

# **C# PROGRAMMING**

## **TEXT BOOK:**

**E. Balagurusamy, “Programming in C# :A Primer”, Fourth Edition, McGraw Hill Education Private Limited, 2016.**

**Prepared by  
B.Loganathan**



# C# PROGRAMMING

- ▶ **UNIT-III: Classes and objects:** Basic principles of OOP – Defining a class – Adding variables – Adding Methods – Member access modifiers – Creating objects – Accessing class members – Constructors – Overloaded constructors – Static members – Static constructors – Private constructors – copy constructors – Destructors - Advanced Features of C#: **Inheritance:** Classical Inheritance – Containment Inheritance – Defining a subclass – Visibility control – Defining sub-class constructors – Multilevel inheritance – Hierarchical inheritance – **Overriding methods:** Hiding methods - Abstract classes- Abstract methods – Sealed classes - Preventing inheritance – Sealed Methods – Polymorphism

# Classes and Objects

- ▶ C# is a true object-oriented language and therefore the underlying structure of all C# programs is classes. Anything we wish to represent in a C# program must be encapsulated in a class that defines the *state* and *behaviour* of the basic program components known as *objects*. Classes create objects and objects use methods to communicate between them.
- ▶ Classes provide a convenient approach for packing together a group of logically related data items and functions that work on them.

# BASIC PRINCIPLES OF OOP

- ▶ All object-oriented languages employ three core principles, namely,
  - encapsulation,
  - inheritance,
  - polymorphism.
- ▶ These are often referred to as three 'pillars' of OOP.
- ▶ *Encapsulation* provides the ability to hide the internal details of an object from its users. The outside user may not be able to change the state of an object directly. However, the state of an object may be altered indirectly using what are known *accessor* and *mutator* methods.

- ▶ *Inheritance* is the concept we use to build new classes using the existing class definitions. Through inheritance we can modify a class the way we want to create new objects. The original class is known as *base* or *parent* class and the modified one is known as *derived class* or *subclass* or *child class*.
- ▶ *Polymorphism* is the third concept of OOP. It is the ability to take more than one form. For example, an operation may exhibit different behaviour in different situations. The behaviour depends upon the types of data used in the operation. For example, an addition operation involving two numeric values will produce a sum and the same addition operation will produce a string if the operands are string values instead of

# DEFINING A CLASS

- ▶ A class is a user-defined data type with a template that serves to define its properties. Once the class type has been defined, we can create 'variables' of that type using declarations that are similar to the basic type declarations. In C#, these variables are represented as *instances* of classes, which are the actual *objects*. The basic form of a class definition is:
  - ▶ `class classname`
  - ▶ `{`
  - ▶ `[ variables declaration; ]`
  - ▶ `[ methods declaration; ]`
  - ▶ `}`

- ▶ **class** is a keyword and *classname* is any valid C# identifier. Everything inside the square brackets is optional. This means that the following would be a valid class definition:
- ▶ `class Empty // class name is Empty`
- ▶ `{`
- ▶ `}`
- ▶ Because the body is empty, this class does not contain any properties and therefore cannot do anything.
- ▶ We can, however, compile it and even create objects using it. C++ *programmers may note that there is no semicolon after the closing brace.*

## ▶ **ADDING VARIABLES**

- ▶ Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called *instance variables* because they are created whenever an object of the class is instantiated.

▶ *Example:*

▶ class Rectangle

▶ {

▶ int length;     / / instance variable

▶ int width;     / / instance variable

▶ }

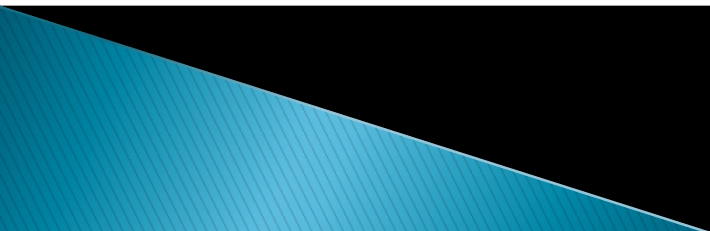
- ▶ The class **Rectangle** contains two integer type instance variables.




