QUICK-START C#

Understanding Parameter Modifiers

For a detailed breakdown on everything here, Microsoft Learn provides excellent resources on this.
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/method-parameters
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types

What are they?

You can skip this. It's just the opening. Head to Params for the real stuff.

In C# there are four common parameter modifiers that you will likely come across, be it through other programmers or the tools you're working with day-to-day. These allow you to change how the method receives the arguments and what the method can do with the parameters. They also affect what the caller will be able to see when the called method changes the values stored in each parameter.

Parameters refer to any value type (int, bool, struct) or reference type (Class, object, string) that represent the values which will be passed into the method when it is called, like aliases for them.

Arguments refer to the values you are assigning to each ***parameter*** of a method.

Why should we use them?

If you're working with a **value type**, then *Optimisation*. If you're working with a **reference type,** it can be to change what the caller and method both see *(changing what the caller's variable is pointing to*). You'll likely find them used with value types most often as it's more performant to pass access of a variable than it is to pass a copy of it for a parameter. Being able to successfully apply these modifiers and justify your reasoning for doing so could be the difference between grade boundaries too.

Our first modifier, **params**, is the quickest to understand. This modifier changes the way our method recieves it's arguments. Instead of passing an array of a defined type, you can pass any sized comma-separated list of that type (e.g. **string**) as the last arguments in the call.

Params

Description:

This modifier allows you to specify any amount of values of the defined object type, which is compiled into an array for the method to iterate over.

Restriction:

Whichever parameter has this type must come last in the list of parameters as there could be any amount of following parameters and the C# compiler can't account for that variance otherwise.

Method Example

```
// Print out a list of names.
private void PrintNames(params string[] names)
{
    for (int i = 0; i < names.Length; i++)
    {
        Debug.Log(names[i]);
    }
}
```

Calling Convention

```
Method("Anna", "Dante", "Merlin");

/* Add as many names as you want in the list.
As long as the parameter with params is the
last parameter in the list, it won't error. */
```

Value Type & Reference Type

When using a parameter modifier, think about '***what type of type*** is the parameter?'. C# defines a few but our focus is on two, **Value Types** and **Reference Types**. For this guide we'll define these types for clarity:

A **Value Type** variable contains it's own data. If multiple variables have the same value and one is changed, only the one that was changed is affected.

When one is a *value* parameter of a method, the argument it represents is copied and the copy is modified by (and local to) the method.

When one is a *reference* parameter, the method is given direct access to the original variable and may be able to modify the data or replace the object, depending on the ***parameter modifier***.

Examples: ***int, bool, float, char, double, byte***

A **Reference Type** variable points to an object with it's data. multiple variables can point to the same object. If the object's state is changed and there are multiple variables pointing to it, all of them can see the change and are affected by it.

When one is a *value* parameter of a method, the reference to the object represents is copied. Any changes made to the object's state are visible to both the method and caller but the method can't change the object the caller's reference points to, only it's own.

When one is a *reference* parameter, the method is given access to the original reference and can change the state of the object being pointed to but also what object of that type is being pointed to.

Examples: ***class, interface, string, object, delegate, array***

Value Parameters are parameters that do not use any modifiers, instead they pass a copy of their value to the method (pass-by-value).
Reference Parameters are parameters that use a modifier and give the method direct access to the variable holding the value (pass-by-reference).

An easy way to remember the difference is with the question: '***Does this object hold all it's data?***'. An example for this is an **array of ints** compared to an **int**. *An array of ints doesn't hold that data in itself*, instead holding an address to the array. *An int* only holds one number and can hold the data without having to point elsewhere.

This document has been written for **University Centre Leeds** - Alexander Gilbertson

In

Description:

More memorable as '**input**', this modifier lets a method access a variable with read-only permissions.

Restriction:

A parameter prefixed with **in** is a read-only reference that cannot be modified and **must already have a value.** Trying overwrite the parameter will cause an error.

Method Example

```
// Print out a name then try to modify it.
private void TryNewName(in string name)
{
    Debug.Log(name); // Print beforehand.
    name = "Thomas"; // ERROR - Can't overwrite.

    Debug.Log(name);
    // If there wasn't an error, this would have
    // printed out the same as the first debug.
}
```

Calling Convention

```
string person = "Geralt";

DefaultName(in person);
```

Out

Description:

More memorable as '**output**', this modifier lets a method access a variable with read-write permissions.

Restriction:

A parameter prefixed with **out** is a read-write reference that does not need to have an initial value when passed into it but **must be assigned to by the method before the method exits.** Failing to do so will cause an error.

Calling Conventions:

In convention **A**, you can define the variable then have it assigned to by the method. In convention **B**, you can have the variable defined as you call the method and it will still be visible to any code below, which is useful for if-statements.

Method Example

```
// Set a default name.
private void DefaultName(out string name)
{
    name = "May";
    // An assignment to the name parameter
    // Must occur before we exit or it errors.
}
```

Calling Convention A: *Create Variable Before Call*

```
string person;
DefaultName(out person); // Will set to "May".

Debug.Log(person); // Will print "May".
```

Calling Convention B: *Create Variable On Call*

```
DefaultName(out string person);

Debug.Log(person); // Will also print "May".
```

Ref

Description:

Memorable as '**reference**', this modifier lets a method access a variable with read-write permissions.

Restriction:

A parameter prefixed with **ref** is a read-write reference that **must already have a value** but can be overwritten, though doesn't need to be by the time the method finishes executing.

Method Example

```
// Print out a name then try to modify it.
private void TryNewName(ref string name)
{
    Debug.Log(name); // Print beforehand.
    name = "Thomas"; // Will work fine this time!
    Debug.Log(name); // Will print "Thomas".
}
```

Calling Convention

```
string person = "Geralt";

TryNewName(ref person);
// Will print "Geralt" then "Thomas".
```

When you might use '**ref**':

You might use **ref** mostly if the method needs read or write to the original object and it may do neither, one or both operations. This can be to optimise modifying **Value Type** variables so that excess copies aren't made or if you have conditions that change what operations are carried out.

When you might use '**in**':

You might use **in** if the method needs to read the original object but not write to it. This can be appropriate if you're using the object as a reference to construct another type of object from or if you do not need the full permissions of the **ref** modifier.

When you might use '**out**':

You might use **out** if the method needs to write to the original object and doesn't need an initial value. This would be appropriate if you are using a variable argument when method call in an if-statement or if you do not need the full permissions of the **ref** modifier.

These aren't the only reasons why or when you might use these modifiers, instead being a few examples of how they could be used. You may find more new reasons to use **reference parameters** over **value parameters** as you explore and experiment with C# too.