LESSON 2

98-361 Software Development Fundamentals

2.1 Understand the Fundamentals of Classes

2.2 Understand Inheritance

2.3 Understanding Polymorphism

2.4 Understand Encapsulation

MTA Software Fundamentals 2 Test

Microsoft

LESSON 2.1

98-361 Software Development Fundamentals

# Understand the Fundamentals of Classes

# Lesson Overview

Students will understand the fundamentals of classes.

In this lesson, you will learn:

- Properties, methods, events, and constructors
- How to create a class
- How to use classes in code

# Review Terms

- Class—in object-oriented programming, a generalized category that describes a group of more specific items, called *objects*, that can exist within it.
- Constructor—a method that allows the programmer to set default values, limit instantiation, and write code that is flexible and easy to read. A default constructor has zero parameters.
- Event—an action or occurrence, often generated by the user, to which a program might respond (for example, key presses, button clicks, or mouse movements).
- Method—in object-oriented programming, a process performed by an object when it receives a message.

# Review Terms

- Object—1. short for object code (machine-readable code). 2. In object-oriented programming, a variable comprising both routines and data that is treated as a discrete entity.
- Object-oriented programming—a programming paradigm in which a program is viewed as a collection of discrete objects that are self-contained collections of data structures and routines that interact with other objects.
- Property—members of a class that provide a flexible mechanism to read, write, or compute the values of private fields. Properties are viewed or accessed by “accessor” methods within the class and are changed or modified by “modifier” methods within the class.

# Objects and Classes—The Relationship

- Objects are instances of a given data type. The data type provides a blueprint for the object that is created—or instantiated—when the application is executed.
- New data types are defined using classes.
- Classes form the building blocks of applications, containing code and data. An application will always contain at least one class.

# Objects and Classes—A Comparison

- An object belongs to a class and is an instance of that class. The class is the blueprint of the object and defines the fields and methods.
- The object exists during the time that a program executes and must be explicitly declared and construed by the executing program. For most programming languages, a class cannot be changed during program execution.
- Classes have fields that can change in value and methods that can execute during program execution. The class to which the object belongs defines these attributes and methods.

- Properties can be used as though they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily while still providing the safety and flexibility of methods.
- Constructors are class methods that are executed when an object of a given type is created. Constructors have the same name as the class and usually initialize the data members of the new object.

# Inside a Method

Access modifier | Return type | Method name | Parameter

```
public Sub Magnify(factor as Integer)
    length = length * factor;
    width = width * factor;
End Sub
```

A *method* is a code block containing a series of statements. Every executed instruction is performed in the context of a method.

LESSON 2.1

98-361 Software Development Fundamentals

# Student Lab 2.1

LESSON 2.2

98-361 Software Development Fundamentals

# Understand Inheritance

# Lesson Overview

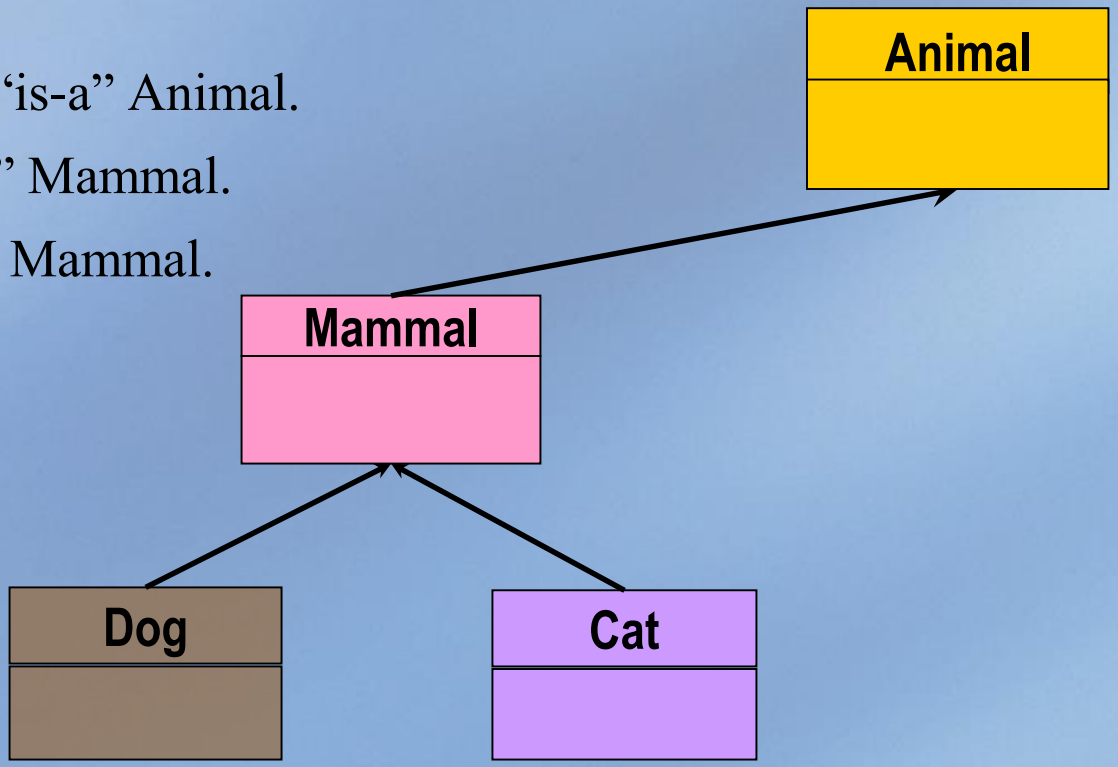
Students will understand the concepts associated with inheritance in object-oriented programming.

