Method Overloading

This lesson discusses method overloading and uses examples to explain the concept in detail

Definition

When *multiple methods* with the **same** name are declared with different *parameters*, it is referred to as *method overloading*.

Method overloading typically represents *functions* that are *identical* in their *purpose* but are written to accept **different** data types as their *parameters*.

Factors Affecting Method Overloading

- **Number of Arguments**
- **Type of arguments**
- **Return Type**

Consider a *method* named `Area` that will perform calculation functions, which will accept various arguments and return the result.

Example

Here’s an example showing the concept of *method overloading*.

```
using System;
class MethodOverloadingExample
{
    static void Main()
    {
        Console.WriteLine(Area(5)); //calling area function for square with int type paramete
        Console.WriteLine(Area(5)); //calling area function for square with int type paramete
        Console.WriteLine("Area of circle is: {0}", Area(5.5)); //calling area function for c
        Console.WriteLine(Area(5.5, 6.5)); //calling area function for rectangle with double

    }
    public static string Area(int value1) //computing area of square
    {
        return String.Format("Area of Square is {0}", value1 * value1);
    }
    public static double Area(double value1) //computing area of circle
    {
        return 3.14 * Math.Pow(value1,2); //using Pow to calculate the power of 2 of value1
    }
    public static string Area(double value1, double value2) //computing area of rectangle
    {
        return String.Format("Area of Rectangle is {0}", value1 * value2);
    }
}
```

Computing Area of Shapes

In the above code:

- In **line 10** the `Area` *method* will accept one `int` type *argument*, compute the *area* of **square** using that value and will return it in a **string**. If we call the *method* with an **integer**(say **5**) the output will be **"Area of Square is 25"**.

- In line **14** the `Area` method is used for finding the **area** of a **circle**, it will accept a `double` **value**(*radius*) and *return* another `double` value as its `Area`.
    - The function used the inbuilt `Pow` function to calculate the `square` of *radius*, `val1`.
    - `Pow` function takes **two** *arguments*. *First*, is the value whose *power* needs to be computed and *second* is the *power* to raise the value by.

- In line **18** the `Area` method is used for finding the **area** of a **rectangle**. We pass **two** `double` values to it and the output will be the *product* of these **two**, also be of type `double`.

- As can be seen from lines **6**, **7**, **8** and the *output* on console, each of these *methods* above can be called normally without conflict - the compiler will examine the *parameters* of each *method* call to determine which version of `Area` needs to be used.

> **Note:** The *return* type alone cannot differentiate between two *methods*.

If we had two *definitions* for `Area` that had the same *parameters*, like so:

```
// This code will NOT compile.

using System;
class MethodOverloadingExample
{
    static void Main()
    {
        Console.WriteLine(Area(3.5,3.5)); //calling area function for square with double type
        Console.WriteLine("Area of Rectangle is {0}", Area(5.5, 6.5)); //calling area functic
    }
    public static string Area(double value1, double value2) //computing area of square
    {
        return String.Format("Area of Square is {0}", value1 * value1);
    }
    public static double Area(double value1, double value2) //computing area of rectangle
    {
        return value1 * value2;
    }
}
```

In the code above, only the **return** types are different while the rest is same in the **two** *methods* defined hence the compiler will not be able to differentiate between the two and will give a **“method already defined”** error.

In the next lesson, we’ll discuss the concepts of *recursion* in C#.