Methods

Two of the hardest things for beginning programmers are learning how to divide a program into separate methods and then learning how to group those methods into classes.

Using Main

So far we have put all of our code in the Main method. This works while the code is simple and only a few lines in length, but is not useful for longer more complex code. The Main method should be used exclusively to start the program. It can do this by calling a method or class that will begin the execution.

Separating Functionality

Early programming languages consisted of on long sequence of commands. This worked fine when there were only a few hundred or even a couple thousand lines of code, but became utterly untenable when the programs started to have tens of thousands of lines of code. For one thing, it was almost impossible to figure out where something went wrong. It could take hours of scanning lines to find the source of a bug. For another, programmer's either had to write the same pieces of code over and over again or use GO TO commands to return to an earlier piece of code that did the thing they wanted. Using GO TO, however, made tracing the sequence of commands difficult or impossible. The resulting code was called spaghetti code, because it was as tangled as a plate full of pasta.

So programming language developers decided, why don't we allow the code to be broken up into functions or methods. Each method could be responsible for one thing. That way if something goes wrong we know where to find it. Also it means that the same code could be used over and over again as needed by just calling the relevant method.

The principle of breaking the code into methods and having each method responsible for one thing is called structured programming.

Structured programming ran into some problems later when programs began to contain thousands of methods and be written by dozens of people. The solution to that was Object Oriented Programming, which we will talk more about later.

The first difficulty for a new programmer is what exactly does "one thing" mean? How do you divide a program into appropriate methods?

First, "one thing" doesn't mean necessarily that each method should contain one command. The one thing can be a logical group of commands. For instance in a simple program, you might have one method to gather input from the user, another method to process the input, and a third method to display the output. Most programs are more complex than that, but that can give you an idea of where to begin.

We will look at some examples below.

Method signatures

A method signature consists of several parts. There is the access, the return type, the name of the method and any parameters in parentheses.

ACCESS RETURNTYPE NAME (Param1, param2, etc.)

Look at the Main method again:

```
static void Main(string[] args)
{
}
```

Static is it access. As explained earlier, Static means that the method is loaded into RAM memory as soon as the program is started and stays there throughout the running of the program. It is accessible everywhere in the program.

Some other access modifiers are:

private	Can only be seen by other methods in the current class
public	Visible to all other running classes
protected	Visible to any class in the current assembly (all the classes that will be compiled together into the current program)

Method return types

The main method has a return type of void. This means it doesn't return any value. It just executes its commands and is done. This can be a little confusing at first. What does it mean to "return" a value or not return a value? Consider this example program.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace firstsample
{
    class Program
    {
        static void Main(string[] args)
        {
            Program p = new Program();
            p.GetValue();
        }

        private void GetValue()
        {
            Console.WriteLine("Enter an integer");
            int number = int.Parse(Console.ReadLine());
            int cube = Cube(number);
```

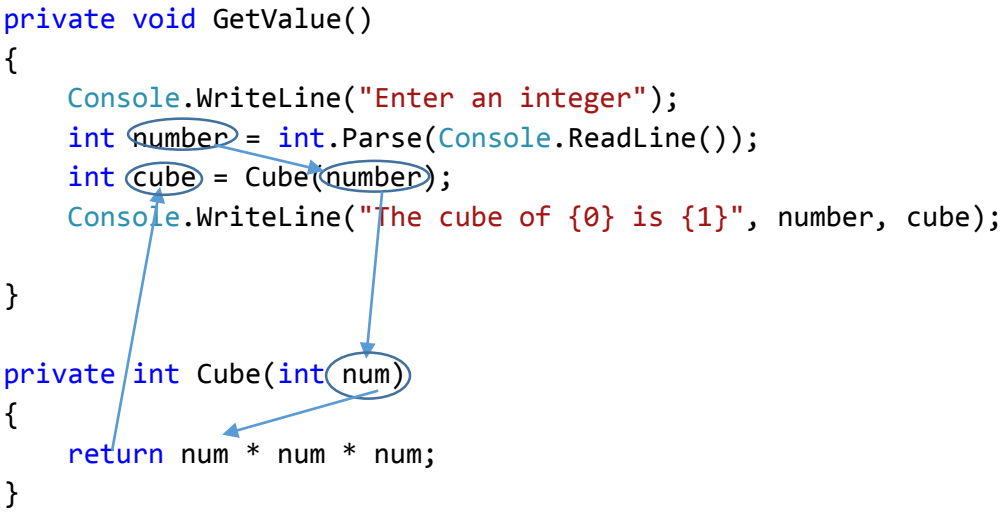
```
            Console.WriteLine("The cube of {0} is {1}", number, cube);

        }

        private int Cube(int num)
        {
            return num * num * num;
        }
    }
}
```

GetValue() is a void method. It prints a prompt to enter an integer and then gets the user's input. Then it calls the method Cube(int). To call a method just name it and provide any parameters it requires. Cube(int) is different from GetValue() in two important ways. It is not void. It returns an integer. Secondly it takes an integer as a parameter.

When Cube(int) is called we assign the results it returns to a variable, and we pass it a number. If the number were 2, for instance, 2 would get copied to the variable num in the Cube method signature. Inside the cube method the 2 would be multiplied by itself twice. The result, 8, would be returned to the place where the method was called and assigned to the variable cube in the GetValue method. The next command, the WriteLine can display the value that was returned.



Methods that have a return type other than void must always use the key word return and return a value of the type specified in the signature. The return is always the last statement in the method. Any code after a return is inaccessible. It can never be executed. The value the method returns is always returned to the place where the method is called. There should be a variable there to store the returned value.

In C# you can return any kind of value including complex values like arrays or objects.

Passing parameters

In the example above, the method Cube takes a parameter of the type integer. Parameters are declared in the parenthesis of a method signature. The method Cube takes an integer as an argument and returns an integer value.

