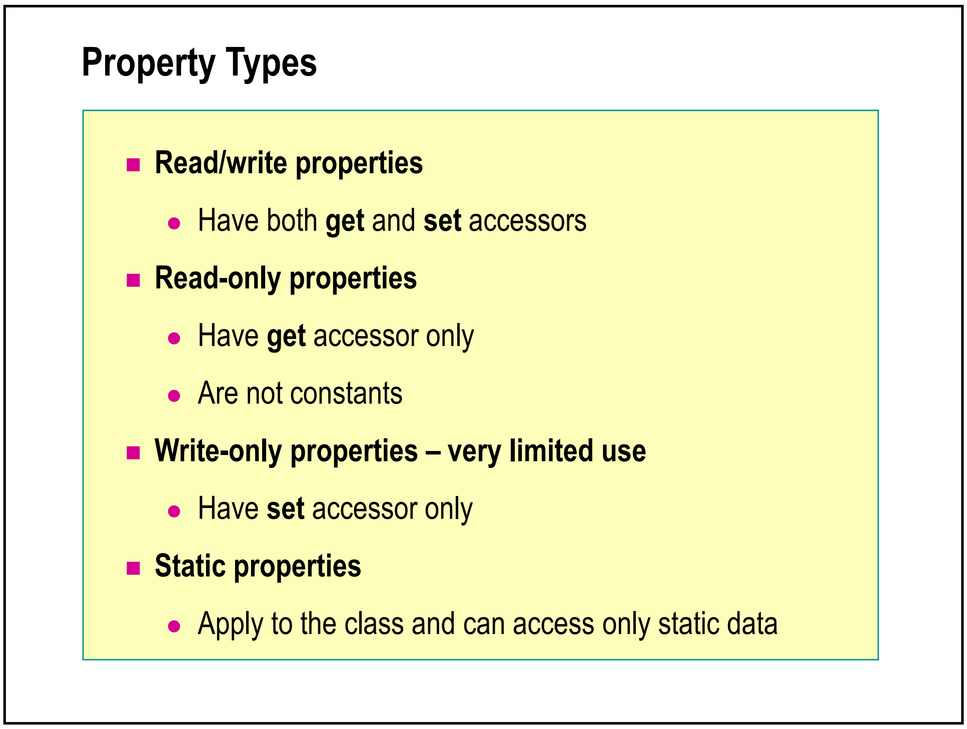
Comparing Properties to Fields

- **Properties are “logical fields”**
  - The **get** accessor can return a computed value
- **Similarities**
  - Syntax for creation and use is the same
- **Differences**
  - Properties are not values; they have no address
  - Properties cannot be used as **ref** or **out** parameters to methods



Comparing Properties to Methods

- **Similarities**
  - Both contain code to be executed
  - Both can be used to hide implementation details
  - Both can be virtual, abstract, or override
- **Differences**
  - Syntactic – properties do not use parentheses
  - Semantic – properties cannot be **void** or take arbitrary parameters



