Constructors & Destructors

The lesson discusses in detail the use of constructors and destructors in classes, using some examples.

WE'LL COVER THE FOLLOWING

Introduction To Constructors

A class's *constructors* control its *initialization*.

A *constructor's* code executes to *initialize* an instance of the **class** when a program requests a **new** *object* of the class's type.

> **Note:** *Constructors* often set *properties* of their *classes*, but they are not restricted to doing so.

Constructor Declaration

Like other methods, a *constructor* can either have or not have *parameters*.

> **Note:** A constructor's **name** must be the **same** as the name of the *class* it

is declared in.

It is also important to keep in mind that **a constructor cannot return a value.**

> **Note:** No *return* type, not even `void` can be used while declaring a *constructor*.

Default Constructors

Constructors without any parameters are called **default** *constructors*.

The **default** *constructor* can be used to set some initial *default* value of the *members* of a class but it's not necessary.

Example

Let's take a look at how **default** constructors are *declared* and *called*.

```
using System;

public class Animal
{
  public Animal() { //default constructor
    Console.WriteLine("This Animal class's default constructor");
  }
  public int age = 5;
}

public class DefaultConstructorExample
{
    public static void Main()
    {
      Animal Dog = new Animal(); //default constructor is called here when a new object is ma
      Console.WriteLine("Doggo's age is {0}",Dog.age);
    }
}
```

> **Note:** When a type is *defined* without a *constructor* then the compiler generates a **default** *constructor* by **default**.

Constructors with Parameters

A *constructor* can be declared with *parameters* as well. To create an **object** using a *constructor* with *parameters*, the `new` command accepts parameters.

The *definition* of any *constructor* of such type will suppress the **default** *constructor* generation.

Example

Let's take a look at how such *constructors* are *declared* and *called*.

```
using System;

public class Animal
{
  public int age;
  public Animal(int _age) { //constructor with parameters
    age = _age;
  }

}

public class ConstructorExample
{
    public static void Main()
    {
      Animal Dog = new Animal(5); //constructor is called with 5 passed as age parameter
      Console.WriteLine("Doggo's age is {0}",Dog.age);
    }
}
```

If you want a *class* to have both a **default** *constructor* and a *constructor* that takes a **parameter**, you can do it by explicitly implementing both *constructors*.

```
using System;

public class Animal
{
  public int age;
  public Animal(int _age) { //constructor with parameters
    age = _age;
  }
  public Animal() { //default constructor
    age = 2;
  }
```

```
}


public class ConstructorExample
{
    public static void Main()
    {
      Animal Dog1 = new Animal(5); //object made by calling the constructor and passing 5 as
      Console.WriteLine("Doggo 1's age is {0}",Dog1.age);

      Animal Dog2 = new Animal(); //object made using default constructor
      Console.WriteLine("Doggo 2's age is {0}",Dog2.age);
    }
}
```

Constructor calling Constructor

Constructors can call each other as well.

Let's take a look at an example.

```
using System;
public class Animal
{
 public string Name;

  public Animal() : this("Dog")
  {
  }

  public Animal(string name)
  {
    Name = name;
  }
}

public class ConstructorCallingConstructor
{
    public static void Main()
    {
      Animal Dog1 = new Animal(); // dog.Name will be set to "Dog" by default.
      Console.WriteLine("Doggo's name is {0}",Dog1.Name);

      Animal cat = new Animal("Cat"); // cat.Name is "Cat", the empty constructor is not call
      Console.WriteLine("Cat's name is {0}",cat.Name);
    }
}
```

Introduction To Destructors

Introduction To Destructors

The *opposite* of **constructors**, **destructors** define the *final* behavior of an *object* and execute when the object is no longer in use.

> **Note:** Although **destructors** are often used in **C++** to **free** resources reserved by an *object*, they are less frequently used in **C#**.

An object's **destructor**, which takes **no** *parameters*, is called sometime after an *object* is no longer *referenced*, but the complexities of *garbage* collection make the specific timing of *destructors* uncertain.

> **Note: Destructors** cannot be called in **C#**. They are invoked *automatically*.

```
using System;

public class Animal
{
    public Animal(string text) //constructor definition
    {
        System.Console.WriteLine(text);
    }

    public void Print() //method prints name of the Animal
    {
      System.Console.WriteLine("Doggo's name is Lucy");
    }

    ~Animal() //destructor for the class Animal which gets called at the end
    {
        System.Console.WriteLine("Destructor Called!");
    }

    public static void Main()
    {
        Animal Dog = new Animal("Constructor Called"); //making new object of the class
        Dog.Print(); //calling the print function
    }
}
```

This marks the end of our discussion on **constructors** and **destructors** in C#.