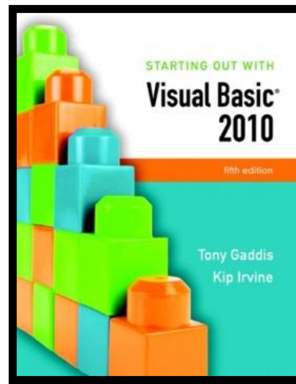
Visual Basic® 2010

fifth edition

Tony Gaddis
Kip Irvine



Chapter 12

Classes, Collections, and Inheritance

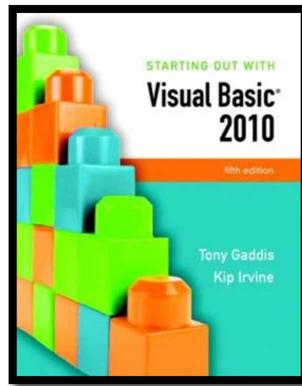
Addison Wesley
is an imprint of

© 2011 Pearson Addison-Wesley. All rights reserved.

Introduction

- This chapter introduces:
  - Abstract Data Types
    - How to create them with classes
  - The process of analyzing a problem
    - Determining its classes
  - Techniques
    - For creating objects, properties, and methods
  - The Object Browser
    - Provides information about classes in your project
  - Collections
    - Structures for holding groups of objects
  - Inheritance
    - A way for new classes to be created from existing ones

Copyright © 2011 Pearson Addison-Wesley



Section 12.1

CLASSES AND OBJECTS

Classes are program structures that define abstract data types and are used to create objects.

Addison Wesley
is an imprint of

© 2011 Pearson Addison-Wesley. All rights reserved.

Object-Oriented Programming

- **Object-oriented programming** (**OOP**) is a way of designing and coding applications with interchangeable software components that can be used to build larger programs
  - First languages appeared in the 1980's
    - SmallTalk, C++, and ALGOL
    - The legacy of these languages has been the gradual development of object-like visual tools for building programs
  - In Visual Basic, forms, buttons, check boxes, list boxes and other controls are all examples of objects
  - These designs help produce programs that are well suited for ongoing development and expansion

Copyright © 2011 Pearson Addison-Wesley

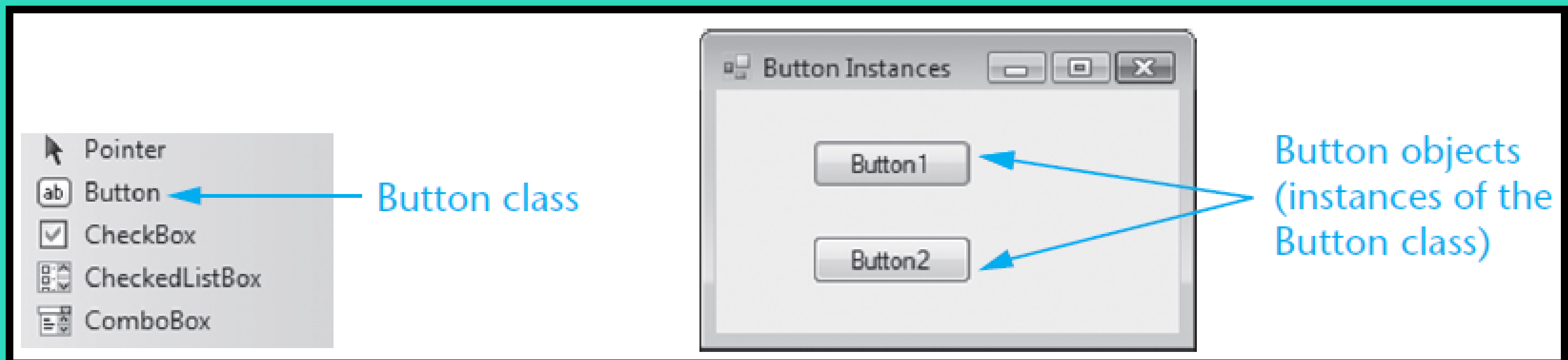
Abstract Data Types

- An **abstract data type** (**ADT**) is a data type created by a programmer
- ADTs are important in computer science and object-oriented programming
- An **abstraction** is a model of something that includes only its general characteristics
- Dog is a good example of an abstraction
  - Defines a general type of animal but not a specific breed, color, or size
  - A dog is like a data type
  - A specific dog is an instance of the data type

Copyright © 2011 Pearson Addison-Wesley

Classes

- A **class** is a program structure that defines an abstract data type
  - Create the class first
  - Then create an instance of the class
    - also called an **object**
  - Class instances share common characteristics
  - Visual Basic forms and controls are classes



Copyright © 2011 Pearson Addison-Wesley

Class Properties, Methods, and Event Procedures

- Programs communicate with an object using the properties and methods of the class
- Class properties:
  - Buttons have Location, Text, and Name properties
- Class methods:
  - The Focus method functions identically for every single button
- Class event procedures:
  - Each button on a form has a different click event procedure

Copyright © 2011 Pearson Addison-Wesley

Object-Oriented Design

- The challenge is to design classes that effectively cooperate and communicate
- Analyze application requirements to determine ADTs that best implement the specifications
- Classes are fundamental building blocks
  - Typically represent nouns of some type
- A well-designed class may outlive the application
  - Other uses for the class may be found

Copyright © 2011 Pearson Addison-Wesley

Finding the Classes

- **Object-oriented analysis** starts with a detailed specification of the problem to be solved
- A term often applied to this process is **finding the classes**
  - For example, specifications for a program that involves scheduling college classes for students:

    We need to keep a ***list of students*** that lets us track the courses they have completed. Each student has a ***transcript*** that contains all information about his or her completed courses. At the end of each semester, we will calculate the grade point average of each ***student***. At times, users will search for a particular ***course*** taken by a student.

