**Delegates In C# Programming**

* Delegate meaning from google: a person sent or authorized to represent others, in particular an elected representative sent to a conference.
* Delegate is a type which holds a method’s reference in an object.
* It is also called function pointer.
* Delegate is of reference type.
* Delegate signature should be as same as the method signature referencing by a delegate.
* Delegate can point to a parameterized method or non-paramterized method.
* Delegate has no implementation means no body with { }.
* We can use invoke() method with delegates.
* Delegates are used to encapsulate methods.
* In the .Net framework, a delegate points to one or more methods. Once you instantiate the delegate, the corresponding methods invoke.
* Delegates are objects that contain references to methods that need to be invoked instead of containing the actual method names
* Using delegates, you can call any method, which is identified only at run-time.
* A delegate is like having a general method name that points to various methods at different times and invokes the required method at run-time.
* In c#, invoking a delegate will execute the referenced method at run-time.
* To associate a delegate with a particular method, the method must have the same return type and parameter type as that of the delegate.
* Consider two methods, Add() and Subtract(). The method Add() takes two parameters of type integer and returns their sum as an integer value. Similarly, the method Subtract() takes two parameters of type integer and returns their difference as an integer value.
* Since both methods have the same parameter and return types, a delegate, Calculation, can be created to be used to refer to Add() or Subtract(). However, when the delegate is called while pointing to Add(), the parameters will be added. Similarly, if the delegate is called while pointing to Subtract(), the parameters will be subtracted.

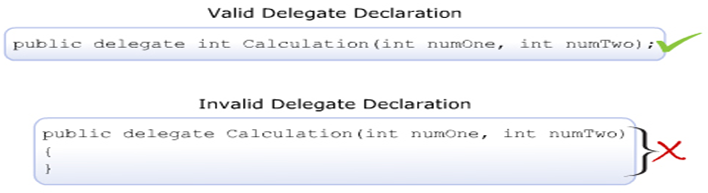
**Following are the features of delegates in C# that distinguish them from normal methods:**

* Methods can be passed as parameters to a delegate. In addition, a delegate can accept a block of code as a parameter. Such blocks are referred to as anonymous methods because they have no method name.
* A delegate can invoke multiple methods simultaneously. This is known as multicasting.
* A delegate can encapsulate static methods.
* Delegates ensure type-safety as the return and parameter types of the delegate are the same as that of the referenced method. This ensures secured reliable data to be passed to the invoked method.

**Declaring Delegates**

* Delegates in C# are declared using the delegate keyword followed by the return type and the parameters of the referenced method.
* Declaring a delegate is quite similar to declaring a method except that there is no implementation. Thus, the declaration statement must end with a semi-colon.

The following figure displays an example of declaring delegates:



**The following syntax is used to declare a delegate:**

<access\_modifier> delegate <return\_type> DelegateName([list\_of\_parameters]);

**where,**

* **access\_modifier**: Specifies the scope of access for the delegate. If declared outside the class, the scope will always be public.
* **return\_type**: Specifies the data type of the value that is returned by the method.
* **DelegateName**: Specifies the name of the delegate.
* **list\_of\_parameters**: Specifies the data types and names of parameters to be passed to the method.

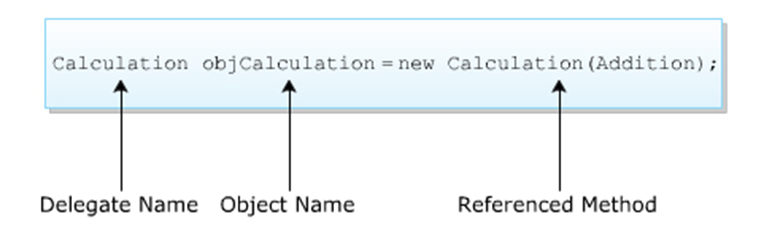
The following code declares the delegate Calculation with the return type and the parameter types as integer:

public delegate int Calculation(int numOne, int numTwo);

**Instantiating Delegates**

* The next step after declaring the delegate is to instantiate the delegate and associate it with the required method by creating an object of the delegate.
* Like all other objects, an object of a delegate is created using the new keyword.
* This object takes the name of the method as a parameter and this method has a signature similar to that of the delegate.
* The created object is used to invoke the associated method at run-time.

