Visual Programming
Classes and Objects

Indah Permatasari, M.Kom.

(7th meeting)

Outline

- Classes and Objects
- Object-Oriented programming (C#) techniques
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

Classes and Objects

Introduction: Classes and Objects

- Classes are the most fundamental of C#'s types.
- A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit.
- A class provides a definition for dynamically created *instances of the class*, also known as **objects**.
- Classes support **inheritance and polymorphism**, mechanisms whereby derived classes can extend and specialize base classes.

Class declarations

- A class declaration starts with a header that specifies the attributes and modifiers of the class, the name of the class, the base class (if given), and the interfaces implemented by the class.
- The header is followed by the class body, which consists of a list of member declarations written between the delimiters { and } .

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Cont.

- Classes are declared by using the class keyword followed by a unique identifier
- Example:

```
//[access modifier] - [class] - [identifier]
public class Customer
```

- The class keyword is preceded by the access level

Members of Class

- The members of a class are either static members or instance members.
- Static members belong to classes, and instance members belong to objects (instances of classes).

Member	Description
Constants	Constant values associated with the class
Fields	Variables of the class
Methods	Computations and actions that can be performed by the class
Properties	Actions associated with reading and writing named properties of the class
Indexers	Actions associated with indexing instances of the class like an array
Events	Notifications that can be generated by the class
Operators	Conversions and expression operators supported by the class
Constructors	Actions required to initialize instances of the class or the class itself
Destructors	Actions to perform before instances of the class are permanently discarded
Types	Nested types declared by the class

Accessibility of class

Accessibility	Meaning
public	Access not limited
protected	Access limited to this class or classes derived from this class
internal	Access limited to this program
protected internal	Access limited to this program or classes derived from this class
private	Access limited to this class

Type parameters in class

- The type parameters can then be used in the body of the class declarations to define the members of the class.
- A class type that is declared to take type parameters is called a generic class type.

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

Creating objects

- Although they are sometimes used interchangeably, a class and an object are different things.
- **A class** defines a type of object, but it is not an object itself.
- **An object** is a concrete entity based on a class, and is sometimes referred to as an instance of a class.
- Objects can be created by using the `new` keyword followed by the name of the class that the object will be based on, like this:

```
Customer object1 = new Customer();
```

  - In the previous example, `object1` is a reference to an object that is based on `Customer`

Object-Oriented programming (C#)

- C# is an object-oriented language.
- Four of the key techniques used in object-oriented programming are:
  - *Abstraction* means that a group of related properties, methods, and other members are treated as a single unit or object.
  - *Encapsulation* means hiding the unnecessary details from type consumers.
  - *Inheritance* describes the ability to create new classes based on an existing class.
  - *Polymorphism* means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.

Abstract Classes

- **Data abstraction** is the process of hiding certain details and showing only essential information to the user.
- **Abstract class** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class)

```
abstract class Animal
{
  public abstract void animalSound();
  public void sleep()
  {
    Console.WriteLine("Zzz");
  }
}
```

Abstract method can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

Encapsulation classes

- Encapsulation is defined as the wrapping up of data under a single unit.
- It is the mechanism that binds together code and the data it manipulates.
- In a different way, encapsulation is a protective shield that prevents the data from being accessed by the code outside this shield.
  - Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of own class in which they are declared.
  - As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.
  - Encapsulation can be achieved by: Declaring all the variables in the class as private and using C# Properties in the class to set and get the values of variables.

