Module name: PROGRAMMING 2A
Module code : PROG6211

Ishmael Chikoo

Email: ichikoo@iie.ac.za

Varsity College

SCHOOL OF INFORMATION TECHNOLOGY

Learning Unit 4

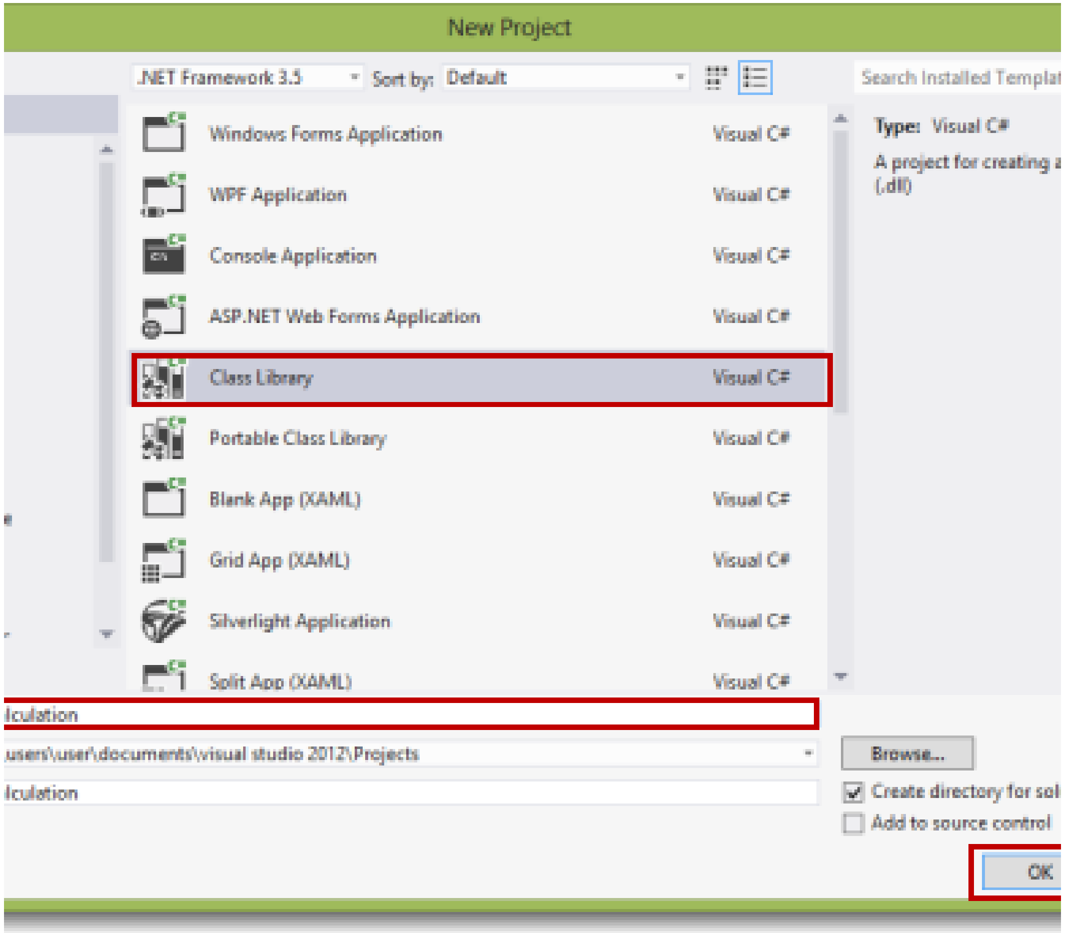
Theme Breakdown:

- ❑ DDL
- ❑ Delegates
- ❑ TESTING

DDL

- A Dynamic Link library (DLL) is a library that contains functions and codes
- Used by more than one program at a time.
- Created once as a DLL file and use it in many applications.
- To use it - add the reference/import the DLL File.
- Both DLL and .exe files are executable program modules but the difference is that we cannot execute DLL files directly.

Creating DDL file



.NET Framework 3.5

Sort by: Default

Search Installed Templat

Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
ASP.NET Web Forms Application	Visual C#
Class Library	Visual C#
Portable Class Library	Visual C#
Blank App (XAML)	Visual C#
Grid App (XAML)	Visual C#
Silverlight Application	Visual C#
Split App (XAML)	Visual C#

Type: Visual C#

A project for creating a
(.dll)

