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-IV: Interfaces**: Defining an interface – Extending an interface – Implementing interface – Interfaces and inheritance - **Operator overloading**: Overloadable operators – Need for operator overloading – Defining operator overloading – Overloading unary operators – Overloading binary operators – Overloading comparison operators - **Delegates and events** : Delegates-Delegate declaration-Delegate methods-Delegate instantiation-delegate invocation-using delegates-multicast delegates-events.

Interface

- A large number of real-life applications require the use of multiple inheritance whereby we inherit methods and properties from several distinct classes. Since 'C++ - like' implementation of multiple inheritance proves difficult and adds complexity to the language, C# provides an alternate approach known as *interface* to support the concept of multiple inheritance.
- An interface in C# is a reference type. It is basically a kind of class with some differences. Major differences include:
  - All the members of an interface are implicitly public and **abstract.**
  - An interface cannot contain constant fields, constructors and destructors.

- Its members cannot be declared **static.**
- Since the methods in an interface are abstract, they do not include implementation code.
- An interf ace can inherit multiple interfaces.

- **DEFINING AN INTERFACE**
- An interface can contain one or more methods, properties, indexers and events but none of them are implemented in the interface itself. It is the responsibility of the class that implements the interface to define the code for implementation of these members.

- The syntax for defining an interface is very similar to that use d for defining a class. The general form of an interface definition is:
- interface *InterfaceName*
- {
- Member declaration;
- }

  Here, **interface** is the keyword and *InterfaceName* is a valid C# identifier (just like class names). Member declarations will contain only a list of members with out implementation code. Given below is a simple interface that defines a single method:

- interface Show
- {
- void Display ( ); // Note semicolon here
- }
- In addition to methods, interfaces can declare properties, indexers and events. *Example:*
- inter face Example
- {
- int Aproperty
- {
- get ;
- }
- event someEvent Changed;
- void Display ( );
- }

- The accessibility of an interface can be controlled by using the modifiers **public, protected, internal** and **private.** The use of a particular modifier depends on the contact in which the interface declaration occurs.
- **EXTENDING AN INTERFACE**
- Like classes, interfaces can also be extended. That is, an interface can be sub-interfaced from other interfaces. The new sub-interface will inherit all the members of the super-interface in the manner similar to subclasses. This is achieved as follows:
- **interface** name2 : *name1*
- {
- Members of name2
- }

- For example, we can put all members of particular behavior category in one interface and the members of another category in the other. Consider the code below:
- interface Addition
- {
- int Add (int x, int y) ;
- }
- Interface Compute : Addition
- {
- int Sub (int x, int y);
- }

- The interface **Conıpute** will have both the methods and any class implementing the interface **Conıpute** should implement both of them; otherwise, it is an error. We can also combine several inteıfaces together into a s ingle inteıface. Following declarations are valid:
- interface I1
- {
- .....
- }
- interface I2
- {
- .....
- }
- interface I3 : I1, I2 // multiple inheritance
- {
- .....
- }

- While interfaces are allowed to extend other interfaces, sub-interfaces cannot define the methods declared in the super-interfaces. After all, sub interfaces are still interfaces, not classes. It is the responsibility of the class that implements the derived inter face to define all the methods. Note that when an interface extends two or more interfaces, they are separated by commas.
- **IMPLEMENTING INTERFACES**
- Interfaces are used as ' superclasses' whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface.

This is done as follows:

- **class** *classname* : **interfacename**
- {
- body of classname
- }
- Here the class **classname** ' implements' the interface **interfacename.** A more general form of implementation may look like this:
- **class** *classname* : superclass, interface1, interface2....
- {
- body of classname
- }

- ln C#, we can derive from a single class and, in addition, implement as many interfaces as the class needs. When a class inherits from a superclass, the name of each interface to be implemented must appear after the superclass name. *Example:*
- class A: B, I1, I2, ...
- {
- ....
- }
- where B is a base class and I1, I2, ... are interfaces . The base class and interfaces are separated by commas.

INTERFACES AND INHERITANCE

- Most often we have situations where the base class of a derived class implements an interface. In such situations, when an object of the de rived class is converted to the interface type, the inheritance hierarchy is searched until it finds a class that directly implements the interface.
- dis.Print ( );
- calls the method **Print** () in base class B but not the one available in the derived class itself. This is because the derived class does not implement the interface.