## ▶ ADDING METHODS

- ▶ A class with only data fields and without methods that operate on that data has no life.
- ▶ We must therefore add methods that are necessary for manipulating the data contained in the class. Methods are declared inside the body of the class. The general form of a method declaration is:
  - ▶ **type** **methodname** (*parameter-list*)
  - ▶ {
  - ▶ **method-body**;
  - ▶ }
- ▶ The *body* actually describes the operations to be performed on the data.

- ▶ Let us consider the **Rectangle** class again and add a method **GetData ( )** to it.
- ▶ class Rectangle
- ▶ {
- ▶ int length; int width;
- ▶ public void GetData (i nt x , in t y)
- ▶ {
- ▶ length = x ;
- ▶ width = y;
- ▶ }
- ▶ }
- ▶ Note that the method has a return type **void** because it does not return any value.

- ▶ We pass two integer values to the method which are then assigned to the instance variables **length** and **width**.
  - ▶ Notice that we are able to use directly **length** and **width** inside the method. We have used the keyword **public** in defining the method.
  - ▶ **MEMBER ACCESS MODIFIERS**
  - ▶ One of the goals of object-oriented programming is 'data hiding'. That is, a class may be designed to hide its members from outside accessibility. C# provides a set of 'access modifiers' that can be used with the members of a class to control their visibility to outside users. The following table lists various access modifiers provided by C# and their visibility control. These modifiers are a part of C# keywords
- 

- ▶ Private :Member is accessible only with in the class containing the member.
  - ▶ Public : Member is accessible from anywhere outside the class as well. It is also accessible in derived classes.
  - ▶ protected :Member is visible only to its own class and its derived classes.
  - ▶ Internal :Member is available within the assembly or component that is being created but not to the clients of that component.
  - ▶ Protected internal :Available in the containing program or assembly and in the derived classes.
- 

# CREATING OBJECTS

- ▶ Objects in C# are created using the new operator. The new operator creates an object of the specified class and returns a reference to that object.

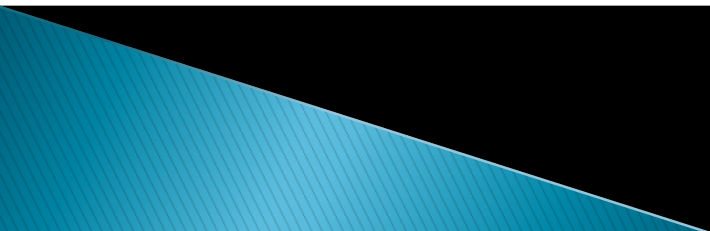
Here is an example of creating an object of type **Rectangle**.

- ▶ `Rectangle rect1 ; // declare`
- ▶ `rect1 = new Rectangle( ); // instantiate`
- ▶ The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable. The variable **rect1** is now an object of the **Rectangle** class .

## ACCESSING CLASS MEMBERS

- ▶ Now that we have created objects, each containing its own set of variables, we should assign values to these variables in order to use them in our program. Remember, all variables must be assigned values before they are used. Since we are outside the class, we cannot access the instance variables and the methods directly.
- ▶ we must use the concerned object and the *dot* operator as shown below:
- ▶ *objectname.variable name;*
- ▶ *objectname.methodname(parameter -list );*