In this lesson, students will learn:

- The “is-a” relationship
- How to create class hierarchies through inheritance

# The “is-a” Relationship

- A Mammal “is-a” Animal.
- A Dog “is-a” Mammal.
- A Cat “is-a” Mammal.



# Review Terms

- **Abstract class** - a class from which no objects can be created.
  - Used to defined subclasses.
  - Objects are created from the subclasses.
- **Base class** - a class from which other classes have been or can be derived by inheritance.
- **Derived class** - a class created from another class, referred to as the base class.
  - A derived class inherits all the features of its base class.
  - It can include additional data elements and routines, redefine routines from the base class, and restrict access to base class features.

# Review Terms (continued)

- **Inheritance** - the transfer of the characteristics of a class to other classes derived from it.
  - Example: If "vegetable" is a class, the classes "legume" and "root" can be derived from it, and each will inherit the properties of the "vegetable" class.
    - Inherited properties might include name, growing season, and water requirements.
- **Interface** - contains only the signatures of methods, delegates, or events.
  - The implementation of the methods is created in the class that implements the interface.

# Inheritance

- Classes can inherit from another class. The syntax requires a colon after the class name in the declaration, and identifying the class to inherit from—the base class—after the colon, as follows:

```
public class A

    public A() { }
    public void doA(){ }

public class B : A

    public B() { }
    public void doB(){ }
```

**Base class**

# Inheritance (continued)

- The new class—the derived class—then includes all the non-private data and behavior of the base class, in addition to any other data or behaviors that it defines for itself. The new class then has two effective types: the type of the new class and the type of the class that it inherits.

```
public class A

    public A()
    public void doA()


public class B : A

    public B()
    public void doB()
```

**Derived class**

# Inheritance (continued)

```
public class Tester

    A = new A()
    B = new B()
    b.doB()    'legal
    b.doA()    'legal
    a.doA()    'legal
    a.doB()    'illegal
```

**B is derived from A, so it can doA(), but A is not derived from B, so it cannot doB(). The inheritance relationship is not reciprocal.**

# Inheritance (continued)

- In the previous examples, class B is effectively both type B and type A. When you access a B object, you can use the cast operation to convert it to an A object. The B object is not changed by the cast, but your view of the B object becomes restricted to A's data and behaviors.
- Multiple inheritance is not supported. A class can inherit from only one other class.

```
B = new B()
A = (A)b;   'B object being cast as an A object
            'this is ok because B "is-a" A
```

# Interfaces

- Are defined using the `interface` keyword.
- Describe a group of related behaviors that can belong to any class.
- Can be made up of methods, properties, events, indexers, or any combination of those four member types.
- Cannot contain fields.
- Members are automatically public.
- A class can inherit more than one interface.
  - When a class inherits an interface, it inherits only the method names and signatures because the interface itself contains no implementations.

```
interface IComparable

    int CompareTo(object obj);
```

**Interfaces and interface members are abstract; they do not provide a default implementation.**

```
public class Minivan : Car, IComparable

    public int CompareTo(object obj)

        'implementation of CompareTo
```

**Minivan inherits from Car and implements the IComparable interface.**

# Abstract Classes

- The `abstract` keyword enables you to create classes and class members solely for the purpose of inheritance—to define features of derived classes.
- An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.
- Abstract classes may also define abstract methods. This is accomplished by adding the keyword `abstract` before the return type of the method.