Property Types

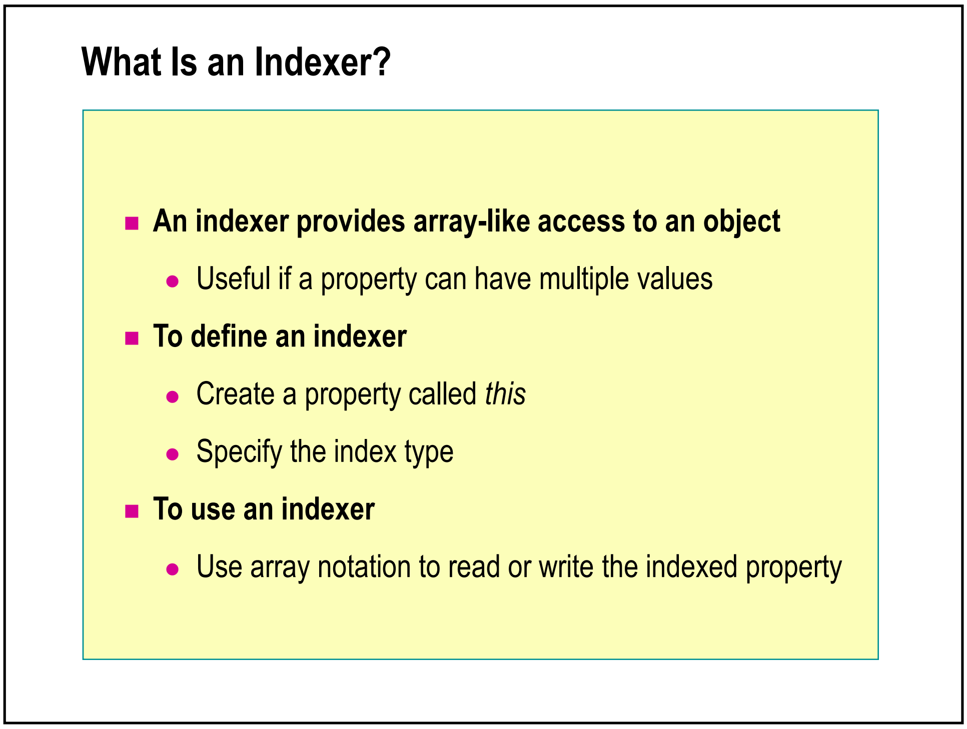
- **Read/write properties**
  - Have both **get** and **set** accessors
- **Read-only properties**
  - Have **get** accessor only
  - Are not constants
- **Write-only properties – very limited use**
  - Have **set** accessor only
- **Static properties**
  - Apply to the class and can access only static data

Property Example

```
public class Console
{
    public static TextReader In
    {
        get {
            if (reader == null) {
                reader = new StreamReader(...);
            }
            return reader;
        }
    }
    ...
    private static TextReader reader = null;
}
```

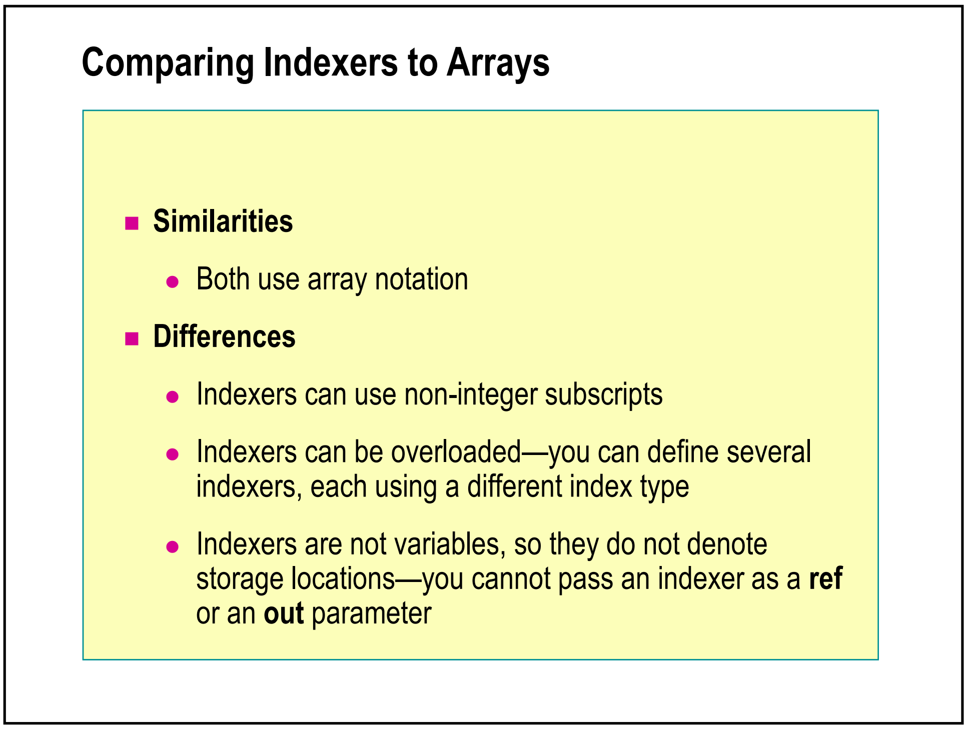
◆ Using Indexers

- What Is an Indexer?
- Comparing Indexers to Arrays
- Comparing Indexers to Properties
- Using Parameters to Define Indexers
- String Example
- BitArray Example