lculation

users\user\documents\visual studio 2012\Projects

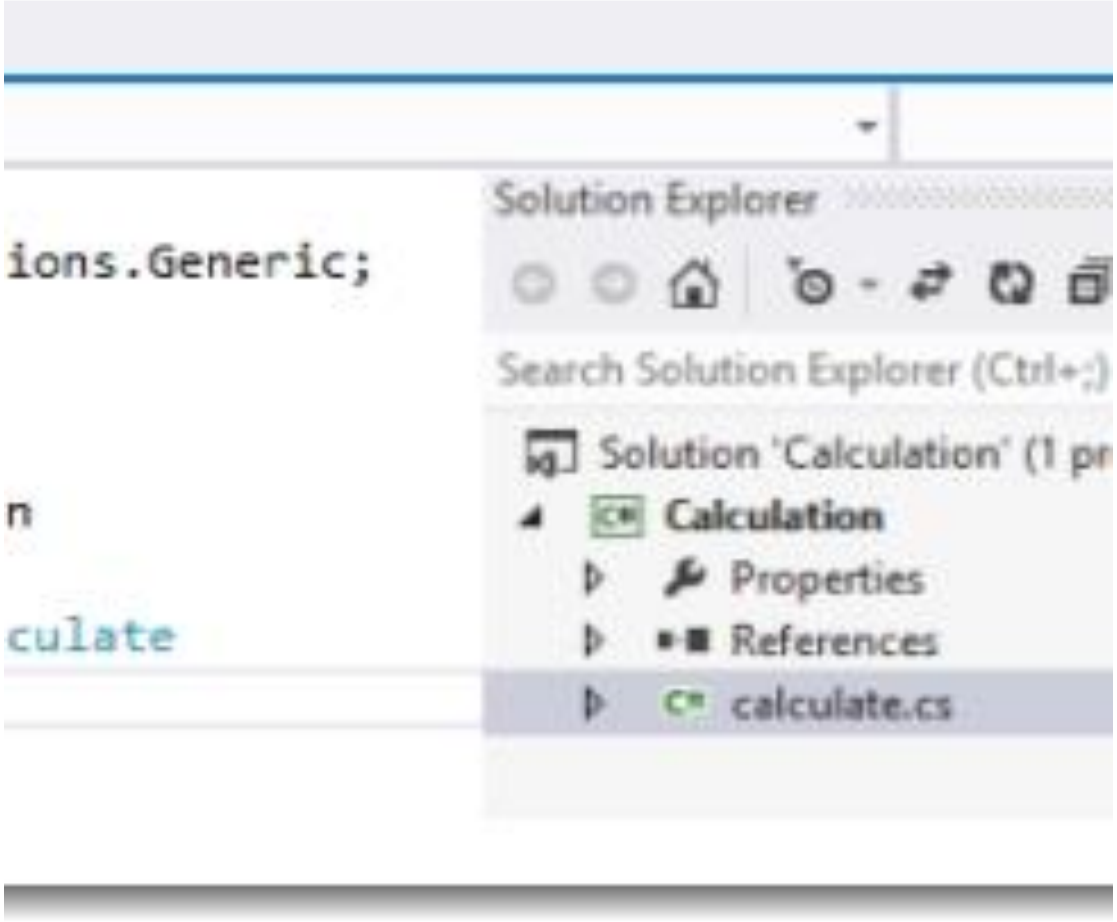
lculation

☑ Create directory for sol

□ Add to source control

- Create a new project of type “Class library”

Create calculate

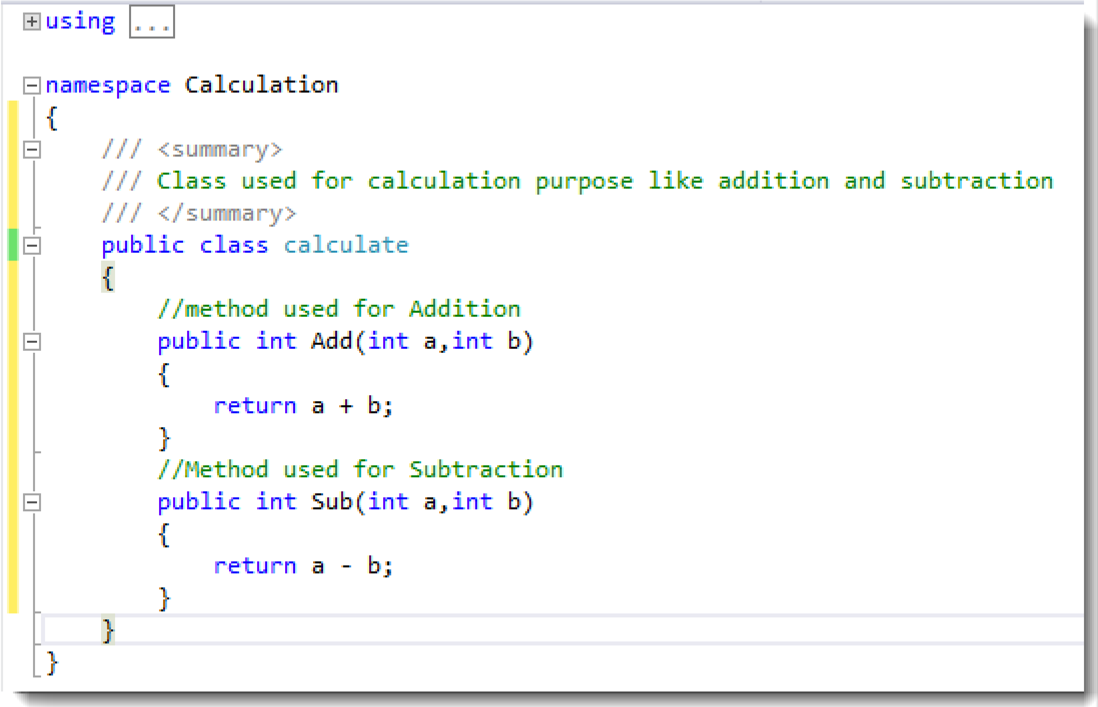


```
ions.Generic;

n

culate
```

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Calculation' (1 pr
Calculation
Properties
References
calculate.cs