  - Notice the italicized nouns and noun phrases:
    - *List of students*, *transcript*, *student*, and *course*
  - These would ordinarily become classes in the program's design

Copyright © 2011 Pearson Addison-Wesley

Looking for Control Structures

- Classes can also be discovered in
  - The description of processing done by the application
  - The description of control structures
    - For example, a description of the scheduling process:

We also want to schedule classes for students, using the college's master schedule to determine the times and room numbers for each student's class. When the optimal arrangement of classes for each student has been determined, each student's class schedule will be printed and distributed.

    - A controlling agent could be implemented with a class
    - For example, a class called **Scheduler**
    - Can be used to match each student's schedule with the college's master schedule

Copyright © 2011 Pearson Addison-Wesley

Describing the Classes

- The next step is to describe classes in terms of attributes and operations
  - **Attributes** are implemented as properties
    - Characteristics of each object
    - Describe the common properties of class objects
  - **Operations** are implemented as methods
    - Actions the class objects perform
    - Messages they can respond to

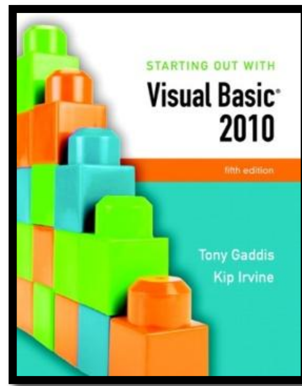
Class	Attributes (properties)	Operations (methods)
`Student`	LastName, FirstName, IdNumber	`Display, Input`
`StudentList`	AllStudents, Count	`Add, Remove, FindStudent`
`Course`	Semester, Name, Grade, Credits	`Display, Input`
`Transcript`	CourseList, Count	`Display, Search, CalculateGPA`

Copyright © 2011 Pearson Addison-Wesley

Interface and Implementation

- The **class interface** is the portion of the class that is visible to the programmer
- The **client program** is written to use a class
  - Refers to the client-server relationship between a class and the programs that use it
- The **class implementation** is the portion of the class that is hidden from client programs
  - Created from private member variables, properties, and methods
  - The hiding of data and procedures in a class is achieved through a process called **encapsulation**
  - Visualize the class as a *capsule* around its data and procedures

Copyright © 2011 Pearson Addison-Wesley



Section 12.2

CREATING A CLASS

To create a class in Visual Basic, you create a class declaration. The class declaration specifies the member variables, properties, methods, and events that belong to the class.

Addison Wesley
is an imprint of

© 2011 Pearson Addison-Wesley. All rights reserved.

Class Declaration and Adding a Class

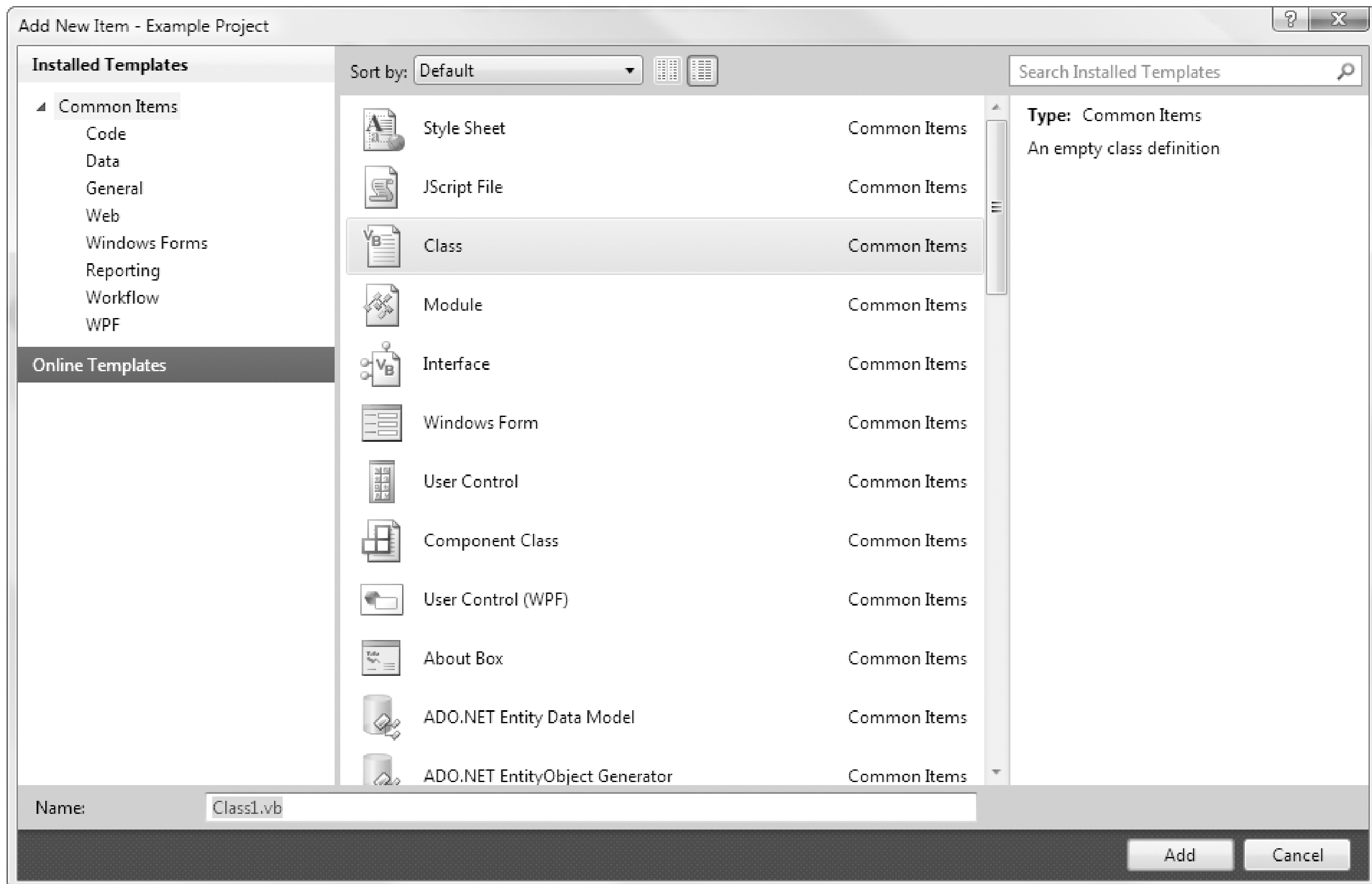
- You create a class in Visual Basic with a **class declaration** using the following general format:

```
Public Class ClassName
   MemberDeclarations
End Class
```