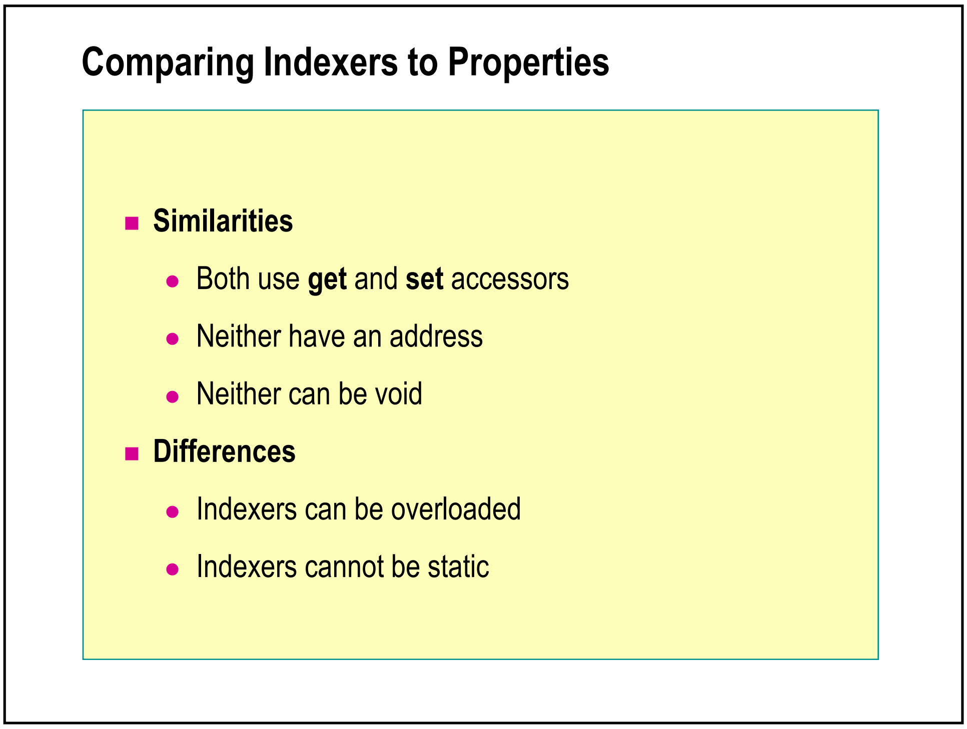
What Is an Indexer?

- **An indexer provides array-like access to an object**
  - Useful if a property can have multiple values
- **To define an indexer**
  - Create a property called *this*
  - Specify the index type
- **To use an indexer**
  - Use array notation to read or write the indexed property



