Lesson 3 - RollingDie in C# .NET - Constructors and random numbers

[C# .NET](https://www.ict.social/csharp) [OOP](https://www.ict.social/csharp/oop) RollingDie in C# .NET - Constructors and random numbers

In the previous lesson, [First object-oriented application in C# - Hello object world](https://www.ict.social/csharp/oop/first-object-oriented-application-in-csharp-hello-object-world), we programmed our first object-oriented application in C# .NET. We are already able to create new classes with fields and parameterized methods with return values. Today, we're going to start working on an arena in which two warriors will battle against each other. The battle will be turn-based and one warrior will decrease the health points of other one based on his attack and on the other warrior's defense. Essentially, we'll be simulating a board game, so we'll add a rolling "die" to our game so as to add a bit of randomness. We're also going to learn to declare custom constructors.

We'll start by creating a new console application and naming it Arena. We will also add a new class to our project and name it RollingDie. Let's think about the fields we'll have to add to our die in order for it to function according to our current needs. We should be able to choose the number of sides. Typically 6 or 10 sides as is standard in this type of game. Our die will also need a random number generator. The .NET framework provides one for us, which includes the Random class for these sort of projects. Our class will now have two fields:

* *sidesCount* of the int type
* *random* of the Random type, where the random number generator will be stored.

Last time, for simplicity's sake, we set all the fields of our class as publicly accessible; however, in most cases, we don't want our fields to be modified externally. In this case, we will use the *private* modifier. A field is from then on only accessible from the inside of the class and from the outside C# treats it like it doesn't exist. When we design a class we usually set everything to private and then we make public only those class members we really need to expose. Our class should now look something like this:

/// <summary>

/// Class representing a die for a board game

/// </summary>

**class** RollingDie

{

/// <summary>

/// Random number generator

/// </summary>

**private** Random random;

/// <summary>

/// Number of sides that the die has

/// </summary>

**private** **int** sidesCount;

}

**Сonstructors**

We would now only be able to set the value of the public fields from outside of the class since the private fields are not visible from the outside. We've already discussed constructors a bit. A constructor is a method that is called **while an object instance is being created**. We use it to set the internal state of the object and perform any necessary initializations. We'd create a die in the Program.cs now like this:

RollingDie die = **new** RollingDie();

The RollingDie() method is the constructor. Since our class doesn't have a constructor, C# automatically generated an empty method. However, we'll now add a constructor to the class. We declare it as a method, but it **doesn't have a return type**. Also, the constructor must have the **same name as the class name**, which in our case is RollingDie. In the constructor, we'll set the number of sides to a fixed value and create an instance of the Random class. The constructor will look like this:

**public** RollingDie()

{

sidesCount = 6;

random = **new** Random();

}

If we create the die now, it'll have the *sidesCount* field set to 6 and a Random class instance will be stored in the *random*field. We'll print the number of sides to the console, so as to visually confirm that the value is there. It's not a good idea to set the field as public since we don't want somebody to be able to change the number of sides once the die is created.

We'll add a GetSidesCount() method to the class which returns the value of the *sidesCount* field. We have now successfully created a read-only field. Meaning that, the field is not visible and can only be read by using the "GetSidesCount()" method, so it can't be changed from outside the class. C# has other structures made for these purposes, but we'll go over them later. The new method would look something like this:

/// <summary>

/// Returns the number of sides the die has

/// </summary>

/// <returns>Number of sides the die has</returns>

**public** **int** GetSidesCount()

{

**return** sidesCount;

}

Let's move on to Program.cs so we could create our first die and print the number of its sides to the console:

[Run code](https://www.ict.social/) Click to edit

RollingDie die = **new** RollingDie(); // the constructor is called right here

Console.WriteLine(die.GetSidesCount());

Console.ReadKey();

The output:

Console application6

Now we have visual confirmation that the constructor had been called. However, we'd like to be able to specify how many sides our die will have. Let's add a parameter to our constructor:

**public** RollingDie(**int** aSidesCount)

{

sidesCount = aSidesCount;

random = **new** Random();

}

Notice that we prefixed the parameter name with "a", otherwise it would have the same name as the field and cause an error. Let's go back to Program.cs and pass a value for this parameter in the constructor:

[Run code](https://www.ict.social/) Click to edit

RollingDie die = **new** RollingDie(10); // a constructor with a par. 10 is called

Console.WriteLine(die.GetSidesCount());

Console.ReadKey();

The output:

Console application10

Everything works as expected. C# will not generate an empty, aka parameterless, constructor now, so we can't create a die without passing the parameter it needs. We can make it possible by adding another constructor, the parameterless one this time. We'll set the number of sides to 6 in it since the user probably expects this value as default:

**public** RollingDie()

{

sidesCount = 6;

random = **new** Random();

}

Let's create 2 dice instances now, one with a parameter and one without (in Program.cs):

[Run code](https://www.ict.social/) Click to edit

RollingDie sixSided = **new** RollingDie();

RollingDie tenSided = **new** RollingDie(10);

Console.WriteLine(sixSided.GetSidesCount());

Console.WriteLine(tenSided.GetSidesCount());

Console.ReadKey();

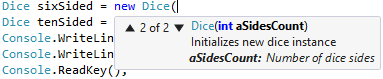
The output:

Console application

6

10

C# doesn't mind us having two methods with the same name as long as their parameters are different. We say that the RollingDie() method, which in this case is the constructor, is **overloaded**. This works for all methods with same names and different parameters, not just constructors. Visual Studio offers overloaded versions of methods when we're typing them. We can scroll through the method variants with the arrow keys. This "scroll-tool" is called IntelliSense. We can see our two constructor overloads on the list:



Many methods in .NET have several overloads. Take a look at the Remove() method on a string. It's good to look through method overloads, so you don't end up programming something what has already been done for you. E.g. the WriteLine() method, which you use for writing to the console, has 18 variants.