**The following figure displays an example of instantiating delegates:**



**The following syntax is used to instantiate a delegate:**

<DelegateName><objName> = new <DelegateName>(<MethodName>);

**where,**

* DelegateName: Specifies the name of the delegate.
* objName: Specifies the name of the delegate object.
* MethodName: Specifies the name of the method to be referenced by the delegate object.

**The following code declares a delegate Calculation outside the class Mathematics and instantiates it in the class:**

public delegate int Calculation (int numOne, int numTwo);

class Mathematics

{

static int Addition(int numOne, int numTwo)

{

return (numOne + numTwo);

}

static int Subtraction(int numOne, int numTwo)

{

return (numOne - numTwo);

}

static void Main(string[] args)

{

int valOne = 5;

int valTwo = 23;

Calculation objCalculation = new Calculation(Addition);

Console.WriteLine (valOne + “ + “ + valTwo + “ = “ +

objCalculation (valOne, valTwo));

}

}

**In Above Code,**

* The delegate called Calculation is declared outside the class Mathematics.
* In the Main() method, an object of the delegate is created that takes the Addition() method as the parameter. The parameter type of the method and that of the delegate is the same, which is type int.

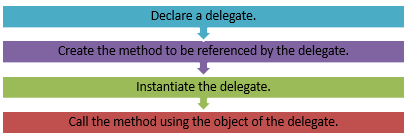
**Output**

5 + 23 = 28

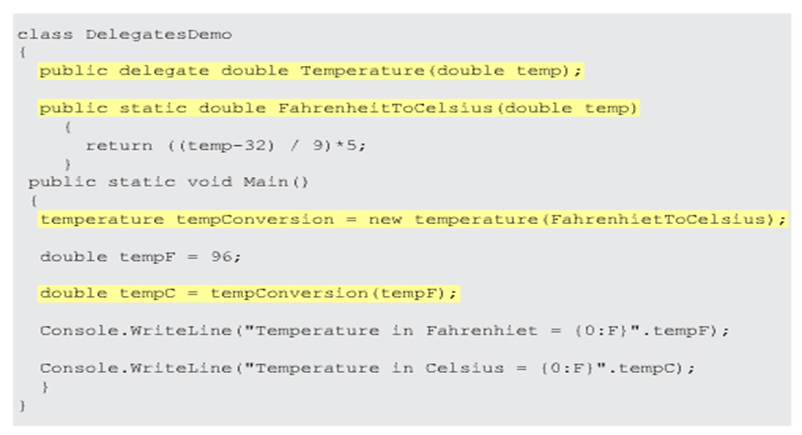
## Using Delegates

A delegate can be declared either before creating the class (having the method to be referenced) or can be defined within the class.

**The following are the four steps to implement delegates in C#:**



**Each of these step is demonstrated with an example shown in the following figure:**



## Delegate-Event Model

* The delegate-event model is a programming model that enables a user to interact with a computer and computer-controlled devices using graphical user interfaces. This model consists of:
  + An event source, which is the console window in case of console-based applications.
  + Listeners that receive the events from the event source.
  + A medium that gives the necessary protocol by which every event is communicated.
* In this model, every listener must implement a medium for the event that it wants to listen to by using the medium, every time the source generates an event, the event is notified to the registered listeners.

# Example

* Consider a guest ringing a doorbell at the doorstep of a home. The host at home listens to the bell and responds to the ringing action by opening the door.
* Here, the ringing of the bell is an event that resulted in the reaction of opening the door. Similarly, in C#, an event is a generated action that triggers its reaction.
* For example, pressing Ctrl+Break on a console-based server window is an event that will cause the server to terminate.
* This event results in storing the information in the database, which is the triggered reaction. Here, the listener is the object that invokes the required method to store the information in the database.
* Delegates can be used to handle events as they take methods that need to be invoked when events occur which are referred to as the event handlers.

