

Week 3 Assignment

Name :Siddarth S

Date : 10/08/2024

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.

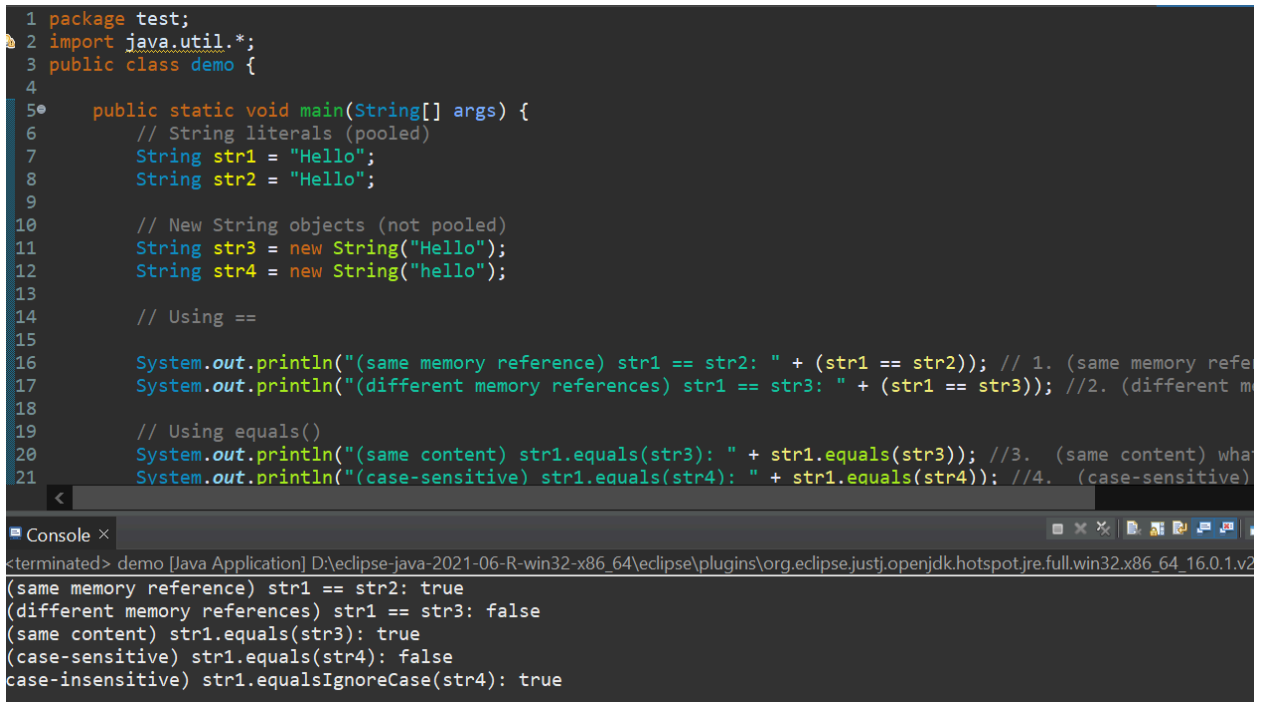
Result: false

5) str1.equalsIgnoreCase(str4):

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

Result: true

Output ScreenShot:



```
1 package test;
2 import java.util.*;
3 public class demo {
4
5     public static void main(String[] args) {
6         // String literals (pooled)
7         String str1 = "Hello";
8         String str2 = "Hello";
9
10        // New String objects (not pooled)
11        String str3 = new String("Hello");
12        String str4 = new String("hello");
13
14        // Using ==
15
16        System.out.println("(same memory reference) str1 == str2: " + (str1 == str2)); // 1. (same memory reference)
17        System.out.println("(different memory references) str1 == str3: " + (str1 == str3)); // 2. (different memory references)
18
19        // Using equals()
20        System.out.println("(same content) str1.equals(str3): " + str1.equals(str3)); // 3. (same content) what about str1.equals(str2)?
21        System.out.println("(case-sensitive) str1.equals(str4): " + str1.equals(str4)); // 4. (case-sensitive) what about str1.equalsIgnoreCase(str4)?
22    }
23 }
```

