



University of Maryland College Park

Department of Computer Science

CMSC132 Spring 2026

Exam #1

FIRSTNAME, LASTNAME (PRINT IN UPPERCASE):

KEY

STUDENT ID (e.g. 123456789):

Instructions

- Please print your answers and use a pencil.
- This exam is a closed-book, closed-notes exam with a duration of 50 minutes and 100 total points.
- **Do not remove the exam's staple.** Removing it will interfere with the scanning process (even if you staple the exam again).
- Write your directory id (e.g., terps1, not UID) at the bottom of pages with **DirectoryId**.
- Provide answers in the rectangular areas.
- Do not remove any exam pages. Even if you don't use the extra pages for scratch work, return them with the rest of the exam.
- Your code must be efficient and as short as possible.
- If you continue a problem on the extra page(s) provided, make a note on the particular problem.
- For multiple choice questions you can assume only one answer is expected, unless stated otherwise.
- You don't need to use meaningful variable names; however, we expect good indentation.
- **You must write your name and id at this point (we will not wait for you after time is up).**
- You must stop writing once time is up.

Grader Use Only

#1	Problem #1 (Short Answer)	30	
#2	Problem #2 (Code 1)	35	
#3	Problem #3 (Code 2)	35	
Total	Total	100	

Problem #1 (Short Answer – 2 points each)

Note: A blank maybe filled by one or two words

1. When overriding the `equals` method, the parameter type must be **Object**.
2. A method declared as `protected` can be accessed by classes in the same **package** or by subclasses.
3. The `compareTo` method returns a **negative int** when the calling object is **less than** the parameter object.
4. Every class in Java inherits the `getClass` method from the `Object` class, which returns the runtime class of an object.
5. When one class contains an instance variable that is an object of another class in order to reuse its functionality (rather than inheriting from it), this design technique is called **composition**.
6. In Java, the ability of a single method or object reference to take on many forms at runtime, allowing a superclass reference to refer to subclass objects and invoke overridden methods, is called **polymorphism**.
7. An abstract data type in which elements are inserted at one end and removed from the other end, following the FIFO (First-In, First-Out) principle, is called a(n) **queue**.
8. An exception that must either be caught using a `try-catch` block or declared using the `throws` keyword is called a(n) **checked exception**.
9. The object-oriented principle that includes data hiding and bundles data together with the methods that operate on it in order to achieve abstraction is called **encapsulation**.
10. In Java, a class uses the keyword **implements** to agree to provide implementations for all methods declared in an interface.

Given the 4 public classes all in the same package.

<pre>public abstract class Device { abstract void powerOn(); } </pre>	<pre>public class SmartPhone extends Phone { void powerOn() { System.out.println("Smart"); } void powerOn(boolean silent) { if (silent) System.out.println("Silent"); else System.out.println("Loud"); }} </pre>
<pre>public class Phone extends Device { void powerOn() { System.out.println("Phone"); } } </pre>	<pre>public class Driver { public static void main(String[] args) { //Code from problem will be here } } </pre>

For problems 11 to 15, assume the given code is the only code added to main. If it will compile, write the output. If it will not compile, write NC and explain why it will not compile. If it will compile but throw an exception at runtime, write CE and explain why an exception is thrown. Limit your explanation to no more than 2 sentences.

11. `Device d = new Device();`

NC - You cannot make an instance of an abstract class

12. `Phone p = new Phone();`
`p = new SmartPhone();`
`p.powerOn();`

Smart

13. `Phone p = new SmartPhone();`
`p.powerOn(false);`

NC - Static type of the variable (Phone) does not have a powerOn with a Boolean parameter

14. `Device d = new Phone();`
`((SmartPhone)d).powerOn(true);`

CE - The variable d references an object of the base class type Phone and is not a SmartPhone object at runtime.

15. `SmartPhone s = new Phone();`
`s.powerOn();`

NC - Type mismatch, a Phone object cannot be assigned to subtype variable.

Problem #2 (Code 1)

```
import java.util.ArrayList;
public class ListTransformer {
    public static void transformList(ArrayList<Object> list, int defaultMultiplier) {
        //YOU WILL CODE THIS
    }

    public static void main(String[] args) {
        ArrayList<Object> list = new ArrayList<>();
        list.add(4); // 4 * 2 = 8 (uses default)
        list.add(null); // index 1 → 1 * 2 = 2
        list.add(3.5); // Double → unchanged
        list.add(new StringBuffer("hello")); // unchanged
        list.add("cat"); // length 3 → multiplier becomes 3
        list.add(5); // 5 * 3 = 15
        list.add(2.5f); // Float → unchanged
        list.add(null); // index 7 → 7 * 3 = 21
        list.add("hi"); // length 2 → multiplier becomes 2
        list.add(6); // 6 * 2 = 12
        System.out.println("Before: " + list);
        transformList(list, 2);
        System.out.println("After: " + list);
    }
}
```

Output

```
Before: [4, null, 3.5, hello, cat, 5, 2.5, null, hi, 6]
After: [8, 2, 3.5, hello, cat, 15, 2.5, 21, hi, 12]
```

Finish the method below that processes an `ArrayList<Object>` in one pass from left to right. The method takes an `ArrayList<Object>` and an integer `defaultMultiplier`.