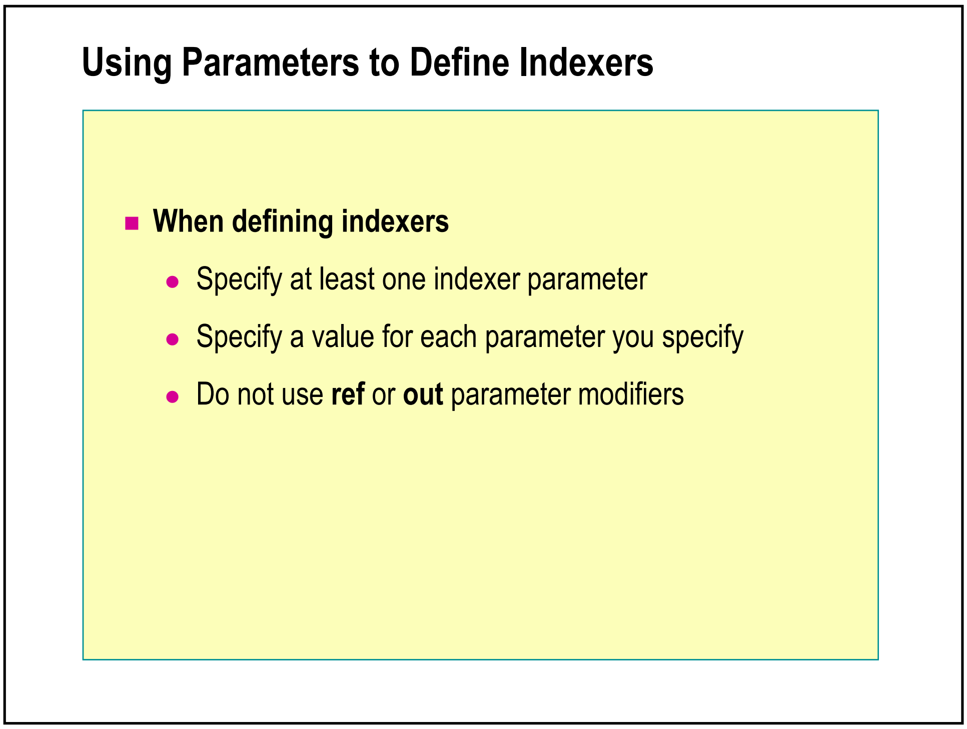
Comparing Indexers to Arrays

- **Similarities**
  - Both use array notation
- **Differences**
  - Indexers can use non-integer subscripts
  - Indexers can be overloaded—you can define several indexers, each using a different index type
  - Indexers are not variables, so they do not denote storage locations—you cannot pass an indexer as a **ref** or an **out** parameter



Comparing Indexers to Properties

- **Similarities**
  - Both use **get** and **set** accessors
  - Neither have an address
  - Neither can be void
- **Differences**
  - Indexers can be overloaded
  - Indexers cannot be static



Using Parameters to Define Indexers

- **When defining indexers**
  - Specify at least one indexer parameter
  - Specify a value for each parameter you specify
  - Do not use **ref** or **out** parameter modifiers

String Example

- **The String class**
  - Is an immutable class
  - Uses an indexer (**get** accessor but no **set** accessor)

```
class String
{
    public char this[int index]
    {
        get {
          if (index < 0 || index >= Length)
              throw new IndexOutOfRangeException( );
          ...
        }
    }
    ...
}
```

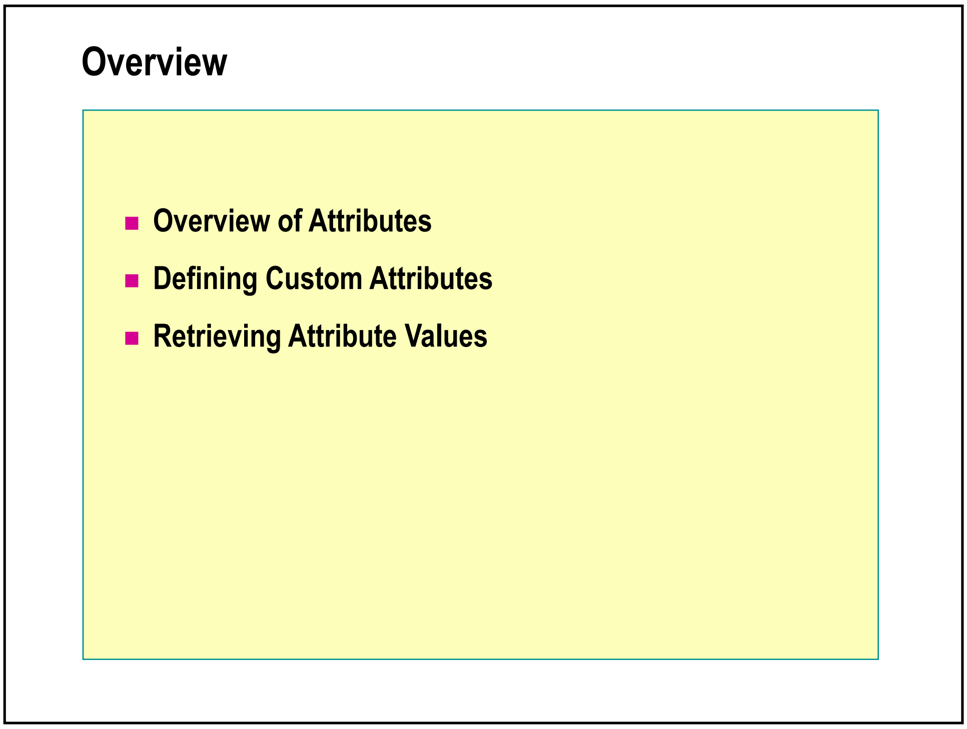
BitArray Example

```
class BitArray
{
   public bool this[int index]
   {
     get {
        BoundsCheck(index);
        return (bits[index >> 5] & (1 << index)) != 0;
     }
     set {
        BoundsCheck(index);
        if (value) {
            bits[index >> 5] |= (1 << index);
        } else {
            bits[index >> 5] &= ~(1 << index);
        }
     }
   }
   private int[ ] bits;
}
```

