Callback Interface

Not only for polymorphism, interfaces can also be used for a callback mechanism. With call back mechanism, object can engage in two-way communication. In this article we will get how to achieve callback mechanism with interfaces in C#.

In this example, we will measure processing of any request and returns whenever it completes 100%.

Create call back interface as shown below.

```
public interface IProcessing
{
    void AboutToFinish(string sMsg);
    void Finished(string sMsg);
}
```

As shown above we have two methods AboutToFinish() & Finished(). AboutToFinish() method needs to call whenever the processing percentage is greater than 50% and less than 100%. Finished() method needs to call once the processing percentage becomes 100%. Callback interfaces are not implemented by helper object called sink object. Let's create sink class which implements IProcessing interface as shown below.

```
public class clsSink : IProcessing
{
    public void AboutToFinish(string sMsg)
    {
```

NAME:NIKSHITHA M
REG NO:193323724

```
        Console.WriteLine(sMsg);
    }
    public void Finished(string sMsg)
    {
        Console.WriteLine(sMsg);
    }
}
```

Now we have to create the class which takes the sink reference as input. This class should have methods for attaching and detach the sink reference as shown below.

```
public class ClsProcessing
{
    ArrayList clientSinks = new ArrayList();
    public void Attach(IProcessing sink)
    {
        clientSinks.Add(sink);
    }
    public void Detach(IProcessing sink)
    {
        clientSinks.Remove(sink);
    }
}
```

As of now we have created the callback mechanism by using interfaces. Let's add the notification method to the class ClsProcessing as shown below.

```
public class ClsProcessing
{
```

NAME:NIKSHITHA M
REG NO:193323724

```
ArrayList clientSinks = new ArrayList();
public void Attach(IProcessing sink)
{
 clientSinks.Add(sink);
}
public void Detach(IProcessing sink)
{
 clientSinks.Remove(sink);
}
int ipercentage=0;
public void Process(int iDelta)
{
 if(ipercentage>=100)
 {
  foreach(IProcessing sink in clientSinks)
  sink.Finished("100% completed");
 }
 else
 {
  ipercentage = iPercentage+10;
  foreach(IProcessing sink in clientSinks)
  sink.AboutToFinish(iPercentage.ToString()+"% completed");
 }
}
}
```

Lets implement the main() method as shown below which shows the callback interfaces functionality.

NAME:NIKSHITHA M
REG NO:193323724

```
class Program
{
  static void Main(string[] args)
  {
    ClsProcessing obj=new ClsProcessing();
    //create sink object
    ClsSink sink=new ClsSink();
    //attach sink object
     obj.Attach(sink);
     for(int i=0;i<10;i++)
     obj.Process(20);
     //detach sink object
     obj.Detach(sink);
     Console.ReadLine();
    }
  }
```

The output is as shown below:

```
10% completed
20% Completed
30% Completed
40% Completed
50% Completed
60% Completed
70% completed
80% completed
90% Completed
100% completed.
```

NAME:NIKSHITHA M
REG NO:193323724

Operator Overloading

C# supports the idea of operator overloading. It means that C# operator 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 type. For instance, C# permits us to add two classes objects with the same syntax that is applied to the basic types.

To define additional 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 describe the task. The general form of operator method is

```
public static retval operator op(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 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 argument passed.

NAME:NIKSHITHA M
REG NO:193323724

Creating Custom Conversion Routine

Begin by creating a new console Application named custom Conversion. C# provides two keywords, explicit and implicit, that you can use to control how your types respond during and attempted conversion. Assume you have the following structure definitions:

```
public struct Rectangle
{
  public int width {get; set}
  public int Height {get; set}
  public Rectangle (int w, int h) : this ()
  {
    width = w; Height = h;
  }
  public void Draw ()
  {
    for (int i=0; i<Height; i++)
    {
      for (int j=0; j<width; j++)
      {
        Console.Write ("*");
      }
      Console.WriteLine ();
    }
  }
  public struct Square
  {
```

NAME:NIKSHITHA M
REG NO:193323724

```
public int Length{get; set;}
public Square(int l): this()
{
Length = l;
}
public void Draw()
{
for (int i=0; i<Length; i++)
{
for (int j=0; j< Length; j++)
{
  Console.Write("*");
}
console.WriteLine();
}
}
public override string ToString()
{
return string.Format("[Length = {0}]", Length); }
public static explicit operator Square(Rectangle r)
{
Square s= new Square();
s.Length = r.Height;
return s;
}
}
```

NAME:NIKSHITHA M
REG NO:193323724

Notice that this iteration of the Square type defines an explicit conversion operator. Like the process of overloading an operator, conversion routines make use of the C# operator keyword in conjunction with the explicit or implicit keyword, and must be defined as static. The incoming parameter is the entity you are converting from, while the operator type is entity you are converting to.

Defining Implicit Conversion Routines

Consider the following implicit conversion code,

```
static void main(String[] args)
{
...
Square S3 = new Square();
S3.Length = 83;
Rectangle rect2 = S3;
Console.ReadLine();
}
```

This code will not compile, given that you have not provided an implicit conversion routine for the Rectangle type. Now here is the catch: It is illegal to explicit and implicit conversion functions on limitation; however the second catch is that when a type defines an implicit conversion routine, it is legal for the caller to make use of the explicit cast sytax. Lets add an implicit conversion routine to the Rectangle structure using the C# implicit keyword &

NAME:NIKSHITHA M
REG NO:193323724

```
public struct Rectangle
{
...
 public static implicit operator Rectangle (Square s)
 {
  Rectangle r=new Rectangle();
  r.Height =s.Length;
  r.Width =s.Length * 2;
  return r;
 }
}
Static void Main (String[] args)
{
...
  Square s3=new Square();
  s3.Length = 7;
  Rectangle rect2 =s3;
  Console.WriteLine("rect2={0}", rect2);
  Square s4 =new square();
  s4.Length = 3;
  Rectangle rect3 = (Rectangle)s4;
  Console.WriteLine("rect3 = {0}", rect3);
  Console.ReadLine();
 }
```

That wraps up our look at defining custom conversion routine. As with overloaded operators, remember that this bit of syntax is simply a shorthand notation for normal member functions and in this light it

NAME:NIKSHITHA M
REG NO:193323724

always optional. When used correctly however custom structure can be used more naturally, as they can be treated as true class types related by inheritance.

NAME:NIKSHITHA M
REG NO:193323724