## Types Of Delegates In C#

* Multiple Delegates
* Single Cast Delegates
* Multi Cast Delegates

## Multiple Delegates

* In C#, a user can invoke multiple delegates within a single program. Depending on the delegate name or the type of parameters passed to the delegate, the appropriate delegate is invoked.
* The following code demonstrates the use of multiple delegates by creating two delegates CalculateArea and CalculateVolume that have their return types and parameter types as double:

using System;

public delegate double CalculateArea(double val);

public delegate double CalculateVolume(double val);

class Cube

{

static double Area(double val)

{

return 6 \* (val \* val);

}

static double Volume(double val)

{

return (val \* val);

}

static void Main(string[] args)

{

CalculateArea objCalculateArea = new CalculateArea(Area);

CalculateVolume objCalculateVolume = new

CalculateVolume(Volume);

Console.WriteLine ("Surface Area of Cube: " +

objCalculateArea(200.32));

Console.WriteLine("Volume of Cube: " +

objCalculateVolume(20.56));

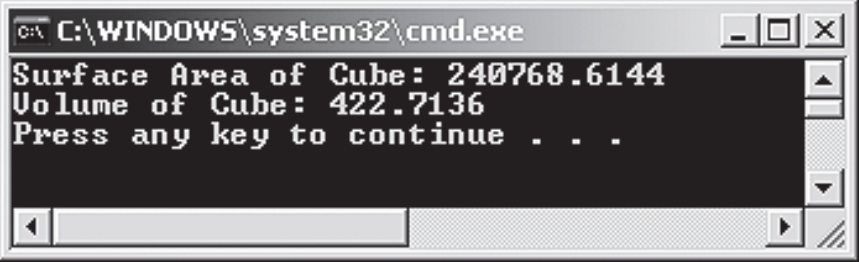
}

}

**In Above code,**

* When the delegates CalculateArea and CalculateVolume are instantiated in the Main() method, the references of the methods Area and Volume are passed as parameters to the delegates CalculateArea and CalculateVolume respectively.
* The values are passed to the instances of appropriate delegates, which in turn invoke the respective methods.

**The following figure shows the use of multiple delegates:**



## Single Cast Delegates

Singlecast delegate point to single method at a time. In this the delegate is assigned to a single method at a time. They are derived from System.Delegate class.

## Multicast Delegates

* A single delegate can encapsulate the references of multiple methods at a time to hold a number of method references.
* When a delegate is wrapped with more than one method that is known as a multicast delegate.
* In C#, delegates are multicast, which means that they can point to more than one function at a time. They are derived from System.MulticastDelegate class.
* Such delegates are termed as ‘Multicast Delegates’ that maintain a list of methods (invocation list) that will be automatically called when the delegate is invoked.
* Multicast delegates in C# are sub-types of the System.MulticastDelegate class. Multicast delegates are defined in the same way as simple delegates, however, the return type of multicast delegates can only be void.
* If any other return type is specified, a run-time exception will occur because if the delegate returns a value, the return value of the last method in the invocation list of the delegate will become the return type of the delegate resulting in inappropriate results. Hence, the return type is always void.
* To add methods into the invocation list of a multicast delegate, the user can use the ‘+’ or the ‘+=’ assignment operator. Similarly, to remove a method from the delegate’s invocation list, the user can use the ‘-‘ or the ‘-=’ operator. When a multicast delegate is invoked, all the methods in the list are invoked sequentially in the same order in which they have been added.
* We can use += and -= assignment operators to implement multi cast delegates.