Review

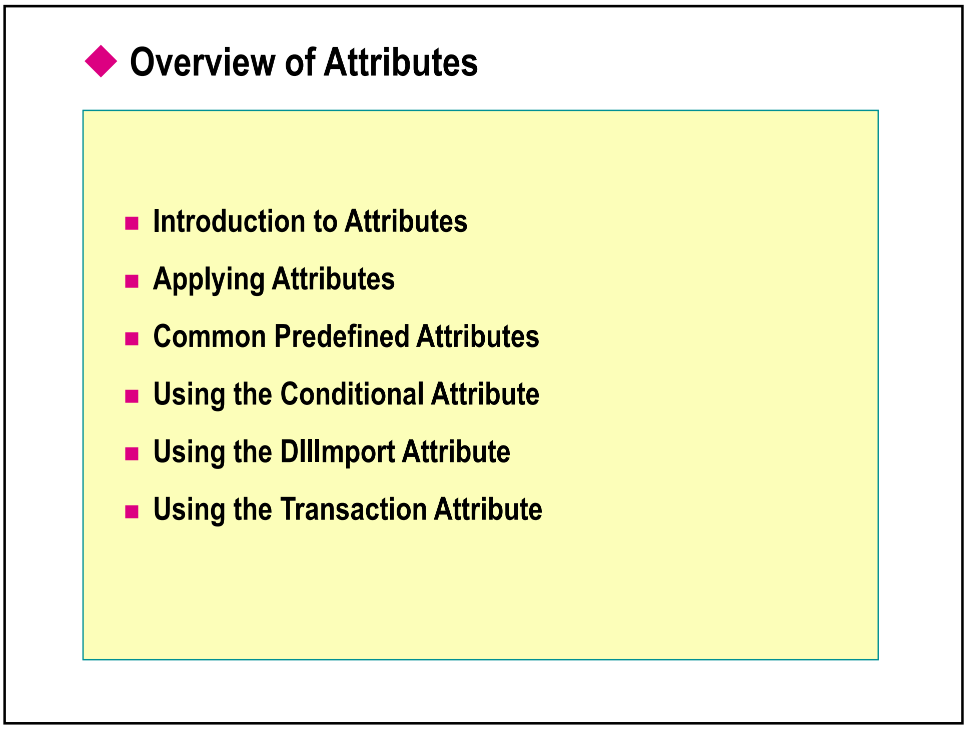
- Using Properties
- Using Indexers

Module 14: Attributes



Overview

- Overview of Attributes
- Defining Custom Attributes
- Retrieving Attribute Values



◆ Overview of Attributes

- Introduction to Attributes
- Applying Attributes
- Common Predefined Attributes
- Using the Conditional Attribute
- Using the DllImport Attribute
- Using the Transaction Attribute

Introduction to Attributes

- **Attributes are:**
  - Declarative tags that convey information to the runtime
  - Stored with the metadata of the element
- **.NET Framework provides predefined attributes**
  - The runtime contains code to examine values of attributes and act on them

Applying Attributes

- **Syntax: Use square brackets to specify an attribute**

```
[attribute(positional_parameters,named_parameter=value, ...)]
element
```

- **To apply multiple attributes to an element, you can:**
  - Specify multiple attributes in separate square brackets
  - Use a single square bracket and separate attributes with commas
  - For some elements such as assemblies, specify the element name associated with the attribute explicitly

