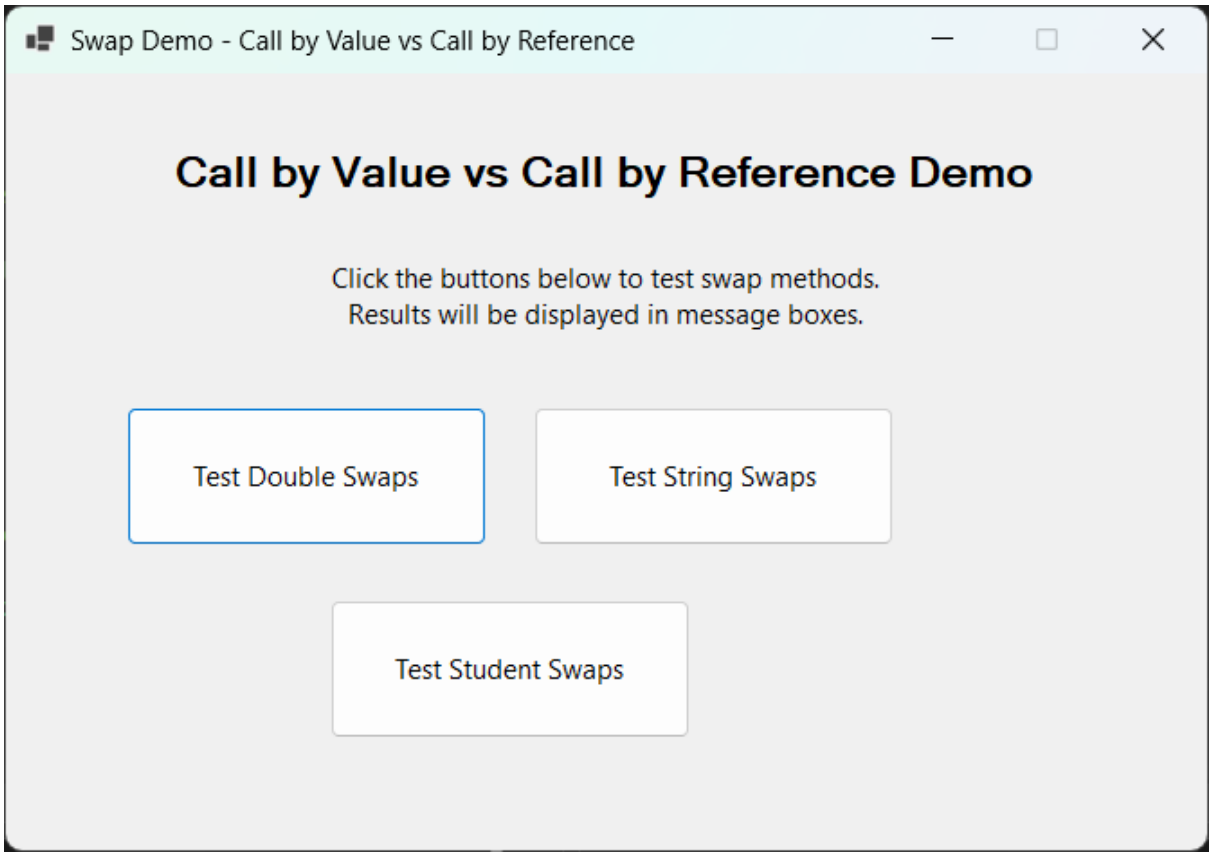
Assignment 2

Student Name: Durvank Deorukhkar
Course: Survey of System & Programming language

What This App Does:

You will see three buttons on the screen, each testing a different type of data: numbers, text, and objects. Each button runs the same experiment twice: once the "normal" way (call by value) and once with the special ref keyword (call by reference).



Code Structure and Implementation

SwapMethods Class Implementation

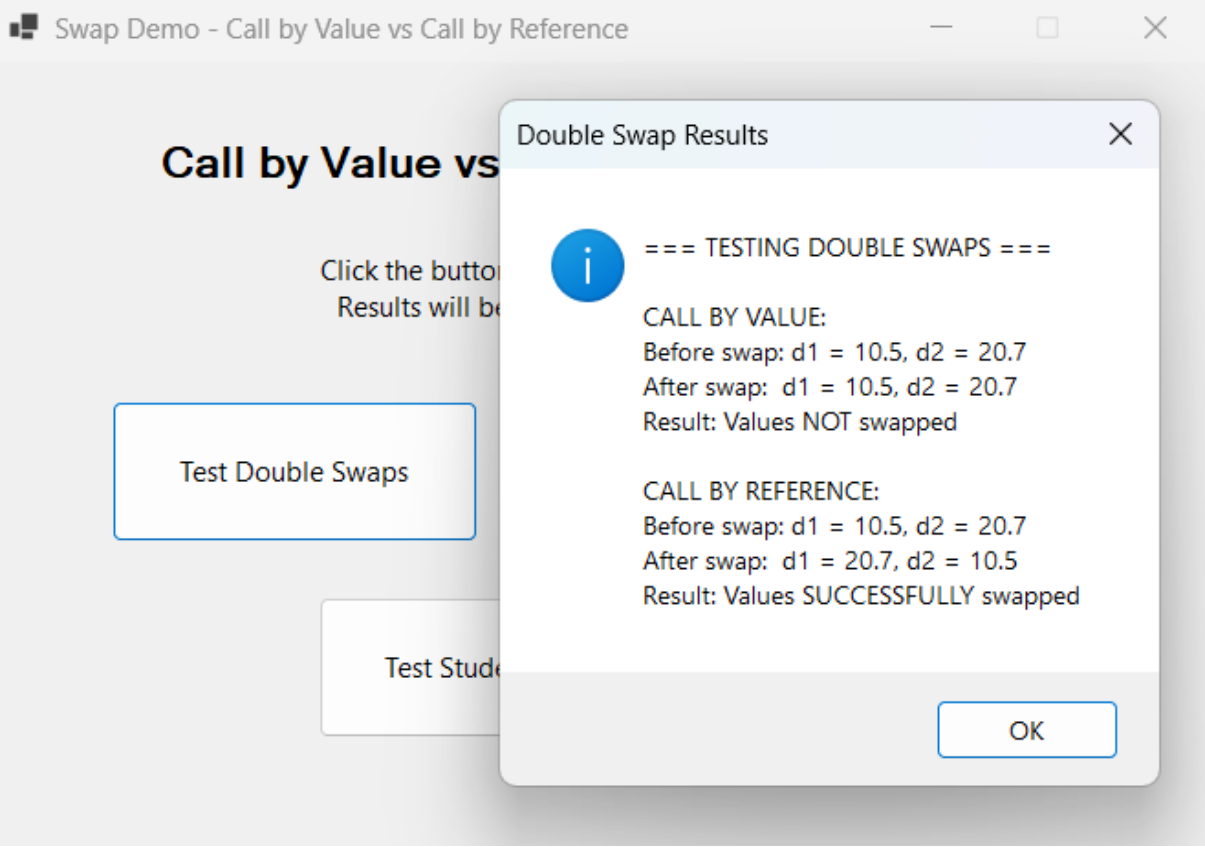
```
public class SwapMethods
{
    // Swap strings by value - won't work
    public void SwapStringsByValue(string str1, string str2)
    {
        string temp = str1;  // temp holds copy of str1's reference
        str1 = str2;         // str1 points to str2's object (local only)
        str2 = temp;         // str2 points to temp's object (local only)
        // Local parameter changes are lost when method ends
    }

    // Swap strings by reference - will work
    public void SwapStringsByRef(ref string str1, ref string str2)
    {
        string temp = str1;  // temp holds reference to original str1
        str1 = str2;         // original str1 now points to str2's object
        str2 = temp;         // original str2 now points to temp's object
        // Changes affect original variables directly
    }
}
```

