Department Mathematik/Informatik, Abteilung Informatik
Software & System Engineering
Weiterführende Konzepte der Programmierung, SS 2025

Niklas Bockholt, Chiara Hafner, Alicja Bezpalko,
Jiyan Alis, Dilara Günay,
Simon Wolf, Adrian Bajraktari, Dr. Mersedeh Sadeghi

Homework 4. Abstract Classes, Wrapper Classes, Strings

Registration Deadline: 04.05.2025, 23:59
Hand-in Deadline: 07.05.2025, 23:59

Exercise 1. Discount Drama

Create:

- `abstract class Article` with `int` attributes `discountThreshold` and `householdLimit`. It also defines two abstract methods:
  - `int getBulkDiscount()`
  - `boolean showWarning()`
- Example: If `discountThreshold` = 5 for the article "salt", you receive a bulk discount when buying at least 5 packs of salt. If `householdLimit` = 7 for salt, then buying 8 or more packs exceeds the household-appropriate quantity.
- `abstract class GroceryItem extends Article` with a `boolean` attribute `isPopular`.
- `abstract class PopularGrocery extends GroceryItem` with `int` attributes `popularityLevel` and `quantity`. It includes:
  - A constructor `PopularGrocery(int popularityLevel)` which sets `isPopular = true` and sets `this.popularityLevel`.
  - Implementation of `showWarning()`, which returns `true` if `quantity > householdLimit`.
  - Setter for `quantity`.
- `class Milk extends PopularGrocery`. It sets:
- It has a constructor `Milk(int quantity, int discountThreshold, int popularityLevel)`.
  - `this.discountThreshold = discountThreshold`
  - `this.quantity = quantity`
  - `this.householdLimit = 20`

  It implements `getBulkDiscount()` to return 12 if `quantity >= discountThreshold`, otherwise 0.
- `class Flour extends PopularGrocery`. It sets:
- It has a constructor `Flour(int quantity, int discountThreshold, int popularityLevel)`.

- `this.quantity = quantity`
- `this.discountThreshold = discountThreshold`
- `this.householdLimit = 15`

It implements `getBulkDiscount()` to return 5 if `quantity >= discountThreshold`, otherwise 0.

Exercise 2. Letter Soup Deluxe

Write a program that checks whether two strings are **anagrams** of each other — that is, they must contain the same letters in any order.

- In your `main` method:
  - Use the `Scanner` class to read two strings from the user.
  - Process the strings to ignore capitalization and spaces before checking for anagrams.
- Write a method `boolean isAnagram(String a, String b)` that returns `true` if the two strings are anagrams.
  - Implement a helper method `int[] countLetters(String s)` which counts how many times each letter (a–z) appears in the string.
  - Use two such arrays to compare the letter frequencies of `a` and `b`.
- Letter count output:
  Write a method `void printLetterCounts(String s)` that prints only the letters used in the input string, one per line, in the following format:
  `a: 2`
  `e: 1`
  `r: 3`

You may use the following as a starting point:

```
import java.util.Scanner;

public class AnagramChecker {

    /**
     * Returns true if the two strings are anagrams of each other.
     */
    public static boolean isAnagram(String a, String b) {
        // TODO: implement using countLetters()
        return false;
    }

    /**
     * Helper method to count how many times each letter (a-z) appears.
     */
    public static int[] countLetters(String s) {
        int[] counts = new int[26];
        // TODO: fill array with letter frequencies
```

```
        return counts;
    }

    /**
     * Print only letters that appear in the string with their counts.
     */
    public static void printLetterCounts(String s) {
        // TODO: implement nicely formatted output
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Get two strings from the user
        System.out.print("Enter the first string: ");
        String word1 = scanner.nextLine();

        System.out.print("Enter the second string: ");
        String word2 = scanner.nextLine();

        // TODO: Process the strings here (ignore spaces, case)

        System.out.println("Are they anagrams? " + isAnagram(word1, word2));
        printLetterCounts(word1);
    }
}
```

- Comparing Strings: Explain the difference between == and `equals()` when comparing `String` objects in Java. Provide a short example showing why == may not be reliable.
- Consider the following code snipplet and explain why this comparison returns `true`.
- What happens if you create the second string using `new String("abc")` instead?

```
String string1 = "abc";
String string2 = "abc";
System.out.println(string1 == string2);  // true
```

Exercise 3. Smart Switch

Design a tiny smart-home device exercise focusing on Java **interfaces**.

