Phanthira Kositjaroenkul - 6630003
Kaung Khant Lin - 6540131
Thit Lwin Win Thant - 6540122

Task 1: Trace the Outputs

1. a = 10, b = 250
2. **Explanation**:
    a. In C, when `updateValue(a);` is called, a copy of the number `10` is passed to the function, so `x` inside `updateValue` receives `10`.
    b. Changing `x` to `250` only alters this local copy, and the original `a` in `main` remains `10` because it's at a different memory location.
    c. When `updateReference(&b);` is called, the function receives the *address* of `b`.
    d. Even though the pointer `y` itself is a copy, it still points to the *original* `b` in `main`.
    e. Therefore, `*y = 250` directly changes `b`'s memory location, and this alteration persists after the function completes.

Task 2: Trace the Outputs

1. a = [1, 2, 3, 55] b = 10
2. **Explanation**:
    a. Python's argument passing uses "call by object reference." Functions receive a reference to the same object, with the parameter as a new local variable.
    b. `update_list(a)`: `lst` refers to the same list as `a`. `lst.append(55)` modifies the list in place, so `a` gains 55.
    c. `update_number(b)`: `n` is a new local name for the integer 10. `n = 99` rebinds `n` to a new integer (99) without altering the original (immutable) 10, thus `b` remains 10.

Task 3: Big Hunt (1)

1. What will be printed on the console?
   "**number = 5**"
2. Java is **pass-by-value**. For primitives (like `int`), the *value* of num is copied into the parameter n. Reassigning n to `100` only changes the **local copy** so the original `num` in `main` is untouched.

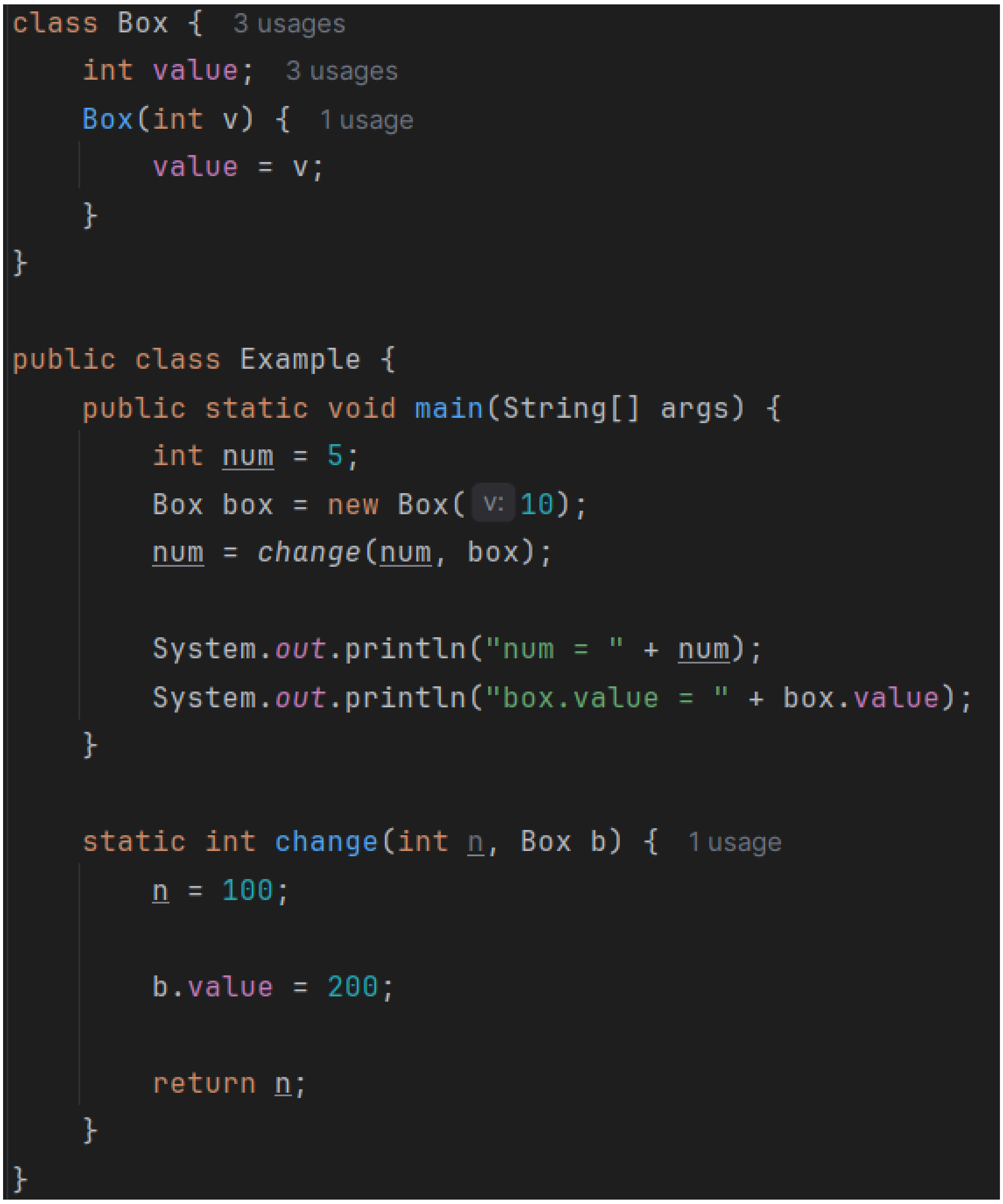
3.
```
public class ParameterPassingExample {
  public static void main(String[] args) {
    int num = 5;
    num = justChange(num);
    System.out.println("number = " + num);
  }

 static int justChange(int n) {
    n = 100;
    return n;
  }
}
```
4. This line: `num = justChange(num);`
   By compute a new value inside `justChange` and **return** it so `main` explicitly reassigns `num` to that value.