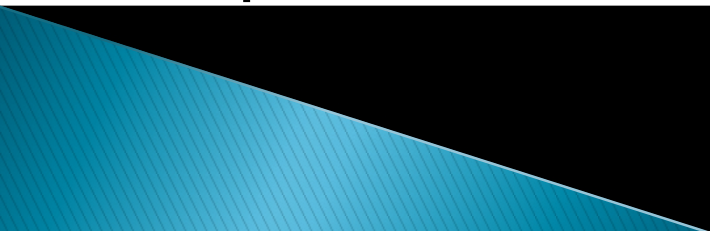
# CONSTRUCTORS

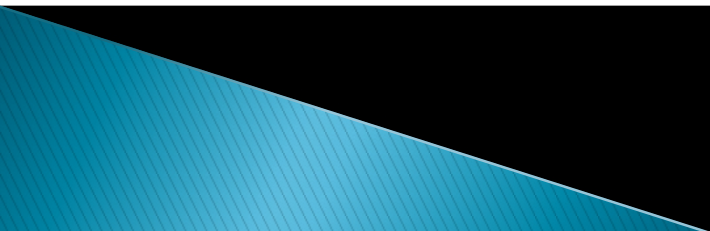
- ▶ We know that all objects that are created must be given initial values. The first approach uses the dot operator to access the instance variables and then assigns values to them individually. It can be a tedious approach to initialize all the variables of all the objects.
  - ▶ C# supports a special type of method, called a *constructor*, that enables an object to initialize itself when it is created.
  - ▶ Constructors have the same name as the class itself. Secondly, they do not specify a return type, not even **void**. This is because they do not return any value.
- 

# OVERLOADED CONSTRUCTORS

- ▶ Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, C# matches up the method name first and then the number and type of parameters to decide which one of the definitions *to* execute. This process is known as *polymorphism*. We can extend the concept of method overloading to provide more than one constructor to a class.



- ▶ To create an overloaded constructor method, all we have to do is to provide several different constructor definitions with different parameter lists. The difference may be in either the number or type of arguments. That is, each parameter list should be unique.
  - ▶ **STATIC MEMBERS**
  - ▶ One declares variables and the other declares methods. These variables and methods are called *instance variables* and *instance methods*. This is because every time the class is instantiated, a new copy of each is created. They are accessed using the objects (with dot operator).
- 

- ▶ Let us assume that we want to define a member that is common to all the objects and accessed without using a particular object. That is, the member belongs to the class as a whole rather than the objects created from the class. Such members can be defined as follows:
    - ▶ **static** int count ;
    - ▶ **static** int max(int x, int y);
    - ▶ The members that are declared **static** as shown in the above are called *static members*.
- 

# STATIC CONSTRUCTORS

- ▶ Like any other static members, we can also have static constructors. A static constructor is called before any object of the class is created. This is useful to do any housekeeping work that needs to be done once. It is usually used to assign initial values to static data members.
- ▶ A static constructor is declared by prefixing a **static** keyword to the constructor definition. It cannot have any parameters. *Example:*
- ▶ Class Abc
- ▶ {
- ▶ **static** Abc( )            / /No parameters
- ▶ {
- ▶ ..... // set values for static members here
- ▶ }
- ▶ .....
- ▶ }

# PRIVATE CONSTRUCTORS

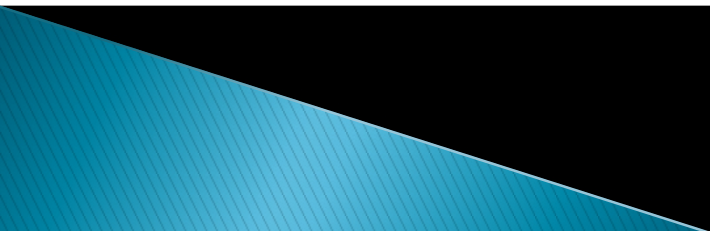
- ▶ C# does not have global variables or constants. All declarations must be contained in a class. In many situations, we may wish to define some utility classes that contain only static members. Such classes are never required to instantiate objects. Creating objects using such classes may be prevented by adding a **private** constructor to the class.
- ▶ **COPY CONSTRUCTORS**
- ▶ A copy constructor creates an object by copying variables from another object. For example, we may wish to pass an **Item** object to the **Item** constructor so that the new **Item** object has the same values as the old one.

- ▶ Since C# does not provide a copy constructor, we must provide it ourselves if we wish to add this feature to the class. A copy constructor is defined as follows:
  - ▶ `public Item (Item item)`
  - ▶ `{`
  - ▶ `code = item.code;`
  - ▶ `price = item.price;`
  - ▶ `}`
- ▶ The copy constructor is invoked by instantiating an object of type `Item` and passing it the object to be copied. *Example:*
  - ▶ `Item item2 = new Item (item1);`
  - ▶ Now, `item2` is a copy of `item1`.