Operator Overloading

- Operator overloading is one of the many exciting features of object-oriented programming. *C#* supports the idea of operator overloading. It means that C# operators can be defined to work with the user-defined data types such as **structs** and **classes** in much the same way as the built -in types.
- **OVERLOADABLE OPERATORS**
- There are quite a number of operators in C# that can be overloaded. There are also many others that cannot be overloaded. They are listed below:

- ***Overloadable operators*** :
- Binary arithmetic **:** + , * ,/ , - , %
- Unary arithmetic **: +, ~, ++, --**
- Binary bitwise : &, !, ^, <<, >>
- Unary bitwise : !,~ , true , false
- Logical operators : = = ,! =, > = , <, < = , >
- ***Operators that cannot be overloaded***
- Conditional operators : && ,||
- Compound assignment **: +=, -=,*=,/=,%=**
- Other operators : [ ], ( ), =,?: , - > , **new, sizeof, typeof, is, as**
- When we overload a binary operator, its compound assignment equivalent is implicitly over- loaded.

NEED FOR OPERATOR OVERLOADING

- Although operator overloading gives us syntactical convenience, it also help us greatly to generate more readable and intuitive code in a number of situations. These include:
  - Mathematical or physical modeling where we use classes to represent objects such as co- ordinates, vectors, matrices, tensors, complex numbers and so on.
  - Graphical programs where co-ordinate-related objects are used to represent positions on the screen.
  - Financial programs where a class represents an amount of money.
  - Text manipulations where classes are used to represent strings and sentences.

DEFINING OPERATOR OVERLOADING

- To define an additional task to an operator, we must specify what it means in relation to the class (or struct) to which the operator is applied. This is done with the help of a special method called *operator method*, which describes the task.
- The general form of an operator method is:
- public **static** *retval* **operator** *ap (arglist)*
- {
- Method body //task defined
- }

- The operator is defined in much the same way as a method, except that we tell the compiler it is actually an operator we are defining by the **operator** keyword, followed by the operator symbol *op*. The key features of operator methods are:
  - They must be defined as **public** and **static.**
  - The *retval* (return value) type is the type that we get when we use this operator. But, technically, it can be of any type.
  - The *arglist* is the list of arguments passed. The number of arguments will be one for the unary operators and two for the binary operators.

- In the case of unaıy operators, the argument must be the same type as that of the enclosing class or struct.
- In the case of binary operators, the first argument must be of the same type as that of the enclosing class or struct and the second may be of any type.

- Examples of overloaded operators are:
- // vector addition
- public static Vector operator + (Vector a, Vecto r b)
- // unary minus
- public static Vector operator -(Vecto r a)
- // comparison
- public static boot operator ==(Vector a, Vector b)

OVERLOADING UNARY OPERATORS

- Let us consider the unary minus operator. A minus operator, when used as unary, takes just one argument. We know that this operator changes the sign of an operand when applied to a basic data item.
- We shall see here how to overload this operator so that **it** can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items.
- The method **operator -**() takes one argument of type **Space** and changes the sign of data members of the object s. Since it is a member method of the same class, it can directly access the members of the object which activated it.

OVERLOADING BINARY OPERATORS

- We have just seen how to overload a unary operator. The same mechanism can be used to overload a binary operator. We can always use methods to add two objects. A statement like
- C = sum (A, B); //functional notation
- is possible. The functional notation can be replaced by a natural looking expression
- C = A + B; // arithmetic notation.
- by overloading the + operator using an operator + ( ) method. Here is the syntax used to define the operator + () method.

- public static Vector operator + (Vector u1, Vector u2)
- {
- //Create a new Vector object
- //Add the contents of u1 and u2
- // to the new Vector object
- // Return the new vector object
- }
- In this program overloads the binary plus operator to add two complex numbers of type:
- x =a + jb

OVERLOADING COMPARISON OPERATORS

- C# supports six comparison operators that can be considered in three pairs:
- = = and ! =
  - > **and** > =
  - < **and** < =
- The significance of pairing is two-fold.
- 1.Within each pair, the second operator should always give exactly the opposite result to the first. That is, whenever the first returns **true**, the second returns **false** and vice versa.
- 2.C# always requires us to overload the comparison operators in pairs. That is, if we overload ==, then we must overload ! = also, otherwise it is an error.

- There is one fundamental difference between overloading comparison operators and overloading arithmetic operators. Comparison operators must return a **bool** type value. Apart from these differences, overloading comparison operators follows the same principles as overloading the arithmetic operators.
- The overloading of == operator can also be done by using the method **Equal** ( ) as shown below:
- Public static boot operator == (Vector u1, Vector u2)
- {
- return (u1, Equals (u2));
- }

- In this case, we must override the method **Equals** () defined in **System** namespace as given below:
- public override bool Equals (object value)
- {
- Vector u = (Vector) value;
- return ((this.x == u.x) && (this.y == u.y) && (this.z == u.z));
- }
- Overloads the == Boolean operator and displays the value on the basis of the division done and the value of the output is compared with another value to check whether they are equal. Another method Equal() uses the same functionality of the == operator to display the data.

