Week 3 Assignment

Name : DHANAPAL

Date : 10/08/2024 - 11/08/2024

1. Please find case 1 and mention the result for the mentioned statements using strings.

2. Find case 2 and mention the result for the statements using integers.

3. Find case 3 and mention how Basic I/O resources are getting closed and the difference that you implemented earlier in the code - copyBytes.java

4. Find case 4 and mention the order for 1,2 and 3 using collections

Case 1:

```
public class StringComparisonExample {
  public static void main(String[] args) {
    // String literals (pooled)
    String str1 = "Hello";
    String str2 = "Hello";

    // New String objects (not pooled)
    String str3 = new String("Hello");
    String str4 = new String("hello");

    // Using ==
    System.out.println("str1 == str2: " + (str1 == str2)); // 1. (same memory reference) what's the result?
    System.out.println("str1 == str3: " + (str1 == str3)); //2. (different memory references) what's the result?
```

```
        // Using equals()
        System.out.println("str1.equals(str3): " + str1.equals(str3)); //3.
(same content) what's the result?
        System.out.println("str1.equals(str4): " + str1.equals(str4)); //4.
(case-sensitive) what's the result?

        // Using equalsIgnoreCase()
        System.out.println("str1.equalsIgnoreCase(str4): " +
str1.equalsIgnoreCase(str4)); //5. (case-insensitive) what's the result?
    }
}
```

Result :

1) **str1 == str2:**
   Since str1 and str2 are string literals with the same content, they refer to the same object in the string pool.
   Result: true

2) **str1 == str3:**
   Here, str3 is created using the new keyword, which creates a new String object in the heap, not in the string pool. Even though str1 and str3 have the same content, they refer to different objects.
   Result: false

3) **str1.equals(str3):**

The equals() method compares the content of the strings, not the memory references. Since str1 and str3 have the same content ("Hello"), this will return true.

Result: true

4) **str1.equals(str4):**

The equals() method is case-sensitive, so comparing "Hello" (str1) and "hello" (str4) will return false because of the difference in case.

Result: false

5) **str1.equalsIgnoreCase(str4):**

The equalsIgnoreCase() method compares strings without considering case, so "Hello" and "hello" will be considered equal.

Result: true

Output

```
str1 == str2: true
str1 == str3: false
str1.equals(str3): true
str1.equals(str4): false
str1.equalsIgnoreCase(str4): true
```

Case 2:

```
public class IntegerComparisonExample {
  public static void main(String[] args) {

//Mention what's the result in 1, 2, 3,4 and 5
    // Primitive int
    int int1 = 100;
    int int2 = 100;
```

```
    // Integer objects
    Integer intObj1 = 100;
    Integer intObj2 = 100;
    Integer intObj3 = new Integer(100);
    Integer intObj4 = new Integer(200);

    // Using == with primitive int
    System.out.println("int1 == int2: " + (int1 == int2)); // 1. (compares
values)

    // Using == with Integer objects (within -128 to 127 range)
    System.out.println("intObj1 == intObj2: " + (intObj1 == intObj2)); // 2.
(cached objects)

    // Using == with Integer objects (new instance)
    System.out.println("intObj1 == intObj3: " + (intObj1 == intObj3)); // 3.
(different instances)

    // Using equals() with Integer objects
    System.out.println("intObj1.equals(intObj3): " +
intObj1.equals(intObj3)); // 4. (same content)
    System.out.println("intObj1.equals(intObj4): " +
intObj1.equals(intObj4)); // 5. (different content)
  }
}
```

Result:

1) **int1 == int2:**
   Since int1 and int2 are primitive int types, the == operator compares their values directly. Both int1 and int2 have the value 100, so this comparison will return true.
   Result: true

2) **intObj1 == intObj2:**
   Integer objects within the range -128 to 127 are cached by the JVM. Therefore, intObj1 and intObj2, both holding the value 100, point to the same object in memory, so the == operator will return true.
   Result: true

3) **intObj1 == intObj3:**
   Here, intObj3 is created using the new keyword, which creates a new Integer object in the heap, different from the one referenced by intObj1. Therefore, intObj1 and intObj3 refer to different objects, so the == comparison will return false.
   Result: false

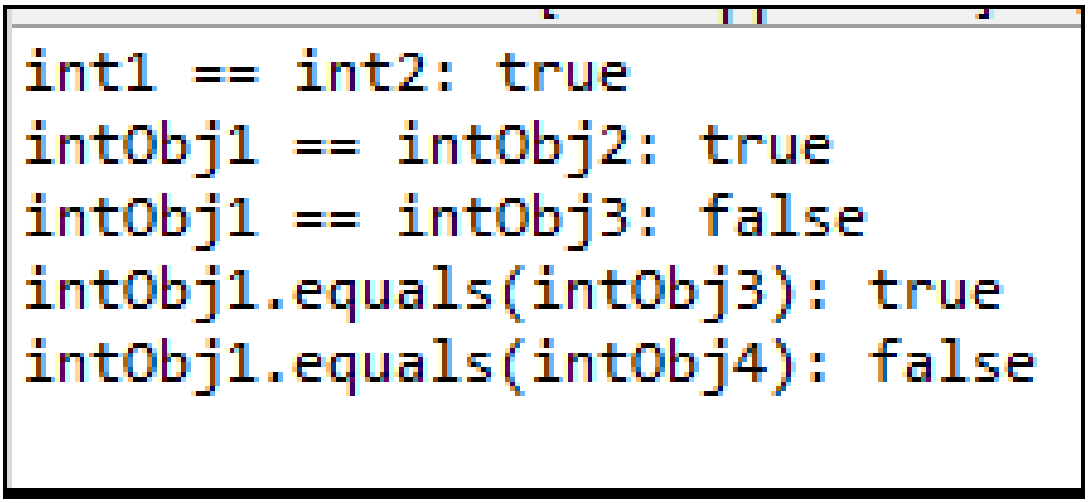
4) **intObj1.equals(intObj3):**
   The equals() method compares the values inside the Integer objects. Since intObj1 and intObj3 both contain the value 100, this comparison will return true.
   Result: true

5) **intObj1.equals(intObj4):**

The equals() method compares the values inside the Integer objects. intObj1 contains 100, while intObj4 contains 200, so this comparison will return false.

