Instance constructors (C# programming guide)

Article • 11/15/2023

You declare an instance constructor to specify the code that is executed when you create a new instance of a type with the new expression. To initialize a static class or static variables in a nonstatic class, you can define a static constructor.

As the following example shows, you can declare several instance constructors in one type:

C#

```
class Coords
{
    public Coords()
        : this(0, 0)
    {  }

    public Coords(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; set; }
    public int Y { get; set; }

    public override string ToString() => $"({X},{Y})";
}

class Example
{
    static void Main()
    {
        var p1 = new Coords();
        Console.WriteLine($"Coords #1 at {p1}");
        // Output: Coords #1 at (0,0)

        var p2 = new Coords(5, 3);
        Console.WriteLine($"Coords #2 at {p2}");
        // Output: Coords #2 at (5,3)
    }
}
```

In the preceding example, the first, parameterless, constructor calls the second constructor with both arguments equal `0`. To do that, use the `this` keyword.

When you declare an instance constructor in a derived class, you can call a constructor of a base class. To do that, use the `base` keyword, as the following example shows:

C#

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {  }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x
* y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63

        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder =
```

```
{tube.Area():F2}");
        // Output: Area of the cylinder = 86.39
    }
}
```

Parameterless constructors

If a *class* has no explicit instance constructors, C# provides a parameterless constructor that you can use to instantiate an instance of that class, as the following example shows:

C#

```
public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output:  Name: unknown, Age: 0
    }
}
```

That constructor initializes instance fields and properties according to the corresponding initializers. If a field or property has no initializer, its value is set to the default value of the field's or property's type. If you declare at least one instance constructor in a class, C# doesn't provide a parameterless constructor.

A *structure* type always provides a parameterless constructor. The parameterless constructor is either an implicit parameterless constructor that produces the default value of a type or an explicitly declared parameterless constructor. For more information, see the Struct initialization and default values section of the Structure types article.

Primary constructors

Beginning in C# 12, you can declare a *primary constructor* in classes and structs. You place any parameters in parentheses following the type name:

C#

```
public class NamedItem(string name)
{
    public string Name => name;
}
```

The parameters to a primary constructor are in scope in the entire body of the declaring type. They can initialize properties or fields. They can be used as variables in methods or local functions. They can be passed to a base constructor.

A primary constructor indicates that these parameters are necessary for any instance of the type. Any explicitly written constructor must use the `this(...)` initializer syntax to invoke the primary constructor. That ensures that the primary constructor parameters are definitely assigned by all constructors. For any `class` type, including `record class` types, the implicit parameterless constructor isn't emitted when a primary constructor is present. For any `struct` type, including `record struct` types, the implicit parameterless constructor is always emitted, and always initializes all fields, including primary constructor parameters, to the 0-bit pattern. If you write an explicit parameterless constructor, it must invoke the primary constructor. In that case, you can specify a different value for the primary constructor parameters. The following code shows examples of primary constructors.

C#

```
// name isn't captured in Widget.
// width, height, and depth are captured as private fields
public class Widget(string name, int width, int height, int depth) :
NamedItem(name)
{
    public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube

    public int WidthInCM => width;
    public int HeightInCM => height;
    public int DepthInCM => depth;

    public int Volume => width * height * depth;
}
```

You can add attributes to the synthesized primary constructor method by specifying the `method:` target on the attribute:

C#

```
[method: MyAttribute]
public class TaggedWidget(string name)
{
```

```
    // details elided
}
```

If you don't specify the `method` target, the attribute is placed on the class rather than the method.

In `class` and `struct` types, primary constructor parameters are available anywhere in the body of the type. They can be used as member fields. When a primary constructor parameter is used, the compiler captures the constructor parameter in a private field with a compiler-generated name. If a primary constructor parameter isn't used in the body of the type, no private field is captured. That rule prevents accidentally allocating two copies of a primary constructor parameter that's passed to a base constructor.

If the type includes the `record` modifier, the compiler instead synthesizes a public property with the same name as the primary constructor parameter. For `record class` types, if a primary constructor parameter uses the same name as a base primary constructor, that property is a public property of the base `record class` type. It isn't duplicated in the derived `record class` type. These properties aren't generated for non-`record` types.

See also

- Classes, structs, and records
- Constructors
- Finalizers
- base
- this
- Primary constructors feature spec