  - ***ClassName*** is the name of the class
  - ***MemberDeclarations*** are the declarations for all the variables, constants, and methods that will belong to the class
- To add a class declaration to a Windows application project:
  1. Click *Project* on the menu bar, the click *Add Class*
  2. Change the default name that appears in the *Name* text box
  3. Click the *Add* button on the *Add New Item* dialog box

Copyright © 2011 Pearson Addison-Wesley

The *Add New Item* Dialog Box



Copyright © 2011 Pearson Addison-Wesley

Member Variables

- A **member variable** is a variable that is declared inside a class declaration using the following general format:

  AccessSpecifer VariableName As DataType

  - ***AccessSpecifier*** determines the accessibility of the variable
    - **Public** access outside of the class or assembly
    - **Friend access** only by other classes inside the same assembly
    - **Private** access only by statements inside the class declaration
  - ***VariableName*** is the name of the variable
  - ***DataType*** is the variable's data type
- As with structures, a class declaration does not create an instance of the class
  - To work with a class, you must create **class objects**, which are instances of the class

Copyright © 2011 Pearson Addison-Wesley

Creating an Instance of a Class

- A two-step process creates an *instance* of a class
- Declare a variable whose type is the class

Dim freshman As Student

- Create instance of the class with **New** keyword and assign the instance to the variable

freshman = New Student

- Or you can accomplish both steps in one statement

Dim freshman As New Student

Copyright © 2011 Pearson Addison-Wesley

Accessing Members

- Once created, you can work with a class object's Public members in code
  - Access the Public members with the dot (**.**) operator
  - Suppose the Student class was declared as follows:

```
Public Class Student
   Public strLastName As String
   Public strFirstName As String
   Public strId As String
End Class
```

  - The following assigns values to each of the member variables for an instance of the **Student** class named **freshman**:

```
' Assign values to the object's members.
freshman.strFirstName = "Joy"
freshman.strLastName = "Robinson"
freshman.strId = "23G794"
```

Copyright © 2011 Pearson Addison-Wesley

Property Procedures

- A **property procedure** is a function that defines a class property using the following general format:

```
Public Property PropertyName() As DataType
    Get
        Statements
    End Get
    Set(ParameterDeclaration)
        Statements
    End Set
End Property
```

- ***PropertyName*** is the name of the property procedure
- ***DataType*** is the type of data that can be assigned to the property
- The **Get section** holds the code that executes when the value is retrieved
- The **Set section** hold the code that executes when the value is stored

Copyright © 2011 Pearson Addison-Wesley

Example Class Property

```
Public Class Student
   Private strLastName As String              ' Holds last name
   Private strFirstName As String             ' Holds first name
   Private strId As String                    ' Holds ID number
   Private dblTestAverage As Double           ' Holds test average

   Public Property TestAverage() As Double
   Get
      Return dblTestAverage
   End Get
   Set(ByVal value As Double)
      dblTestAverage = value
   End Set
End Property
```

Copyright © 2011 Pearson Addison-Wesley

Example Class Property Use

```
Dim freshman As New Student
freshman.TestAverage = 82.3
```

- Stores the value **82.3** in the **TestAverage** property using the **Set** section of the property procedure
- Any statement that retrieves the value in the **TestAverage** property causes the **Get** section of the property procedure to execute

```
dblAverage = freshman.TestAverage

MessageBox.Show(freshman.TestAverage.ToString())
```

Copyright © 2011 Pearson Addison-Wesley

Read-Only Properties

- Client programs can query a **read-only property** and get is value, but cannot modify it
- Here is the general format of a read-only property procedure:

```
Public ReadOnly Property PropertyName() As DataType
    Get
        Statements
    End Get
End Property
```

  - Uses the **ReadOnly** keword
  - Has no **Set** section
  - Only capable of returning a value

Copyright © 2011 Pearson Addison-Wesley

Read-Only Property Example

```
Public ReadOnly Property Grade() As String
    Get
        Dim strGrade As String

        If dblTestAverage >= 90.0 Then
            strGrade = "A"
        ElseIf dblTestAverage >= 80.0 Then
            strGrade = "B"
        ElseIf dblTestAverage >= 70.0 Then
            strGrade = "C"
        ElseIf dblTestAverage >= 60.0 Then
            strGrade = "D"
        Else
            strGrade = "F"
        End If

        Return strGrade
    End Get
End Property
```