Common Predefined Attributes

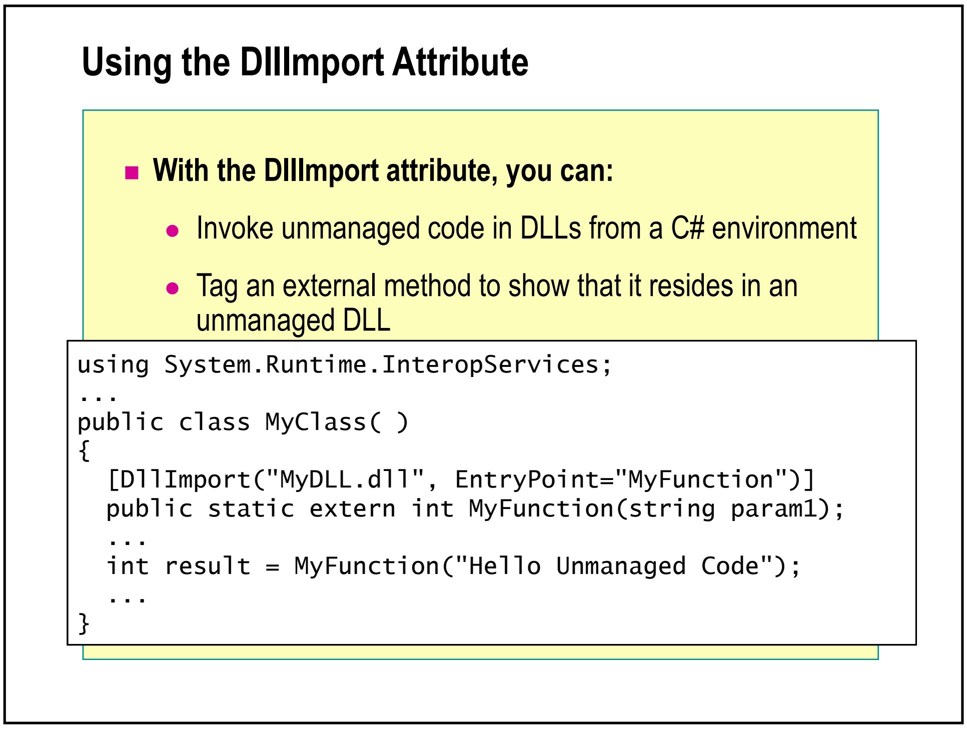
- **.NET provides many predefined attributes**
  - General attributes
  - COM interoperability attributes
  - Transaction handling attributes
  - Visual designer component building attributes

Using the Conditional Attribute

- **Serves as a debugging tool**
  - Causes conditional compilation of method calls, depending on the value of a programmer-defined symbol
  - Does not cause conditional compilation of the method itself

```
using System.Diagnostics;
...
class MyClass
{
  [Conditional ("DEBUGGING")]
  public static void MyMethod( )
  {
    ...
  }
}
```

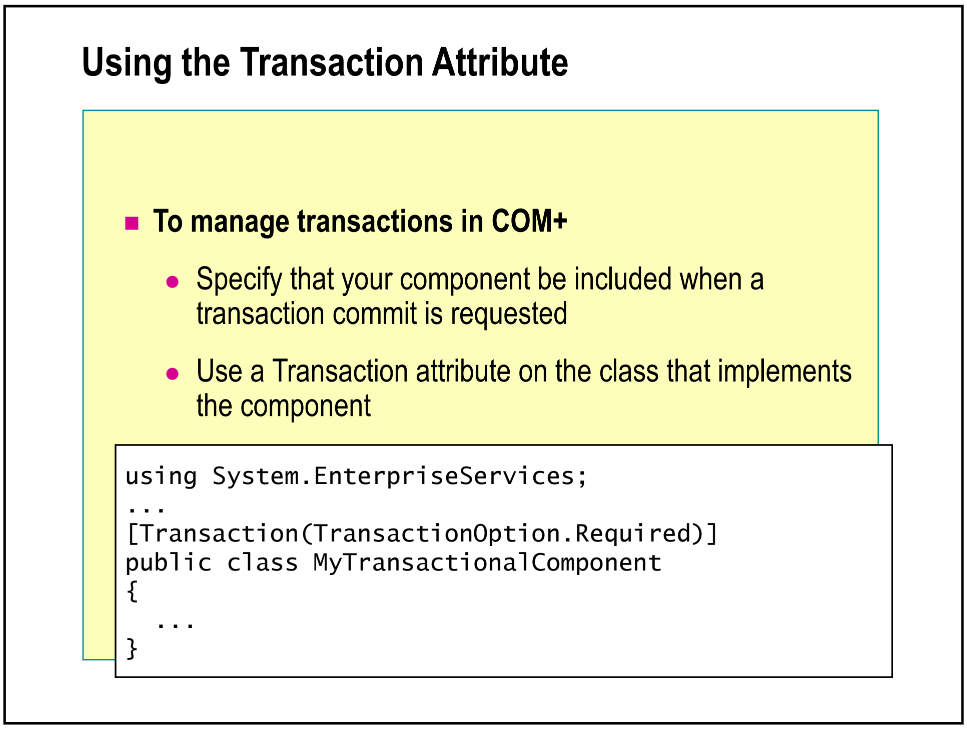
- **Restrictions on methods**
  - Must have return type of **void**
  - Must not be declared as **override**
  - Must not be from an inherited interface