Form Event Handler Code

```
// Test string swaps
private void testStringsButton_Click(object sender, EventArgs e)
{
    string result = "=== TESTING STRING SWAPS ===\n\n";

    // Test by value - will FAIL
    string str1 = "Hello";
    string str2 = "World";
    result += "CALL BY VALUE:\n";
    result += $"Before swap: str1 = \"{str1}\", str2 = \"{str2}\"\n";

    swapMethods.SwapStringsByValue(str1, str2);  // Copies passed to method

    result += $"After swap:  str1 = \"{str1}\", str2 = \"{str2}\"\n";
    result += "Result: Strings NOT swapped\n\n";

    // Test by reference - will SUCCEED
    str1 = "Hello";  // Reset values for second test
    str2 = "World";
    result += "CALL BY REFERENCE:\n";
    result += $"Before swap: str1 = \"{str1}\", str2 = \"{str2}\"\n";

    swapMethods.SwapStringsByRef(ref str1, ref str2);  // References passed

    result += $"After swap:  str1 = \"{str1}\", str2 = \"{str2}\"\n";
    result += "Result: Strings SUCCESSFULLY swapped";
    MessageBox.Show(result, "String Swap Results", MessageBoxButtons.OK,
MessageBoxIcon.Information);
}
```

1) Double Swap Results

When you press "Test Double Swaps", you get a message box with the results showing both call by value and call by reference behavior.



Code Execution Analysis:

Call by Value Process:

```
// In main method:
double d1 = 10.5;  // d1 contains value 10.5
double d2 = 20.7;  // d2 contains value 20.7

// Method call - VALUES are copied
swapMethods.SwapDoublesByValue(d1, d2);

// Inside SwapDoublesByValue method:
public void SwapDoublesByValue(double a, double b)  // a=10.5, b=20.7
{
```

```
    double temp = a;  // temp = 10.5
    a = b;          // a = 20.7 (local parameter only)
    b = temp;       // b = 10.5 (local parameter only)
}
// When method ends: d1 still = 10.5, d2 still = 20.7
```

Call by Reference Process:

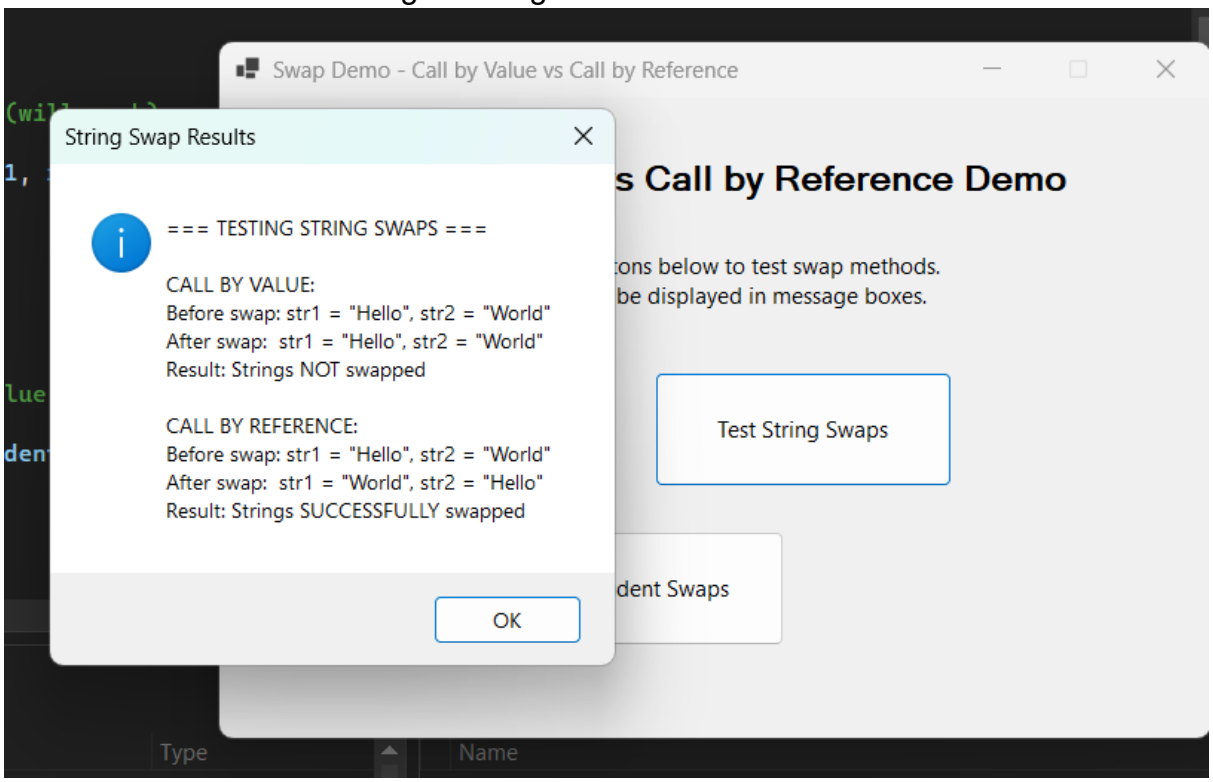
```
// Method call - REFERENCES are passed
swapMethods.SwapDoublesByRef(ref d1, ref d2);


// Inside SwapDoublesByRef method:
public void SwapDoublesByRef(ref double a, ref double b)  // a=ref to d1, b=ref to d2
{
    double temp = a;  // temp = 10.5 (reading through reference)
    a = b;          // d1 = 20.7 (modifying original through reference)
    b = temp;       // d2 = 10.5 (modifying original through reference)
}
// When method ends: d1 now = 20.7, d2 now = 10.5
```