Example

```
// C# program to illustrate encapsulation
using System;

public class DemoEncap {
   // private variables declared these can only be accessed
   // by public methods of class
    private String studentName;
    private int studentAge;

     // using accessors to get and set the value of studentName
    public String Name {
      get {
            return studentName;
          }
      set {
            studentName = value;
          }
    }
     // using accessors to get and set the value of studentAge
    public int Age {
      get {
            return studentAge;
          }
      set {
            studentAge = value;
          }
    }
}
```

```
// Driver Class
class GFG {
    // Main Method
    static public void Main() {
        // creating object
        DemoEncap obj = new DemoEncap();

        // calls set accessor of the property Name,
        // and pass "Ankita" as value of the
        // standard field 'value'
        obj.Name = "Ankita";

        // calls set accessor of the property Age,
        // and pass "21" as value of the
        // standard field 'value'
        obj.Age = 21;

        // Displaying values of the variables
        Console.WriteLine("Name: " + obj.Name);
        Console.WriteLine("Age: " + obj.Age);
     }
}
```

Explanation

- In the above program the class `DemoEncap` is **encapsulated as the variables are declared as private**. To access these private variables we are using the `Name` and `Age` accessors which contains the `get` and `set` method to retrieve and set the values of private fields. Accessors are defined as public so that they can access in other class.
- Advantages of Encapsulation:
  - Data Hiding: The user will have no idea about the inner implementation of the class.
  - Increased Flexibility: We can make the variables of the class as read-only or write-only depending on our requirement.
  - Reusability: Encapsulation also improves the re-usability and easy to change with new requirements.
  - Testing code is easy: Encapsulated code is easy to test for unit testing.

Inheritance Classes

- Classes fully support inheritance, a fundamental characteristic of object-oriented programming.
- Inheritance is accomplished by using a derivation, which means a class is declared by using a base class from which it inherits data and behavior.
- Inheritance enables you to create new classes that reuse, extend, and modify the behaveior defined in other classes.
- For example, if you have a base class `Animal`, you might have one derived class that is named `Mammal` and another derived class that is named `Reptile`. A `Mammal` is an `Animal`, and a `Reptile` is an `Animal`, but each derived class represents different specializations of the base class.
- Syntax:
  - `class derived-class : base-class {…}`

Important terminology in Inheritance

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

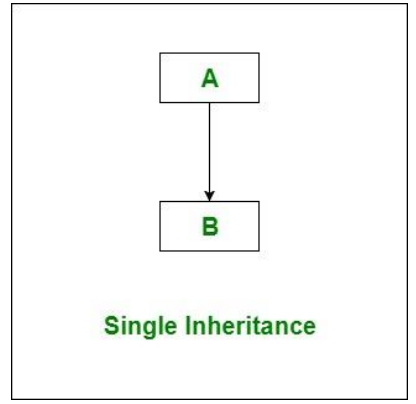
Example

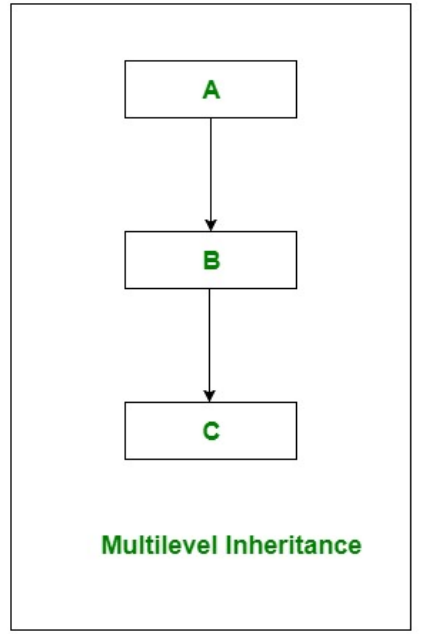
```
// C# program to illustrate the
// concept of inheritance
using System;
namespace ConsoleApplication1 {

// Base class
class GFG {
// data members
    public string name;
    public string subject;

    // public method of base class
    public void readers(string name, string subject){
        this.name = name;
        this.subject = subject;
        Console.WriteLine("Myself: " + name);
        Console.WriteLine("My Favorite Subject is: " +
subject);
    }
}
```

```
// inheriting the GFG class using :
class GeeksforGeeks : GFG {
// constructor of derived class
    public GeeksforGeeks() {
        Console.WriteLine("GeeksforGeeks");
    }
}
// Driver class
class Sudo {

    // Main Method
    static void Main(string[] args)
    {

        // creating object of derived class
        GeeksforGeeks g = new GeeksforGeeks();

        // calling the method of base class
        // using the derived class object
        g.readers("Kirti", "C#");
    }
 }
}
```