- Name the class calculate

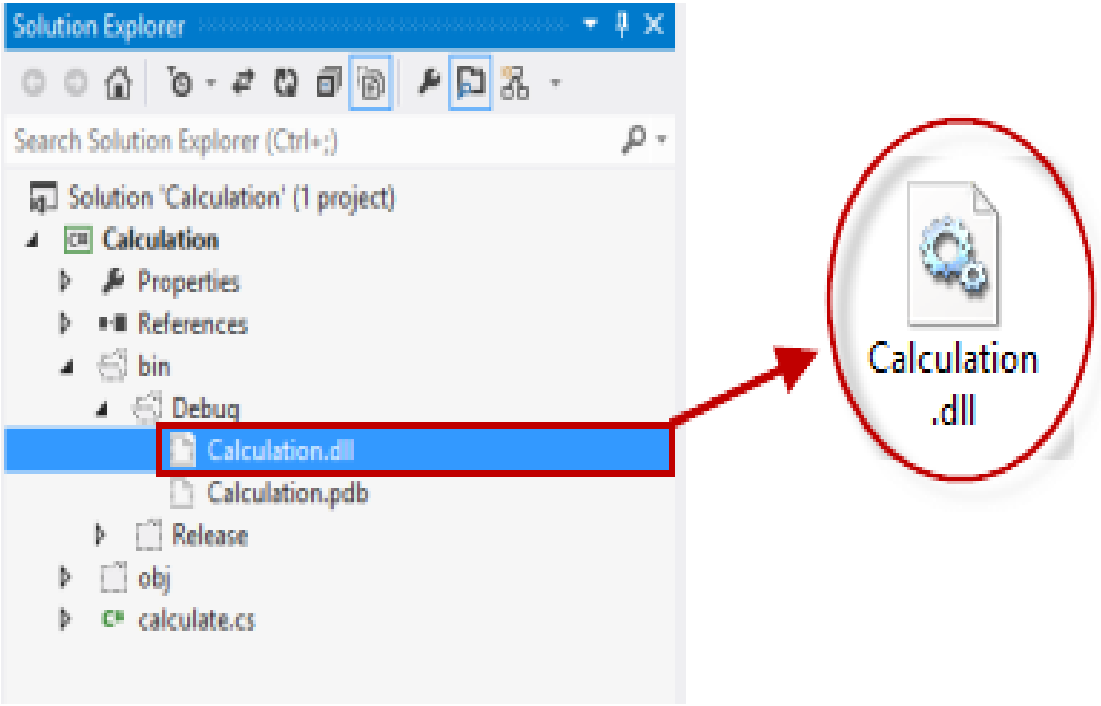
Add methods to the class



```
using ...

namespace Calculation
{
    /// <summary>
    /// Class used for calculation purpose like addition and subtraction
    /// </summary>
    public class calculate
    {
        //method used for Addition
        public int Add(int a,int b)
        {
            return a + b;
        }
        //Method used for Subtraction
        public int Sub(int a,int b)
        {
            return a - b;
        }
    }
}
```

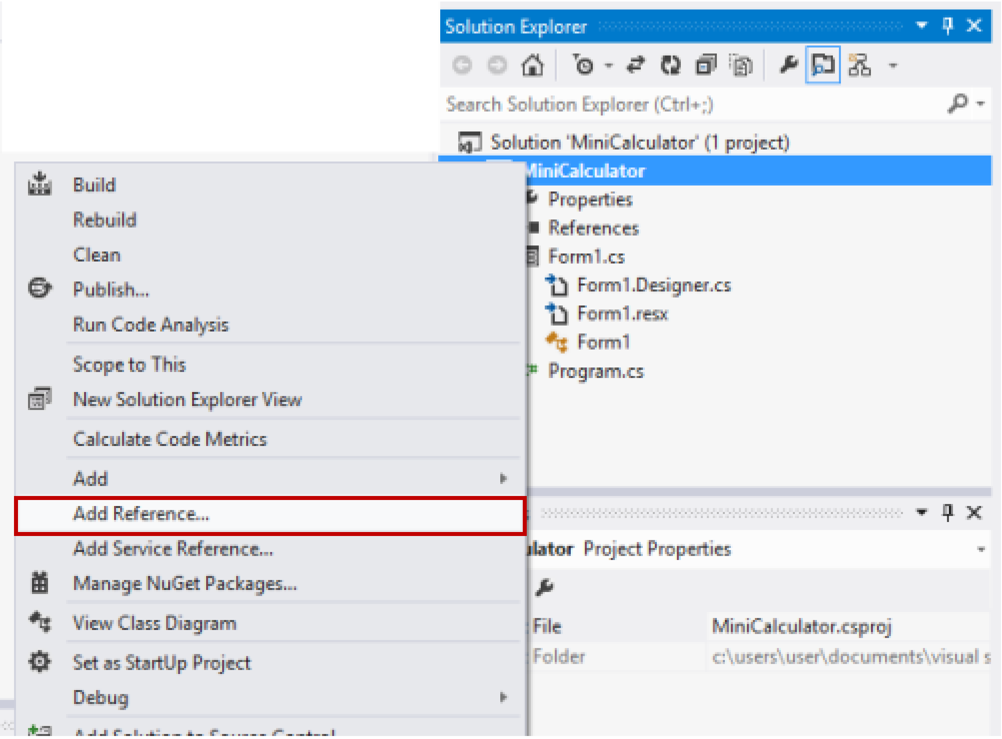
- Add methods to do calculations in the calculate class

Compile the DDL



- Build the solution (F6).
- If the build is successful, then you will see a "calculation.dll" file in the "bin/debug" directory of your project.