Classes can be declared as abstract by putting the keyword `abstract` before the keyword `class` in the class definition.

```
public abstract class A
```

```
public abstract void DoWork(int i);
```

Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods.

# Review of Interfaces

- An interface is similar to an abstract base class.
  - Any non-abstract type inheriting the interface must implement all its members.
- An interface cannot be instantiated directly.
- Interfaces can contain events, indexers, methods, and properties.
- Interfaces cannot contain implementation of methods.
- Classes can inherit from more than one interface.
- An interface can itself inherit from multiple interfaces.

## Abstract Classes

- Can have implementation code
- Use to provide related derived classes with common method signatures, and also common methods and instance variables
- Can contain instance variables
- Can declare constants

## Interfaces

- Are fully abstract
- Are used to represent a set of abstract behaviors that can be implemented by unrelated classes
- Cannot contain instance variables
- Can declare constants

LESSON 2.2

98-361 Software Development Fundamentals

# Student Lab 2.2

Microsoft

LESSON 2.3

98-361 Software Development Fundamentals

# Understanding Polymorphism

(with Visual Basic)

LESSON 2.3

98-361 Software Development Fundamentals

# Lesson Overview

Students will understand polymorphism.

In this lesson, you will learn about:

- Extending the functionality in a class after inheriting from a base class
- Overriding methods in a derived class

# Review Terms

- `MyBase`—a keyword is used to access members of the base class from within a derived class.
- `New`—when used as a modifier, this keyword explicitly hides a member inherited from a base class. When you hide an inherited member, the derived version of the member replaces the base-class version.
- `Overrides`—a modifier required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

# Review Terms (continued)

- polymorphism—the ability to redefine a method in a derived class (a class that inherited its data structures and methods from another class).
  - A class can be used as more than one type; it can be used as its own type, any base types, or any interface type if it implements interfaces.
- `NotOverridable`—cannot be inherited. A sealed method overrides a method in a base class, but itself cannot be overridden further in any derived class.
- `Overridable`—a keyword used to modify a method or property declaration, in which case the method or the property is called a *virtual member*.
  - — The virtual member allows its implementation to be replaced within derived classes.

# Polymorphism

- Refers to the ability to redefine methods in a derived class and use a class as more than one type; it can be used as its own type, any base types, or any interface type if it implements interfaces.
- Is important not only to the derived classes, but to the base classes as well. Using the base class could be the same as using an object of the derived class that has been cast to the base class type.
- When a derived class inherits from a base class, it gains all the methods, fields, properties, and events of the base class.
- To change the data and behavior of a base class, you have two choices: You can replace the base member with a new derived member, or you can override a virtual base member.

# Using the `Shadows` Keyword

- Replacing a member of a base class with a new derived member requires the `Shadows` keyword.
- If a base class defines a method, field, or property, the `Shadows` keyword is used to create a new definition of that method, field, or property in a derived class.

# Using the `Shadows` Keyword (continued)

The `Shadows` keyword is placed before the return type of a class member that is being replaced:

```
 Public Class BaseClass
    Public Function WorkField() As Integer
    End Function
    Public Sub DoWork()        End Sub
End Class

Public Class DerivedClass : Inherits BaseClass
    Public Shadows Function WorkField() As Integer
    End Function
    Public Shadows Sub DoWork()        End Sub
End Class
```

# Using the `Shadows` Keyword (continued)

- When the `Shadows` keyword is used, the new class members are called instead of the base class members that have been replaced.
- Those base class members are called *hidden members*. Hidden class members can still be called if an instance of the derived class is cast to an instance of the base class.

