[0420] Method Parameters

Pass-By-Value vs Pass-By-Reference Parameters

When you call a method and pass in literal values as arguments, things are easy. If I call *PrintName("Fred")* then I know that the letters F-r-e-d will appear on the display and nothing else will happen (assuming "PrintName()" is coded correctly).

But now consider the following code:

```
string myName = "Fred";
PrintName(myName);
```

Question: What exactly got passed to the *PrintName()* method? Was it a **copy** of the data in the *myName* string (i.e., a string with F-r-e-d inside of it)? or was it the **actual variable** *myName* itself (in which case the *PrintName()* method could potentially reach in and change *myName*'s value from "Fred" to something else)?

Answer: It depends on how PrintName()'s parameters are defined. Unlike Java programmers, C# programmers have the power to do it either way! The first way - where a copy of myName's data is sent - is known as **Call-by-Value** (since you are sending the value of the variable in the call). The second way - where the variable itself is sent and can be changed - is known as **Call-by-Reference**.

For "simple" data types like int, string, double, etc., using call-by-value parameters makes tons of sense and is, indeed, C#'s default behavior. While it is possible to use call-by-reference with these data types, that is generally frowned upon as poor design because you are violating the Structured Programming principles of "**Well-designed methods should only return one value"** and, potentially, **"Well-designed methods should do one thing and one thing only."**

IF YOU REALLY WANT TO USE CALL-BY-REFERENCE WITH SIMPLE DATA TYPES...
(Did I mention that this is a BAD idea? #its-a-bad-idea)
...you can use either the "**out**" or "**ref**" modifier in-front of the parameter in your method signature. See the book for the gory details.

Strangely, things change 180-degrees when we talk about more complex data types like structs and classes. In the case of those data types, call-by-reference is the norm and call-by-value is rarely used. That is also C#'s default behavior. That means that if you pass an object into a method, the method will get access to the actual object itself - not a copy of the object.

Methods with a Variable Number of Parameters

(These things are normally called "Variable Parameters" which is possibly THE most confusing and misleading use of terminology in the history of programming. In this context, "Variable" doesn't mean "data variable" - it means "changeable", "non-fixed." So "Variable Parameters" means "a non-fixed

number of parameters." In other words "A method which can take 1, 2, 3 or even more parameters depending on how it is called.")

Check out the "CalculateAverage()" example in the book. It's not too bad.

Variable Parameters are *rarely* used.

The Big Problem with Method Parameters

The big problem with method parameters is that, if you have a method that takes a large number of parameters, it can get very confusing very quickly as to which argument matches up with which parameter in the call. For example:

```
CreatePerson("Tom", "Smith", 24, 190, 127, 3.4, 4.3, "Indiana", "France", 450.10, 72);
```

"Umm... what?"

Because arguments are normally assigned to parameters by position, methods with lots of parameters can easily become very confusing!

C# has two features (missing from Java atm) that help with this problem: Optional Parameters and Named Parameters.

Optional Parameters

Optional parameters can be left off of the method call. If they are not used in the call, they will get "default" values provided in the method's signature. This helps because often lots of parameters in a long method call are just using default values and don't really need to be specified every time. For example *CreateDice(2);* and *CreateDice(2, 6, "white");* would be identical given the following method signature:

```
void CreateDice(int numDice, int numSides = 6, string color = "white");
```

Note that the optional parameters must come at the END of the method's parameter list.

Named Parameters

Named Parameters attack the problem of long method calls straight-on and, for the most part, solves it. #yay Consider the following:

```
CreateDice(color: "blue", numDice: 2, numSides: 6);
```

Now, even though the parameters are in a completely different order from their order in the method signature, everything works fine. It is also much more obvious which argument value gets assigned to which parameter regardless of how many parameters the method takes.

Note: Named parameters are even more awesome when combined with Optional parameters!
Note #2: In theory, you can combine Named parameters with traditional, unnamed, positional

parameters. DON'T!!