**The following code creates a multicast delegate Maths. This delegate encapsulates the reference to the methods Addition, Subtraction, Multiplication, and Division:**

using System;

public delegate void Maths (int valOne, int valTwo);

class MathsDemo

{

static void Addition(int valOne, int valTwo)

{

int result = valOne + valTwo;

Console.WriteLine("Addition: " + valOne + " + " +

valTwo + "= " + result);

}

static void Subtraction(int valOne, int valTwo)

{

int result = valOne - valTwo;

Console.WriteLine("Subtraction: " + valOne + " - " +

valTwo + "= " + result);

}

static void Multiplication(int valOne, int valTwo)

{

int result = valOne \* valTwo;

Console.WriteLine("Multiplication: " + valOne + " \* "

+ valTwo + "= " + result);

}

static void Division(int valOne, int valTwo)

{

int result = valOne / valTwo;

Console.WriteLine("Division: " + valOne + " / " +

valTwo + "= " + result);

}

static void Main(string[] args)

{

Maths objMaths = new Maths(Addition);

objMaths += new Maths(Subtraction);

objMaths += new Maths(Multiplication);

objMaths += new Maths(Division);

if (objMaths != null)

{

objMaths(20, 10);

}

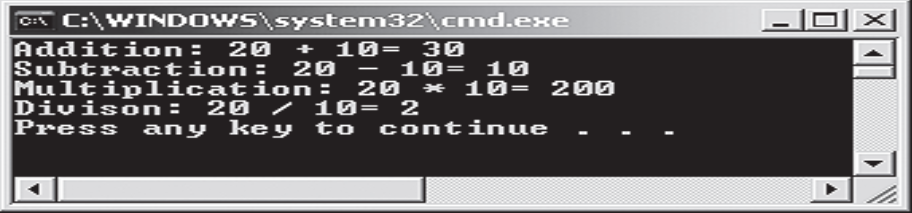
}

}

**In Above code,**

* The delegate Maths is instantiated in the Main() method. Once the object is created, methods are added to it using the ‘+=’ assignment operator, which makes the delegate a multicast delegate.

**The following figure shows the creation of a multicast delegate:**



## System.Delegate Class

* The Delegate class of the System namespace is a built-in class defined to create delegates in C#.
* All delegates in C# implicitly inherit from the Delegate class. This is because the delegate keyword indicates to the compiler that the defined delegate in a program is to be derived from the Delegate class. The Delegate class provides various constructors, methods, and properties to create, manipulate, and retrieve delegates defined in a program.
* The following table lists the constructors defined in the Delegate class:

| **Constructor** | **Description** |
| --- | --- |
| Delegate(object, string) | Calls a method referenced by the object of the class given as the parameter |
| Delegate(type, string) | Calls a static method of the class given as the parameter |

**The following table lists the properties defined in the Delegate class:**

| **Property** | **Description** |
| --- | --- |
| Method | Retrieves the referenced method |
| Target | Retrieves the object of the class in which the delegate invokes the referenced method |

**The following table lists some of the methods defined in the Delegate class:**

| **Method** | **Description** |
| --- | --- |
| Clone | Makes a copy of the current delegate |
| Combine | Merges the invocation lists of the multicast delegates |
| CreateDelegate | Declares and initializes a delegate |
| DynamicInvoke | Calls the referenced method at run-time |
| GetInvocationList | Retrieves the invocation list of the current delegate |

**The following code demonstrates the use of some of the properties and methods of the built-in Delegate class:**

using System;

public delegate void Messenger(int value);

class CompositeDelegates

{

static void EvenNumbers(int value)

{

Console.Write("Even Numbers: ");

for (int i = 2; i <= value; i+=2)

{

Console.Write(i + " ");

}

}

void OddNumbers(int value)

{

Console.WriteLine();

Console.Write("Odd Numbers: ");

for (int i = 1; i <= value; i += 2)

{

Console.Write (i + " ");

}

}

static void Start(int number)

{

CompositeDelegates objComposite = new CompositeDelegates();

Messenger objDisplayOne = new Messenger(EvenNumbers);

Messenger objDisplayTwo = new Messenger

(objComposite.OddNumbers);