Copyright © 2011 Pearson Addison-Wesley

Removing Objects and Garbage Collection

- Memory space is consumed when objects are instantiated
- Objects no longer needed should be removed
- Set object variable to **Nothing** so it no longer references the object

freshman = Nothing

- Object is a candidate for garbage collection when it is no longer referenced by any object variable
- The **garbage collector** monitors for and automatically destroys objects no longer needed

Copyright © 2011 Pearson Addison-Wesley

Going Out of Scope

- An object variable is local to the procedure in which it is declared
  - Will be removed from memory when the procedure ends
  - This is called **going out of scope**
  - The object variable will not be removed from memory if it is referenced by a variable that is outside of the procedure

```
Sub CreateStudent()
Dim sophomore As New Student           ' Create an instance of the Student class.
' Assign values to its properties.
sophomore.FirstName = "Travis"
sophomore.LastName = "Barnes"
sophomore.IdNumber = "17H495"
sophomore.TestAverage = 94.7
g_studentVar = sophomore               ' Assign the object to a global variable.
End Sub
```

Copyright © 2011 Pearson Addison-Wesley

Comparing Object Variables with the **Is** and **IsNot** Operators

- The **Is operator** determines if two variables reference the same object

```
If collegeStudent Is transferStudent Then
   ' Perform some action
End If
```

- The **IsNot operator** determines if two variables do not reference the same object

```
If collegeStudent IsNot transferStudent Then
   ' Perform some action
End If
```

- The special value **Nothing** determines if the variable references any object

```
If collegeStudent Is Nothing Then
   ' Perform some action
End If
If transferStudent IsNot Nothing Then
   ' Perform some action
End If
```

Copyright © 2011 Pearson Addison-Wesley

Creating an Array of Objects

- You can create an array of object variables
- Then create an object for each element to reference

```
Dim mathStudents(9) As Student
Dim intCount As Integer
For intCount = 0 To 9
   mathStudents(intCount) = New Student
Next
```

- Use another loop to release the memory used by the array

```
Dim intCount As Integer

For intCount = 0 To 9
   mathStudents(intCount) = Nothing
Next
```

Copyright © 2011 Pearson Addison-Wesley

Writing Procedures and Functions That Work with Objects

- Can use object variables as arguments to a procedure or function
  - Example: student object **s** as an argument

```
Sub DisplayStudentGrade(ByVal s As Student)
   ' Displays a student's grade.
   MessageBox.Show("The grade for " & s.FirstName &
                   " " & s.LastName & " is " &
                   s.TestGrade.ToString())
End Sub
```

- Pass object variable with the procedure call

```
DisplayStudentGrade(freshman)
```

Copyright © 2011 Pearson Addison-Wesley

Passing Objects by Value and by Reference

- If argument is declared using **ByRef**
  - Values of object properties may be changed
  - The original object variable may be assigned to a different object
- If argument is declared using **ByVal**
  - Values of object properties may be changed
  - The original object variable may ***not*** be assigned to a different object

Copyright © 2011 Pearson Addison-Wesley

Returning an Object from a Function

- Example below instantiates a student object
- Prompts the user for and sets its property values
- Then returns the instantiated object

```
Dim freshman As Student = GetStudent()
```

```
Function GetStudent() As Student
    Dim s As New Student
    s.FirstName = InputBox("Enter the student's first name.")
    s.LastName = InputBox("Enter the student's last name.")
    s.IdNumber = InputBox("Enter the student's ID number.")
    s.TestAverage = CDbl(InputBox("Enter the student's test average."))
    Return s
End Function
```

Copyright © 2011 Pearson Addison-Wesley

Methods

- A **method** is a procedure or function that is a member of a class
  - Performs some operation on the data stored in the class
  - For example, the following statement calls the **Clear** method of the Student object freshman

    freshman.Clear()

```
Public Class Student
    ' Member variables
    Private strLastName As String
    Private strFirstName As String
    Private strId As String
    Private dblTestAverage As Double

   (...Property procedures omitted...)

    ' Clear method
    Public Sub Clear()
        strFirstName = String.Empty
        strLastName = String.Empty
        strId = String.Empty
        dblTestAverage = 0.0
    End Sub
End Class
```

Copyright © 2011 Pearson Addison-Wesley

Constructors

- A **constructor** is a method that is automatically called when an instance of the class is created
  - Think of constructors as initialization routines
  - Useful for initializing member variables or other startup operations
- To create a constructor:
  - Create a method named **New** inside the class
  - Alternatively, select **New** from the method name drop-down list

```
Public Class Student
    ' Member variables
    Private strLastName As String
    Private strFirstName As String
    Private strId As String
    Private dblTestAverage As Double

    ' Constructor
    Public Sub New()
        strFirstName = "(unknown)"
        strLastName = "(unknown)"
        strId = "(unknown)"
        dblTestAverage = 0.0
    End Sub
        (The rest of this class is omitted.)
End Class
```