The output then clearly demonstrates the basic difference between the two approaches. In the call-by-value scenario, SwapDoublesByValue takes the copies of the original double variables. When SwapDoublesByValue swaps parameters a and b within the method, it's merely changing these local copies - the original variables d1 and d2 within the calling method are left completely unaffected.

Conversely, if I use the ref keyword in SwapDoublesByRef, the function can directly access the memory locations of the original variables. The swap function can now indeed alter the values held in d1 and d2, resulting in the successful value swap.

2) String Swap Results

When you press "Test String Swaps", you get a message box with results showing the critical difference in string handling.



String Reference Behavior Analysis:

This was a surprise to me at first because strings are reference types in C#. But the behavior makes sense when I consider what is being passed into the method.

Call by Value - String References:

```
// In main method:
string str1 = "Hello";  // str1 points to "Hello" object in heap
string str2 = "World";  // str2 points to "World" object in heap

// Method call - COPIES of references are passed
swapMethods.SwapStringsByValue(str1, str2);

// Inside method:
public void SwapStringsByValue(string str1, string str2)
{
    // str1 parameter = copy of main's str1 reference → "Hello"
```

```
    // str2 parameter = copy of main's str2 reference → "World"

    string temp = str1;  // temp → "Hello"
    str1 = str2;         // local str1 copy → "World"
    str2 = temp;         // local str2 copy → "Hello"

    // Only the copied references are swapped!
    // Main method's str1 and str2 variables are unchanged
}
```

Call by Reference - Direct Access:

```
// Method call - REFERENCES to actual variables passed
swapMethods.SwapStringsByRef(ref str1, ref str2);

// Inside method:
public void SwapStringsByRef(ref string str1, ref string str2)
{
    // str1 parameter = reference to main's actual str1 variable
    // str2 parameter = reference to main's actual str2 variable

    string temp = str1;  // temp → "Hello"
    str1 = str2;         // main's actual str1 → "World"
    str2 = temp;         // main's actual str2 → "Hello"

    // The original variables are modified directly!
}
```

In the call by value scenario, the function accepts copies of the references pointing to the string objects instead of copies of the strings themselves. While I may have the theoretical capability of altering the string objects using these reference copies (if strings were not immutable), I cannot modify what the original reference variables str1 and str2 point to.

The ref parameter does exactly the opposite by enabling the actual reference variables themselves to be passed, and thus the method can alter what these variables refer to.