# DESTRUCTORS

- ▶ A destructor is opposite to a constructor. It is a method called when an object is no more required. The name of the destructor is the same as the class name and is preceded by a tilde (~). Like constructors, a destructor has no return type. *Example:*
- ▶ class Fun
- ▶ {
- ▶ ...
- ▶ ...
- ▶ ~Fun (J / /No arguments
- ▶ {
- ▶ ...
- ▶ }
- ▶ }
- ▶ Note that the destructor takes no arguments.

# CLASSICAL INHERITANC

- ▶ Inheritance represents a kind of relationship between two classes. Let us consider two classes **A** and **B**. We can create a class hierarchy such that **B** is derived from **A**.
  - ▶ Class **A**, the initial class that is used as the basis for the derived class is referred to as the *base class*, *parent class* or *super class*.
  - ▶ Class **B**, the derived class, is referred to as *derived class* *child class* or *sub class*. A derived class is a completely new class that incorporates all the data and methods of its base class. It can also have its own data and method members that are unique to itself.
- 

- ▶ We can now create objects of classes A and B independently.
- ▶ *Example:*
- ▶ **A** a; // a is object of A
- ▶ **B** b; // b is object of B
- ▶ In such cases, we say that the object b is a type of a. Such relationship between a and b is referred to as 'is-a' relationship.
- ▶ The classical inheritance may be implemented in different combinations :
- ▶ Single inheritance (only one base class)
- ▶ Multiple inheritance (several base classes)
- ▶ Hierarchical inheritance (one base class, many subclasses)
- ▶ Multilevel inheritance (derived from a derived class)



# CONTAINMENT INHERITANCE

- ▶ We can also define another form of inheritance relationship known as *containership* between class A and B.

*Example:*

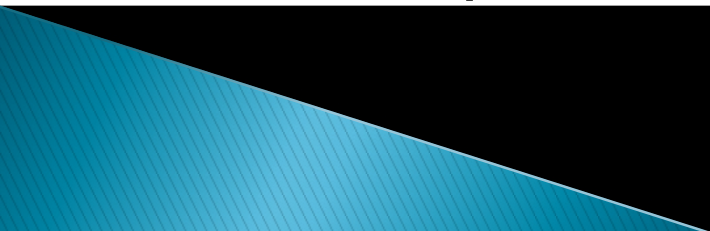
- ▶ class A
- ▶ {
- ▶ .....
- ▶ }
- ▶ class B
- ▶ {

- ▶ ....
- ▶ A a; // a is contained in b
- ▶ }
- ▶ B b;
- ▶ ...
- ▶ In such cases, we say that the object **a** is contained in the object **b**. This relationship between **a** and **b** is referred to as 'has-a' relationship. The outer class **B** which contains the inner class **A** is termed the 'parent' class and the contained class **A** is termed a 'child' class.
- ▶ Examples are:
  - Car has-a radio
  - House has-a store room
  - City has-a road

# DEFINING A SUBCLASS

- ▶ A subclass is defined as follows:
- ▶ Class *subclass-name* : *baseclass-name*
- ▶ {
- ▶ variables declaration ;
- ▶ methods declaration ;
- ▶ }
- ▶ The definition is very similar to a normal class definition except for the use of colon: and *base-class - name*. The colon signifies that the properties of the base-class are extended to the *subclass-name*. When implemented the subclass will contain its own members as well those of the base-class.

# VISIBILITY CONTROL

- ▶ **1. Class Visibility**
  - ▶ Each class needs to specify its level of visibility. Class visibility is used to decide which parts of the system can create class objects.
  - ▶ A C# class can have one of the two visibility modifiers: **public** or **internal**. If we do not explicitly mark the visibility modifier of a class, it is implicitly set to 'internal' ; that is, by default all classes are **internal**. Internal classes are accessible within the same program assembly and are not accessible from outside the assembly.
- 

## ▶ 2. Class Members Visibility

▶ A class member can have any one of the five visibility modifiers:

- public
- protected
- private
- internal
- protected internal

## ▶ 3. Accessibility of Baseclass Members

▶ When a class inherits from a baseclass, all members of the base class, except constructor and destructors, are inherited and become members of the derived class. The declared accessibility of a base class member has no control over its inheritability. However, an inherited member may not be accessible in a derived class, either because of its declared accessibility or because it is hidden by a declaration in the class itself.

