Methods

Software Craftsmanship Guild

Coding Bootcamp

Lesson Goals

- Learn about how to create and call methods, as well as how they behave

What is a Method?

- A method is a block of code that can be called by other functions.
- When it is called, control of the program is passed to it, it performs all of the statements in its block, and then it returns control back to the location it was called from.

Why Do We Use Methods?

1. Reduce code duplication
2. Organize complex problems into small steps
3. Improve the readability of code

The Parts of a Method

1. **Access Level —** Who can call the method?
2. **Return Type —** Does the method return any data? If so, what kind of data?
3. **Name —** What name should other code blocks use to run the method?
4. **Parameters —** What data needs to be given to the method to do its job?
5. **Code Block —** What statements should be run when the method is called?

The first 4 parts make up the *method signature*.

A Simple Method

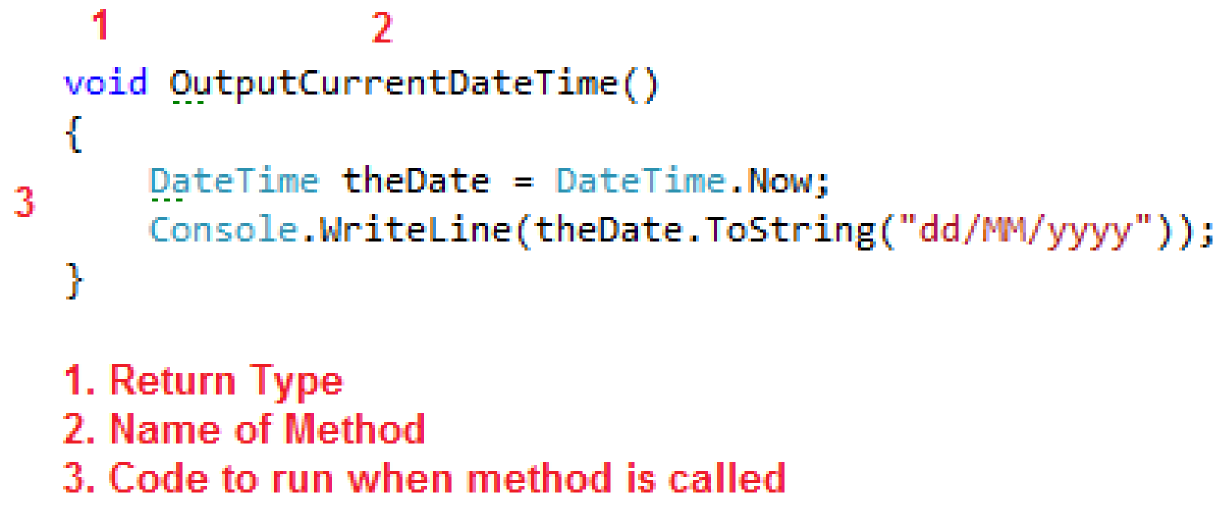
The simplest method is one that has no return data and no parameters.

If a method has no return value, then its type is set to *void*.

Methods must always have parenthesis after the name to contain parameters it is passed. An empty set of parentheses means no parameters (data) is passed into the method for use.

Example: Write Current Date to Console

Here, we are going to write the current date to the console.



```
void OutputCurrentDateTime()
{
    DateTime theDate = DateTime.Now;
    Console.WriteLine(theDate.ToString("dd/MM/yyyy"));
}
```

1. Return Type
2. Name of Method
3. Code to run when method is called

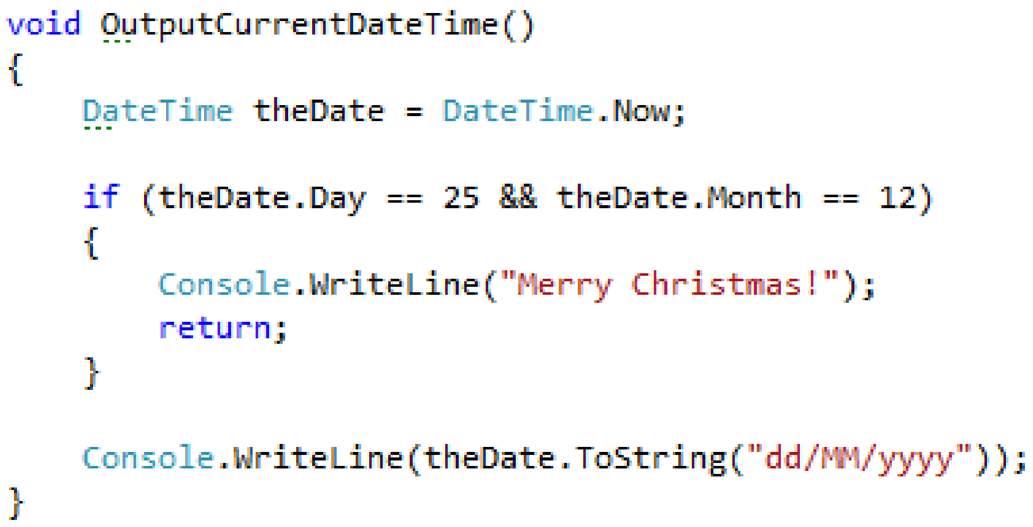
Returning From a Method

In the prior example, the method returns after the code is executed, so no action is required from us.

Sometimes, we want to return early from a method if a condition is met. For this we can use the *return* statement.

In our new code, if it is December 25th, the method will write Merry Christmas to the console, and the return statement will exit the code block, so the dd/MM/yyyy statement will not be executed.

We can have as many return statements as we want in a method. When the work is done, we should return.