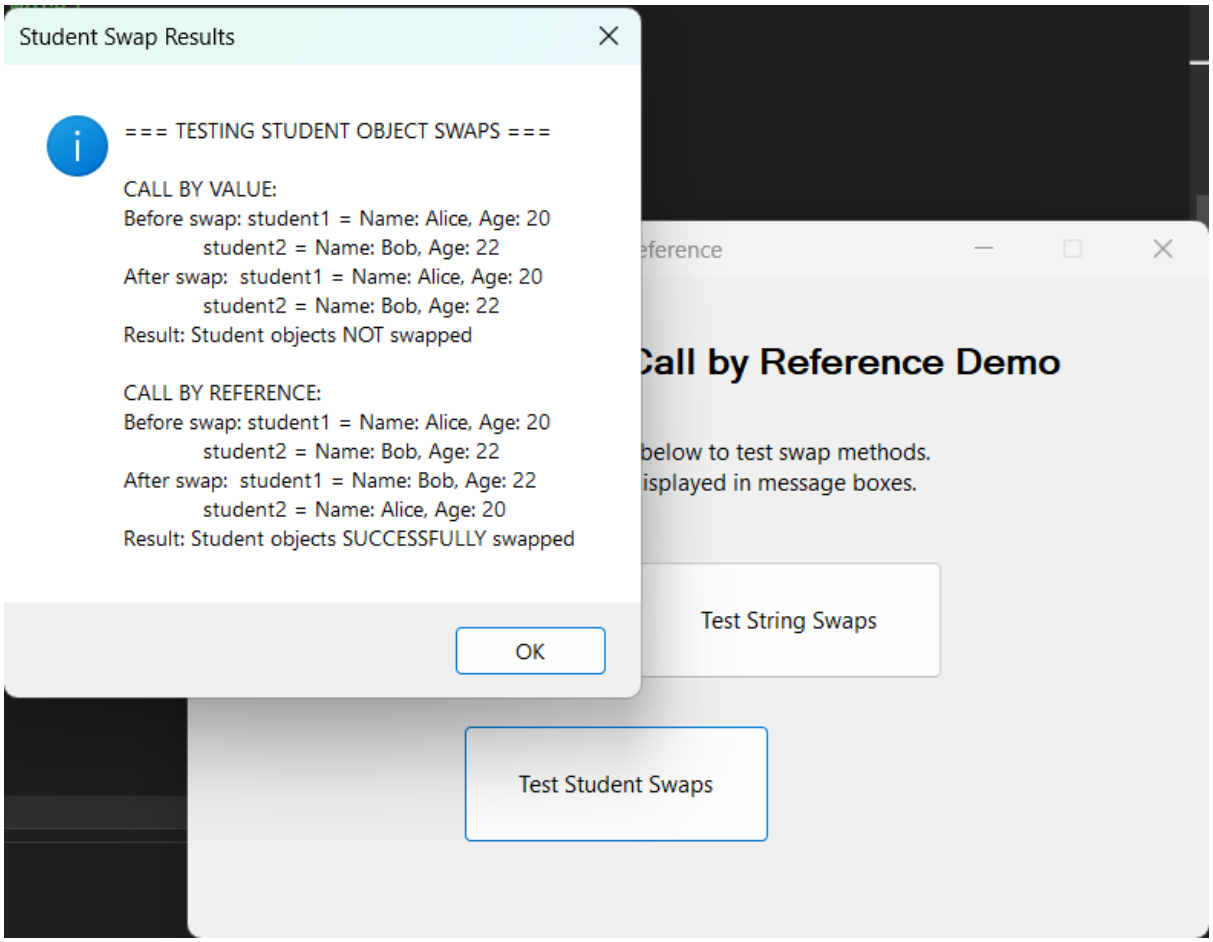
3) Student Object Swap Results

When you click on "Test Student Swaps", you get a message box demonstrating object reference behavior.

Object Reference Swapping Analysis:

```
// Student class definition:
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Student(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

// In test method:
Student student1 = new Student("Alice", 20);  // student1 points to Alice object
Student student2 = new Student("Bob", 22);    // student2 points to Bob object

// Call by value - copies of object references
swapMethods.SwapStudentsByValue(student1, student2);

public void SwapStudentsByValue(Student student1, Student student2)
{
```

```
    // student1 parameter = copy of reference to Alice object
    // student2 parameter = copy of reference to Bob object

    Student temp = student1;  // temp points to Alice object
    student1 = student2;      // local student1 copy points to Bob object
    student2 = temp;          // local student2 copy points to Alice object

    // Only the copied references are swapped!
    // Original student1 and student2 variables unchanged
}
```



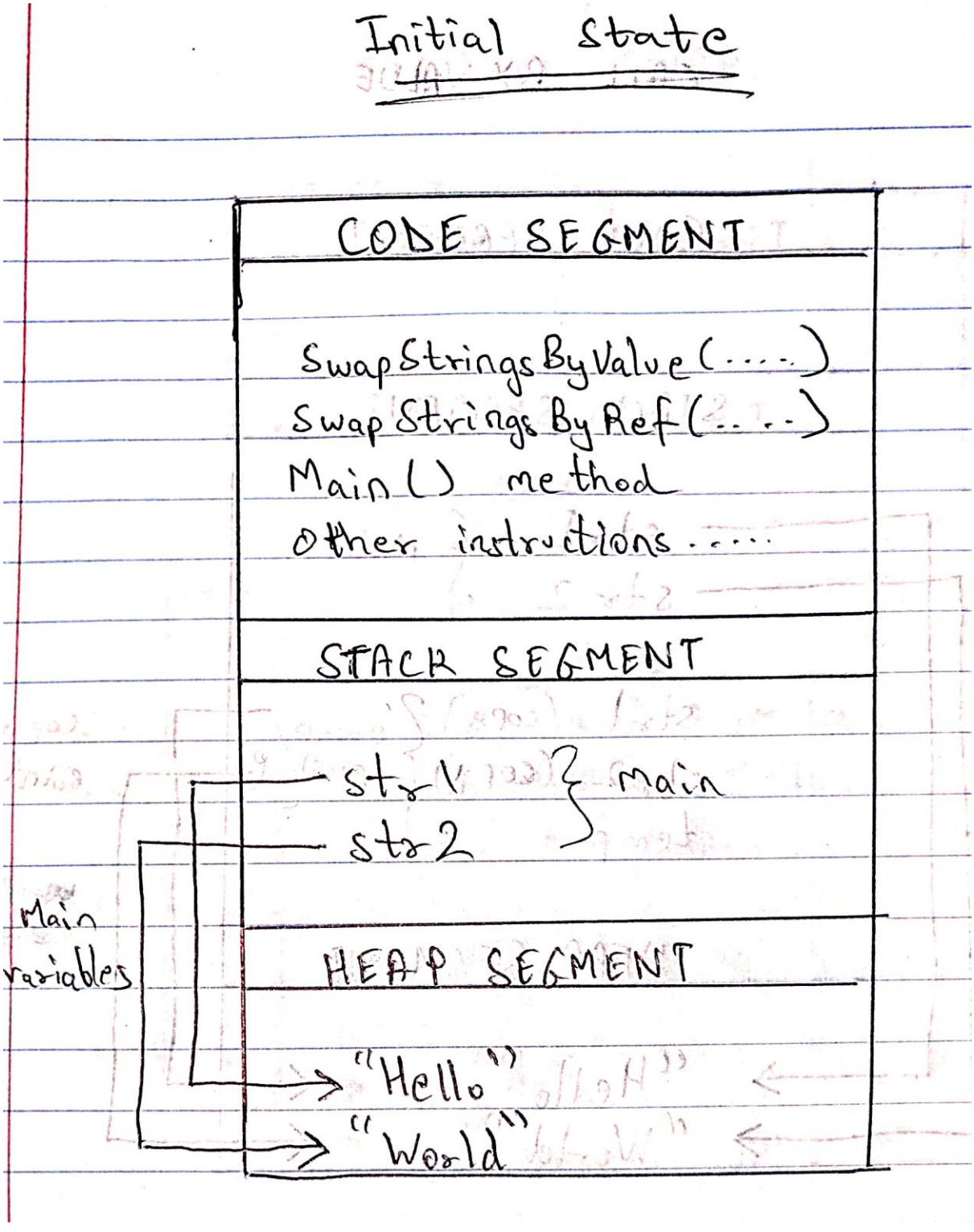
This test demonstrates an important distinction that I had conceptual difficulty with at first. When passing Student objects by value, I am not creating copies of the entire Student objects, that would be expensive and inefficient. Instead, I am creating copies of the references pointing to those objects.