Console ×

```
<terminated> demo [Java Application] D:\eclipse-java-2021-06-R-win32-x86_64\eclipse\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_16.0.1.v20210519-1\jre\bin\java.exe
(same memory reference) str1 == str2: true
(different memory references) str1 == str3: false
(same content) str1.equals(str3): true
(case-sensitive) str1.equals(str4): false
(case-insensitive) 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.

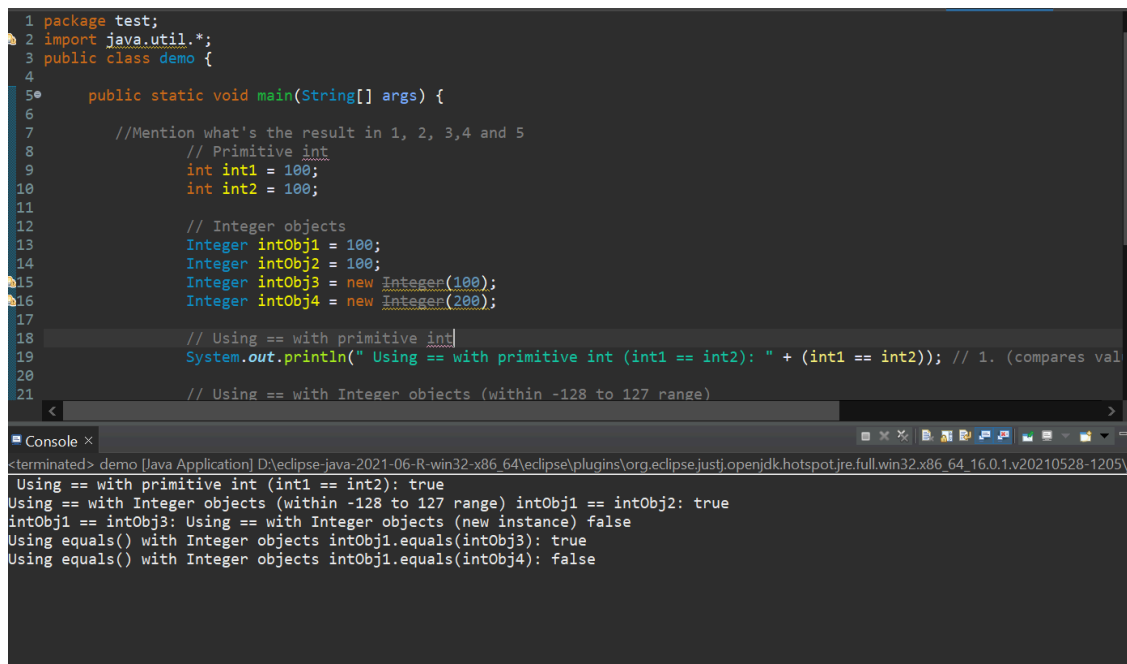
Result: true

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 ScreenShot:



```
1 package test;
2 import java.util.*;
3 public class demo {
4
5     public static void main(String[] args) {
6
7         //Mention what's the result in 1, 2, 3,4 and 5
8         // Primitive int
9         int int1 = 100;
10        int int2 = 100;
11
12        // Integer objects
13        Integer intObj1 = 100;
14        Integer intObj2 = 100;
15        Integer intObj3 = new Integer(100);
16        Integer intObj4 = new Integer(200);
17
18        // Using == with primitive int
19        System.out.println(" Using == with primitive int (int1 == int2): " + (int1 == int2)); // 1. (compares val
20
21        // Using == with Integer objects (within -128 to 127 range)
```

Console

```
<terminated> demo [Java Application] D:\eclipse-java-2021-06-R-win32-x86_64\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.16.0.1.v20210528-1205\
Using == with primitive int (int1 == int2): true
Using == with Integer objects (within -128 to 127 range) intObj1 == intObj2: true
intObj1 == intObj3: Using == with Integer objects (new instance) false
Using equals() with Integer objects intObj1.equals(intObj3): true
Using equals() with Integer objects intObj1.equals(intObj4): false
```

Case 3:

Code:

```
package test;

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\\\\siddarth.s\\\\\\eclipse-workspace\\\\\\Training\\\\\\src\\\\\\tes
t\\\\\\xanad.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();
        }
    }
}
```