Using the DllImport Attribute

- **With the DllImport attribute, you can:**
  - Invoke unmanaged code in DLLs from a C# environment
  - Tag an external method to show that it resides in an unmanaged DLL

```
using System.Runtime.InteropServices;
...
public class MyClass( )
{
  [DllImport("MyDLL.dll", EntryPoint="MyFunction")]
  public static extern int MyFunction(string param1);
  ...
  int result = MyFunction("Hello Unmanaged Code");
  ...
}
```



Using the Transaction Attribute

- **To manage transactions in COM+**
  - Specify that your component be included when a transaction commit is requested
  - Use a Transaction attribute on the class that implements the component

```
using System.EnterpriseServices;
...
[Transaction(TransactionOption.Required)]
public class MyTransactionalComponent
{
  ...
}
```

◆ Defining Custom Attributes

- Defining Custom Attribute Scope
- Defining an Attribute Class
- Processing a Custom Attribute
- Using Multiple Attributes

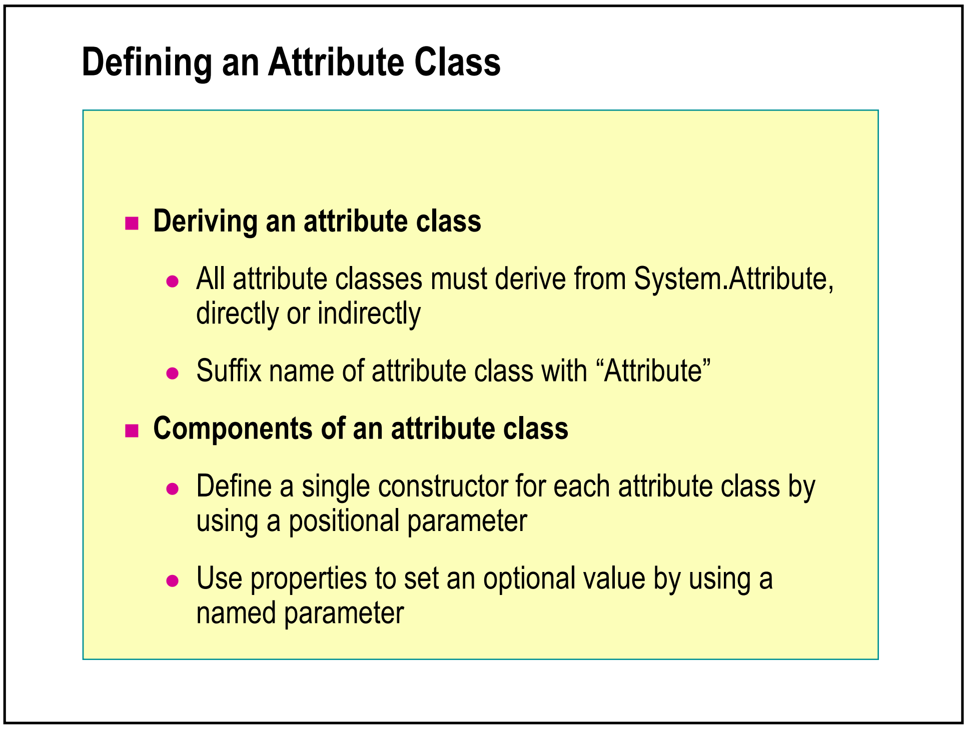
Defining Custom Attribute Scope

- Use the AttributeUsage tag to define scope
  - Example

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute: System.Attribute
{ ... }
```

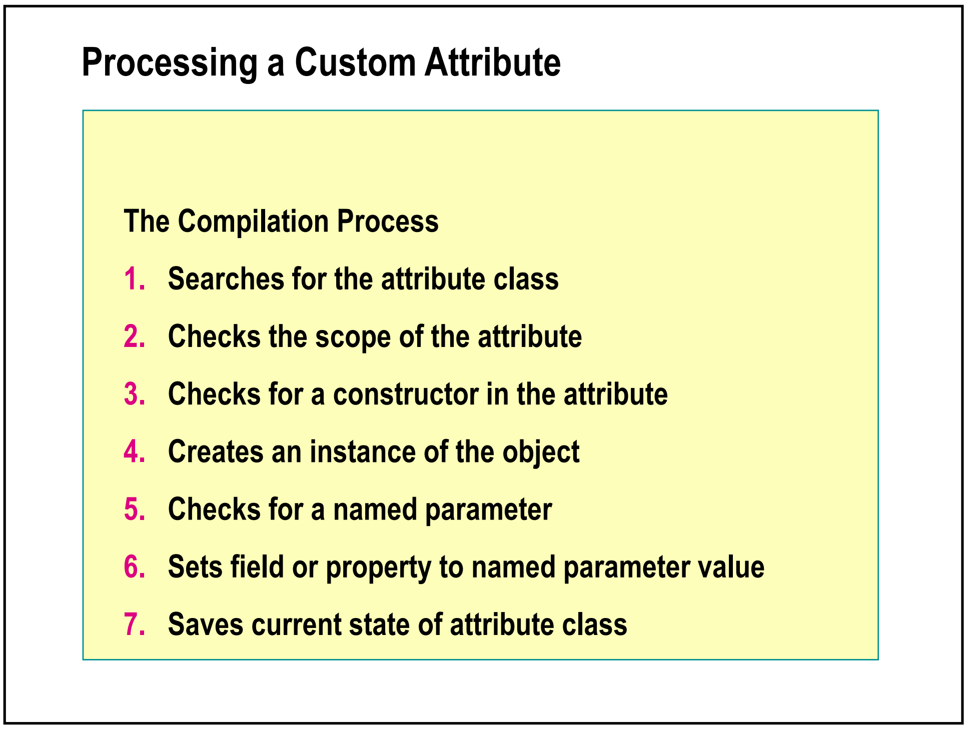
- Use the bitwise “or” operator (|) to specify multiple elements
  - Example

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttribute: System.Attribute
{ ... }
```



Defining an Attribute Class

- **Deriving an attribute class**
  - All attribute classes must derive from System.Attribute, directly or indirectly
  - Suffix name of attribute class with “Attribute”
- **Components of an attribute class**
  - Define a single constructor for each attribute class by using a positional parameter