What this means is that if I were to modify properties of the Student objects within the method (like Alice's age), those modifications would be visible outside the method because I am still working with the same objects in memory.

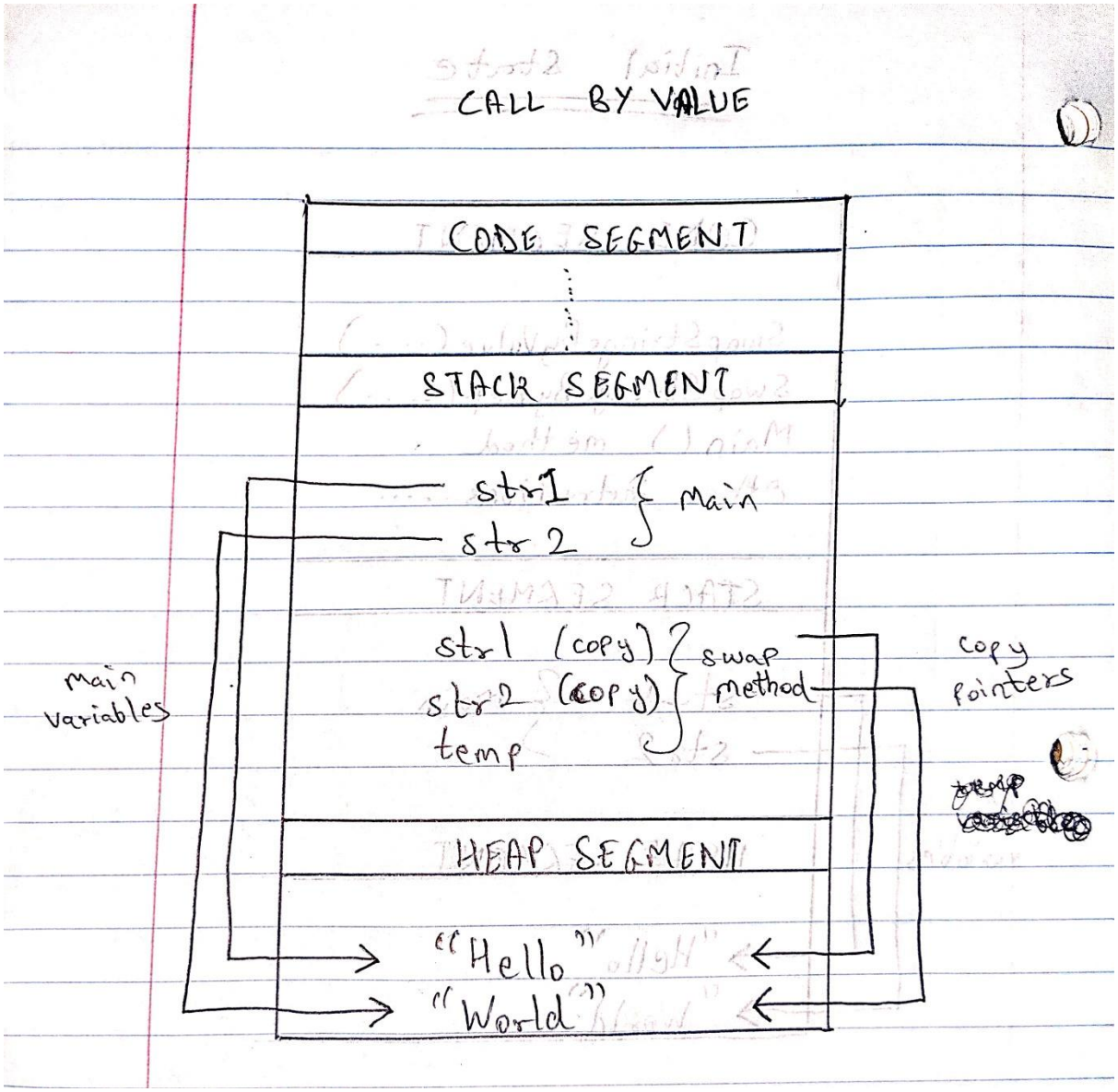
However, in the case of swapping, reassigning which objects the variables refer to, the call by value system breaks down because I'm swapping the copied pointers, not the actual reference variables in the calling method.