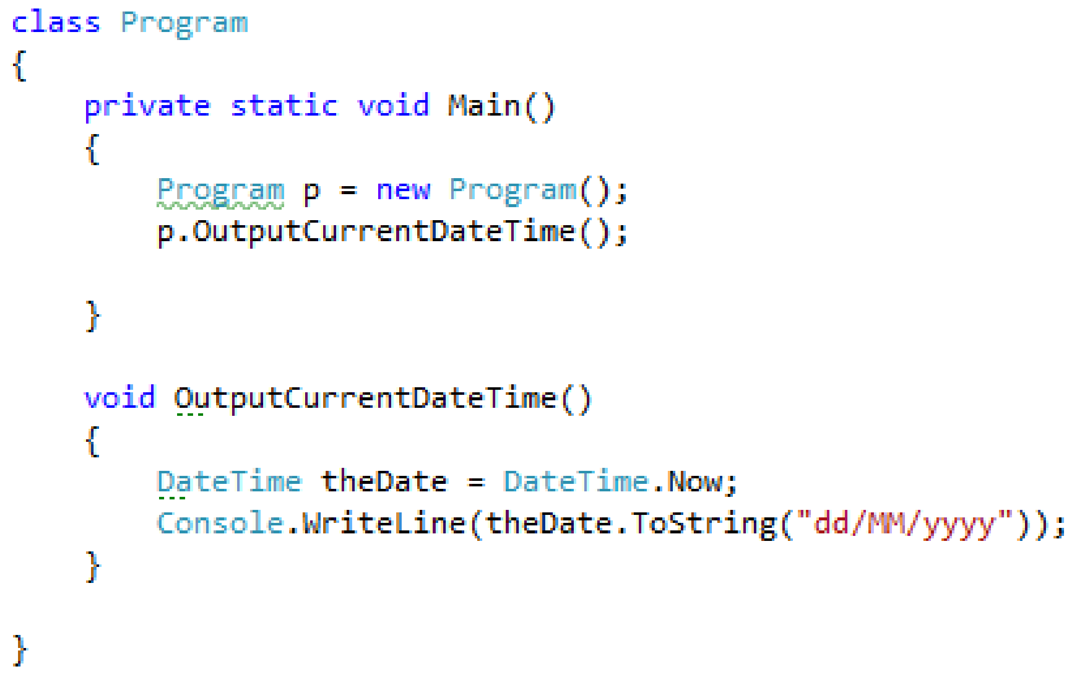
```
void OutputCurrentDateTime()
{
    DateTime theDate = DateTime.Now;

    if (theDate.Day == 25 && theDate.Month == 12)
    {
        Console.WriteLine("Merry Christmas!");
        return;
    }

    Console.WriteLine(theDate.ToString("dd/MM/yyyy"));
}
```

Calling a Method

The purpose of a method is to be called from other code blocks. Methods are called by name from the class they are in.

Here, because the method is not static, we must have an *instance* of the Program class to use it. We create instances of classes with the new keyword.

In programmer-speak: “The main method has called the OutputCurrentDateTime method of the class Program.”



```
class Program
{
    private static void Main()
    {
        Program p = new Program();
        p.OutputCurrentDateTime();

    }

    void OutputCurrentDateTime()
    {
        DateTime theDate = DateTime.Now;
        Console.WriteLine(theDate.ToString("dd/MM/yyyy"));
    }

}
```

Creating a Method that Returns a Value

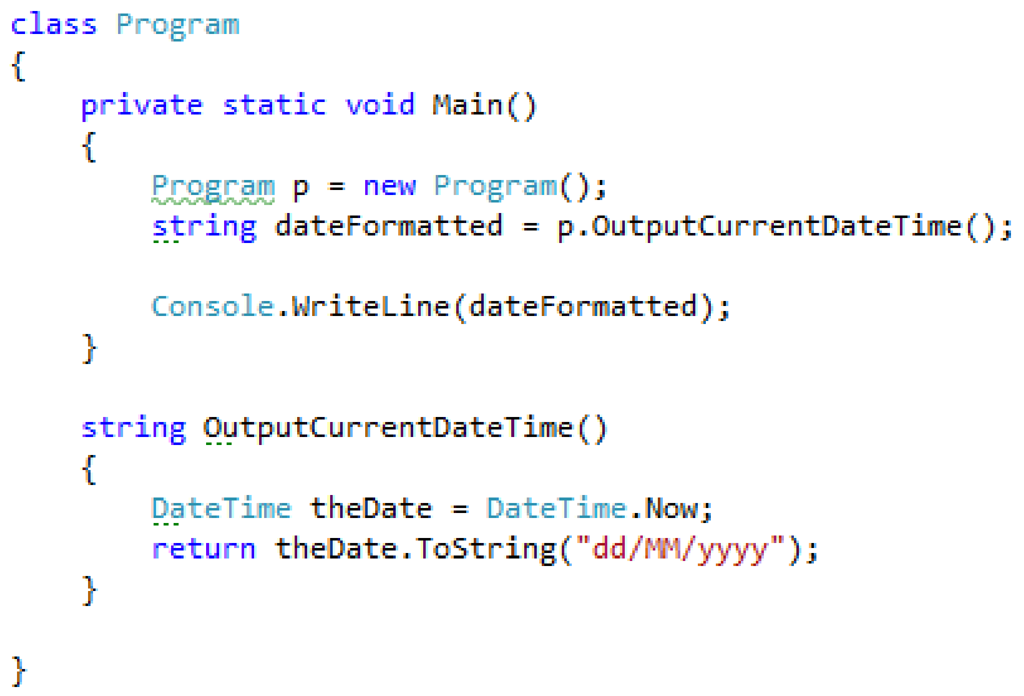
A method can return any value type (int, decimal, string, etc.) or reference type (class, array, struct, etc.) as a return value.

