**Constructors**

One of the biggest advantages of an OOP language such as C# is that you can define special methods that are always called whenever an instance of the class is created. These methods are called *constructors*. C# introduces a new type of constructor called a *static constructor,* which you'll see in the next section

A key benefit of using a constructor is that it guarantees that the object will go through proper initialization before being used. When a user instantiates an object, that object's constructor is called and must return before the user can perform any other work with that object. It's this guarantee that helps ensure the integrity of the object and helps make applications written with object-oriented languages much more reliable.

constructors in C# must have the same name as the class itself. Here's a simple class with an equally simple constructor:

using System;

**class Constructor1App**

{

**Constructor1App()**

{

Console.WriteLine("I'm the constructor.");

}

public static void Main()

{

Constructor1App app = new Constructor1App();

}

}

Constructors do not return values. If you attempt to prefix the constructorwith a type, the compiler will emit an error stating that you cannot define members with the same names as the enclosing type.

You should also note the way in which objects are instantiated in C#. This is done using the *new* keyword with the following syntax:

*<class>* *<object>* = new *<class>* (*constructor arguments*)

## Constructor Initializers

All C# object constructors—with the exception of the *System.Object* constructors—include an invocation of the base class's constructor immediately before the execution of the first line of the constructor. These constructor initializers enable you to specify which class and which constructor you want called. This takes two forms:

* An initializer of the form *base(.)* enables the current class's base class constructor—that is, the specific constructor implied by the form of the constructor called—to be called

To see the order of events in action, note that the following code will execute the constructor for class A first and then the constructor for class B:

using System;

class A

{

**public A()**

{

Console.WriteLine("A");

}

}

class B : A

{

**public B()**

{

Console.WriteLine("B");

}

}

class DefaultInitializerApp

{

public static void Main()

{

B b = new B();

}

}

This code is the functional equivalent to the following code in which the base class's constructor is called explicitly:

using System;

class A

{

public A()

{

Console.WriteLine("A");

}

}

class B : A

{

**public B() : base()**

{

Console.WriteLine("B");

}

}

class BaseDefaultInitializerApp

{

public static void Main()

{

B b = new B();

}

}

Let's look at a better example of when constructor initalizers are useful. Once again, I have two classes: *A* and *B*. This time, class *A* has two constructors, one that takes no arguments and one that takes an *int*. Class *B* has one constructor that takes an *int*. The problem arises in the construction of class *B*. If I run the following code, the class *A* constructor that takes no arguments will be called:

using System;

class A

{

**public A()**

{

Console.WriteLine("A");

}

**public A(int foo)**

{

Console.WriteLine("A = {0}", foo);

}

}

class B : A

{

**public B(int foo)**

{

Console.WriteLine("B = {0}", foo);

}

}

class DerivedInitializer1App

{

public static void Main()

{

B b = new B(42);

}

}

So, how do I ensure that the desired class *A* constructor will be called? By explicitly telling the compiler which constructor I want called in the initalizer list, like so:

using System;

class A

{

public A()

{

Console.WriteLine("A");

}

**public A(int foo)**

{

Console.WriteLine("A = {0}", foo);

}

}

class B : A

{

**public B(int foo) : base(foo)**

{

Console.WriteLine("B = {0}", foo);

}

}

class DerivedInitializer2App

{

public static void Main()

{

B b = new B(42);

}

}