## 4. Accessibility Constraints

- ▶ C# imposes certain constraints on the accessibility of members and classes when they are used in the process of inheritance.
  - The direct base class of a derived class must be at least as accessible as the derived class itself.
  - Accessibility domain of a member is never larger than that of the class containing it.
  - The return type of method must be at least as accessible as the method itself.

# DEFINING SUBCLASS CONSTRUCTORS


- ▶ We have seen that an object is created when constructor is called. The same principle may be applied for constructing the derived class objects as well.
- ▶ We can define an appropriate constructor for the derived class that may be invoked when a derived class object is created. Remember, the purpose of a constructor is to provide values to the data fields of the class.
- ▶ `BedRoom room1 = new Bed Room (14,12,10);`
- ▶ calls first the `BedRoom` constructor method which in turn calls the `room` constructor method by using the `base` keyword.

# MULTILEVEL INHERITANCE

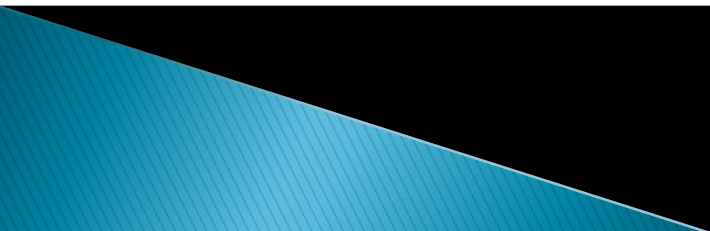
- ▶ A common requirement in object-oriented programming is the use of a derived class as a super-class. C# supports this concept and uses it extensively in building its class library. This concept allows us to build a chain of classes.
- ▶ The class **A** serves as a base class for the derived class **B** which in turn serves as a base class for the derived class **C**. The chain **ABC** is known as *inheritance path*.



- ▶ A derived class with multilevel base classes is declared as follows:
- ▶ **class A**
- ▶ {
- ▶ ....
- ▶ }
- ▶ **class B : A // First level derivation**
- ▶ {
- ▶ ...
- ▶ }
- ▶ **class C : B // Second level derivation**
- ▶ {
- ▶ ....
- ▶ }

- ▶ This process may be extended to any number of levels. The class **C** can inherit the members of both **A** and **B**.
  - ▶ The constructors are executed from the top downwards, with the bottom most class constructor being executed last.
  - ▶ **HIERARCHICAL INHERITANCE**
  - ▶ Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below the level.
- 

# OVERRIDING METHODS

- ▶ We have seen that a method defined in a super-class is inherited by its subclass and is used by the objects created by the subclass. Method inheritance enables us to define and use methods repeatedly in subclasses.
  - ▶ However, there may be occasions when we want an object to respond to the same method but behave differently when that method is called. That means, we should override the method defined in the super-class.
  - ▶ This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass. Then, when that method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass,
- 

# HIDING METHODS

- ▶ We declared the base class method as virtual and the subclass method with the keyword `override`. This resulted in 'hiding' the base class method from the subclass.
- ▶ Now, let us assume that we wish to derive from a class provided by someone else and we also want to redefine some method contained in it. Here, we cannot declare the base class methods as virtual. Then, how do we override a method without declaring it virtual? This is possible in C#. We can use the modifier **`new`** to tell the compiler that the derived class method "hides" the base class method.

# ABSTRACT CLASSES

- ▶ In a number of hierarchical applications, we would have one base class and a number of different derived classes. The top-most base class simply acts as a base for others and is not useful on its own. In such situations, we might not want any one to create its objects. We can do this by making the base class **abstract**.
- ▶ The **abstract** is a modifier and when used to declare a class indicates that the class cannot be instantiated. Only its derived classes (that are not marked abstract) can be instantiated.