Let's change our method to return the formatted date as a string to the caller

Whenever we see a method call in the code that has a return value, imagine the data that comes out of the method being placed in that spot, so:

string dateFormatted = "01/05/2013";

Methods that return values execute and put the value in place where the method was called.



```
class Program
{
    private static void Main()
    {
        Program p = new Program();
        string dateFormatted = p.OutputCurrentDateTime();

        Console.WriteLine(dateFormatted);
    }

    string OutputCurrentDateTime()
    {
        DateTime theDate = DateTime.Now;
        return theDate.ToString("dd/MM/yyyy");
    }

}
```

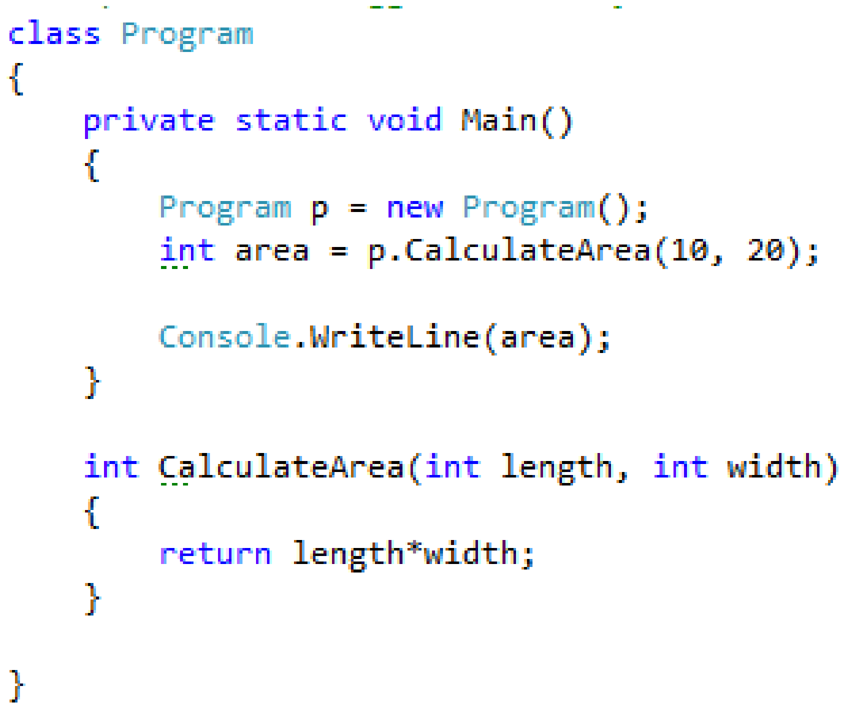
Adding Parameters

So far, our methods have only preformed a single task that does not change. By adding parameters, we can pass data into our method from the calling method.

Any number of parameters (sometimes called *arguments*) can be passed to a method separated by commas.

Here is an example of a method to calculate the area of a rectangle. The calling code will provide the length and width and the method will return an integer of the calculated height.

In programmer-speak we “can pass two arguments to CalculateArea, which will return an integer.”



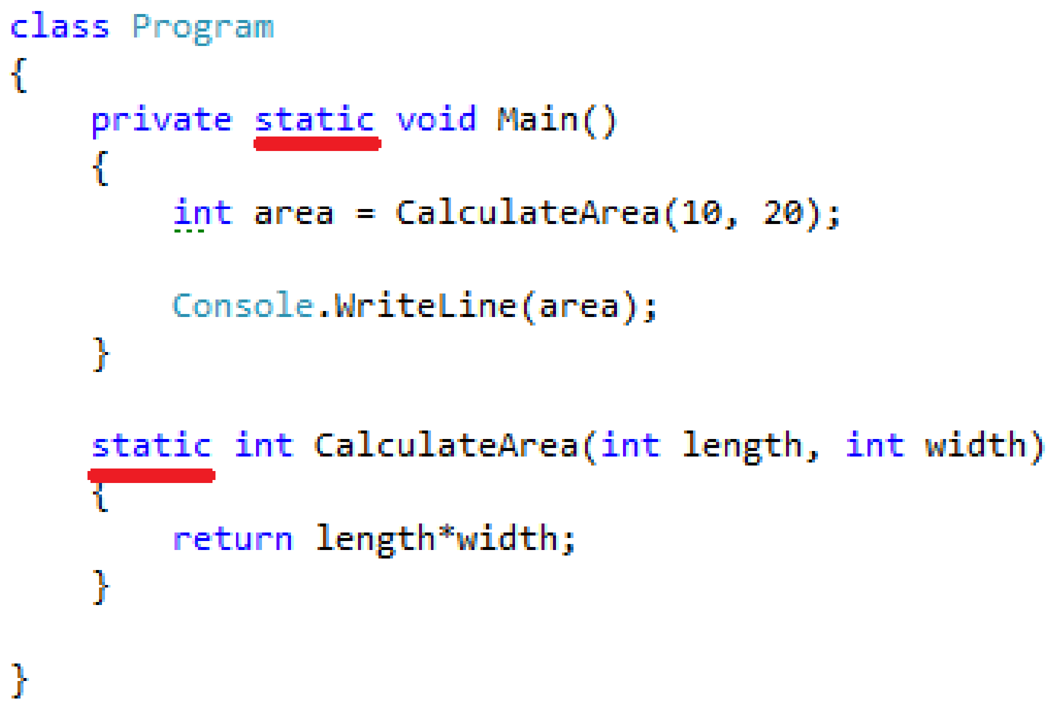
```
class Program
{
    private static void Main()
    {
        Program p = new Program();
        int area = p.CalculateArea(10, 20);

        Console.WriteLine(area);
    }

    int CalculateArea(int length, int width)
    {
        return length*width;
    }

}
```

Static Methods

- In the previous examples, we had to create a new instance of the class Program so that we could call our method.
- The reason for this is because the Main() method is marked with the *static* keyword. Static means that you don’t need an instance of a class to use the static member.
- When static is used, only one copy is ever loaded into memory. Normally, we can create many copies of a class.
- The repercussion of this is that if a class has both static and non-static members, the static members can not call the non-static ones (because they are separated in memory and can’t determine which instance to talk to).