Now, let's go over how to name constructor parameters better than aSidesCount in our case. If we named them the same as the fields they initialize, we would cause an error:

**public** RollingDie(**int** sidesCount)

{

sidesCount = sidesCount;

random = **new** Random();

}

C# can't tell the two apart. In this case, the parameter would be assigned to the parameter again. We can refer to the current instance inside a class because it's stored in the **this** variable. We would use this e.g. if our die had aGiveToPlayer(player: Player) method, calling player.PickUpDie(this) inside. We'd essentially be passing itself, the concrete die instance with which we're working, to the other player by means of the **this** keyword. We won't deal with any of the "player" stuff for now, but we will set things up to reference the current instance while setting fields:

**public** RollingDie(**int** sidesCount)

{

**this**.sidesCount = sidesCount;

random = **new** Random();

}

Using this, we specified that the left variable, *sidesCount*, belongs to the instance. The right one is treated as a parameter by C#. Now we have two constructors that allow us to create different dice.

**Random numbers**

Next, we'll define a *Roll()* method in the RollingDie class, which will return a random number from 1 to the number of sides. The method will be public, meaning that it can be called from outside the class, and will have no parameters. The return value will be of the int type. We can obtain a random number by calling the Next() method on the generator. It has several overloads:

* Next(): The Parameterless variant returns a random number in the entire range of the int data type.
* Next(To): Returns a non-negative number lesser than the To bound. random.Next(100) therefore returns a number between 0 and 99.
* Next(From, To): Returns a random number between specified interval. From still belongs to the interval because it is inclusive, but To does not, because it is exclusive. Therefore random.Next (1, 101) would return a random number from 1 to 100;

The third overload suits our purposes the most, so we'll write:

/// <summary>

/// Rolls a die

/// </summary>

/// <returns>A number from 1 to sides count</returns>

**public** **int** Roll()

{

**return** random.Next(1, sidesCount + 1);

}

Do not create a random number generator in the method returning the random number. It would create a new generator for every random number and would result in the numbers hardly ever being random if at all. Always create a single shared instance of the generator, e.g. as a private field using the constructor, and then call the Next() method on it.

**Overriding the ToString() method**

Our RollingDie class is almost ready, let's add one more method before we wrap up this class. The ToString() method that we've briefly gone over in previous lessons is contained in **every** object, including our die. The method is designed to return a **textual representation of the instance**. It's handy in all cases where we need to print the instance or work with it as with text. Even numerical datatypes have this method.

We already know that C# performs implicit conversions. When we want to print a number or any other object to the console, C# calls the ToString() method and prints its return value. When creating a class, we should consider whetherToString() will come in handy. We should never make our own method such as Print() when there is already a way to handle string conversions in C#. It makes no sense to use it in our die, but when implementing warriors it could return their names. Still, for education's sake, we'll override ToString() in our RollingDie class anyway. It'll say that it's a die and print how many sides it has. First, let's try to print a RollingDie instance directly to the console:

[Run code](https://www.ict.social/) Click to edit

Console.WriteLine(sixSided);

Only the path to our class is printed to the console, which is Arena.RollingDie.

We can't just declare a ToString() method since it's already there (we'll find out why in upcoming tutorials). We must **override** it. We won't go too far in detail for now, but I want you to know what the proper usage of ToString() is. We use the **override** keyword for overriding:

[Run code](https://www.ict.social/) Click to edit

/// <summary>

/// Returns a textual representation of our die

/// </summary>

/// <returns>Textual representation of the die</returns>

**public** **override** **string** ToString()

{

**return** String.Format("Rolling a die with {0} sides", sidesCount);

}

Let's print out the die instance to the console again.

The output:

Console applicationRolling a die with 6 sides

Let's test our rolling dice now. We'll roll them in loops and see if they work as expected:

[Run code](https://www.ict.social/) Click to edit

// Create instances

RollingDie sixSided = **new** RollingDie();

RollingDie tenSided = **new** RollingDie(10);

// Rolls the 6-sided die

Console.WriteLine(sixSided);

**for** (**int** i = 0; i < 10; i++)

Console.Write(sixSided.Roll() + " ");

// Rolls the 10-sided die

Console.WriteLine("\n\n" + tenSided);

**for** (**int** i = 0; i < 10; i++)

Console.Write(tenSided.Roll() + " ");

Console.ReadKey();

The output should look something like this:

Console application

Rolling a die with 6 sides

3 6 6 1 6 3 6 2 6 3

Rolling a die with 10 sides

5 9 9 2 10 4 9 3 10 5

We've created a nice, customizable class that represents a rolling die. It'll come in handy when we get to the arena part of the application, but you can use it whenever you'd like. Hopefully, you now understand how OOP allows you to reuse components. In the next lesson, [Reference and value data types in C# .NET](https://www.ict.social/csharp/oop/reference-and-value-data-types-in-csharp-net), we'll talk about the differences between reference data types (objects) and value data types (e.g. int). :)

Download

[Download Arena.zip](https://www.ict.social/api/Articles-Article/file/56e9d2af82217)

Downloaded 28x (36.52 kB)   
 Application includes source codes in language C#

[Mark this lesson as absolved](https://www.ict.social/)

Article has been written for you by [David Capka](https://www.ict.social/portfolio/1)

[](https://www.ict.social/portfolio/1)

Do you like this article?

Top of Form

Bottom of Form

1 votes

The author is a programmer, who likes web technologies and being the lead/chief article writer at ICT.social. He shares his knowledge with the community and is always looking to improve. He believes that anyone can do what they set their mind to.



<https://www.ict.social/csharp/oop/rolling-die-in-csharp-net-constructors-and-random-numbers>