Delegates and Events

- In object oriented programming, it is the usual practice for one object to send messages to other objects.
- However in real-life applications, it is quite common for an object to report back to the object that was responsible for sending a message. This, in effect, results in a two-way conversation between objects. The methods used to call back messages are known as *callback methods*.
- C# implements the callback technique in a much safer and more object-oriented manner, using a kind of object called **delegate** object.

- A delegate object is a special type of object that contains the details of a method rather than data.
- Delegates *in* C# are used for two purposes:
  - Callback
  - Event handling
- **DELEGATES**
- The dictionary meaning of **delegate** is "a person acting for another person". In C#, it really means a method acting for another method. As pointed out earlier, a delegate in C# is a class type object and is used to invoke a method that has been encapsulated into it at the time of its creation.

- Creating and using delegates involve four steps. They include:
- Delegate declaration
- Delegate methods definition
- Delegate instantiation
- Delegate invocation
- A delegate declaration defines a class using the class **System.Delegate** as a base class. Delegate methods are any functions (defined in a class) whose signature matches the delegate signature exactly.
- The delegate instance holds the reference to delegate methods. The instance is used to invoke the methods indirectly.

DELEGATE DECLARATION

- A delegate declaration is a type declaration and takes the following general form:
- *modifier* **delegate** *return-type delegate-name (parameters);*
- **delegate** is the keyword that signifies that the declaration re presents a class type derived from **System.Delegate.** The *return-type* indicates the return type of the delegate. *Parameters* identifies the signature of the delegate. The *delegate-name* is any valid C# identifier and is the name of the delegate that will be used to instantiate delegate objects.

- The *modifier* controls the accessibility of the delegate. It is optional. Depending upon the context in which they are declared, delegates may take any of the following modifiers:
- **new**
- **public**
- **protected**
- **internal**
- **private**
- The **new** ınodifier is only permitted on delegates declared within another type. It signifies that the delegate hides an inherited member by the same name.

- Some examples of delegates are:
- delegate void SimpleDelegate ( );
- delegate int MathOperation(int x, int y);
- public delegate int Compareltems(object 01, object 02);
- private delegate string GetAString( );
- delegate double Doubl eOperation(double x);
- Although the syntax is similar to that of a method definition (without method body), the use of keyword delegate tells the compiler that it is the definition of a new class using the System.Delegate as the base class.

- Since it is a class type, it can be defined in any place where a class definition is permitted. That is, a delegate may be defined in the following places:
- Inside a class
- Outside all classes
- As the top level object in a namespace
- Depending on how visible we want the delegate to be, we can apply any of the visibility modifiers to the delegate definition.
- Delegate types are implicitly **sealed** and therefore it is not possible to derive any type from a delegate type. It is also not permissible to derive a non-delegate class type from **System.Delegate.**

DELEGATE METHODS

- The methods whose references are encapsulated into a delegate instance are known as *delegate methods* or *callable entities*. The signature and return type of delegate methods must exactly match the signature and return type of the delegate.
- One feature of delegates, as pointed out earlier, is that they are type-safe to the extent that they ensure the matching of signatures of the delegate methods. However, they do not care :
- what type of object the method is being called against
- whether the method is a static or an instance method.

- For instance, the delegate
- delegate string Get AString( )
- can be made to refer to the method **ToString()** using an **int** object **N** as follows:
- ....
- int **N** = 100
- GetAString s1 = new GetAString(N.ToString);
- The delegate
- delegate void Delegate1();
- can encapsulate references to the following methods:

- public void F1( ) / /instance method
- {
- Console.WriteLine(" F1");
- }
- static public void F2( ) //static method
- {
- Console.WriteLine("F2");
- }
- In the above code, the signature and return type of methods match the signature and type of the delegate.

DELEGATE INSTANTIATION

- Although delegates are of class types and behave like classes, C# provides a special syntax for instantiating their instances. A *delegate-creation-expression* is used to create a new instance of a delegate.
- new *delegate-type (expression)*
- The *delegate-type* is the name of the delegate declared whose object is to be created. The *expression* must be a method name or a value of a *delegate-type*. If it is a method name its signature and return type must be the same as those of the delegate.