Static Methods

If we make the CalculateArea() method static, we can now call it from Main() without instantiating a new program class.



```
class Program
{
    private static void Main()
    {
        int area = CalculateArea(10, 20);

        Console.WriteLine(area);
    }

    static int CalculateArea(int length, int width)
    {
        return length*width;
    }

}
```

Value Parameters

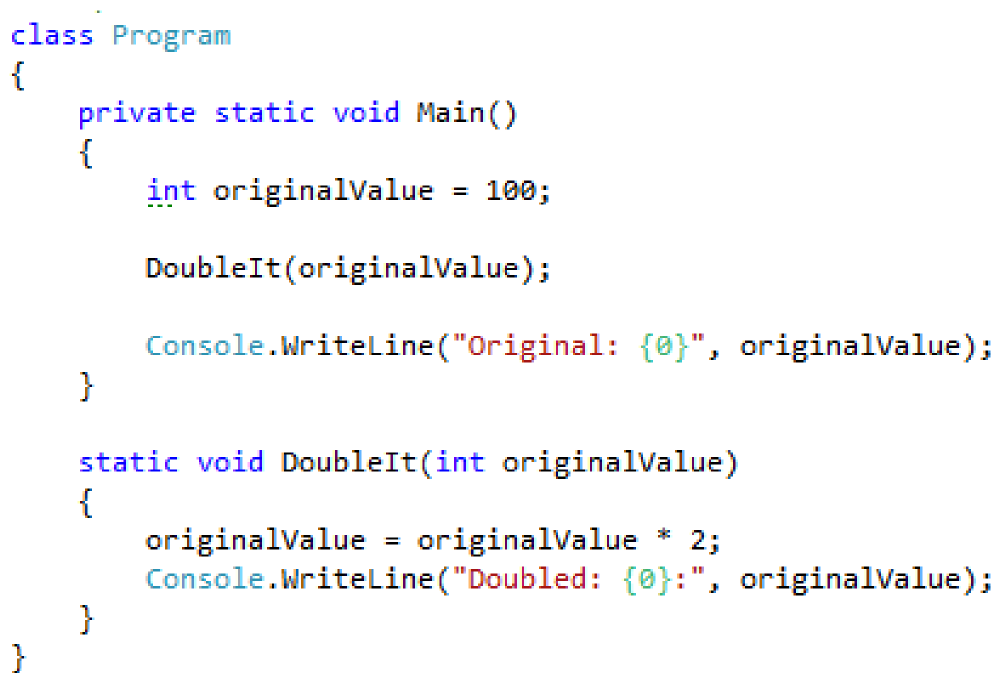
When we pass parameters that are value types by default, they are *passed by value*. This means that a **copy** of the value is created and passed into the method.

If the value passed into the method from the calling code (*external value*) changes inside the method, the original value will **not be changed**. It is a copy and, therefore, is in a different memory location.

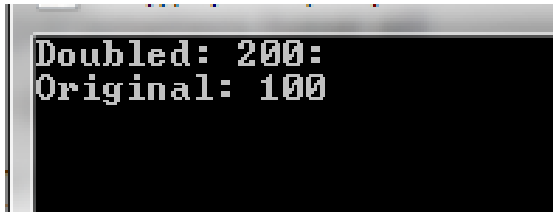
Value Parameters Example

The code to the right demonstrates the behavior of value parameters passed by value.

The original value is created and the data 100 is stored in it. When the DoubleIt() method is called, the data 100 is copied into a new variable… which also happens to be called originalValue. The variable's data is doubled and printed, then control is returned to the Main method, where the other variable's data is still 100.



```
class Program
{
    private static void Main()
    {
        int originalValue = 100;

        DoubleIt(originalValue);

        Console.WriteLine("Original: {0}", originalValue);
    }

    static void DoubleIt(int originalValue)
    {
        originalValue = originalValue * 2;
        Console.WriteLine("Doubled: {0}:", originalValue);
    }
}
```

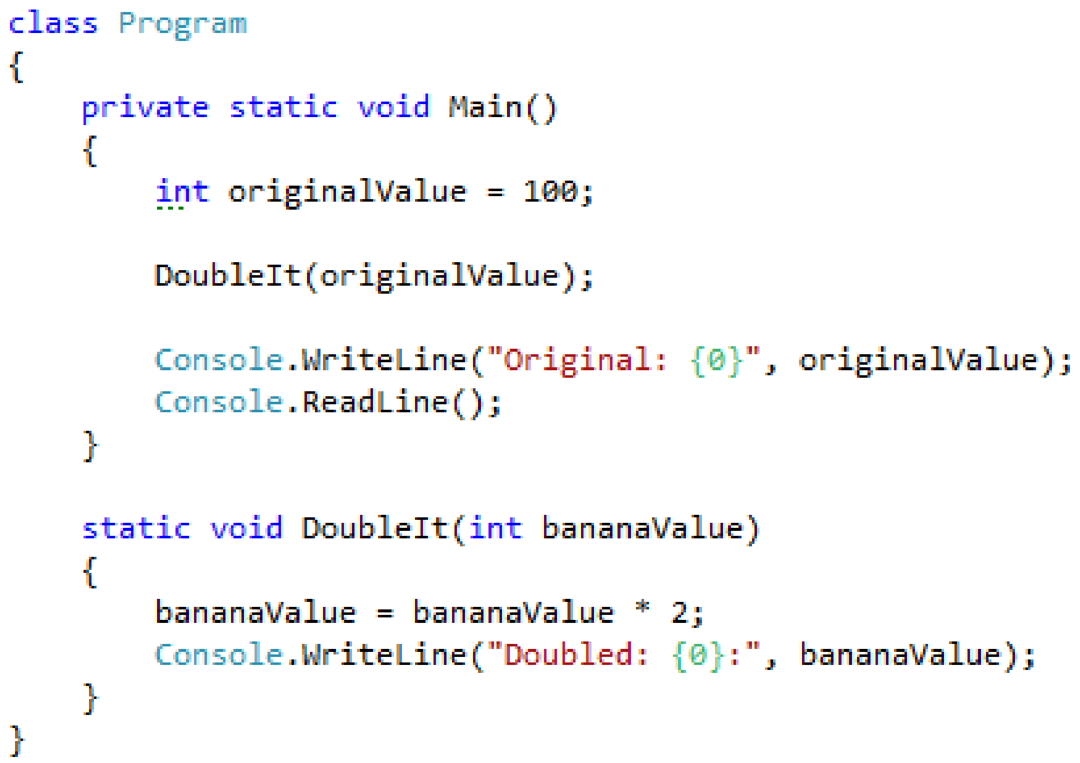


```
Doubled: 200:
Original: 100
```

It is Important to Note

Method parameter variable names **are completely separate from calling variable names.** Oftentimes, developers will name them the same for clarity, but that's not required.