# Using `Overridable` and `Overrides` Keywords

- For an instance of a derived class to completely take over a class member from a base class, the base class has to declare that member as `Overridable in VB.`
- A derived class then uses the `Overrides` keyword, instead of `new`, to replace the base class implementation with its own.

```
Public Class BaseClass
  Public Function WorkField() As Integer  End Function
  Public Overridable Sub DoWork()  End Sub
End Class

Public Class DerivedBlass : Inherits BaseClass
  Public Function WorkField() As Integer  End Function
  Public Overrides Sub DoWork()  End Sub
End Class
```

# Using `Overridable` and `Overrides` Keywords (continued)

- Fields cannot be virtual; only methods, properties, events, and indexers can be virtual.
- When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class.

```
DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method


BaseClass A = (BaseClass) B;
A.DoWork(); // Also calls the new method
```

# Virtual members remain virtual

- If class A declares a virtual member, class B derives from A, and class C derives from B, class C inherits the virtual member and has the option to override it, regardless of whether class B declared an override for that member .

```
Public Class A
  Public Overridable Sub DoWork()  End Sub
End Class
Public Class B : Inherits A
  Public Overrides Sub DoWork()  End Sub
End Class
Public Class C : Inherits B
  Public Overrides Sub DoWork()  End Sub
End Class
```

# Using the `NotOverridable` Keyword

- A derived class can stop virtual inheritance by declaring an override as NotOverridable
- In the code samples below, the method `DoWork` is no longer virtual to any class derived from C. It is still virtual for instances of C, even if they are cast to type B or type A.

```
Public Class A
  Public Overridable Sub DoWork()  End Sub
End Class
Public Class B : Inherits A
  Public Overrides Sub DoWork()  End Sub
End Class
Public Class C : Inherits B
  Public NotOverridable Overrides Sub DoWork()  End Sub
End Class
```

# Using the `NotOverridable` Keyword (continued)

- Sealed methods can be replaced by derived classes using the `Shadows` keyword.
- If `DoWork` is called on D using a variable of type D, the new `DoWork` is called. If a variable of type C, B, or A is used to access an instance of D, a call to `DoWork` will follow the rules of virtual inheritance, routing those calls to the implementation of `DoWork` on class C.

```
Public Class C : Inherits B
  Public NotOverridable Overrides Sub DoWork()   End Sub
End Class
Public Class D : Inherits C
  Public Shadows Sub DoWork()  End Sub
End Class
```

# Using the `MyBase` Keyword

- A derived class that has replaced or overridden a method can still access the method on the base class using the `MyBase` keyword.

```
Public Class A
  Public Overridable Sub DoWork() End Sub
End Class
Public Class B : Inherits A
  Public Overrides Sub DoWork()   End Sub
End Class
Public Class C : Inherits B
  Public Overrides Sub DoWork()
    MyBase.DoWork()    '  B's DoWork()
  '  Behaviors specific to C's DoWork()
  End Sub
End Class
```

# No Student Lab 2.3

LESSON 2.4

98-361 Software Development Fundamentals

# Understand Encapsulation

# Lesson Overview

Students will understand encapsulation in object-oriented programming.

In this lesson, you will learn about:

- Creating classes that hide their implementation details while still allowing access to the required functionality through the interface
- Access modifiers

# Review Terms

- **Encapsulation** - In object-oriented programming, encapsulation packages attributes (properties) and functionality (methods or behaviors) to create an object that is essentially a “black box”—one whose internal structure remains private and whose services can be accessed by other objects only through messages passed by a clearly defined interface. Also known as *information hiding*.
- **`Internal`** - An access modifier for classes and class members. Internal members are accessible only within files in the same assembly.

# Review Terms (continued)

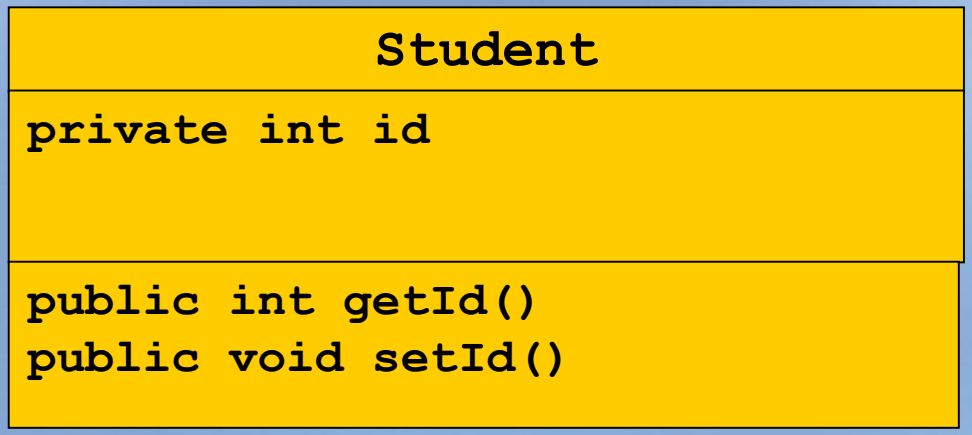
- **`Private`** - a member access modifier. Private access is the least permissive access level. Private members are accessible only within the body of the class in which they are declared.
- **`Protected`** - a member access modifier. A protected member is accessible from within the class in which it is declared, and from within any class derived from the class that declared this member.
- **`Public`** - an access modifier for classes and class members. Public access is the most permissive access level. There are no restrictions on accessing public members.