Use DDL

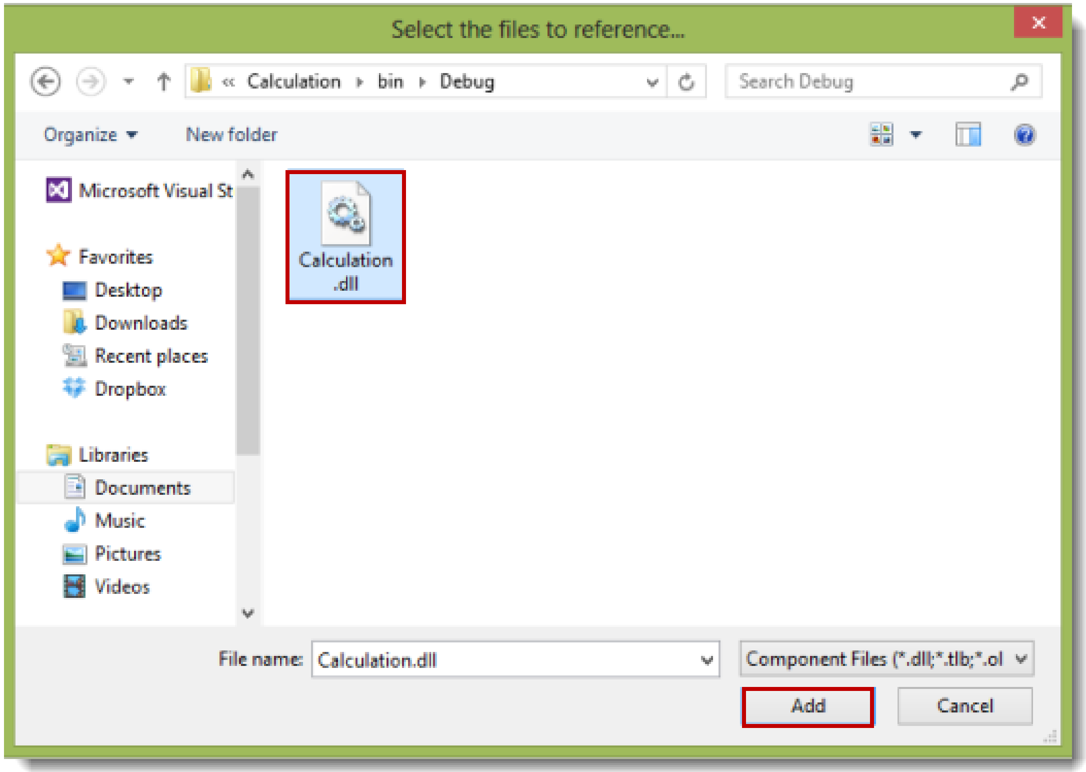


File MiniCalculator.csproj
Folder c:\users\user\documents\visual s

- In your projects we created before.

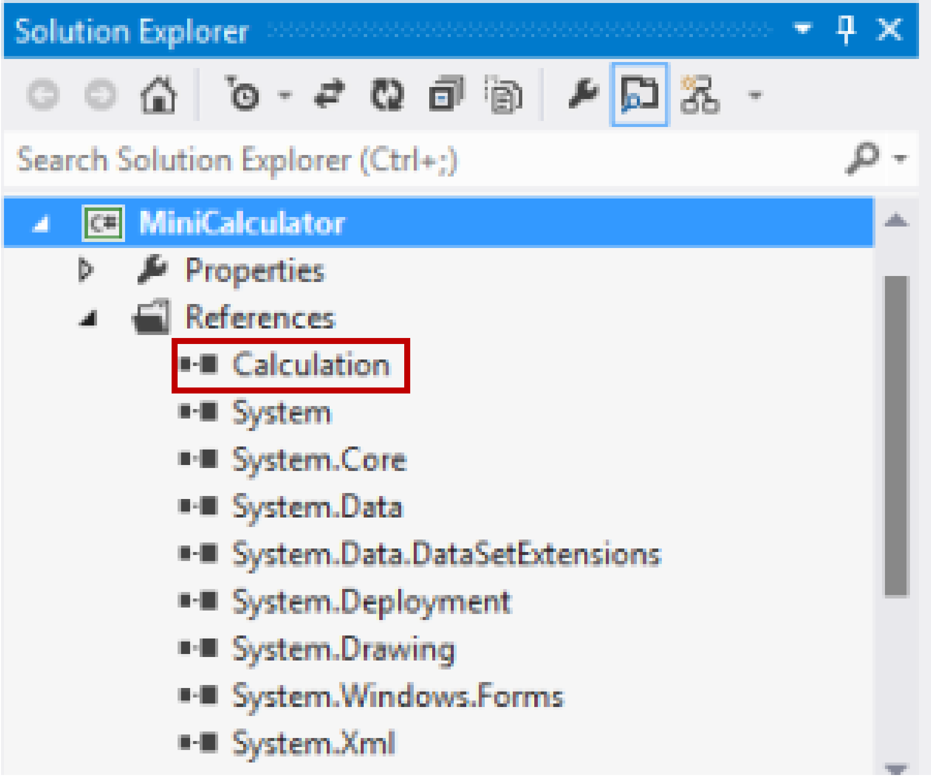
Use DDL

- Select the DLL file and add it to the project.