Notice how I can pass originalValue to DoubleIt and have the method's local variable be named something ridiculous like bananaValue.



```
class Program
{
    private static void Main()
    {
        int originalValue = 100;

        DoubleIt(originalValue);

        Console.WriteLine("Original: {0}", originalValue);
        Console.ReadLine();
    }

    static void DoubleIt(int bananaValue)
    {
        bananaValue = bananaValue * 2;
        Console.WriteLine("Doubled: {0}:", bananaValue);
    }
}
```

Reference Parameters

When we pass a reference type by value, a copy of the value is not made. Instead, only a pointer to the value in memory is passed.

This means that any changes made to a reference type inside a method passed by value will impact all references to that value

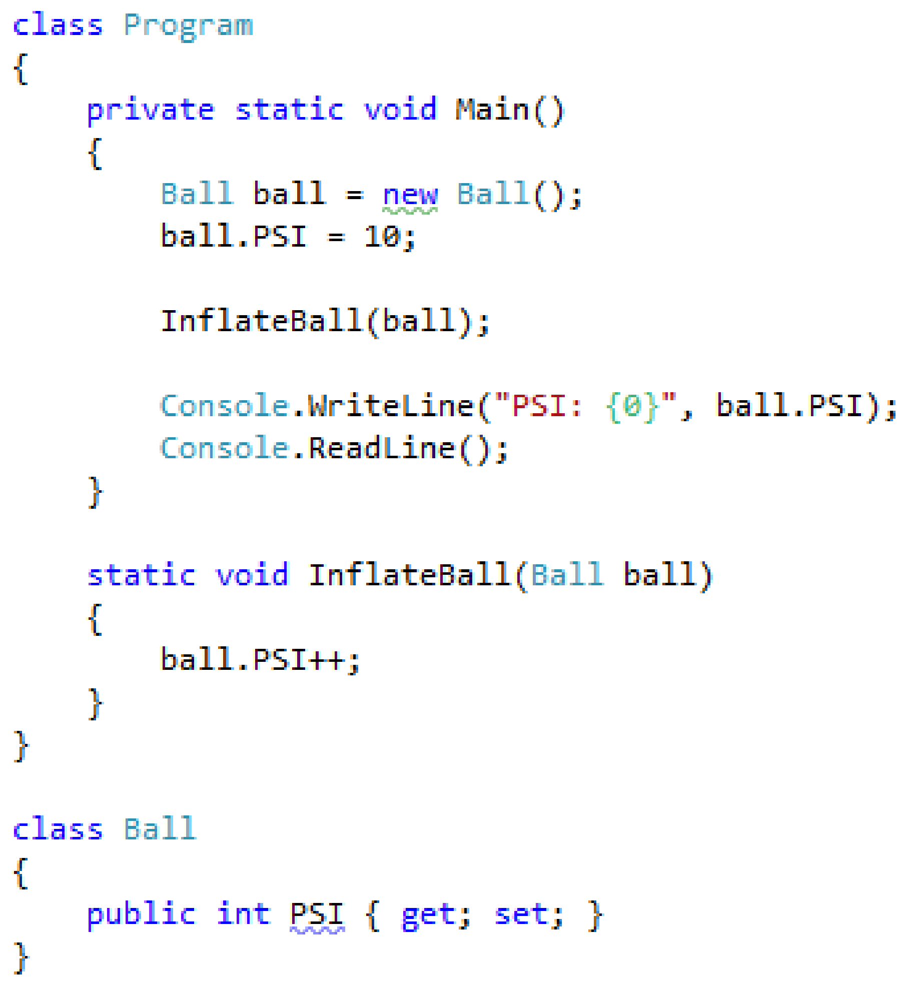
Reference Type By Value Example

Here we create an instance of the Ball class, which is a reference type.

We set the current PSI to 10, then call the InflateBall method, passing it the ball instance variable.

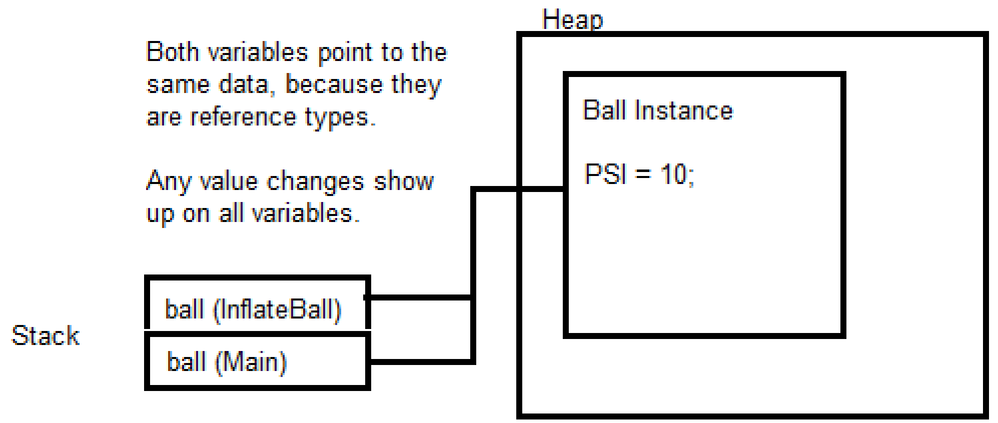
Because Ball is a class, and therefore a reference type, only the memory pointer to the real value is passed. So, when we increment the PSI, it does so for that value which is referenced by multiple pointers.

Thus all pointers, referencing the same value, will return 11 for PSI.