Memory Stack

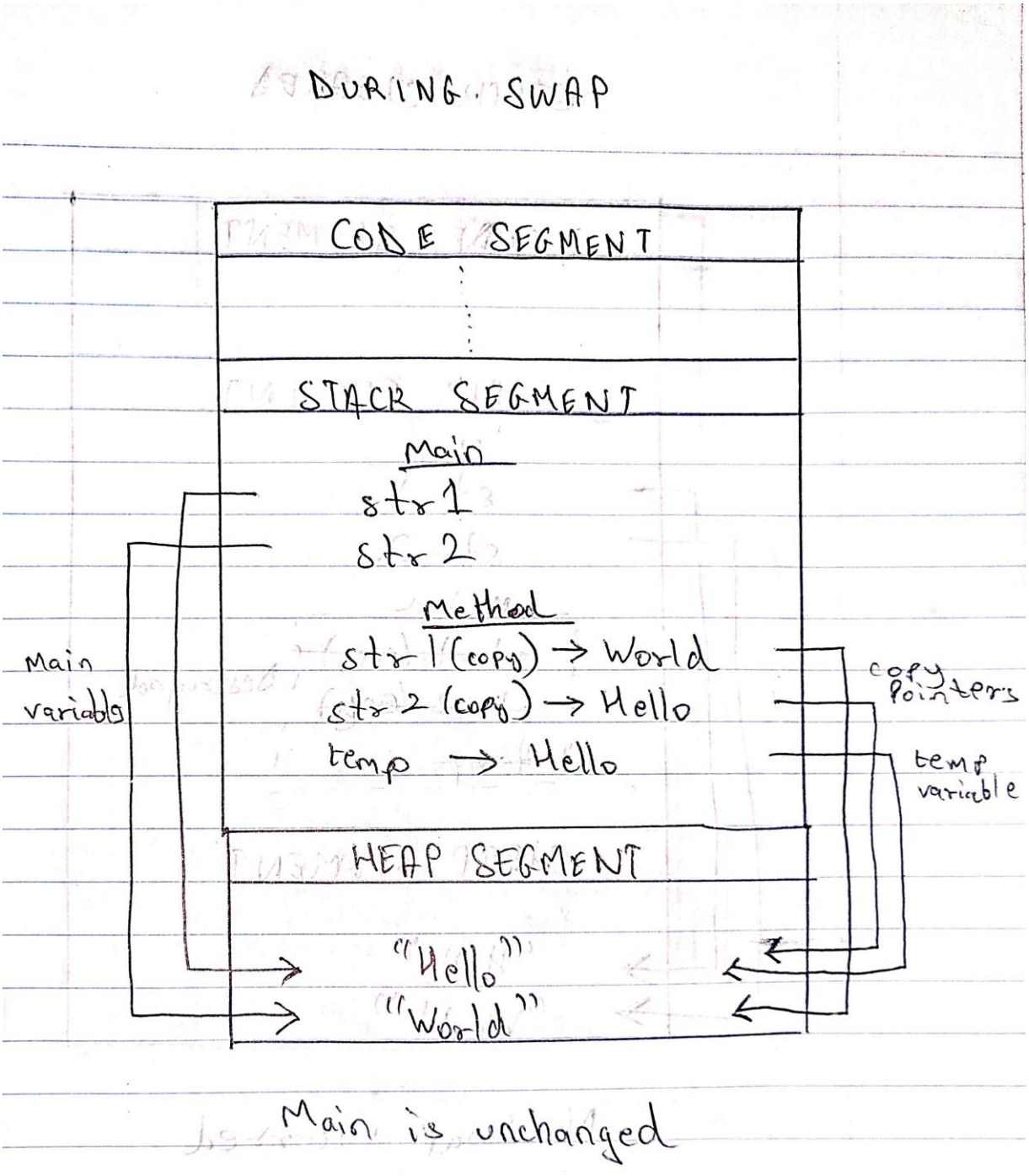
1) **Initial State**



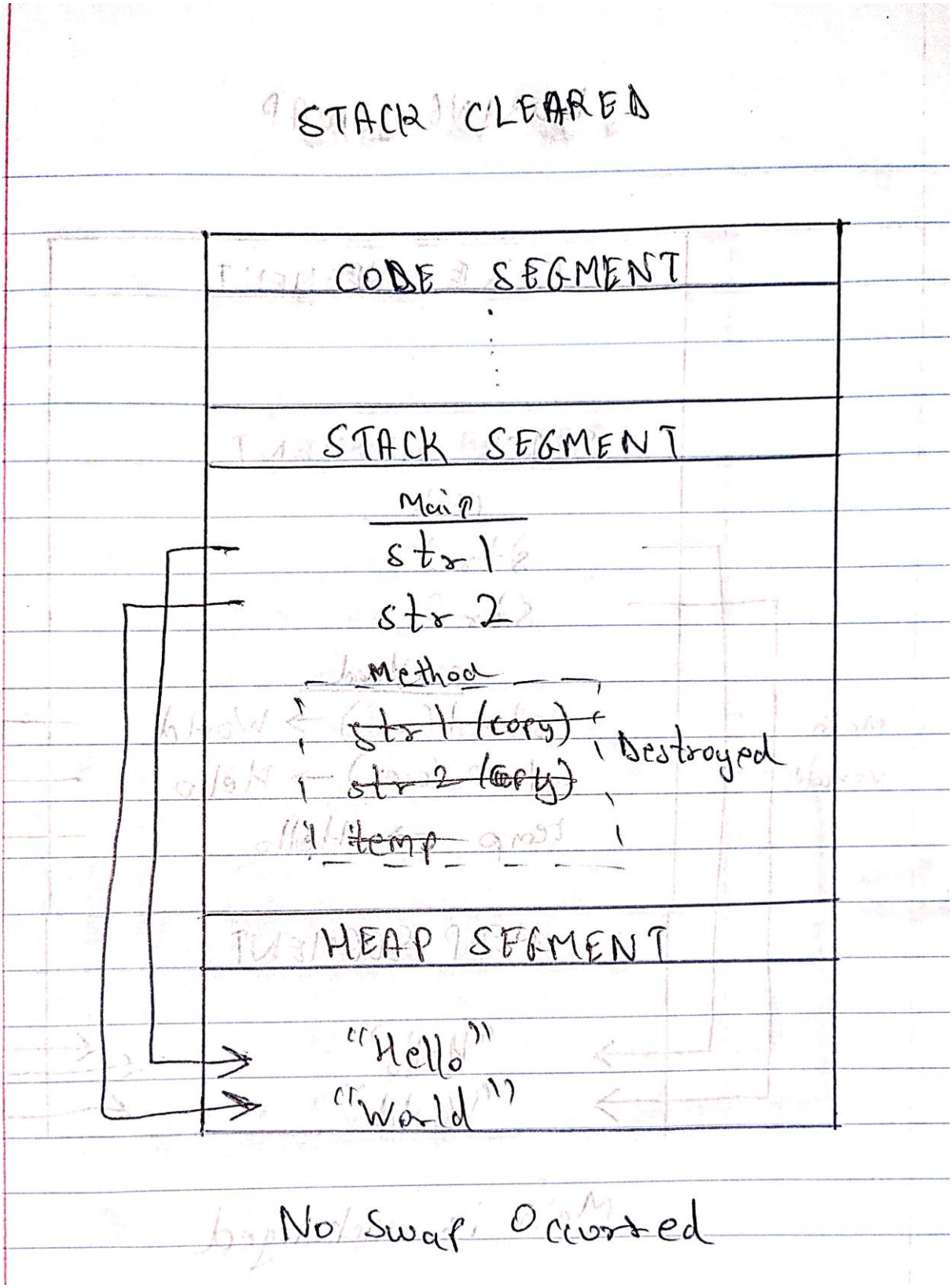
2) Call By Value method is called