  - Use properties to set an optional value by using a named parameter



Processing a Custom Attribute

The Compilation Process

1. **Searches for the attribute class**
2. **Checks the scope of the attribute**
3. **Checks for a constructor in the attribute**
4. **Creates an instance of the object**
5. **Checks for a named parameter**
6. **Sets field or property to named parameter value**
7. **Saves current state of attribute class**

Using Multiple Attributes

- **An element can have more than one attribute**
  - Define both attributes separately
- **An element can have more than one instance of the same attribute**
  - Use AllowMultiple = true

◆ Retrieving Attribute Values

- **Examining Class Metadata**
- **Querying for Attribute Information**

Examining Class Metadata

- **To query class metadata information:**
  - Use the MemberInfo class in System.Reflection
  - Populate a MemberInfo object by using System.Type
  - Create a System.Type object by using the typeof operator
- **Example**

```
System.Reflection.MemberInfo typeInfo;
typeInfo = typeof(MyClass);
```

Querying for Attribute Information

- **To retrieve attribute information:**
  - Use **GetCustomAttributes** to retrieve all attribute information as an array

```
System.Reflection.MemberInfo typeInfo;
typeInfo = typeof(MyClass);
object[ ] attrs = typeInfo.GetCustomAttributes(false);
```

  - Iterate through the array and examine the values of each element in the array
  - Use the **IsDefined** method to determine whether a particular attribute has been defined for a class

Review

- Overview of Attributes
- Defining Custom Attributes
- Retrieving Attribute Values

Course Evaluation