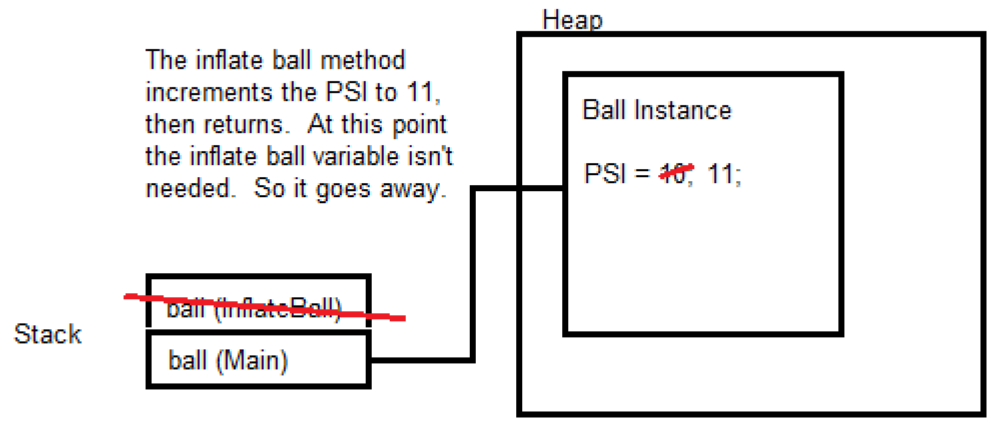
```
class Program
{
    private static void Main()
    {
        Ball ball = new Ball();
        ball.PSI = 10;

        InflateBall(ball);

        Console.WriteLine("PSI: {0}", ball.PSI);
        Console.ReadLine();
    }

    static void InflateBall(Ball ball)
    {
        ball.PSI++;
    }
}

class Ball
{
    public int PSI { get; set; }
}
```

Reference Type Parameter Illustrated



Both variables point to the same data, because they are reference types.

Any value changes show up on all variables.



The inflate ball method increments the PSI to 11, then returns. At this point the inflate ball variable isn't needed. So it goes away.

The ball variable in Main is still pointing to the value, so it prints 11.

```
private static void Main()
{
    Ball ball = new Ball();
    ball.PSI = 10;

    InflateBall(ball);

    Console.WriteLine("PSI: {0}", ball.PSI);
    Console.ReadLine();
}

static void InflateBall(Ball ball)
{
    ball.PSI++;
}
```

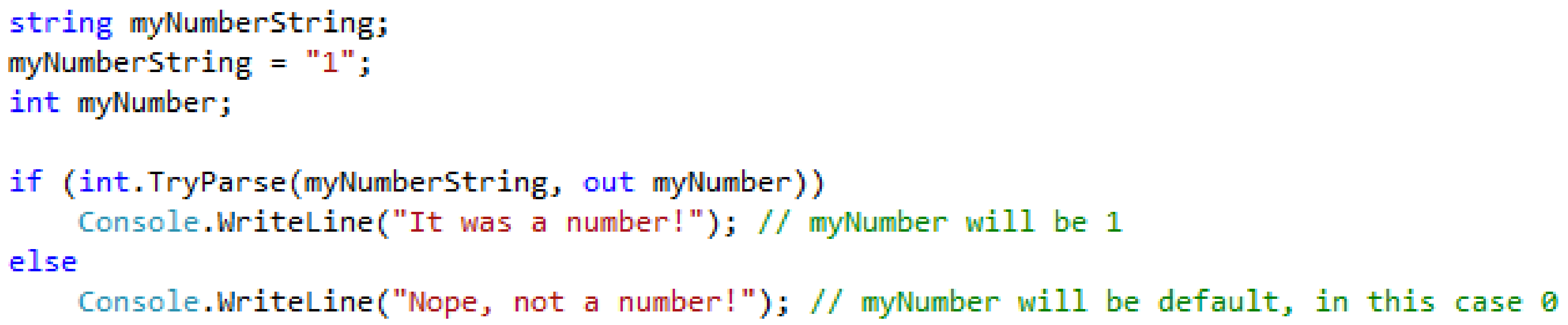
Output Parameters

- Output parameters can pass data from inside the method back to the calling code.
  - They cannot take in a value.
  - They are uninitialized, so you must assign a value in the method before using them.
  - Unlike the return statement, which can only return a single type, output parameters let you return multiple values.
- In practice, these aren’t used often. The built-in TryParse function is the most common example.
  - Many developers prefer to return class objects if more than one value needs to come out of a method.

Int.TryParse as an Example

- Returns true/false, indicating a successful conversion.
- Assigns the converted value to the out parameter if successful; otherwise, it remains the default for the type.
- Signature is:

  bool TryParse(string input, out int result)

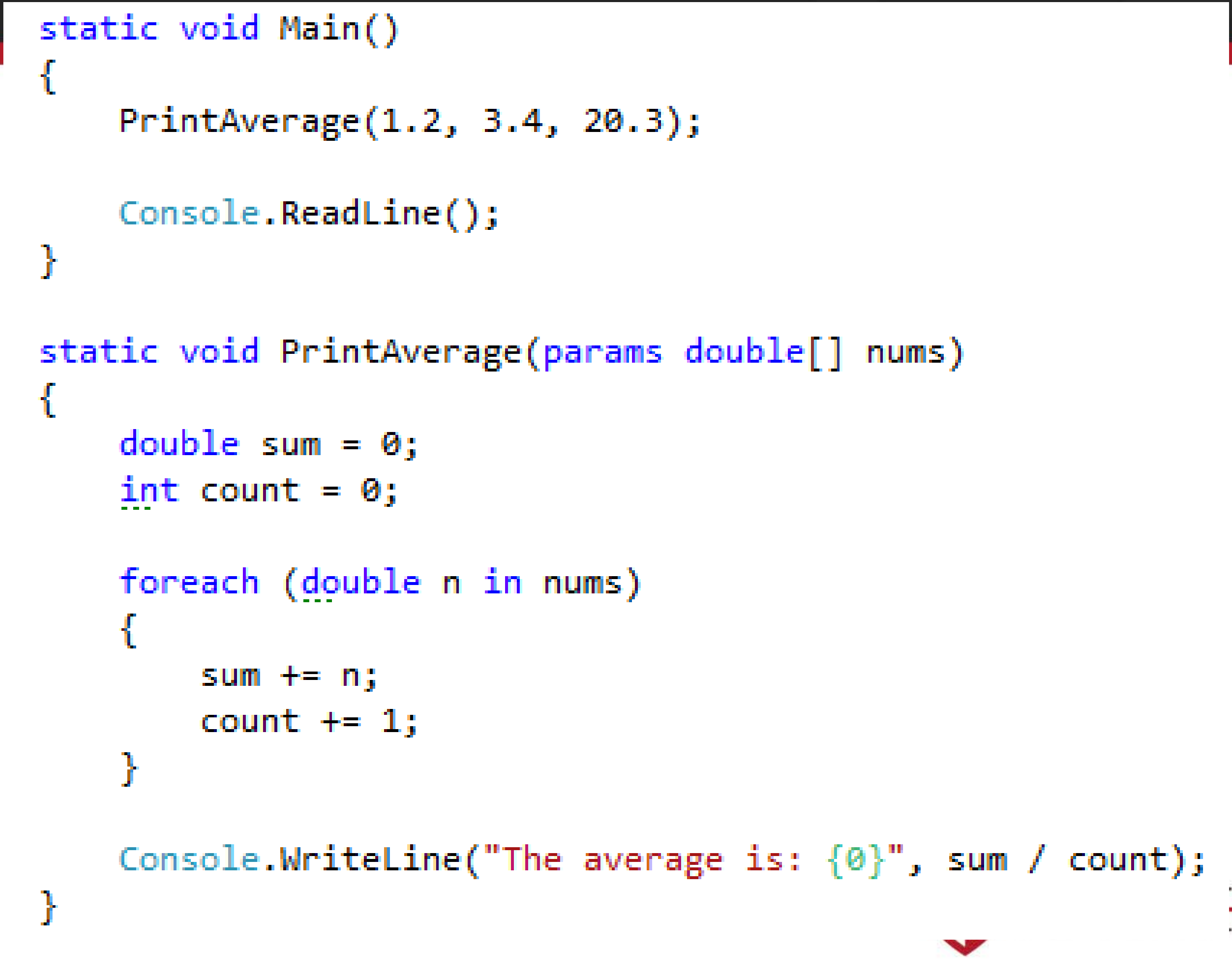


```
string myNumberString;
myNumberString = "1";
int myNumber;

if (int.TryParse(myNumberString, out myNumber))
    Console.WriteLine("It was a number!"); // myNumber will be 1
else
    Console.WriteLine("Nope, not a number!"); // myNumber will be default, in this case 0
```