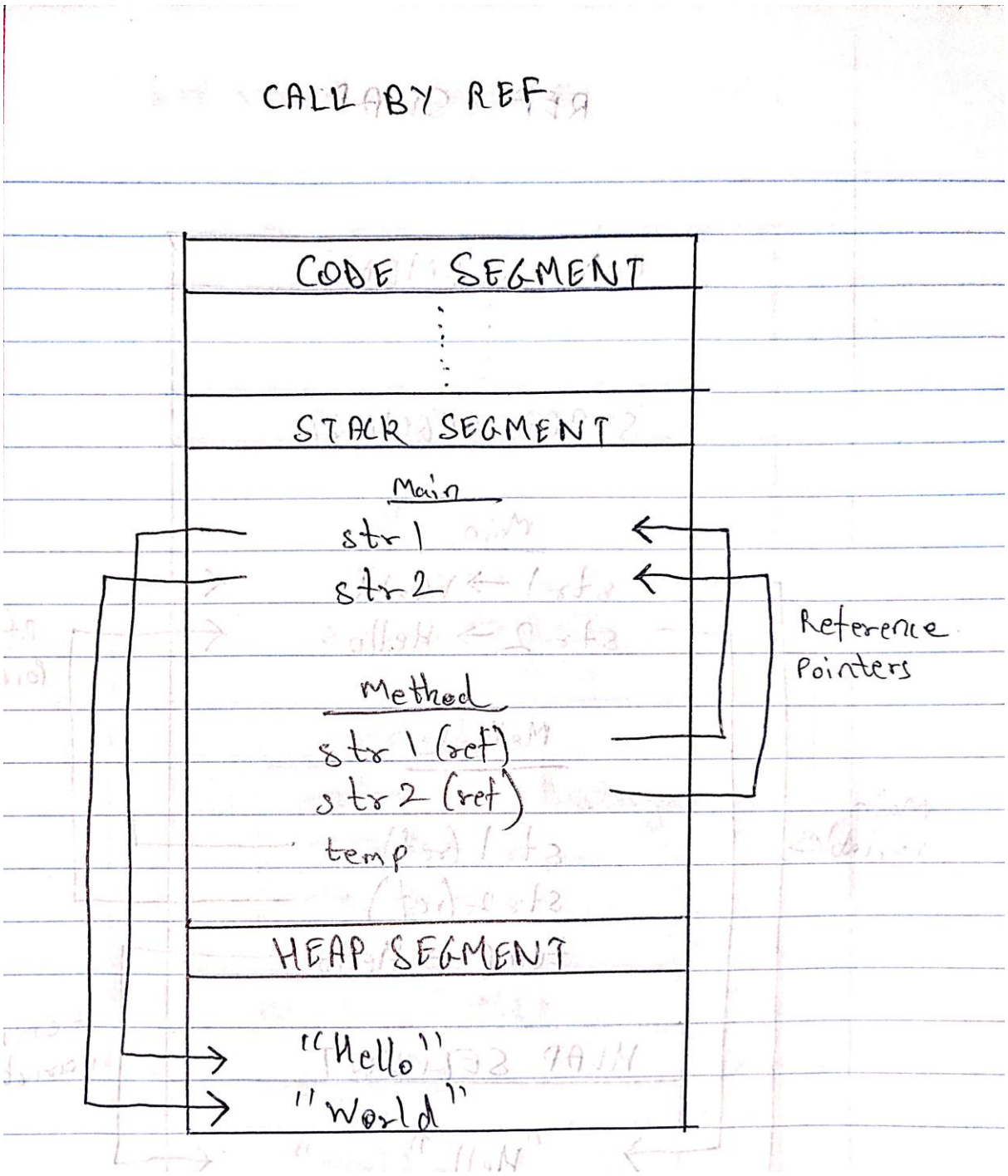
3) **Swapping is performed**



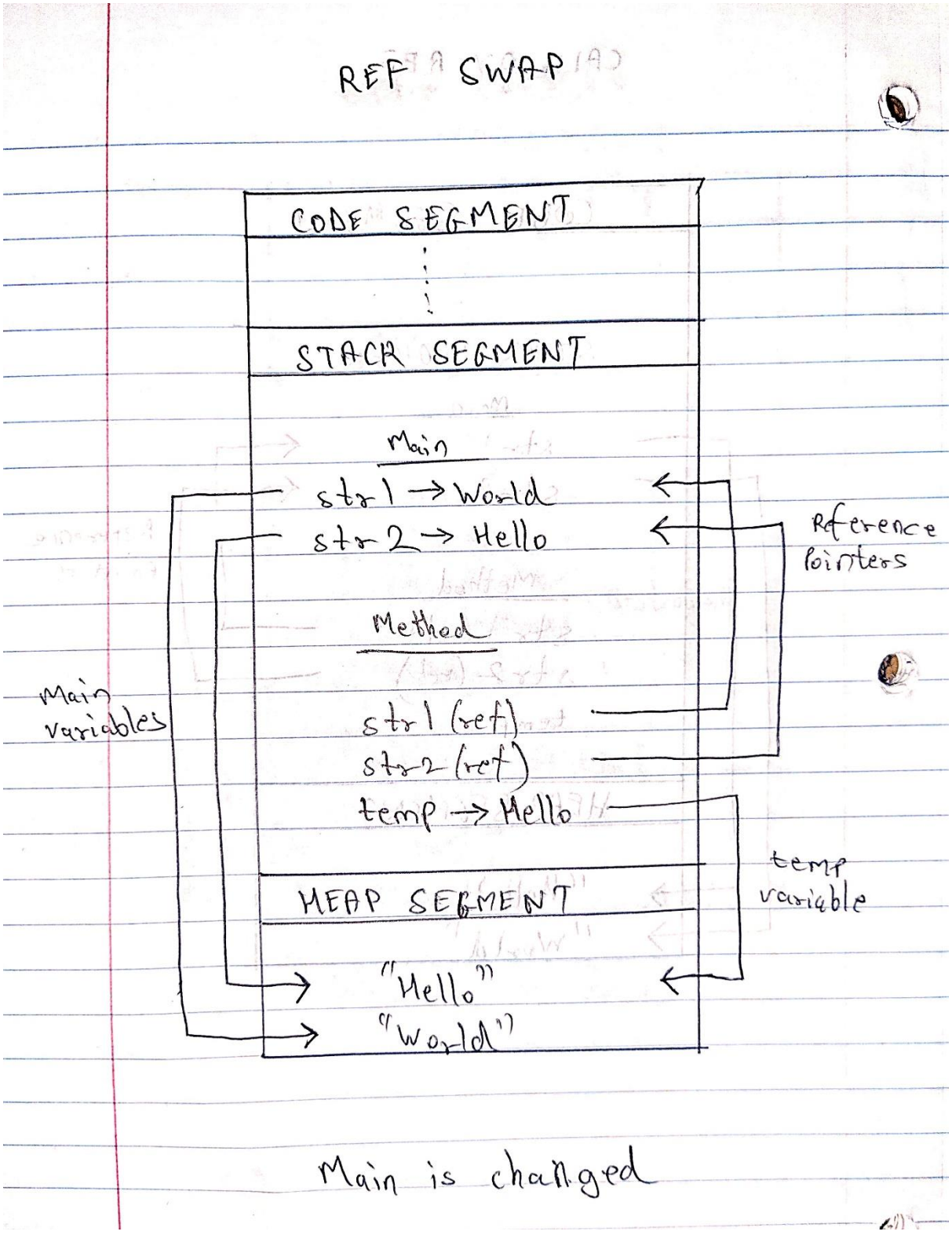
4) **Stack is cleared after performing the swap**



5) Now Call By Ref method is called