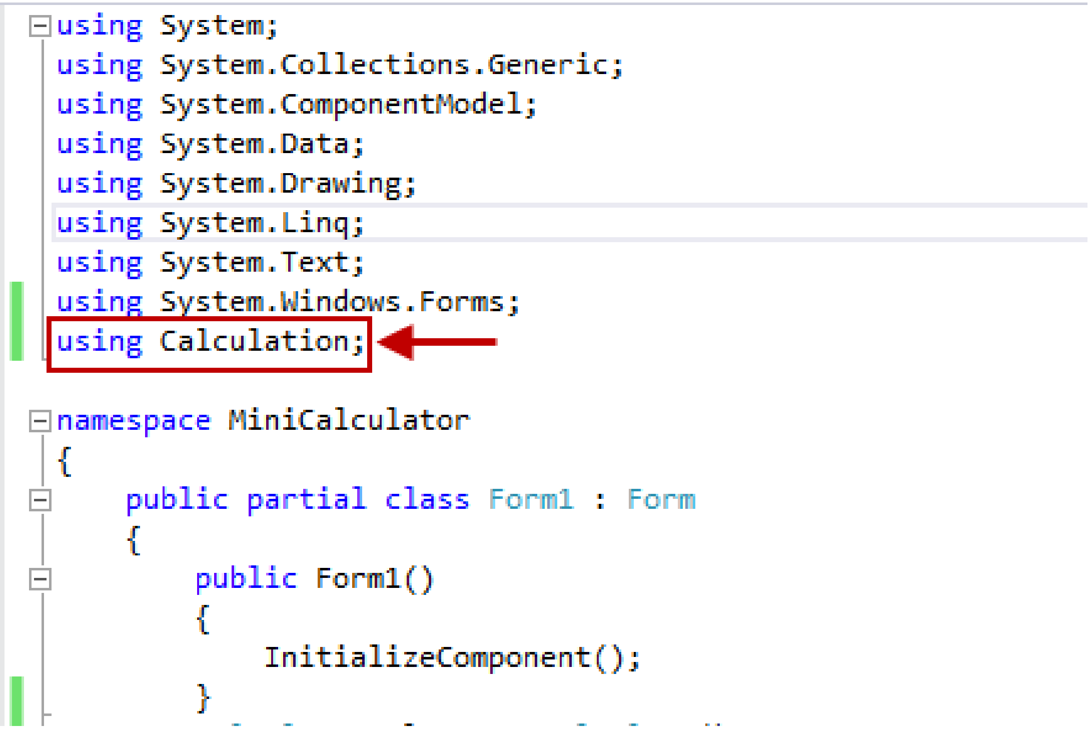
Use DDL

- After adding the file, you will see that the calculation namespace has been added (in references)



Use DDL

- Add the namespace ("using calculation;") in your project



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Calculation;

namespace MiniCalculator
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
```

Use DDL

- You can create an object of the calculate class directly and call the method from the ddl.

```
calculate cal = new calculate();
int i = cal.Add(4,6);
```

Using DDL Activity

1. Create a new class library project called **Validations**
2. In validations project, add a method called **CheckInput** that takes a **string as a parameter**.
3. The method must return a Boolean **true** if the input is a valid input, and **false** if the input is not valid input.
4. Build the DDL.
5. Create a new project called UsingDDL where you will use the DDL, copy and paste it in the project’s bin /debug folder

Using DDL Activity

- In your **UsingDDL** project, create an **arrayList** to add numbers .
- Ask the user to enter a number, then verify it using the method in the DDL, remember to reference the DDL
- If the number is correct , then save it to the arrayList.
- Ask the user if they are **done** or **not**
- If not done ask them to continue capturing
- Create a new class called **PrintDataClass**, where you add a method **PrintData**, that takes an arrayList as a parameter and prints data in that arrayList when user is done.

Delegates

- ❑ Delegate in C# is a type that allows you to pass a method as a parameter and get a return value.
- ❑ Delegates are often used to deal with events
- ❑ The C# delegate type in . NET represents a delegate.
- ❑ Delegate is a special type of an object
- ❑ The object is used to define contained data, but a delegate contains the details of a method

Delegates

- ❑ When dealing with events, methods are passed as parameters of other methods
- ❑ A delegate is a class that encapsulates a method signature.
- ❑ A delegate is something that gives a name to a method signature