```
}  
}
```

Output ScreenShot:

```
1 package test;  
2  
3 import java.io.BufferedReader;  
4  
5  
6  
7 public class TryWithResourcesExample {  
8 //Eliminating finally block to close resources.  
9  
10 public static void main(String[] args) {  
11 // File path (adjust the path as needed)  
12 String filePath = "C:\\\\Users\\\\siddarth.s\\\\eclipse-workspace\\\\Training\\\\src\\\\test\\\\xanad.txt";  
13  
14 // Traditional try-with-resources block  
15 try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {  
16 String line;  
17 while ((line = reader.readLine()) != null) {  
18 System.out.println(line);  
19 }  
20 } catch (IOException e) {  
21 e.printStackTrace();  
22 }  
23 }  
24 }  
25
```

< Problems Javadoc Declaration Console x

<terminated> TryWithResourcesExample [Java Application] D:\\eclipse-java-2021-06-R-win32-x86_64\\eclipse\\plugins\\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_...
hello siddarth

TryWithResourcesExample.java xanad.txt x

```
1 hello siddarth
```


Solution:

In the `TryWithResourcesExample` code, basic I/O resources like `BufferedReader` and `FileReader` are getting closed automatically by using the **try-with-resources** statement. This feature was introduced in Java 7, and it simplifies resource management by ensuring that any resources opened within the `try` block are closed automatically when the block is exited, either normally or because of an exception.

Key Points:

1. **Automatic Resource Closure:** The `try-with-resources` block makes sure that `BufferedReader` and `FileReader` are automatically closed without needing to explicitly call `close()` in a `finally` block.
2. **Simplified Code:** This approach eliminates the need for a `finally` block to close the resources, resulting in cleaner and more readable code.

Difference from the Traditional Approach (`copyBytes.java`):

- In the traditional approach (before Java 7), I had to use a `finally` block to manually close the resources. This was necessary to avoid resource leaks, but it made the code more verbose and error-prone.
- Now, with the `try-with-resources` statement, the code is more concise, and there's no risk of forgetting to close the resources since it's handled automatically.

So, the `try-with-resources` feature not only reduces boilerplate code but also ensures better resource management.

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:

In the `SetExample` code, the order of elements changes depending on which `Set` implementation is used:

1. HashSet:

- **Order of Elements:** The elements in a `HashSet` are unordered, so there's no guaranteed order when I iterate over them.
- **Example Output:** The order could be [Banana, Apple, Orange, Grapes], but it might vary. The key point is that the order is unpredictable.

2. LinkedHashSet:

- **Order of Elements:** The elements in a `LinkedHashSet` maintain the order in which they were added.
- **Example Output:** The order will be exactly as I inserted them: [Banana, Apple, Orange, Grapes].

3. TreeSet:

- **Order of Elements:** The elements in a TreeSet are sorted according to their natural order, which for strings is alphabetical.
- **Example Output:** The order will be [Apple, Banana, Grapes, Orange], sorted alphabetically.

So, HashSet doesn't guarantee any order, LinkedHashSet keeps the insertion order, and TreeSet sorts the elements alphabetically.

Output:

```
4
5 public static void main(String[] args) {
6     // Set 1. What's the order of elements?
7     Set<String> hashSet = new HashSet<>();
8     hashSet.add("Banana");
9     hashSet.add("Apple");
10    hashSet.add("Orange");
11    hashSet.add("Grapes");
12
13    System.out.println("HashSet: " + hashSet);
14
15    // LinkedHashSet 2. What's the order of elements ?
16    Set<String> linkedHashSet = new LinkedHashSet<>();
17    linkedHashSet.add("Banana");
18    linkedHashSet.add("Apple");
19    linkedHashSet.add("Orange");
20    linkedHashSet.add("Grapes");
21
22    System.out.println("LinkedHashSet: " + linkedHashSet);
23
24    // TreeSet 1. What's the order of elements ?
25
```

<

Console ×

<terminated> demo [Java Application] D:\eclipse-java-2021-06-R-win32-x86_64\eclipse\plugins\org.ec
HashSet: [Apple, Grapes, Orange, Banana]
LinkedHashSet: [Banana, Apple, Orange, Grapes]
TreeSet: [Apple, Banana, Grapes, Orange]