Messenger objDisplayComposite =

(Messenger)Delegate.Combine

(objDisplayOne, objDisplayTwo);

objDisplayComposite(number);

Console.WriteLine();

Object obj = objDisplayComposite.Method.ToString();

if (obj != null)

{

Console.WriteLine ("The delegate invokes an instance

method: " + obj);

}

else

{

Console.WriteLine ("The delegate invokes only

static methods");

}

}

static void Main(string[] args)

{

int value = 0;

Console.WriteLine("Enter the values till which you want

to display even and odd numbers");

try

{

value = Convert.ToInt32(Console.ReadLine());

}

catch (FormatException objFormat)

{

Console.WriteLine("Error: " + objFormat);

}

Start(value);

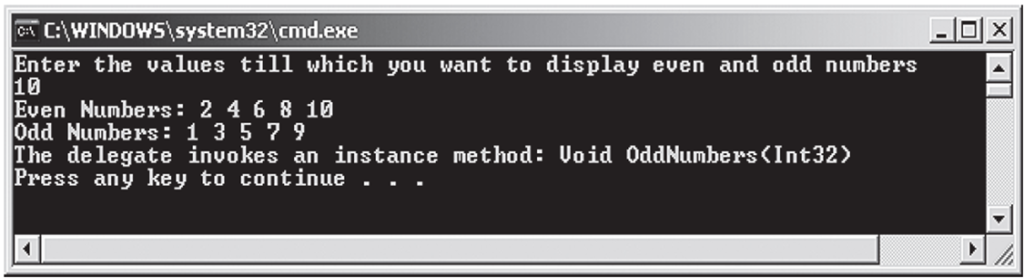
}

}

**In Above Code,**

* The delegate Messenger is instantiated in the Start() method.
* An instance of the delegate, objDisplayOne, takes the static method, EvenNumbers(), as a parameter, and another instance of the delegate, objDisplayTwo, takes the non-static method, OddNumbers(), as a parameter by using the instance of the class.
* The Combine() method merges the delegates provided in the list within the parentheses.
* The Method property checks whether the program contains instance methods or static methods. If the program contains only static methods, then the Method property returns a null value.
* The Main() method allows the user to enter a value. The Start() method is called by passing this value as a parameter. This value is again passed to the instance of the class CompositeDelegates as a parameter, which in turn invokes both the delegates.
* The code displays even and odd numbers within the specified range by invoking the appropriate methods.

**The following figure shows the use of some of the properties and methods of the Delegate class:**



## Source Code Of Delegates

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DELEGATES\_2

{

public delegate void Calculation(int a, int b);

public delegate void Calculation2();

class Program

{

public static void show()

{

Console.WriteLine("this is show method !!");

}

public static void Addition(int a, int b)

{

int result = a + b;

Console.WriteLine("Addition result is: {0}", result);

}

public static void Subtraction(int a, int b)

{

int result = a - b;

Console.WriteLine("Subtraction result is: {0}", result);

}

public static void Multiplication(int a, int b)

{

int result = a \* b;

Console.WriteLine("Multiplication result is: {0}", result);

}

public static void Division(int a, int b)

{

int result = a / b;

Console.WriteLine("Division result is: {0}", result);

}

static void Main(string[] args)

{

//Calculation2 obj5 = new Calculation2(show);

//obj5.Invoke();

Calculation obj = new Calculation(Program.Addition);

//obj.Invoke(20,10); // 30

obj += Program.Subtraction;

obj += Program.Multiplication;

obj.Invoke(20,10);

//Calculation obj1 = new Calculation(Program.Subtraction);

//obj1(20,10); // 10

//Calculation obj2 = new Calculation(Program.Multiplication);

//obj2(2, 2); // 4

//Calculation obj3 = new Calculation(Program.Division);

//obj3(10, 5); // 2

//obj = Subtraction;

//obj(20,10); // 10

//obj = Multiplication;

//obj(2,4); // 8

//obj = Division;

//obj(10,5); // 2

Console.ReadLine();

}

}

}