```
 public delegate int DelegateMethod(int x, int y);
```

- ❑ Signature has a return type and parameters if the method takes parameters

Delegates

- ❑ In class define its object or its instance in delegate
- ❑ Class of the object it reference doesn’t matter to a delegate
- ❑ Delegates are object-oriented, type-safe, and very secure, as they ensure that the signature of the method being called is correct
- ❑ Good for event handling

Delegates

- ❑ Two types of delegates
- ❑ Singlecast delegate :
  - Point to a single method at a time
  - Assigned to a single method at a time
  - Derived from System.Delegate class
- ❑ Multicast delegate:
  - Delegate wrapped with more than one method
  - Point to more than one function at a time
  - Derived from System,MulticastDelegate Class

Defining delegates in C#

```
[attributes] [modifiers] delegate ReturnType Name ([formal-parameters]);
```

- Attributes factor - a normal C# attribute.
- Modifier - new, public, private, protected, or internal.
- ReturnType - data types we have used. It can also be a type void or the name of a class.
- Name - must be a valid C# name.
- Eg delegate with not parameter below

```
public delegate void DelegateExample();
```

Delegates

- ❑ Instantiation

```
DelegateExample d1 = new DelegateExample(Display);
```

- ❑ Invocation

```
d1();
```

- ❑ Singlecast delegate:

Delegates

- ❑ Special types of .NET classes whose **instances store references** (addresses) to **methods** as opposed to storing actual data.
- ❑ Delegates enable you to pass methods as parameters into other methods.
- ❑ Encapsulate a reference to a method inside a delegate object.
- ❑ A delegate object can be passed to code which can call the referenced method, without having to know at compile time

Defining Delegates

- delegate base class type is defined in the System namespace

```
delegate string ReturnsSimpleString( );
```

- Every delegate type has a signature (0 or more parameters)
- A methods signature includes its name, parameters, and parameter types
- Signature of a delegate include a return type or the keyword void as part of its heading.

Creating Delegate Instances

- ❑ delegate instance is defined using the method name as the argument inside the parentheses
- ❑ instantiates the ReturnsSimpleString delegate with the EndStatement( ) method as the argument

```
ReturnsSimpleString saying3 = new
ReturnsSimpleString(EndStatement);
```

- ❑ the EndStatement argument does not include the parentheses, even though EndStatement is a method.
- ❑ A reference to the address of the method is sent as an argument.

Using Delegates

```
ReturnsSimpleString saying3 = new
ReturnsSimpleString(EndStatement);
Console.WriteLine(saying3( ));
```

- Console.WriteLine( ) method of the class calls the delegate instance, saying3( ), which calls the EndStatement( ) method to display “in 10 years.”
- A reference to the address of the method is sent as an argument.

```
delegate string ReturnsSimpleString( );
class DelegateExample
{
```

Using Delegates

```
static void Main ( )
{
    int age = 18;
    ReturnsSimpleString saying1 = new
    ReturnsSimpleString(AHeading);
    ReturnsSimpleString saying2 = new
    ReturnsSimpleString((age + 10).ToString);
    ReturnsSimpleString saying3= new
    ReturnsSimpleString(EndStatement);
    Console.Write(saying1( ) + saying2( ) +
    saying3( ));
}
    // Method that returns a string.
static string AHeading( )
    {return "Your age will be ";}
    // Method that returns a string.
    static string EndStatement( )
    {return " in 10 years.";}
}
```

Relationship of Delegates to Events

- Delegate - someone who acts as a bridge between two things
- A delegate serves as a bridge with event-driven applications.Import the following using statement
- .creating a three-dimensional array

```
using System.Collections;
```

- To instantiate objects of the ArrayList class

Component-based development

- Emphasizes a reuse-based approach to defining, implementing, and composing independent components into systems
- Object-oriented development techniques work well for constructing multitier applications
- In C#, in addition to creating .EXE files, you can create class library files with a dynamic link library (DLL) extension

Unit Testing

- ❑ Create a unit test project
- ❑ C# MSTest Unit Test Project (.NET Core) for .NET Core template
- ❑ Solution Explorer, select Dependencies under the PROJECT NAME project
- ❑ And then choose Add Reference (or Add Project Reference) from the right-click menu.
- ❑ In the Reference Manager dialog box, expand Projects, select Solution, and then check the PROJECT item.

Unit Testing

- Create the test class
- Rename the UnitTest1 file to your testname

Inheritance

- ❑ Allows you to create a general class and then define specialized
- ❑ classes that have access to the members of the general class.
- ❑ classes can extend functionality by adding their own new unique data and behaviors.
- ❑ Inheritance is associated with an “is a” relationship
- ❑ Inheriting from the Object Class
- ❑ Inheriting from Other .NET FCL Classes
- ❑ Windows forms classes created inherited from the System.Windows.Forms.Form class.

Creating Base Classes for Inheritance

- super or parent class.