Result: false

Output



Case 3:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TryWithResourcesExample {
//Eliminating finally block to close resources.

  public static void main(String[] args) {
    // File path (adjust the path as needed)
```

```
    String filePath =
"C:\\Users\\dhanapal.m\\eclipse-workspace\\sample\\src\\tesst\\Input.
txt";

    // Traditional try-with-resources block
    try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
      String line;
      while ((line = reader.readLine()) != null) {
        System.out.println(line);
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Result :

How Basic I/O Resources Are Closed

In the current code, I'm using a **try-with-resources** statement to handle the closing of I/O resources like `BufferedReader` and `FileReader`. This statement ensures that as soon as the try block finishes (whether it completes normally or due to an exception), these resources are automatically closed. This eliminates the need for a `finally` block that would traditionally be used to manually close resources.

Differences from Earlier Code (`copyBytes.java`)

Earlier Code:

- In the earlier code, I used a regular try block with a `finally` block to manually close the resources. The `finally` block ensures that the resources are closed even if an exception occurs during the execution of the try block.
- This approach required extra code and careful handling to avoid resource leaks.

Current Code:

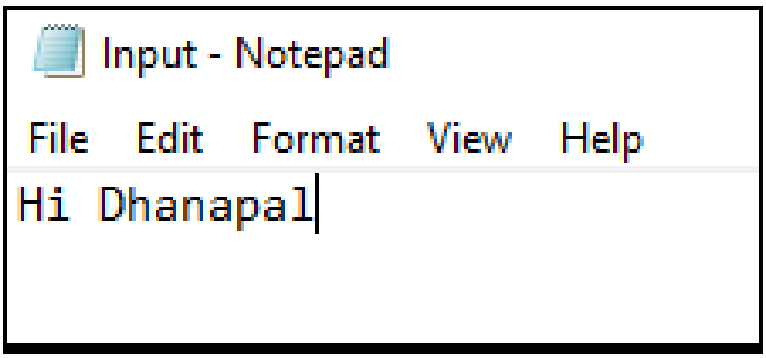
- In the current implementation, I've replaced the manual resource management with a try-with-resources statement. This automatically manages the closing of the `BufferedReader` and `FileReader`, making the code cleaner and less error-prone.

Key Difference

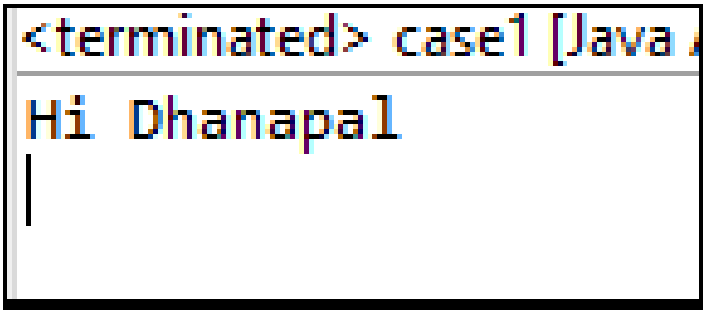
Resource Management:

- **Earlier Code:** Resources were manually closed using a `finally` block. This approach worked, but it required more code and was more prone to errors if not handled properly.
- **Current Code:** Resources are automatically closed by the try-with-resources statement. This makes the code simpler, more reliable, and reduces the risk of resource leaks.

Input.txt



Output:



Case 4:

```
 import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

public class SetExample {
   public static void main(String[] args) {
      // Set 1. What's the order of elements?
      Set<String> hashSet = new HashSet<>();
      hashSet.add("Banana");
      hashSet.add("Apple");
```

```
        hashSet.add("Orange");
        hashSet.add("Grapes");

        System.out.println("HashSet: " + hashSet);

        // LinkedHashSet 2. What's the order of elements ?
        Set<String> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add("Banana");
        linkedHashSet.add("Apple");
        linkedHashSet.add("Orange");
        linkedHashSet.add("Grapes");

        System.out.println("LinkedHashSet: " + linkedHashSet);

        // TreeSet 1. What's the order of elements ?
        Set<String> treeSet = new TreeSet<>();
        treeSet.add("Banana");
        treeSet.add("Apple");
        treeSet.add("Orange");
        treeSet.add("Grapes");

        System.out.println("TreeSet: " + treeSet);
    }
}
```

Result:

1) **HashSet:**

   Order of Elements: No guaranteed order. The HashSet does not maintain any order of elements. The elements could be printed in any order depending on their hash codes.

2) **LinkedHashSet:**

   Order of Elements: Insertion order. The LinkedHashSet maintains the order in which the elements were inserted.

3) **TreeSet:**

   Order of Elements: Natural ordering (sorted order). The TreeSet sorts the elements according to their natural order (for strings, it is lexicographical order).

Output:

```
HashSet: [Apple, Grapes, Orange, Banana]
LinkedHashSet: [Banana, Apple, Orange, Grapes]
TreeSet: [Apple, Banana, Grapes, Orange]
```