- If no matching method exists, or more than one matching method exists, an error occurs. The matching method may be either an instance method or a static method.
- The method and the object to which a delegate refers are determined when the delegate is instantiated and then remain constant for the entire lifetime of the delegate. It is, therefore, not possible to change them, once the delegate is created.
- It is also not possible to create a delegate that would refer to a constnıctor, indexer, or user defined operator.

- Consider the following code:
- // delegate declaration
- delegate int ProductDelegate (int x, int y);
- class Delegate
- {
- static float Product (float a, float b) // signature does
- // not match
- {
- return(a*b);
- }
- static int Product (int a, int b) / / signature matches
- {
- return (a * b);
- }

- // delegate instantiation
- **ProductDelegate p = new Product Delegate(Product);**
- }
- Here, we have two methods with the same name but with different signatures. The delegate **p** is initialized with the reference to the second **Product** method because that method exactly matches the signature and return type of **ProductDelegate.**
- The code defines two delegate methods in two different classes. Since class **A** defines an instance method, an **A** type object is created and used with the method name to initialize the delegate object **d1.** The delegate method defined in class **B** is static and therefore the class name is used directly with the method name in creating the delegate object **d2.**

DELEGATE INVOCATION

- C# uses a special syntax for invoking a delegate. When a delegate is invoked, it in nırn invokes the method whose reference has been encapsulated into the delegate, (only if their signatures match). Invocation takes the following form:
- delegate_object *(parameters list)*
- The optional *parameters list* provides values for the parameters of the method to be used.
- If the invocation invokes a method that returns **void,** the result is nothing and therefore it cannot be used as an operand of any operator. It can be simply a statement_expression. *Example:*
- delegate1(x, y); // void delegate
- This delegate invokes a method that does not return any value.

- If the method returns a value, then it can be used as an operand of any operator. Usually, we assign the return value to an appropriate variable for further processing. *Example:*
- double result = delegat e2(2.56, 45.73);
- This statement invokes a method (that takes two double values as parameters and returns double type value) and then assigns the returned value to the variable **result.**
- **USING DELEGATES**
- If we need to implement more delegate methods, we have to create more delegate objects. In such cases, we may create an array of delegate objects and then use them in a **for** loop to invoke the methods.

- *Example:*
- ArithOp [ ] operation =
- {new ArithOp(MathOperation.Add),
- new ArithOp(MathOperation.Sub)
- };
- This creates two delegates, **operation[0]** to invoke **Add** method and **operation** [1] to invoke **Sub** method.
- It is also allowed to use delegate objects as method parameters. For instance:
- ProcessMethod(operation[0], 200, 100);
- is valid.

MULTICAST DELEGATE

- A delegate can invoke only one method (whose reference has been encapsulated into the delegate). However, it is possible for certain delegates to bold and invoke ınultiple methods.
- Such delegates arc called *multicast delegates*. Multicast delegates, also known as *combinable delegates*, must satisfy the following co11ditions:
  - The return type of the delegate must be **void.**
  - None of the parameters of the delegate type can be declared as output parameters, using **out** keyword.

- If **D** is a delegate that satisfies the above conditions and **d1 , d2, d3** and **d4** are the instances of **D**, then the statements :
- d3 = d1 + d2; // *d3 refers* to two methods
- d4 = d3 - d2; // d4 refers to only d1 method
- are valid provided that the delegate instances **d1** and **d2** have already been initialized with method references and **d3** and **d4** contain **null** reference.
- For a multicast delegate instance that was created by combining two delegates, the invocation list is formed by concatenating the invocation list of the two operands of the addition operation. Delegates are invoked in the order they are added.

EVENTS

- An *event* is a delegate type class member that is used by the object or class 10 provide a notification to other objects that an event has occurred. The client object can act on an event by adding an *event handler* to the event.
- Events are declared using the simple *event declaration* format as follows:
- *modifier* **event** *type event-name;*
- The *modifier* may be new, a valid combination of the four access modifiers, and a valid combination of **static, virtual**, **override, abstract** and **sealed**. The *type* of an event declaration must be a delegate type and the delegate must be as accessible as the event itself.

- The *event-name* is any valid C# variable name. **event** is a keyword that signifies that the *event-name* is an event.
- Examples of event declaration are:
- public event EventHandler Click;
- public event RateChange Rate;
- **EventHandler** and **RateChange** are delegates and **Click** and **Rate** are events. Since events are based on delegates, we must first declare a delegate and then declare an instance of the delegate using the keyword **event**.