Maintain a multiplier variable that is initially set to `defaultMultiplier`.

If a `String` is encountered in the list, update the multiplier to be the length of that `String`. The `String` itself should remain unchanged.

If an `Integer` is encountered, replace it with its value multiplied by the current multiplier.

If a `null` is encountered, treat it as an `Integer` equal to its index in the list. Multiply that index by the current multiplier and replace the `null` with the resulting `Integer`.

If any other type of object is encountered (including other subclasses of `Number` such as `Double`, `Float`, or `Long`), leave it unchanged.

The list must be modified **in place**, meaning you must modify and update the given list directly. You may not create a new list and return it.

Only one looping construct is allowed, as you may only make one pass from the start to the end of the list.

No library methods may be used other than: `get`, `set`, and `size` from `ArrayList`. `length` from `String`. `instanceof` is an operator and therefore allowed.

Assume the input list is non-null and contains at least one element.

```
public static void transformList(ArrayList<Object> list, int defaultMultiplier) {  
  
    int currentMultiplier = defaultMultiplier;  
  
    for (int i = 0; i < list.size(); i++) {  
  
        Object obj = list.get(i);  
  
        // If String → update multiplier  
        if (obj instanceof String) {  
            currentMultiplier = ((String) obj).length();  
        }  
  
        // If Integer → multiply and replace  
        else if (obj instanceof Integer) {  
            int value = (Integer) obj;  
            list.set(i, value * currentMultiplier);  
        }  
  
        // If null → treat as Integer equal to index  
        else if (obj == null) {  
            list.set(i, i * currentMultiplier);  
        }  
  
        // All other types remain unchanged  
    }  
}
```

Directory ID:

Problem #2 (Code 2)

```
import java.util.Stack;
import java.util.Deque;
import java.util.LinkedList;

public class StackUtils {
    public static boolean stacksEqual(Stack<Integer> s1, Stack<Integer> s2) {
        //YOU WILL CODE THIS
    }
    public static void main(String[] args) {
        Stack<Integer> s1 = new Stack<>();
        Stack<Integer> s2 = new Stack<>();
        Stack<Integer> s3 = null;

        // Example 1: equal stacks
        s1.push(1); s1.push(2); s1.push(3);
        s2.push(1); s2.push(2); s2.push(3);

        System.out.println(stacksEqual(s1, s2)); // true
        System.out.println("s1 after call: " + s1); // [1, 2, 3]
        System.out.println("s2 after call: " + s2); // [1, 2, 3]
        // Example 2: one null stack
        System.out.println(stacksEqual(s1, s3)); // false
        // Example 3: both null stacks
        System.out.println(stacksEqual(null, null)); // true
        // Example 4: empty stacks
        Stack<Integer> empty1 = new Stack<>();
        Stack<Integer> empty2 = new Stack<>();
        System.out.println(stacksEqual(empty1, empty2)); // true
    }
}
```

Output

```
true
s1 after call: [1, 2, 3]
s2 after call: [1, 2, 3]
false
true
true
```

Finish the method below that will compare two stacks for equality. The method should return **true** if both stacks are **null**, if both stacks are empty, or if they contain the same integers in the same order. It should return **false** if only one is **null**, if their sizes differ, or if any corresponding elements differ.

The only allowed Java library methods are **push**, **pop**, **isEmpty**, and **size** methods from **Stack**, and **addFirst**, **removeFirst**, **addLast**, **removeLast**, and **isEmpty** methods from **Deque** (You might not need them all).

You can only use the declared **tempDeque** to help you code. You cannot create additional stacks, deque, or other data structures (e.g. arrays, ArrayList, etc.).

Local variables and looping/decision making constructs are allowed.

Both stacks must be restored to their original state before the method returns. **If the stacks are not restored correctly, the solution is incorrect even if the boolean result is correct.**

```

public static boolean stacksEqual(Stack<Integer> s1, Stack<Integer> s2) {
    Deque<Integer> tempDeque = new LinkedList<>();

    // Null checks
    if (s1 == null && s2 == null) return true;
    if (s1 == null || s2 == null) return false;

    // Empty stacks
    if (s1.isEmpty() && s2.isEmpty()) return true;
    if (s1.size() != s2.size()) return false;

    boolean isEqual = true;

    // Compare while popping
    while (!s1.isEmpty()) {
        int a = s1.pop();
        int b = s2.pop();

        if (a != b) isEqual = false;

        // Add to FRONT so order is preserved for restoration
        tempDeque.addFirst(a);
        tempDeque.addFirst(b);
    }

    // Restore stacks (correct original order)
    while (!tempDeque.isEmpty()) {
        s2.push(tempDeque.removeFirst());
        s1.push(tempDeque.removeFirst());
    }

    return isEqual;
}

```

Directory ID:

Blank Page