Copyright © 2011 Pearson Addison-Wesley

Finalizers

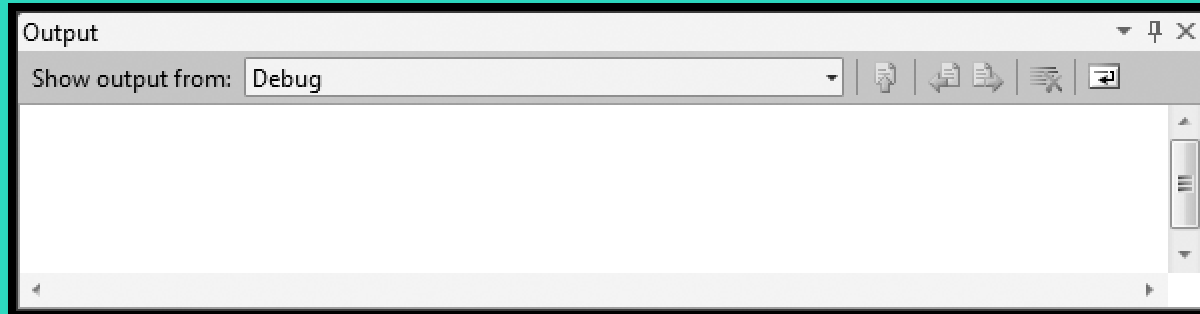
- A **finalizer** is a class method named **Finalize**
  - Automatically called just before an instance of the class is removed from memory
- To create a **Finalize** method:
  - Select **Finalize** from the method name drop-down list
  - The following code template is created for you:

```
Protected Overrides Sub Finalize()
    MyBase.Finalize()
    ' Perform some action
End Sub
```

Copyright © 2011 Pearson Addison-Wesley

Displaying Messages in the *Output* Window

- The ***Output* window** is a valuable debugging tool
- Display it by clicking the *View* menu, *Other Windows*, then *Output* or you can press the **Ctrl + Alt + O** key combination



- Display your own messages with the **Debug.WriteLine** method using the following general format:

Debug.WriteLine(*Output*)

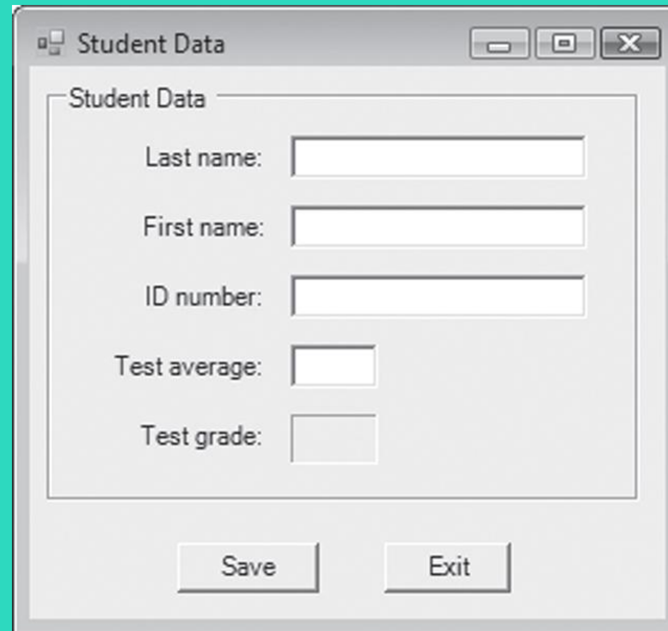
- Enable debug messages by inserting the following in your startup form's **Load** event handler:

Debug.Listeners.Add(New ConsoleTraceListener())

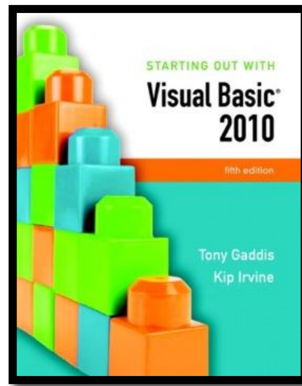
Copyright © 2011 Pearson Addison-Wesley

Tutorial 12-1

- You create the **Student class**
- An application that saves student data to a file
- Display messages in the *output* window



Copyright © 2011 Pearson Addison-Wesley



Section 12.3

COLLECTIONS

A collection holds a group of items. It automatically expands and shrinks in size to accommodate the items added to it. It allows items to be stored with associated key values, which may then be used in searches.

Addison Wesley
is an imprint of

© 2011 Pearson Addison-Wesley. All rights reserved.

Collections

- A **collection** is similar to an array
  - A single unit that contains several items
  - Access individual items with an index value
- Differences from an array include the following:
  - Collections index values begin at 1
  - Collections automatically expand as items are added and shrink as items are removed
  - Items in a collection do not have to be of the same data type

Copyright © 2011 Pearson Addison-Wesley

Creating an Instance of the Collection Class

- Visual Basic provides a class named **Collection**
  - To create an *instance* of the **Collection** class:
  - Declare a variable whose type is the **Collection** class

Dim customers As Collection

  - Create instance of the class with **New** keyword and assign the instance to the variable

customers = New Collection

  - Or you can accomplish both steps in one statement

Dim customers As New Collection

Copyright © 2011 Pearson Addison-Wesley

Adding Items to a Collection

- You add items to a collection with the **Add method** using the following general format:

  CollectionName.Add(Item [, Key])

  - ***CollectionName*** is the name of an object variable that references a collection
  - ***Item*** is the object, variable, or value that is to be added to the collection
  - ***Key*** is an optional string expression that can be used to search for items
    - Must be unique for each member of a collection

Copyright © 2011 Pearson Addison-Wesley

Examples of Adding Items to a Collection

- Declaring a **Collection** object

```
Private customers As New Collection
```

- Inserting a value into the collection

```
customers.Add(myCustomer)
```

- Inserting a value into the collection with an optional key value

```
customers.Add(myCustomer, myCustomer.Name)
```