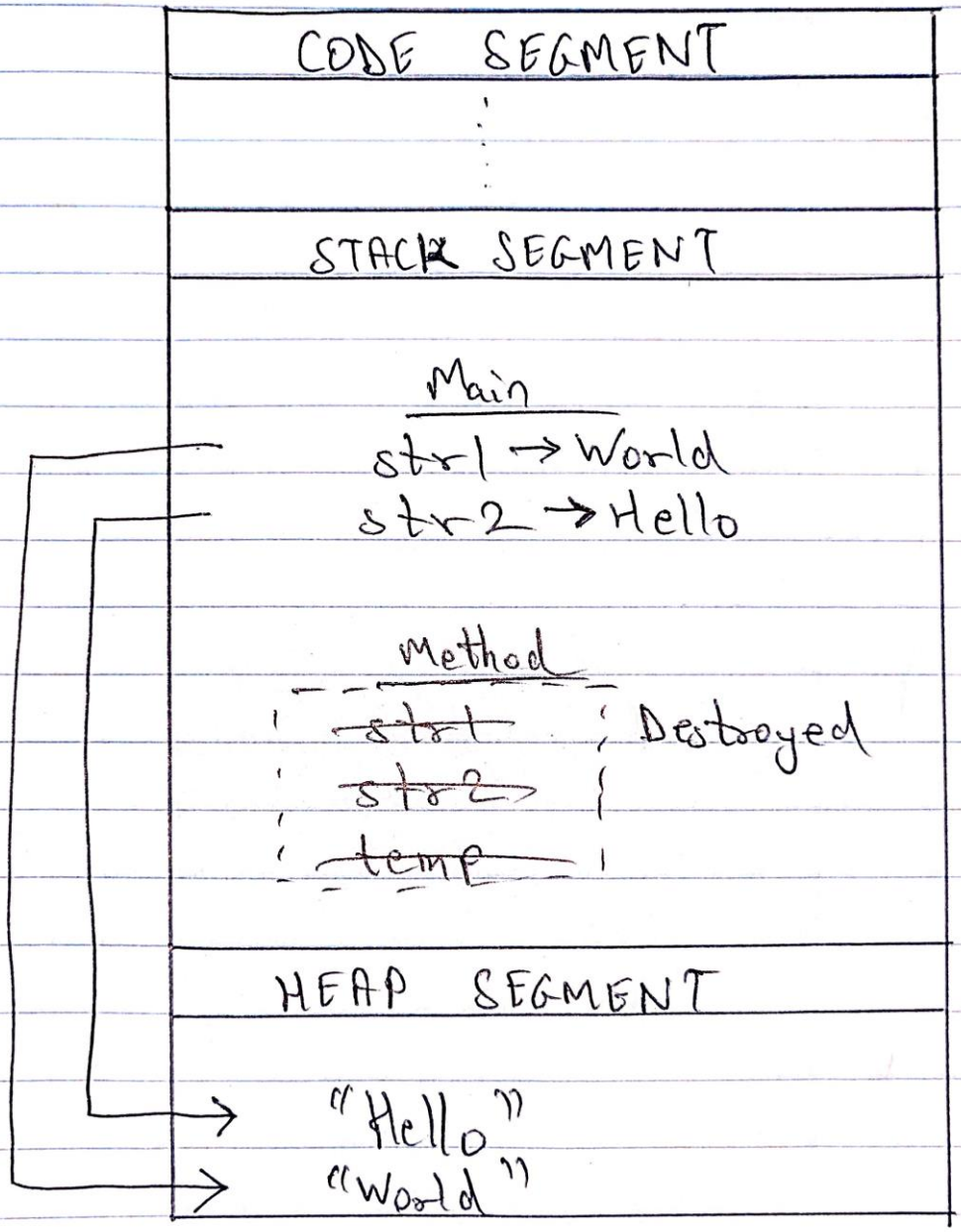


6) **Swapping is performed**



7) **Ref swapping is complete, and stack is cleared**

REF SWAP Complete



Swap Successful