# What is encapsulation?

- Encapsulation allows the programmer to hide (encapsulate) some of the data and functionality of a class while revealing others.
- Data fields are made private to restrict access to within the class, but they are partnered with accessible (public) methods that provide a gateway to the data.
- A method within a class can be called by other classes, but the actual implementation of the method is hidden within the class.
- This allows you to change the code of the method from within the class without affecting the code of the classes outside the class.
- It is now easier to edit or enhance a program because you can localize your changes to particular classes.

# What is encapsulation? (continued)

- The fields and methods pertaining to a student record are encapsulated within the `Student` class.
- The private data fields are accessed by the public methods.



# What is encapsulation?

- To change the `id` field, another class has to call the public `setID` method, which protects the `id` field from invalid numbers.

```
public class Student

    private int id
    public int getID()
        return id

    public void setID(int number)
        'only allow positive numbers
        if (id > 0)
            id = number


End Class
```

# Encapsulation with Properties

- Properties enable a class to expose a public way of getting and setting values while hiding implementation or verification.
- The `value` keyword is used to define the value being assigned by the set indexer.
- Properties that do not implement a set method are read-only.
- Properties that do not implement a get method are write-only.

```
public class Student

    public int Id
       get

          return id

       set

          if (id > 0)
             id = value

End class
```

# Encapsulation deals with access

- Class members can be declared with any of the five types of access: `public`, `private`, `protected`, `internal`, `protected internal`
- The accessibility of a member can never be greater than the accessibility of its containing type. For example, a public method declared in an `internal` type has only internal accessibility.

```
public class Tricycle

    protected void Pedal()
    private int wheels = 3
    protected internal int Wheels

        get

          return wheels

End class
```

# Using the `public` keyword

- The `public` keyword is an access modifier for types and type members. Public access is the most permissive access level.

```
public class Sample

    public int x


public class MainClass

    public static void Main()

        Sample s = new Sample()
        // direct access to public members
        s.x = 15

End class
```

# Using the `private` keyword

- When the `private` keyword is used, members are accessible only within the body of the class in which they are declared.

```
public class Sample

    private int x


public class MainClass

    public static void Main()

        Sample s = new Sample()
        'private denies access
        s.x = 15 'won't compile
```

# Using the `protected` keyword

- A protected member is accessible within its class and by derived class instances.
- A protected member of a base class is accessible in a derived class only if the access occurs through the derived class type.

```
public class A

   protected int x = 123

public class B : A
public class Driver

   public static void Main()
      A = new A()
      B = new B()
      A.x = 10  'error
      B.x = 10  'OK
```

# Using the `internal` keyword

- Internal types or members are accessible only within files in the same assembly.
- A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code.
- For example, a framework for building graphical user interfaces (GUIs) could provide Control and Form classes that cooperate by using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

# Using the `internal` keyword (continued)

- This example contains two files in different assemblies, `Assembly1.cs` and `Assembly2.cs`.
- The first file contains an internal base class, `BaseClass`. In the second file, an attempt to instantiate `BaseClass` will produce an error.

```
'Assembly1.cs, compile with: /target:library
internal class BaseClass

   public static int x = 0

'Assembly2.cs, compile with: /reference:Assembly1.dll
public class TestAccess

   public static void Main()

      BaseClass myBase = new BaseClass()  'error
```

# No Student Lab 2.4

LESSON 2

98-361 Software Development Fundamentals

# Complete the QUIZ Test

## MTA Software Fundamentals 2 Test