Parameter Arrays

- Allow for zero to many parameters of the same type to be added to a method
- Must be the last parameter in the method signature
- Substitution Strings {0}, {1}, etc. are an example of this in practice
- Usage requires the params keyword

Parameter Arrays by Example



```
static void Main()
{
    PrintAverage(1.2, 3.4, 20.3);

    Console.ReadLine();
}

static void PrintAverage(params double[] nums)
{
    double sum = 0;
    int count = 0;

    foreach (double n in nums)
    {
        sum += n;
        count += 1;
    }

    Console.WriteLine("The average is: {0}", sum / count);
}
```

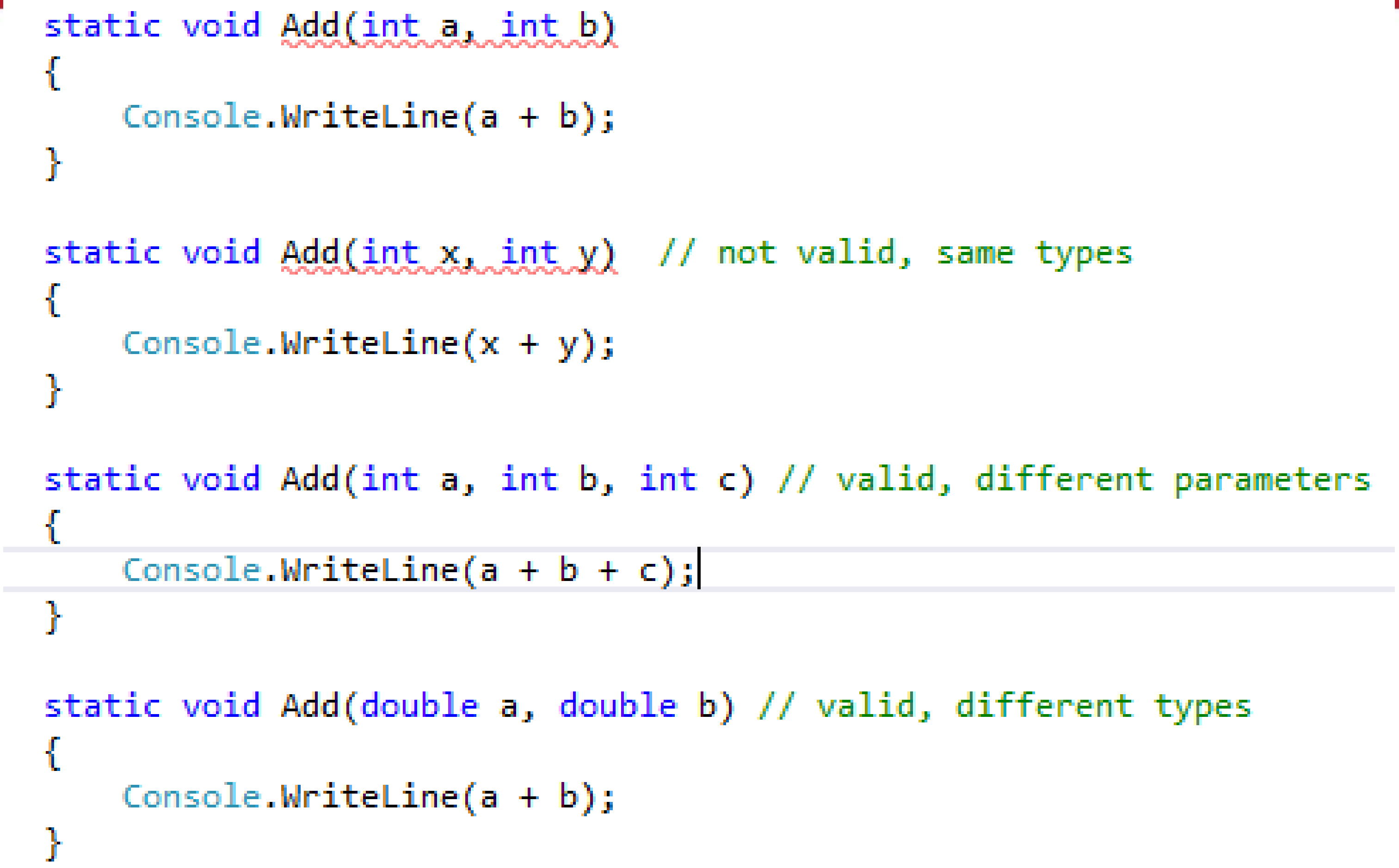
How Does it Work?