Task 4: Big Hunt (2)

1. **Output:**

   num = 5
   box.value = 200
   Explanation:
   - Primitive types (like `int`) are passed by value. This means the method receives a copy of the value. Any changes made to this copy inside the method do not affect the original variable.
     - Result: `num` remains 5.

- Objects (like `Box`) are passed by value of their reference. This means the method receives a copy of the reference. Both the original variable and the method's parameter point to the same object in memory.
    - Modifying the object's fields (e.g., `b.value = 200;`) changes the actual object, so the change is visible in the original variable.
    - Reassigning the reference itself (e.g., `b = new Box(300);`) only changes where the method's local copy of the reference points, not the original reference.
    - Result: `box.value` becomes 200.

2. 

```java
class Box {  3 usages
    int value;  3 usages
    Box(int v) {  1 usage
        value = v;
    }
}

public class Example {
    public static void main(String[] args) {
        int num = 5;
        Box box = new Box( v: 10);
        num = change(num, box);

        System.out.println("num = " + num);
        System.out.println("box.value = " + box.value);
    }

    static int change(int n, Box b) {  1 usage
        n = 100;

        b.value = 200;

        return n;
    }
}
```

3. **Effect on num:**

- The change method was modified to static int change(int n, Box b). This means it now returns an integer value.
- Inside the change method, n = 100; sets the local parameter n to 100.
- The return n; statement sends this value (100) back to where the change method was called.
- In main, the line num = change(num, box); receives this returned value (100) and assigns it back to the original num variable. This is how the value of num is updated to 100.

Effect on box.value:

- The change method still receives a copy of the reference to the Box object. Both the box in main and the parameter b in change point to the same Box instance in memory.
- The line b.value = 200; directly modifies the value field of this shared Box object.
- Because box in main refers to the exact same object whose value was changed via b.value, the change is immediately visible in main.

Essentially, num is updated because the change method returns its new value, and main receives and assigns it. box.value is updated because the method directly manipulates the state of the object that the box reference points to.

Task 5: Discussion

1. Java was designed to be simpler than C/C++ (no pointer arithmetic, no manual memory management) so that means Java always using `pass-by-value` (even for object references) makes parameter passing predictable and prevents accidental side effects from reference aliasing. And it also helps avoiding `Pointer-like Confusion`
2. Understanding parameter passing in Java helps avoid bugs in larger programs by:

   - **Predicting Data Changes:** Knowing that primitives are passed by value (copies) means you won't accidentally expect changes made inside a method to affect the original variable, unless you explicitly return and reassign it.
   - **Controlling Object State:** Understanding that object references are passed by value (copies of the reference) allows you to correctly:
     1. **Modify Objects:** You know that changes to an object's fields will be visible to the caller because both references point to the same object.

   2. **Avoid Unintended Reassignments:** You also know that reassigning the reference inside a method only affects the method's local copy, not the original reference in the calling code.

- **Improving Debugging:** It helps pinpoint the source of unexpected variable values by clearly distinguishing between modifications to primitive copies and mutations of shared objects.
- **Guiding Code Design:** It informs decisions about whether to make objects mutable or immutable and how methods should communicate results (e.g., via return values or by modifying object state).

Essentially, it provides a clear mental model for how data moves and changes between different parts of your Java application, leading to more robust and predictable code.