Constructors and Destructors in C#

.NET basics

Document Prepared by :
Sehul Soni
Call : +91-9925025622
Email : sehulsoni@gmail.com
Chat : sehulsoni@yahoo.com
Talk : sehulsoni1011 (Skype)

Introduction

A constructor is a method in the class which gets executed when its object is created. Usually, we put the initialization code in the constructor. Have a look at the following sample:

```
public class mySampleClass
{
    public mySampleClass()
    {
        // This is the constructor method.
    }
    // rest of the class members goes here.
}
```

When the object of this class is instantiated, this constructor will be executed. Something like this:

```
mySampleClass obj = new mySampleClass()
// At this time the code in the constructor will // be executed
```

Constructor Overloading

C# supports overloading of constructors, that means, we can have constructors with different sets of parameters. So, our class can be like this:

```
public class mySampleClass
{
    public mySampleClass()
    {
        // This is the no parameter constructor method.
        // First Constructor
    }
    public mySampleClass(int Age)
    {
        // This is the constructor with one parameter.
        // Second Constructor
    }
    public mySampleClass(int Age, string Name)
    {
        // This is the constructor with two parameters.
        // Third Constructor
    }
    // rest of the class members goes here.
}
```

Well, note here that call to the constructor now depends on the way you instantiate the object. For example:

```
mySampleClass obj = new mySampleClass()
// At this time the code of no parameter
// constructor (First Constructor)will be executed

mySampleClass obj = new mySampleClass(12)
// At this time the code of one parameter
// constructor(Second Constructor)will be
// executed.
```

The call to the constructors is completely governed by the rules of overloading here.

Behavior of Constructors in Inheritance

Let us first create the inherited class.

```
public class myBaseClass
{
    public myBaseClass()
    {
        // Code for First Base class Constructor
    }
    public myBaseClass(int Age)
    {
        // Code for Second Base class Constructor
    }
    // Other class members goes here
}
public class myDerivedClass : myBaseClass
// Note that I am inheriting the class here.
{
    public myDerivedClass()
    {
        // Code for the First myDerivedClass Constructor.
    }

    public myDerivedClass(int Age):base(Age)
    {
        // Code for the Second myDerivedClass Constructor.
    }
    // Other class members goes here
}
```

Now, what will be the execution sequence here:

If I create the object of the derived class as:

```
myDerivedClass obj = new myDerivedClass()
```

Then the sequence of execution will be:

1. `public myBaseClass()` method.
2. and then `public myDerivedClass()` method.

Note: If we do not provide initializer referring to the base class constructor then it executes the no parameter constructor of the base class.

Note one thing here: we are not making any explicit call to the constructor of base class neither by initializer nor by the base keyword, but it is still executing. This is the normal behavior of the constructor.

If I create an object of the derived class as:

```
myDerivedClass obj = new myDerivedClass(15)
```

Then the sequence of execution will be:

1. `public myBaseClass(int Age)` method
2. and then `public myDerivedClass(int Age)` method

Here, the new keyword base has come into picture. This refers to the base class of the current class. So, here it refers to the `myBaseClass`. And `base(10)` refers to the call to `myBaseClass(int Age)` method.

Introduction

In the enterprise application development world, the buzzwords are Performance, Scalability, and Security. I started my career as a VC++ programmer, and one fine morning, I was transferred to Web development department. Like every C++ programmer, I also was frustrated. I thought every Tom, Dick, and our very own Harry can program in HTML. But, soon I found that the real challenge is to produce high performance, scalable, and reliable applications. And above all that the loosely coupled, stateless nature of web environment is always going to haunt you.

In order to produce high performance scalable applications, it is important to use your resources in an optimized manner. One tip is that use your resource as late as you can and free it at the earliest after your use. My intention here is to describe the object cleaning up mechanism used in C#.

Destructors

As we all know, Destructors are used to destruct instances of classes. When we are using destructors in C#, we have to keep in mind the following things:

- A class can only have one destructor.
- Destructors cannot be inherited or overloaded.
- Destructors cannot be called. They are invoked automatically.
- A destructor does not take modifiers or have parameters.

The following is a declaration of a destructor for the class `MyClass`:

```
~ MyClass()
{
   // Cleaning up code goes here
}
```

The programmer has no control on when the destructor is going to be executed because this is determined by the Garbage Collector. The garbage collector checks for objects that are no longer being used by the application. It considers these objects eligible for destruction and reclaims their memory. Destructors are also called when the program exits. When a destructor executes what is happening behind the scenes is that the destructor implicitly calls the `Object.Finalize` method on the object's base class. Therefore, the preceding destructor code is implicitly translated to:

```
protected override void Finalize()
{
   try
   {
      // Cleaning up .
   }
   finally
   {
      base.Finalize();
   }
}
```

Now, let us look at an example of how destructors are called. We have three classes `A`, `B` and `C`. `B` is derived from `A`, and `C` is derived from `B`. Each class has their own constructors and destructors. In the main of the class `App`, we create an object of `C`.

```
using System;
class A
{
 public A()
 {
  Console.WriteLine("Creating A");
 }
 ~A()
 {
  Console.WriteLine("Destroying A");
 }
}
```

```
class B:A
{
 public B()
 {
  Console.WriteLine("Creating B");
 }
 ~B()
 {
  Console.WriteLine("Destroying B");
 }

}
class C:B
{
 public C()
 {
  Console.WriteLine("Creating C");
 }

 ~C()
 {
  Console.WriteLine("Destroying C");
 }
}
class App
{
 public static void Main()
 {
  C c=new C();
  Console.WriteLine("Object Created ");
  Console.WriteLine("Press enter to Destroy it");
  Console.ReadLine();
  c=null;
  //GC.Collect();
  Console.Read();
 }
}
```

As we expect, the constructors of base classes will be executed and program will wait for the user to press 'enter'. When this occurs, we set the object of class `C` to `null`. But the destructors are not executing ..!!?? As we already said, the programmer has no control on when the destructor is going to be executed because the Garbage Collector determines this. But the destructors are called when the program exits. You can check this by redirecting the o/p of the program to a text file. I have the output here. Notice that the destructors of the base classes are called because behind the scenes `base.Finalize()` is called.

```
Creating A
Creating B
Creating C
Object Created
Press enter to Destroy it
Destroying C
Destroying B
Destroying A
```

So, what do you do if you want to call the destructors once you are finished using the object?

Calling the garbage collector

You can force the garbage collector to do clean up by calling the `GC.Collect` method, but in most cases, this should be avoided because it may result in performance issues. In the above program, remove the comment on `GC.Collect()`. Compile and run it. Now, you can see the destructors being executed in the console itself.