- The compiler examines the number of inputs and creates an array on the heap
- It stores a reference to the array on the stack
- If there are no actual parameters, the compiler creates a length 0 array to use
- If the array is a value type, the values are copied and can't be changed. If it is a reference type, they can be changed.

Method Overloading

- You may reuse an existing method name as long as the signatures are different.
- For the compiler, a signature difference is the name plus the type and order of the parameters.
  - The name of the parameters is ignored; only the types are important.

Overloading by Example

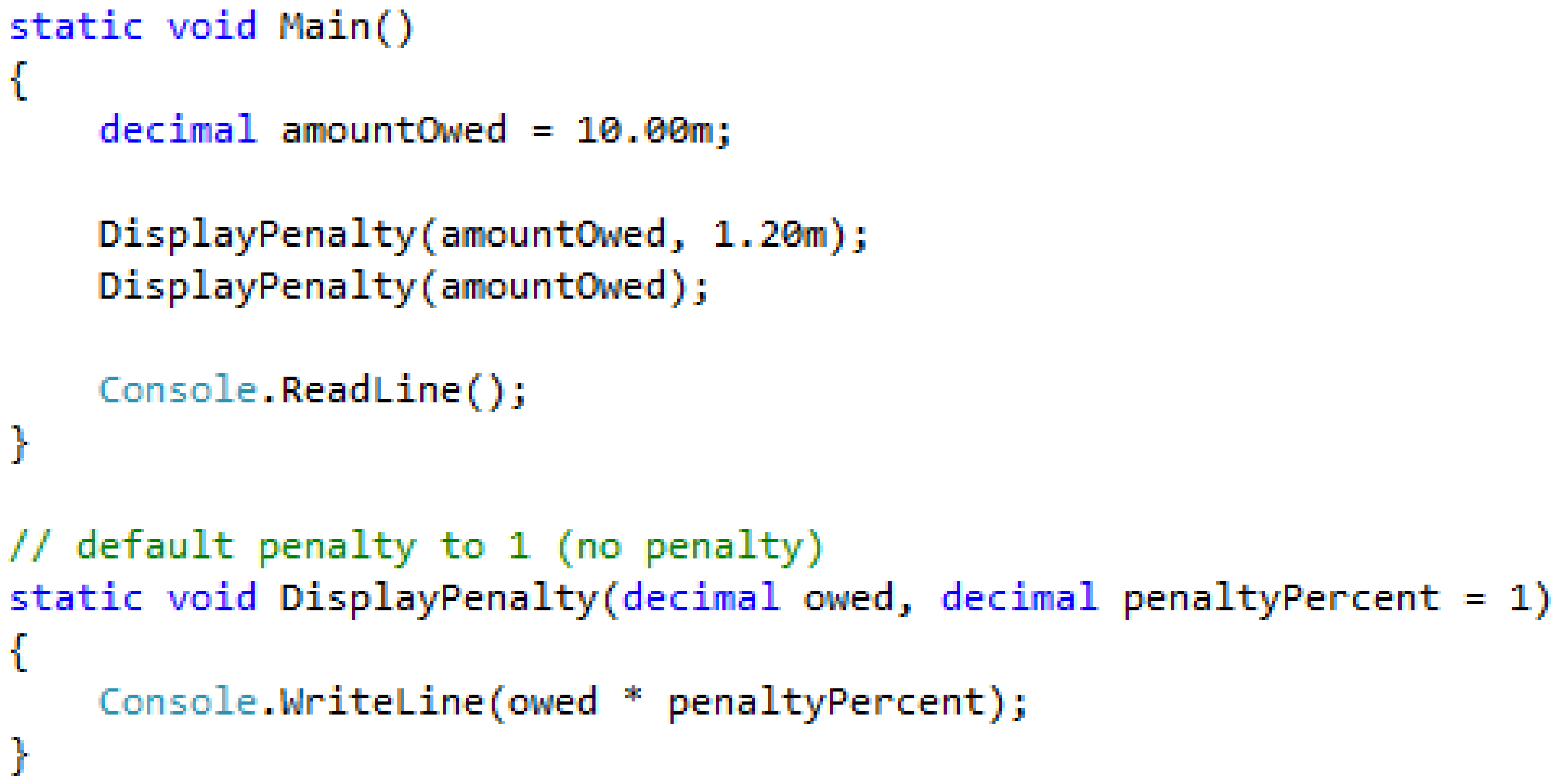


```
static void Add(int a, int b)
{
    Console.WriteLine(a + b);
}

static void Add(int x, int y)  // not valid, same types
{
    Console.WriteLine(x + y);
}

static void Add(int a, int b, int c) // valid, different parameters
{
    Console.WriteLine(a + b + c);
}

static void Add(double a, double b) // valid, different types
{
    Console.WriteLine(a + b);
}
```

Optional Parameters

- By specifying a default value in the signature, you can make a parameter optional.
- This is a newer feature in C#. Some developers don’t like it for stylistic reasons and prefer overloading instead. Ask your teammates if they care.

Optional Parameter Example



```
static void Main()
{
    decimal amountOwed = 10.00m;

    DisplayPenalty(amountOwed, 1.20m);
    DisplayPenalty(amountOwed);

    Console.ReadLine();
}

// default penalty to 1 (no penalty)
static void DisplayPenalty(decimal owed, decimal penaltyPercent = 1)
{
    Console.WriteLine(owed * penaltyPercent);
}
```

Conclusion

- Review these slides a lot because a solid understanding of methods is critical to your success as a developer.
- Methods define *reusable blocks of code* that can *return data* and take in *parameters* to perform tasks.
- Methods are the verbs of our world.
- Good method naming and organization is a key component of clean code.

SOFTWARE GUILD