```
public class Person
{
    private string idNumber;
    private string lastName;
    private string firstName;
    private int age;
    // Constructor with three arguments
    public Person(string id, string lname, string fname)
    {
    idNumber = id;
    lastName = lname;
    firstName = fname;
    }
    // Constructor with one argument
    public Person(string id)
    {
    idNumber = id;
    }
}
```

Creating Base Classes for Inheritance

- ❑ ACCESS MODIFIERS
- ❑ **private** access modifier is restricted to members of the current class.
- ❑ private members are not accessible to other classes that derive from this class or that instantiate objects of this class
- ❑ Private access modifiers – class data protecting
- ❑ Only allow access to the data through its methods or properties

Creating Base Classes for Inheritance

- CONSTRUCTORS USE PUBLIC ACCESS
- **Defined as** same name as the class name
- defined with public access
- no objects can be instantiated from the class
- Can not have return type
- PROPERTIES OFFER PUBLIC ACCESS TO DATA FIELDS

```
// Read-only property. First name cannot be changed.
public string FirstName
{
   get
   {   return firstName;
   }
}
```

Creating Base Classes for Inheritance

- ❑ CONSTRUCTORS USE PUBLIC ACCESS

```
// Property for last name
public string LastName
{
   get
     {
        return lastName;
     }
   set
     {
        lastName = value;
     }
}
```

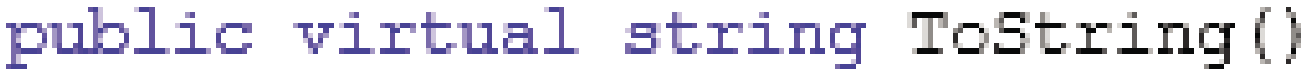
Overriding Methods

- ❑ When you override a method, you replace the method defined at a higher level with a new definition or behavior.
- ❑ Use keyword **override**

```
public override string ToString( ) // Defined in Person
{
   return firstName + " " + lastName;
}

public virtual int GetSleepAmt( )
{
   return 8;
}
```

Overriding Methods

- Override allows a method to provide a new implementation of a method inherited from a base class.
- The signature of the methods must match to override
- To override a base method, the base method must be defined as virtual, abstract, or override



```
public virtual string ToString()
```

PROTECTED ACCESS MODIFIERS

- ❑ Derived classes inherit all the characteristics of the base class, but they do not have direct access to change their private members.
- ❑ **Internal** members are accessible only within files in the same assembly.
- ❑ **Protected** members are accessible to any subclass that is derived from them but not to any other classes
- ❑ Use protected for derived class to have access to change data in the base class

PROTECTED ACCESS MODIFIERS

- Protected data is hidden from other classes but available for subclasses

```
public class Student : Person // Student is derived from Person
{
    private string major;
    private string studentId;
    // Default constructor
    public Student( )
    :base( ) // No arguments sent to base class constructor
{
}
```

PROTECTED ACCESS MODIFIERS

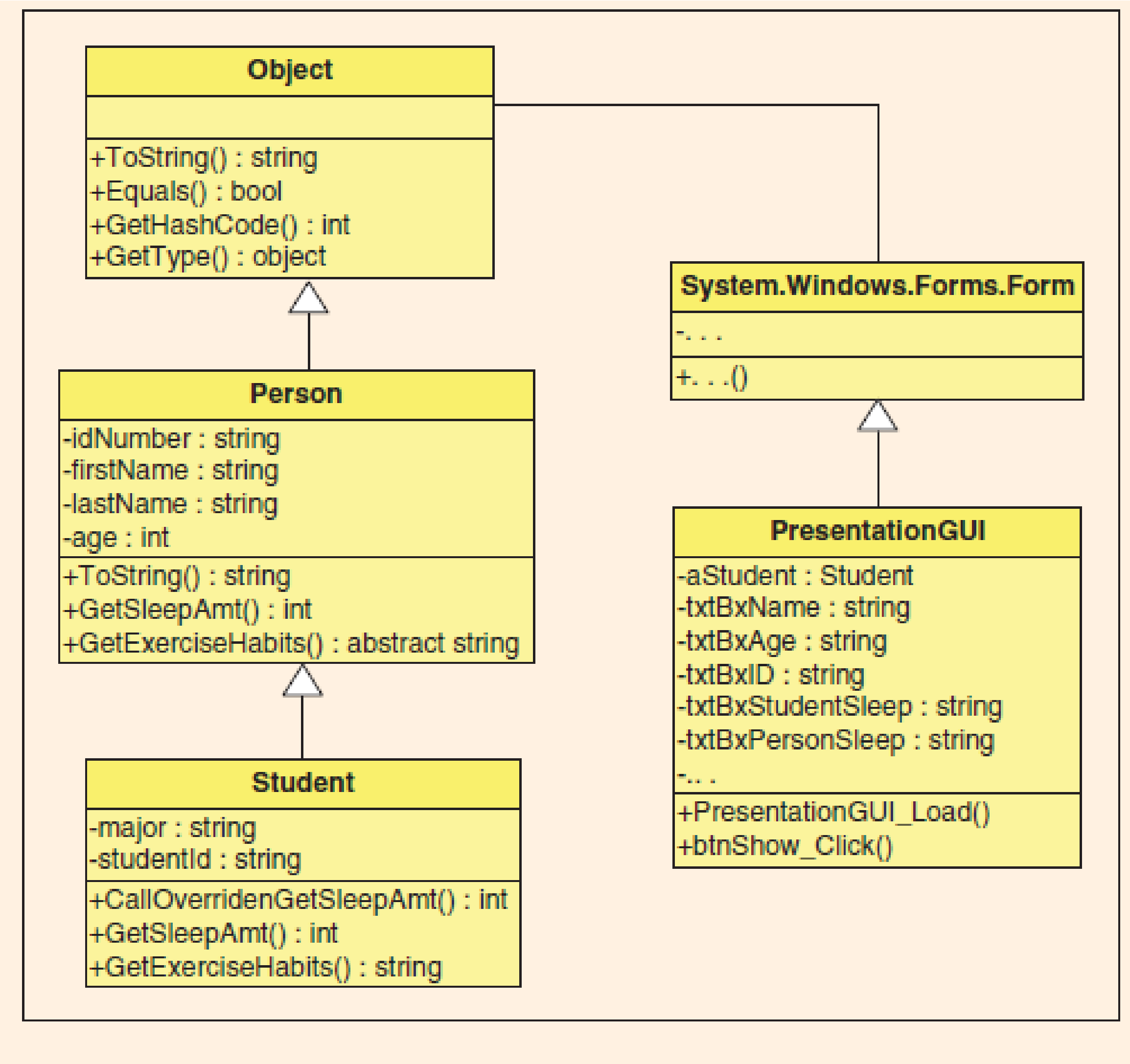
- ❑ Protected data is hidden from other classes but available for subclasses