- Handling duplicate key exceptions

```
Try
   customers.Add(myCustomer, myCustomer.Name)
Catch ex as ArgumentException
   MessageBox.Show(ex.Message)
End Try
```

Copyright © 2011 Pearson Addison-Wesley

Accessing Items by their Indexes

- You can access an item in a collection by passing an integer to the **Item method** as follows:

  CollectionName.Item(index)

  - ***CollectionName*** is the name of the collection object variable
  - ***index*** is the integer index of the item that you want to retrieve
  - The **Item** method returns an Object
  - Call the **Ctype** method to cast the Object to the type needed

```
Dim cust As Customer = CType(customers.Item(1), Customer)
MessageBox.Show("Customer found: " & cust.Name & ": "& cust.Phone)
```

  - **Item** is the default method for collections, so you can use an abbreviated format, as in the following example:

```
Dim cust As Customer = CType(customers(3), Customer)
```

Copyright © 2011 Pearson Addison-Wesley

The **IndexOutOfRange** Exception

- An **IndexOutOfRange** exception occurs if an index is used that does not match any item in a collection
  - The following code example shows how to handle the exception:

```
Try
   Dim cust As Customer

   Dim index As Integer = CInt(txtIndex.Text)

   cust = CType(customers.Item(index), Customer)

   MessageBox.Show("Customer found: " & cust.Name & ": " & cust.Phone)

Catch ex As IndexOutOfRangeException
   MessageBox.Show(ex.Message)
End Try
```

Copyright © 2011 Pearson Addison-Wesley

The Count Property

- Each collection has a **Count** property
  - Holds the number of items in the collection
- The following code example:
  - Uses a **For Next** loop
  - With the **Count** property as the upper limit
  - To add the contents of the collection to a list box

```
Dim intX As Integer
For intX = 1 To names.Count
   lstNames.Items.Add(names(intX).ToString())
Next
```

Copyright © 2011 Pearson Addison-Wesley

Searching for an Item by Key Value Using the **Item** Method

- The **Item** method can be used to retrieve an item with a specific key value using the following general format:

  CollectionName.Item(Expression)

  - ***CollectionName*** is the name of a collection
  - ***Expression*** can be a numeric or string expression
    - If a string expression is used
      - The key value that matches the string is returned
    - If a numeric expression is used, it becomes the index value
      - The member at the specified index is returned
  - If no member exists with an index or key value matching ***Expression***, an **IndexOutOfRange** exception occurs

```
Dim s As Student = CType(studentCollection.Item("49812"), Student)
```

Copyright © 2011 Pearson Addison-Wesley

Using References versus Copies

- When an item in a collection is:
  - A fundamental Visual Basic Type
    - Integer, String, Decimal, and so on
    - Only a copy of the member is returned
    - its value cannot be changed
  - A class object
    - A reference to the object is returned
    - Its value can be changed

Copyright © 2011 Pearson Addison-Wesley

Using the **For Each…Next** Loop with a Collection

- You may use the **For Each…Next** loop to access the individual members of a collection
  - Eliminates the need for a counter variable
  - For example:

```
Dim s As Student
For Each s In studentCollection
   MessageBox.Show(s.LastName)
Next
```

Copyright © 2011 Pearson Addison-Wesley

Removing Members

- Use the **Remove method** to remove a member from a collection using the following general format:

  CollectionName.Remove(Expression)

  - ***CollectionName*** is the name of a collection
  - ***Expression*** can be a numeric or string expression
    - If a string expression is used
      - The key value that matches the string is removed
      - An **ArgumentExeception** occurs if the key value does not match an item in the collection
    - If a numeric expression is used, it becomes the index value
      - The member at the specified index is removed
      - An **IndexOutOfRange** exception occurs if the index does not match any item in the collection

Copyright © 2011 Pearson Addison-Wesley

Preventing Exceptions when Removing Members

- To avoid throwing an exception with the **Remove** method:
  - Always check the range of the index

```
Dim intIndex As Integer
' (assign value to intIndex...)
If intIndex > 0 and intIndex <= studentCollection.Count Then
   studentCollection.Remove(intIndex)
End If
```

  - Make sure a key value exists before using it

```
Dim strKeyToRemove As String
' (assign value to strKeyToRemove...)
If studentCollection.Contains(strKeyToRemove) Then
   studentCollection.Remove(strKeyToRemove))
End If
```

Copyright © 2011 Pearson Addison-Wesley

Writing Sub Procedures and Functions That Use Collections

- Sub procedures and functions can accept collections as arguments
  - Remember that a collection is an instance of a class
  - Follow the same guidelines for:
    - Passing a class object as an argument
    - Returning a class object from a function

Copyright © 2011 Pearson Addison-Wesley

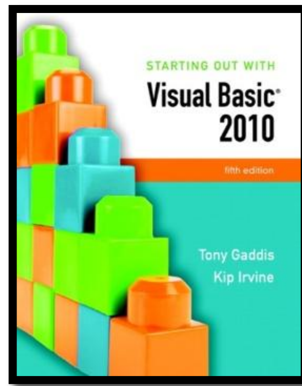
Relating the Items in Parallel Collections

- Sometimes it is useful to store related data in two or more parallel collections
- Use a unique key value to relate the items in the collections
  - An ID or employee number for instance
  - For example, the following code works with items in parallel collections by using the employee number 55678 as the key value

```
Dim hoursWorked As New Collection     ' To hold hours worked
Dim payRates As New Collection        ' To hold hourly pay rates

hoursWorked.Add(40, "55678")          ' Store a value using the key value
payRates.Add(12.5, "55678")           ' Use the same key value again

' The key value is used once  again when retrieving the related  data
sngGrossPay = hoursWorked.Item("55678") * payRate.Item("55678")
```



