1Z0-808 Exam Topic Reviewer

TopicId: 1036
Topic: Passing Data Among Methods

August 5, 2025

Introduction: The Great Debate - Pass-by-Value

Class, let's settle one of the most misunderstood concepts in Java, one that the 1Z0-808 exam loves to test: how data is passed to methods. You will hear people talk about 'pass-by-reference', but let me be perfectly clear: **Java is always, 100% of the time, pass-by-value**. Understanding what this truly means is the key to correctly predicting the outcome of many exam questions. What "pass-by-value" means is that when you pass a variable to a method, a **copy** of the variable's value is made, and that copy is what the method receives. The confusion arises because the "value" of a primitive is different from the "value" of an object reference.

1 Passing Primitive Types

This is the simple case. The value of a primitive variable is the data itself. When you pass a primitive (like an `int`, `double`, or `boolean`) to a method, a copy of its literal value is passed.

- Any changes made to the parameter inside the method have **no effect** on the original variable in the calling code.
- The method works on its own private copy.

Example:

```
public class PrimitiveTester {
    public static void main(String[] args) {
        int myNumber = 10;
        System.out.println("Before call, myNumber is: " + myNumber);
        tryToModify(myNumber);
        System.out.println("After call, myNumber is: " + myNumber);
    }

    public static void tryToModify(int numberParam) {
        // numberParam is a COPY of myNumber's value (10)
        numberParam = 20; // This only changes the copy
        System.out.println("Inside method, parameter is: " + numberParam);
    }
}
```

Output:

```
Before call, myNumber is: 10
Inside method, parameter is: 20
After call, myNumber is: 10
```

2 Passing Object References

This is where the confusion begins. Let's be precise. The "value" of a reference variable is **not the object itself**, but the **memory address** where the object is stored on the heap.

So, when you pass an object to a method, you are passing a **copy of the memory address**.

This means that both the original reference variable (in the caller) and the method's parameter now hold identical copies of a memory address. They both *point to the same single object on the heap.*

This has two critical implications:

2.1 Case 1: Modifying the Object's State (The Insides)

Because the method parameter points to the *same object*, if you use that reference to change the object's internal state (e.g., by calling a setter method or modifying a public field), the changes **will be visible** outside the method. There is only one object, and you just modified it.

Example:

```
public class ReferenceTester {
    public static void main(String[] args) {
        StringBuilder name = new StringBuilder("Alice");
        System.out.println("Before call, name is: " + name);
        modifyState(name);
        System.out.println("After call, name is: " + name);
    }

    public static void modifyState(StringBuilder builderParam) {
        // builderParam is a COPY of the reference, pointing to the SAME object
        builderParam.append(" in Wonderland");
    }
}
```

Output:

```
Before call, name is: Alice
After call, name is: Alice in Wonderland
```

2.2 Case 2: Reassigning the Reference Itself (The Pointer)

What if, inside the method, you try to reassign the parameter to a `new` object? Remember, the method only has a *copy* of the reference. Reassigning the parameter only changes this local copy. It does **not** change the original reference variable in the calling code.

Example:

```
public class ReferenceTester {
    public static void main(String[] args) {
        StringBuilder name = new StringBuilder("Bob");
        System.out.println("Before call, name is: " + name);
        reassignReference(name);
        System.out.println("After call, name is: " + name);
    }

    public static void reassignReference(StringBuilder builderParam) {
        // This changes the method’s local copy ’builderParam’ to point
        // to a brand new object. The original ’name’ reference is unaffected.
        builderParam = new StringBuilder("Charlie");
    }
}
```

Output:

```
Before call, name is: Bob
After call, name is: Bob
```

Key Takeaways for the 1Z0-808 Exam

- Java is **always pass-by-value**. No exceptions.
- For primitives, a **copy of the value** is passed. Changes inside the method are isolated.
- For objects, a **copy of the reference (memory address)** is passed.
- This allows a method to **change the state** of the object that was passed in.
- This does **NOT** allow a method to make the original variable refer to a different object.
- To ace these questions, ask yourself: Is the code changing the object’s internal data, or is it trying to reassign the reference variable itself?