```
    // Constructor sends three arguments to base class constructor
    public Student(string id, string fname, string lname, string
    maj, string sId)
    :base (id, lname, fname) // Base constructor arguments
    {
    major = maj;
    studentId = sId;
    }
    public override int GetSleepAmt( )
    {
    return 6;
    }
}
```

CALLING THE BASE CONSTRUCTOR

- An extra entry is added between the constructor heading for the Student subclass and the opening curly brace

```
public Student( )
:base( ) // No arguments sent to base class constructor
{ ...

public Student(string id, string fname, string lname,
string maj, string sId)
:base (id, lname, fname) // Base constructor arguments
{ ...

// Student object instantiated in a different class
// such as a PresentationGUI class.
Student aStudent = new
Student ("123456789", "Maria", "Woo", "CS", "1111");
```



© Cengage Learning

11-5 Inheritance class diagram

Making Stand-Alone Components

- classes can be compiled and then stored as a dynamic link library (DLL) file
- Any number of applications can then reference the classes. - beauty of component-based development and object-oriented programming.
- DYNAMIC LINK LIBRARY :
- Inheritance does not require the use of DLL components
- C# and Visual Studio – allows creating library components that can be compiled into a dynamic link library (.dll) file (PAGE 716 to 728)

Abstract Classes

- Add an **abstract** modifier to classes that prohibit other classes from instantiating objects of a base class
- can still inherit characteristics from this base class in subclasses
- Creating an abstract class

```
[access modifier] abstract class ClassIdentifier // Base class
```

- No objects can then be instantiated of the base class type

Abstract Methods

- One that does not include the implementation details for the method.
- The method has no body.
- The implementation details of the method are left up to the classes that are derived from the base abstract class.
- The syntax for creating an abstract method is as follows:

```
[access modifier] abstract returnType
MethodIdentifier([parameter list]); // No { } included
```

Abstract Methods

```
public abstract class Person
{
private string idNumber;
private string lastName;
private string firstName;
private int age;
public Person( )
{
}
public Person(string id, string lname, string fname, int anAge)
{
idNumber = id;
lastName = lname;
firstName = fname;
age = anAge;
}
public Person(string id, string lname, string fname)
{
idNumber = id;
lastName = lname;
firstName = fname;
}
```

Abstract Methods

- every class that derives from the Person class must provide the implementation details for the GetExerciseHabits( ) method.
- That is what adding the **abstract** keyword does.
- It is like signing a contract.
- If you derive from an abstract base class, you sign a contract that details how to implement its abstract methods.
- Abstract classes can include regular data field members, regular methods, and virtual methods in addition to abstract methods

Sealed Classes

- The purpose of an abstract class is to provide a common definition of a base class so that multiple derived classes can share that definition.
- Sealed classes provide a completely opposite type of restriction.
- They restrict the inheritance feature of object-oriented programming.

```
public sealed class SealedClassExample
{
// class members inserted here
}
```

Sealed Classes

- ❑ Sealed classes are defined to prevent derivation
- ❑ SealedClassExample shown before cannot be inherited.
- ❑ Objects can be instantiated from the class, but subclasses cannot be derived from it.
- ❑ There are a number of .NET classes defined with the sealed modifier

Interfaces

- Interfaces contain no implementation details for any of their members; all their members are considered abstract.
- Implementing the interface, the class agrees to define details for all of the interface’s methods
- A class can implement any number of interfaces.

```
[modifier] interface InterfaceIdentifier
{
// Members - no access modifiers are used.
}
```

Interfaces

- ❑ Interfaces contain no implementation details for any of their members; all their members are considered abstract.
- ❑ Implementing the interface, the class agrees to define details for all of the interface’s methods
- ❑ A class can implement any number of interfaces.

```
[modifier] interface InterfaceIdentifier
{
// Members - no access modifiers are used.
}
```

Implementing the Interface

- ❑ Student class derives from the base class Person and implements the ITraveler interface

```
public class Student : Person, ITraveler // Base class comes first
```

- ❑ If a class implements more than one interface, they all appear on the class definition line separated by commas.

```
public class Student : Person, ITraveler
{
    private string major;
    private string studentId;
    public Student( )
    : base( )
    {
}
```

Activity 3

❑ Yes

❑ No

Activity 3

❑ Yes

❑ No