Section 12.4

FOCUS ON PROBLEM SOLVING: CREATING THE *STUDENT COLLECTION* APPLICATION

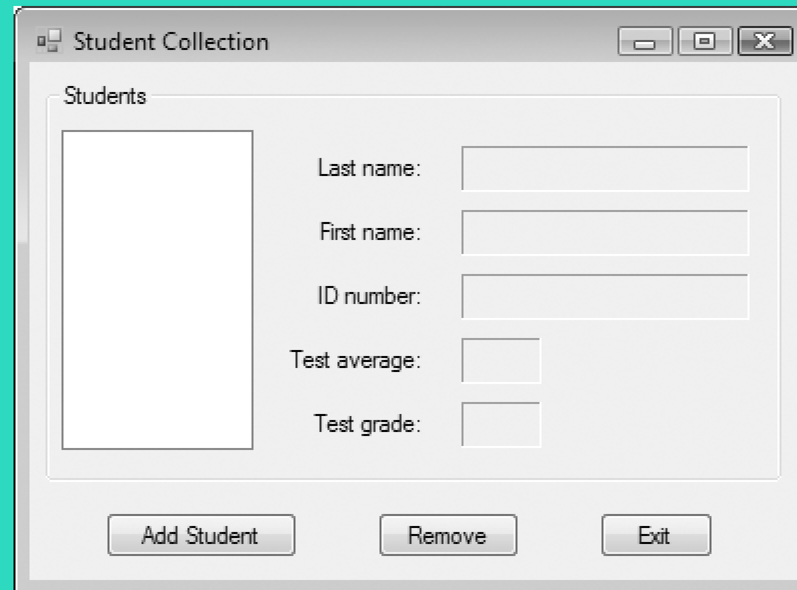
Create the *Student Collection* application

Addison Wesley
is an imprint of

© 2011 Pearson Addison-Wesley. All rights reserved.

The **MainForm** Form

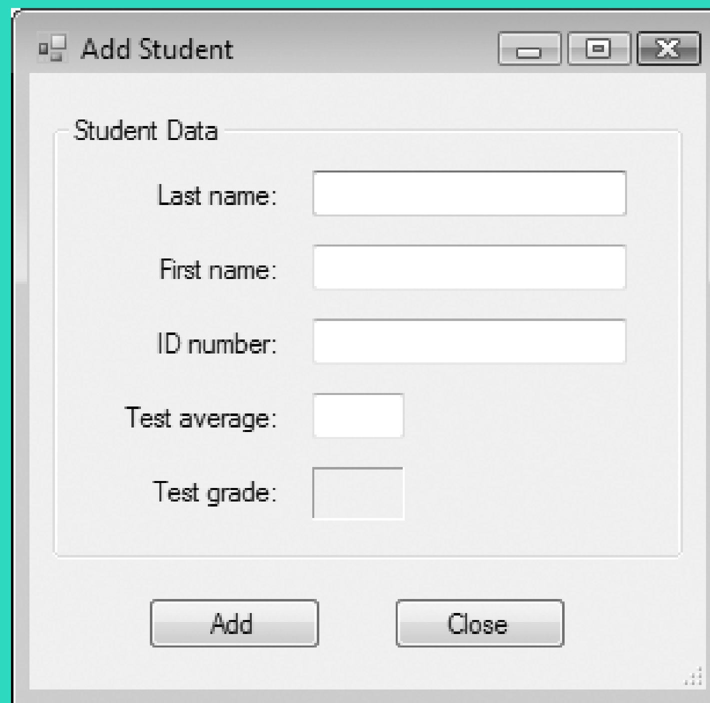
- Displays a list of student ID numbers in the list box
- When an ID number is selected, student data is displayed in the labels
- The *Add Student* button causes the **AddForm** form to be displayed
- The *Remove* button removes a student with the currently selected ID number



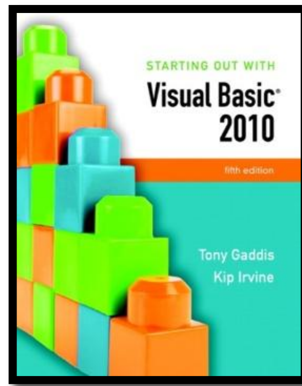
Copyright © 2011 Pearson Addison-Wesley

The **AddForm** Form

- Allows the user to enter student data in the text boxes
- The *Add* button adds the student data to the collections



Copyright © 2011 Pearson Addison-Wesley



Section 12.5

THE OBJECT BROWSER

The Object Browser is a dialog box that allows you to browse all classes and components available to your project.

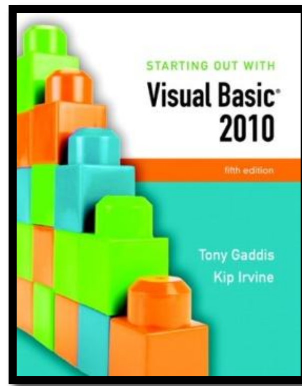
Addison Wesley
is an imprint of

© 2011 Pearson Addison-Wesley. All rights reserved.

The Object Browser

- The **Object Browser** is a dialog box that displays information about objects
- You can use the object browser to examine:
  - Classes you have created in your project
  - Namespaces, classes, and other components that Visual Basic makes available to your project
- Tutorial 12-3 guides you through the process of using the Object browser to examine the classes you created in the *Student Collection* project

Copyright © 2011 Pearson Addison-Wesley



Section 12.6

INTRODUCTION TO INHERITANCE

Inheritance allows a new class to be based on an existing class. The new class inherits the accessible member variables, methods, and properties of the class on which it is based.