Types of Inheritance in C#

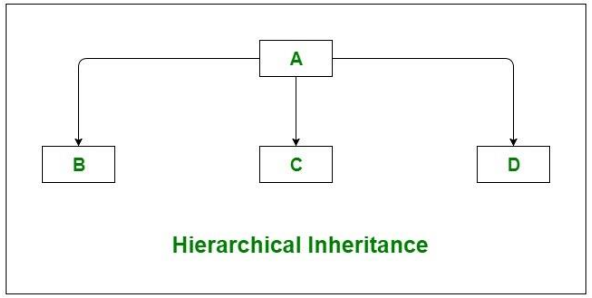
- **Single Inheritance:** subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.
- **Multilevel Inheritance:** a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

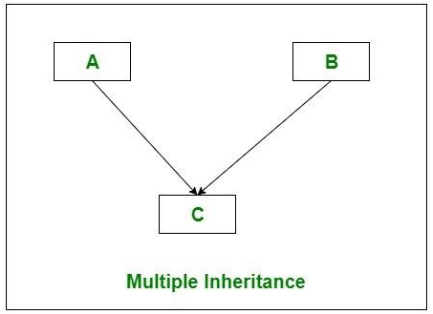


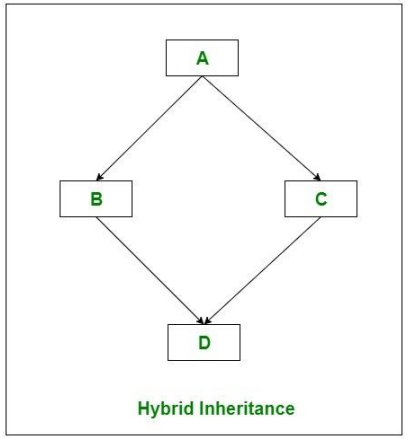


Cont.

- **Hierarchical Inheritance:** one class serves as a superclass (base class) for more than one subclass. In below image, class A serves as a base class for the derived class B, C, and D.
- **Multiple Inheritance(Through Interfaces):** one class can have more than one superclass and inherit features from all parent classes.
- **Hybrid Inheritance(Through Interfaces):** It is a mix of two or more of the above types of inheritance.







Polymorphism classes

- The word polymorphism is made of two words poly and morph, where poly means **many** and morphs means **forms**.
- In programming, polymorphism is a feature that allows one interface to be used for a general class of actions.
- it occurs when we have **many classes** that are related to each other by inheritance.
- *Inheritance* lets us inherit fields and methods from another class. *Polymorphism* uses those methods to perform different tasks. This allows us to perform a single action in different ways.

Example

- Think of a base class called `Animal` that has a method called `animalSound()`. Derived classes of `Animals` could be *Pigs, Cats, Dogs, Birds* - And they also have their own implementation of an animal sound (*the pig oinks, and the cat meows, etc.*)

```
class Animal  // Base class (parent)
{
  public void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Pig : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}
```

Important Notes

- **C# provides** an option to override the base class method, by adding the `virtual` keyword to the method inside the base class, and by using the `override` keyword for each derived class methods

```
class Animal  // Base class (parent)
{
  public virtual void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Pig : Animal  // Derived class (child)
{
  public override void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public override void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}
```

Why And When To Use "Inheritance" and "Polymorphism"?

- It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

Summary

- You used *Abstraction* when you defined classes for each of the different types. Those classes described the behavior for that type.
- You used *Encapsulation* when you kept many details `private` in each class.
- You used *Inheritance* when you leveraged the implementation already created in the class to save code.
- You used *Polymorphism* when you created `virtual` methods that derived classes could **override** to create specific behavior for that type.

Terima kasih.