```
private int Cube(int num)
```

When Cube is called in the method GetValue() the programmer must pass an integer to the method. This integer is stored in the variable number which is give its value by the user.

```
int number = int.Parse(Console.ReadLine());
int cube = Cube(number);
```

The value of number is copied to the value of num in the Cube method signature. This is called "passing by value." Simple data types like int, double, decimal, and Boolean are passed this way. Larger objects, such as arrays and classes are "passed by reference." This means instead of making a copy of the value and passing it to the parameter variable, a reference to the memory address of the original value is passed instead. In C# this happens invisibly, behind the scenes, for the most part, though you can make it explicit if you wish by using the key word ref to force it to pass by reference . But it does have some implications. If you pass a variable by value, you can make any changes you wish to the copy and it does not affect the value of the original. It only works on the copy. If you pass a value by references any changes you make to the value also change the original value. This is because when you pass a variable by reference you didn't pass a copy of the value, you passed the memory location of the original.

Variable scope

Variables have a property called scope. This was not particularly important when all your variables and code were written in the Main() method, but becomes very important when you start to break your code into separate methods.

Scope refers to two aspects of a variable: its life and its visibility. If you declare a variable in a method, the variable lives while the method is executing. After the method is done, the variable is disposed of. This means it can't be seen by other methods. Its value is only visible in the method where it is declared. In our example, for instance, "number" and "cube" are only visible to the GetValue() method. They cannot be referred to in the Cube method. Number, of course, gets copied to num. But the Cube() method can only see the copy "num." It can't refer to number directly.

Scope helps control memory use. A variable is only stored in the computer's memory as long as needed to perform its task. It also helps control debugging. If a variable's scope is limited, it is clear where to look if something is going wrong with the variable.

Here is a general rule to judge a variable's scope: The scope of a variable is as large as the curly braces it is declared within. If you declare a variable inside a method, it has method scope. If you declare it in a for loop it has the scope of that loop. If you declare it inside an if statement, it has the scope of that if statement. That means if you declare a variable inside an if statement, it is invisible to the rest of the method.

Here is an example with various levels of scope:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ScopeExamples
{
    class Program
    {
        const double PI = 3.14;
        static void Main(string[] args)
        {
            Program p = new Program();
            p.GetInput();
            p.EndProgram();
        }

        private void GetInput()
        {
            double r;
            Console.WriteLine("Enter the radius of the circle");
            bool isDouble = double.TryParse(Console.ReadLine(), out r);
            if (!isDouble)
            {
                string err = "You must enter a valid number";
                Console.WriteLine(err);
                return;
            }

            double circum=CalculateCircumferance(r);
            DisplayResults(circum);
        }

        private double CalculateCircumferance(double radius)
        {
            return 2*(PI * radius);
        }

        private void DisplayResults(double circumference)
        {
            Console.WriteLine("the circumference is {0}", circumference);
        }

        private void EndProgram()
        {
            Console.WriteLine("Press any key to end");
            Console.ReadKey();
        }

    }
}
```

Let's go through the variables. PI is a constant rather than a variable. It can't be changed in the body of the program. But it does have scope. It is declared in the class block, between the class curly braces, so it has class scope. That means that it is visible to any method in that class. It retains its value as long as the class is active. You can declare variables at the class level also and they too will be visible to all the methods in the class. As you will see when we create our own classes, there are some variables that should have a class level scope. The general rule, however, is that a variable should be given the most limited scope that still allows it to fulfill its purpose. If you think about it, you should be able to see the reasoning behind this rule. A class level variable can be changed by any method in the class. If something goes wrong with the variable, it is more difficult to pinpoint exactly where the problem is occurring. If a variable has only method scope you know much more precisely where to look for the problem.

In the Main() method, p is a variable standing for Program. This is necessary because of the static access of Main().

In the GetInput() method , r is a variable with a type of double as is circum. The variable isDouble is Boolean. These three variables have method scope. They are declared between the braces of GetInput() and can only be seen within the scope of that method.

Note: *the out parameter of the method TryParse() throws the value out of the method rather than taking it in. The TryParse works by testing whether the string variable (such as the one returned by Console.ReadLine()) is a valid number or not. If it is, it returns true to the Boolean variable and assigns the number that was parsed out to variable specified in the out parameter—in this case r. If it is false, it returns false and assigns 0 to the out parameter.*

The string variable err is declared inside the braces of the if statement. Therefore it can't be seen or accessed outside the if statement, even in the same method.

The variables radius and circumference each have method scope, even though they are declared as parameters of the method.

More advanced topics

There are other aspects of methods. Methods can be overloaded or overridden, but we will look at these and other specialized types of methods when we look at creating objects.