Addison Wesley
is an imprint of

© 2011 Pearson Addison-Wesley. All rights reserved.

What is Inheritance?

- **Inheritance** allows new classes to derive their characteristics from existing classes
- The Student class may have several types of students such as
  - GraduateStudent
  - ExchangeStudent
  - StudentEmployee
- These can become new classes and share all the characteristics of the Student class
- Each new class would then add specialized characteristics that differentiate them

Copyright © 2011 Pearson Addison-Wesley

Base and Derived Classes

- The **Base Class** is a general-purpose class that other classes may be based on
  - Think of the base class as a parent
- A **Derived Class** is based on the base class and inherits characteristics from it
  - Think of the derived class as a child

Copyright © 2011 Pearson Addison-Wesley

The Vehicle Base Class

- Consider a Vehicle class with the following:
  – Private variable for number of passengers
  – Private variable for miles per gallon
  – Public property for number of passengers (Passengers)
  – Public property for miles per gallon (MilesPerGallon)
- This class holds general data about a vehicle
- Can create more specialized classes from the Vehicle class

Copyright © 2011 Pearson Addison-Wesley

The Truck Derived Class

- Truck class derived from Vehicle class
  - Inherits all non-private methods, properties, and variables of Vehicle class
- Truck class defines two properties of its own
  - MaxCargoWeight – holds top cargo weight
  - FourWheelDrive – indicates if truck is 4WD
- The *Vehicle Inheritance* program in the Chapter 12 student sample programs folder contains the code for the Vehicle and Truck classes

Copyright © 2011 Pearson Addison-Wesley

Overriding Properties and Methods

- Sometimes a base class property procedure or method must work differently for a derived class
  - You can **override** base class method or property
  - You must write the method or property as desired in the derived class using same name
- When an object of the derived class accesses the property or calls the method
  - The overridden version in derived class is used
  - The base class version is not used

Copyright © 2011 Pearson Addison-Wesley

Overriding Procedure Example

- Vehicle class has no restriction on number of passengers
- But may wish to restrict the Truck class to two passengers at most
- Can override Vehicle class Passengers property by:
  – Coding Passengers property in derived class
  – Specify **Overridable keyword** in base class property
  – Specify **Overrides keyword** in derived class property

Copyright © 2011 Pearson Addison-Wesley

Overridable Property Procedure in the Base Class Example

- **Overridable** keyword added to **Vehicle** base class property procedure

```
Public Overridable Property Passengers() As Integer
    Get
        Return intPassengers
    End Get
    Set(ByVal value As Integer)
        intPassengers = value
    End Set
End Property
```

Copyright © 2011 Pearson Addison-Wesley

Overridden Property Procedure in the Derived Class Example

- **Overrides** keyword and new logic added to **Truck** derived class property procedure
- The **MyBase keyword** refers to the base class

```
Public Overrides Property Passengers() As Integer
   Get
      Return MyBase.Passengers
   End Get
   Set(ByVal value As Integer)
      If value >= 1 And value <= 2 Then
         MyBase.Passengers = value
      Else
         MessageBox.Show("Passengers must be 1 or 2.", "Error")
      End If
   End Set
End Property
```

Copyright © 2011 Pearson Addison-Wesley

Overriding Methods

- The general format of a procedure that overrides a base class procedure is as follows:

  AccessSpecifier Overrides Sub ProcedureName()
     Statements
  End Sub

- The general format of a function that overrides a base class function is as follows:

  AccessSpecifier Overrides Function FunctionName() As DataType
     Statements
  End Sub

- When overriding methods and procedures, remember that:
  - A derived class cannot access methods or property procedures in the base class that are declared as **Private**
  - A derived class must keep the same access level as the base class

Copyright © 2011 Pearson Addison-Wesley

Overriding the **ToString** Method

- Every class that you create in Visual Basic is derived from a built-in class named **Object**
  - The **Object class** has a method named **ToString**
  - You can override this method so it returns a string representation of the data stored in an object

```
' Overridden ToString method
Public Overrides Function ToString() As String
    ' Return a string representation of a vehicle.
    Dim str As String

    str = "Passengers: " & intPassengers.ToString() &
          " MPG: " & dblMPG.ToString()
    Return str
End Function
```

Copyright © 2011 Pearson Addison-Wesley

Base Class and Derived Class Constructors

- A constructor (named **New**) may be defined for both the base class and a derived class
- When a new object of the derived class is created, both constructors are executed
  - The constructor of the base class will be called first
  - Then the constructor of the derived class will be called

Copyright © 2011 Pearson Addison-Wesley

Base and Derived Class Constructors Example

```
Public Class Vehicle

   Public Sub New()
      MessageBox.Show("This is the base class constructor.")
   End Sub
   ' (other properties and methods...)
End Class

Public Class Truck
      Inherits Vehicle

   Public Sub New()
      MessageBox.Show("This is the derived class constructor.")
   End Sub
   ' (other properties and methods...)
End Class
```

Copyright © 2011 Pearson Addison-Wesley

Protected Members

- The **Protected access specifier** may be used in the declaration of a base class member, such as the following:

  Protected decCost As Decimal

  - Protected base class members are treated as
    - Public to classes derived from this base
    - Private to classes not derived from this base

- In Tutorial 12-4, you complete an application that uses inheritance

Copyright © 2011 Pearson Addison-Wesley