Exercise – Parameter Passing in C#

Introduction:

Having an understanding of how value and reference types work is what separates the best C# programmers from the competent ones.

In the lecture for this session it was mentioned that passing a value type using a reference parameter was *not* the same as passing a reference type using a value parameter.

In this exercise, we explore this further by asking the question:

What is the difference between passing a value object by reference, and a reference object by value?

An answer and explanation is provided for you on the last page of this exercise, but you should attempt to answer this question yourself by completing the programming exercises below first.

Exercises:

Consider the following code:

```
using System;
using System.IO;

public struct IntHolder
{
        public int i;
}

class Program
{
        static void Foo(IntHolder x)
        {
                x = new IntHolder();
        }

        static void Main(string[] args)
        {
                IntHolder y = new IntHolder();
                y.i = 5;

                Foo(y);

                Console.WriteLine(y.i);
        }
}
```

1. Is *IntHolder* a value type or a reference type? Value type
2. Is the variable *y* being passed by reference or by value? Value.
3. Without executing this program, what will be printed to the console? 5
4. Execute this program and see what is printed to the console. Were you correct? Yes

Let's now consider the same program, except this time we will add the *ref* keyword to the *Foo()* function declaration and invocation:

```
        static void Foo(ref IntHolder x)
        {
                x = new IntHolder();
        }

        static void Main(string[] args)
        {
                IntHolder y = new IntHolder();
                y.i = 5;

                Foo(ref y);

                Console.WriteLine(y.i);
        }
```

5. In the updated program, is the variable *y* being passed by reference or by value? Reference
6. Without executing this program, what will be printed to the console? 0
7. Execute this program and see what is printed to the console. Were you correct? Yes

Change *IntHolder* to a class and remove the *ref* keyword from the *Foo()* function declaration and invocation:

```
public class IntHolder {
        public int i;
}

class Program {
        static void Foo(IntHolder x) {
                x = new IntHolder();
        }

        static void Main(string[] args) {
                IntHolder y = new IntHolder();
                y.i = 5;

                Foo(y);

                Console.WriteLine(y.i);
        }
}
```

8. In the updated program, is *IntHolder* a value or reference type? Reference
9. Without executing this program, what will be printed to the console? 5
10. Execute this program and see what is printed to the console. Were you correct? Yes

The result of the last modification to the program may have resulted in an output you weren't expecting.

11. Explain what is happening in memory (regarding the variables *x* and *y*) before, during and after the *Foo()* function is called?

Answers are on the following page.

Answers:

Have you completed the exercises yourself?

Attempting the exercises before looking at the answers will be critical in helping you understand the difference between value and reference types, and parameter passing in C#.

1. *IntHolder* is a value type. All structures, including custom structures, are value types.

   Even if a structure contains reference types as its members, it is still a value type.

   The other value types are:
   - All numeric data types
   - Boolean, Char and Date
   - Enumerations

2. By default, parameters in C# are value parameters. Therefore, the variable *y* is being passed by value.
   A new storage location will be created to store the variable *x*. The value of variable *y* (5) is then copied to variable *x*, so that upon entry into function *Foo()* the value of *x* is also 5.

3. The value *5* will be printed to the console.

   Because the parameter is a value parameter, a new variable is created to store *x*. When *x* is assigned a new *IntHandler*, the change is only seen by *x*.

   When the function *Foo()* exits, the variable *x* is destroyed. The value of *y* has not changed, so the value 5 is written to the console.

4. You should have created a functioning program for this question. Use a breakpoint to inspect the values of *x* and *y* as the program executes.

5. The *ref* modifier makes the parameter a reference parameter. The *ref* keyword must also be used when calling the function so that it is clear to the programmer what the behaviour will be.

   The variable *y* is therefore being passed by reference.

6. The value 0 is written to the console.

   The variable *y* is passed by reference, even though *IntHolder* itself is a value type. This means that instead of allocating storage and creating a new variable to use for *x* (and copying the data to *x*), *x* will store a reference to the original variable *y*.

Any changes we make to variable *x* inside function *Foo()* will be seen by variable *y* (because they both refer to the same variable).

After the call to *Foo(ref y)*, the value of *y* is a new *IntHolder*. The integer member variable is automatically assigned a value of 0.

7. You should have created a functioning program for this question. Use a breakpoint to inspect the values of *x* and *y* as the program executes.

8. All classes are reference types.

   A reference type contains a pointer to another memory location that holds the data. Reference types include the following:
   - String
   - All arrays, even if their elements are value types
   - Class types
   - Delegates

9. 5 is written to the console.

   IntHolder is now a reference type (a class), but the parameter is a *value* parameter (the *Foo()* function no longer uses the *ref* modifier, making *x* a value parameter by default).

   When we assign a reference type variable to another, the data it contains is copied. But reference types contain a pointer to another memory location that holds the data. So when *y* is assigned to *x*, it is this pointer that is copied.

   When the *Foo()* function starts, the variables *x* and *y* both point to the same location in memory. If the function *Foo()* simply modified the variable *x.i*, then the change would be seen by *y*.

   But instead, a new *IntHolder* object is assigned to *x*. Then means that *x* and *y* now both refer to different objects in memory. (We've changed the object that *x* points to). *Y* is a reference to the same object it was before the *Foo()* function call.

   Since *y* hasn't changed, the value 5 is written to the console.

   This difference (why passing a reference type as a value parameter differs to passing a value type as a reference parameter) is absolute crucial to understanding parameter passing in C#.

10. You should have created a functioning program for this question. Use a breakpoint to inspect the values of *x* and *y* as the program executes.

11. The program starts and the variable *y* is assigned a new object of type *IntHandler*. The actual value of *y* is a memory address (or pointer) to where the actual object resides in memory.

When calling the function *Foo()*, a new variable *x* is created. Then *y* is assigned to *x*, the value of *y* is copied to *x*. But, because the value for references types is a memory pointer, it is this memory address that is copied to *x*. *X* and *y* now refer to the same object in memory (with a value 5).

When *x* is assigned a new *IntHandler* object, the object is stored in memory and the memory address is copied into *x*. *X* and *Y* now refer to two different objects stored in different locations in memory. No changes have occurred to *y*.

When the function *Foo()* exits, *x* is destroyed. Because *x* and *y* refer to different objects, this does not affect *y*. *Y* still refers to the original object, containing the original value 5.