- **Create interface `Switchable`**
  - `void turnOn()`, `void turnOff()`, `boolean isOn()`
- **Create interface `Dimmable`**
  - Constant `int MAX_BRIGHTNESS = 100`
  - `void setBrightness(int level)` (ensure the brightness level stays between 0 and `MAX_BRIGHTNESS`)

    - `int getBrightness()`
    - Default method `void dimToHalf()` that sets brightness to `MAX_BRIGHTNESS / 2`
- **class** `Lamp` implements both interfaces with fields `boolean on` and `int brightness`.
    - `setBrightness` only changes the value when the lamp is on.
- **Demo**: In `main` create a lamp, turn it on, set brightness to 80, call `dimToHalf()`, then print the brightness.

You may start with:

```
public interface Switchable {
    // TODO: implement
}

public interface Dimmable {
    // TODO: implement
}

public class Lamp implements Switchable, Dimmable {

    @Override
    public void turnOn() {
        // TODO: implement
    }

    @Override
    public void turnOff() {
        // TODO: implement
    }

    @Override
    public boolean isOn() {
        // TODO: implement
        return false;
    }

    @Override
    public void setBrightness(int level) {
        // TODO: implement
    }

    @Override
    public int getBrightness() {
        // TODO: implement
        return 0;
    }

    public void dimToHalf() {
        // TODO: implement
```

```
    }
}

public class LampDemo {

    public static void main(String[] args) {
        Lamp lamp = new Lamp();

        // turn on, set brightness, dim to half, print result
        lamp.turnOn();
        lamp.setBrightness(80);
        lamp.dimToHalf();
        System.out.println("Brightness: " + lamp.getBrightness());
    }
}
```

1. Unterschied zwischen `==` und `equals()`

Aufgabe2:

- `==` **Operator**:
  - Vergleicht die Speicheradressen (Referenzen) der Objekte
  - Prüft, ob beide Variablen auf dasselbe Objekt im Memory zeigen
  - Nicht geeignet für inhaltlichen String-Vergleich
- `equals()` **Methode**:
  - Vergleicht den tatsächlichen Inhalt der Strings
  - Prüft Zeichen für Zeichen ob die Strings identisch sind
  - Korrekte Methode für String-Inhaltsvergleich

2. Beispiel warum `==` unzuverlässig ist

java Copy Download

```java
String a = "hello";
String b = new String("hello");
String c = "hello";

System.out.println(a == b);  // false (unterschiedliche Objekte)
System.out.println(a == c);  // true (selbes Objekt im String-Pool)
System.out.println(a.equals(b)); // true (gleicher Inhalt)
```

3. Warum das gezeigte Beispiel `true` zurückgibt

java Copy Download

```java
String string1 = "abc";
String string2 = "abc";
System.out.println(string1 == string2); // true
```

Grund: Beide Strings werden aus dem **String-Pool** genommen. Java optimiert String-Literale und verwendet dasselbe Objekt für identische Literale.

4. Mit `new String("abc")`

java Copy Download

```java
String string1 = "abc";
String string2 = new String("abc");
System.out.println(string1 == string2); // false
```

Grund: `new` erzwingt die Erstellung eines neuen String-Objekts im Heap, außerhalb des String-Pools.

Visualisierung des Memory:

Copy Download

```
String-Pool: ["abc"] <-- string1, string2 (im ersten Beispiel)
Heap:        [new String("abc")] <-- string2 (im zweiten Beispiel)
```

Best Practices:

1. **Immer** `equals()` für String-Vergleiche verwenden:

   java Copy Download

   ```java
   if (string1.equals(string2)) { ... }
   ```

2. Für case-insensitive Vergleiche:

   java Copy Download

   ```java
   if (string1.equalsIgnoreCase(string2)) { ... }
   ```

3. **String-Pool Nutzung** mit `intern()`:

   java Copy Download

   ```java
   String s = new String("abc").intern(); // holt aus Pool oder fügt hinzu
   ```

Warum gibt es diesen Unterschied?

- **Performance-Optimierung**: Der String-Pool vermeidet Duplikate von Literalen
- **Objektidentität vs. Wertgleichheit**: `==` prüft Objektidentität, `equals()` Wertgleichheit

Diese Unterscheidung ist fundamental in Java und gilt für alle Objekttypen (nicht nur Strings), während `==` bei primitiven Datentypen (int, char etc.) den Wert vergleicht.