# ABSTRACT METHODS

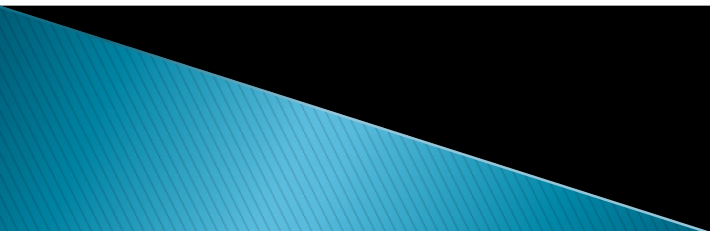
- ▶ Similar to abstract classes, we can also create abstract methods. When an instance method declaration includes the modifier **abstract**, the method is said to be an *abstract method*.
- ▶ An abstract method is implicitly a virtual method and does not provide any implementation. Therefore, an abstract method does not have method body.

*Example:*

- ▶ `public abstract void Draw(int x, int y);`

# SEALED CLASSES: PREVENTING INHERITANCE

- ▶ Sometimes, we may like to prevent a class being further subclassed for security reasons. A class that cannot be subclassed is called a *sealed class*. This is achieved in C# using the modifier **sealed** as follows:
- ▶ **sealed** class Aclass
- ▶ {
- ▶ ...
- ▶ }
- ▶ **sealed** class Bclass: Someclass
- ▶ {
- ▶ ...
- ▶ }

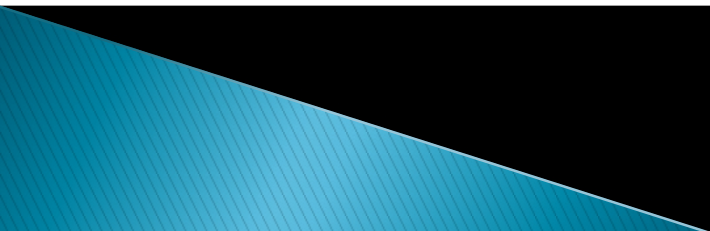
- ▶ Any attempt to inherit these classes will cause an error and the compiler will not allow it.
  - ▶ **SEALED METHODS**
  - ▶ When an instance method declaration includes the sealed modifier, the method is said to be a *sealed method*. It means a derived class cannot override this method.
  - ▶ A sealed method is used to override an inherited virtual method with the same signature. That means, the sealed modifier is always used in combination with the override modifier.
- 



- ▶ *Example:*
- ▶ class A
- ▶ {
- ▶ public virtual void Fun( )
- ▶ {
- ▶ ...
- ▶ }
- ▶ }
- ▶ class B : A
- ▶ {
- ▶ public sealed override void Fun ( )
- ▶ {
- ▶ ....
- ▶ }
- ▶ }

- ▶ The **sealed** method **Fun()** overrides the **virtual** method **Fun( )** defined in Class **A**. Any derived class of **B** cannot further override the method **Fun()**.

# Polymorphism

- ▶ *Operation polymorphism* is implemented using overloaded methods and operators.
  - ▶ The overloaded methods are 'selected' for invoking by matching arguments, in terms of number, type and order. This information is known to the compiler at the time of compilation and, therefore, the compiler is able to select and bind the appropriate method to the object for a particular call at *compile time* itself. This process is called *early binding*, or *static binding*, or *static finding*. It is also known as *compile time polymorphism*.
- 

- ▶ *Inclusion polymorphism* is achieved through the use of virtual functions. Assume that the class **A** implements a virtual method **M** and classes **B** and **C** that are derived from **A** override the virtual method **M**.
- ▶ When **B** is cast to **A**, a call to the method **M** from **A** is dispatched to **B**. Similarly, when **C** is cast to **A**, a call to **M** is dispatched to **C**. The decision on exactly which method to call is delayed until runtime and, therefore, it is also known as *runtime polymorphism*.
- ▶ Since the method is linked with a particular class much later after compilation, this process is termed *late binding*. It is also known as *dynamic binding* because the selection of the appropriate method is done dynamically at runtime.