**Functions**

A function allows you to encapsulate a piece of code and call it from other parts of your code. You may very soon run into a situation where you need to repeat a piece of code, from multiple places, and this is where functions come in. In C#, they are basically declared like this:

<visibility> <return type> <name>(<parameters>)

{

<function code>

}

To call a function, you simply write its name, an open parenthesis, then parameters, if any, and then a closing parenthesis, like this:

DoStuff();

Here is an example of our DoStuff() function:

public void DoStuff()

{

Console.WriteLine("I'm doing something...");

}

The first part, public, is the visibility, and is optional. If you don't define any, then the function will be private. More about that later on. 

Next is the type to return. It could be any valid type in C#, or as we have done it here, void. A void means that this function returns absolutely nothing. Also, this function takes no parameters, as you can see from the empty set of parentheses, so it's actually just a tad bit boring. Let's change that:

public int AddNumbers(int number1, int number2)

{

int result = number1 + number2;

return result;

}

We've changed almost everything. The function now returns an integer, it takes two parameters (both integers), and instead of outputting something, it makes a calculation and then returns the result. This means that we can add two numbers from various places in our code, simply by calling this function, instead of having to write the calculation code each time. While we don't save that much time and effort in this small example, you better believe that you will learn to love functions, the more you use C#. This function is called like this:

int result = AddNumbers(10, 5);

Console.WriteLine(result);

As mentioned, this function actually returns something, and it has to, because we told C# that it's supposed to do so. When declaring anything else than void as a return type, we are forcing our self to return something. You can try removing the return line from the example above, and see the compiler complain:   
  
'AddNumbers(int, int)': not all code paths return a value 

The compiler is reminding us that we have a function which doesn't return something, although we promised. And the compiler is pretty clever! Instead of removing the line, try something like this:

public int AddNumbers(int number1, int number2)

{

int result = number1 + number2;

if(result > 10)

{

return result;

}

}

You will see the exact same error - but why? Because there is no guarantee that our if statement will evaluate to true and the return line being executed. You can solve this by having a second, default like return statement in the end:

public int AddNumbers(int number1, int number2)

{

int result = number1 + number2;

if(result > 10)

{

return result;

}

return 0;

}

This will fix the problem we created for our self, and it will also show you that we can have more than one return statement in our function. As soon as a return statement is reached, the function is left and no more code in it is executed. In this case, it means that as long as the result is higher than 10, the "return 0" is never reached.

**Function parameters**

While parameters are very simple and straight forward to use, there are tricks which can make them a lot more powerful.   
  
The first thing that we will take a look at, is the out and ref modifiers. C#, and other languages as well, differ between two parameters: "by value" and "by reference". The default in C# is "by value", which basically means that when you pass on a variable to a function call, you are actually sending a copy of the object, instead of a reference to it. This also means that you can make changes to the parameter from inside the function, without affecting the original object you passed as a parameter.   
  
With the ref and the out keyword, we can change this behavior, so we pass along a reference to the object instead of its value.

**The ref modifier**

Consider the following example:

static void Main(string[] args)

{

int number = 20;

AddFive(number);

Console.WriteLine(number);

Console.ReadKey();

}

static void AddFive(int number)

{

number = number + 5;

}

We create an integer, assign the number 20 to it, and then we use the AddFive() method, which should add 5 to the number. But does it? No. The value we assign to number inside the function, is never carried out of the function, because we have passed a copy of the number value instead of a reference to it. This is simply how C# works, and in a lot of cases, it's the preferred result. However, in this case, we actually wish to modify the number inside our function. Enter the ref keyword:

static void Main(string[] args)

{

int number = 20;

AddFive(ref number);

Console.WriteLine(number);

Console.ReadKey();

}

static void AddFive(ref int number)

{

number = number + 5;

}

As you can see, all we've done is adding the ref keyword to the function declaration as well as the call to the function. If you run the program now, you will see that the value of number has now changed, once we return from the function call. 

**The out modifier**

The out modifier works pretty much like the ref modifier. They both ensure that the parameter is passed by reference instead of by value, but they do come with two important differences: A value passed to a ref modifier has to be initialized before calling the method - this is not true for the out modifier, where you can use un-initialized values. On the other hand, you can't leave a function call with an out parameter, without assigning a value to it. Since you can pass in un-initialized values as an out parameter, you are not able to actually use an out parameter inside a function - you can only assign a new value to it.   
  
Whether to use out or ref really depends on the situation, as you will realize once you start using them. Both are typically used to work around the issue of only being able to return one value from a function, with C#.   
  
Using the out modifier is just like using the ref modifier, as shown above. Simply change the ref keyword to the out keyword.

**The params modifier**

So far, all of our functions have accepted a fixed amount of parameters. However, in some cases, you might need a function which takes an arbitrary number of parameters. This could of course be done by accepting an array or a list as a parameter, like this:

static void GreetPersons(string[] names) { }

However, calling it would be a bit clumsy. In the shortest form, it would look like this:

GreetPersons(new string[] { "John", "Jane", "Tarzan" });

It is acceptable, but it can be done even smarter, with the params keyword:

static void GreetPersons(params string[] names) { }

Calling it would then look like this:

GreetPersons("John", "Jane", "Tarzan");

Another advantage of using the params approach, is that you are allowed to pass zero parameters to it as well.

Functions with params can even take other parameters as well, as long as the parameter with the params keyword are the last one. Besides that, only one parameter using the params keyword can be used per function. Here is a last and more complete example:

static void Main(string[] args)

{

GreetPersons(0);

GreetPersons(25, "John", "Jane", "Tarzan");

Console.ReadKey();

}

static void GreetPersons(int someUnusedParameter, params string[] names)

{

foreach(string name in names)

Console.WriteLine("Hello, " + name);

}