University of Zurich[UZH]

s.e.a.l. software evolution & architecture lab

Informatics I – EProg HS15

Exercise 05

1 Task: Information Hiding

1.1 Learning Objectives

1. Translate simple UML class diagrams into code.
2. Know the concepts of *Information Hiding* and *Encapsulation*.
3. Modify the internals of a class while maintaining its public interface.

1.2 Assignment

Read chapter 5.2 about *Information Hiding and Encapsulation* in the literature (page 293ff in Savitch 6th Edition) to become familiar with these concepts and UML class diagrams (page 320f in Savitch 6th Edition).

a) UML into Code

Implement a class called `Number` that fulfills the self-explanatory interface described by the UML class diagram in Figure 1.

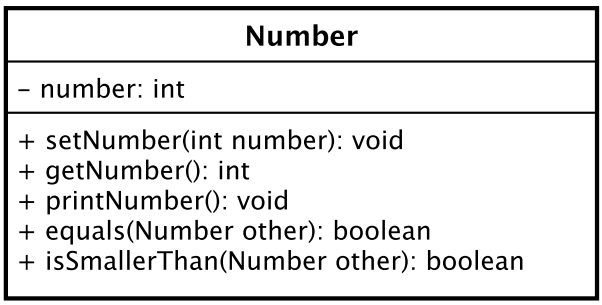

Figure 1: UML class diagram

Solution:

```java
public class Number {
   private int number = 0;


   public void setNumber(int number) {
      this.number = number;
   }

   public int getNumber() {
      return this.number;
   }

   public void printNumber() {
      System.out.println(number);
   }

   public boolean equals(Number other) {
      return this.number == other.number;
   }

   public boolean isSmallerThan(Number other) {
      return this.number < other.number;
   }

}
```

Listing 1: `Number.java`

b) Test Driver

Write a test driver called `NumberTestDriver` that creates two instances of the `Number` class and invokes each method from `Number` at least once. Print the return values to the console.

While writing the test driver we recognized that it would be helpful to have some additional methods. Additionally invoke the following methods from the test driver:

1. `isSmallerThanOrEquals(Number other)`
2. `isLargerThan(Number other)`
3. `isLargerThanOrEquals(Number other)`

Notice that we invoke methods which do not exist yet. It is often a good idea to write the client code (e.g. test driver that invokes the method) first and subsequently implement the actual methods. This helps us to focus on the *what* when defining the interface because we think about the *how* later during the implementation.

Solution:

```java
public class NumberTestDriver {

   public static void main(String[] args) {
      Number x = new Number();
      Number y = new Number();
      x.setNumber(4);
      y.setNumber(4);

      x.printNumber();
      System.out.println(" compared to " + y.getNumber());
      System.out.println(x.equals(y));
      System.out.println(x.isSmallerThan(y));
      System.out.println(x.isSmallerThanOrEquals(y));
      System.out.println(x.isLargerThan(y));
      System.out.println(x.isLargerThanOrEquals(y));
   }

}
```

Listing 2: `NumberTestDriver.java`

c) Extend the Number Class

The mathematical properties of the comparison operators allow for an easy implementation of the additional methods from above. Implement them based on the already present methods in the `Number` class.

Hint: You might want to temporarily comment out some lines in the test driver in order to verify your implementation step-by-step.

Solution:

```java
public class Number {
   ...

   public boolean isSmallerThanOrEquals(Number other) {
      return isSmallerThan(other) || equals(other);
   }

   public boolean isLargerThan(Number other) {
      return !isSmallerThan(other) && !equals(other);
   }

   public boolean isLargerThanOrEquals(Number other) {
      return isLargerThan(other) || equals(other);
   }

}
```

Listing 3: `Number.java` additions

d) Modify Implementation

The requirements have changed and you have to modify your implementation. However, the public interface of the `Number` class must remain backwards compatible. In order to verify this behavior you should't change your `NumberTestDriver` anymore.

The new implementation should save memory[1] and therefore use a variable of type `byte` (1 byte) instead of one single `int` (4 bytes). Prevent overflows within the setter method by checking whether the number is too small or too large to be stored in a `byte` before you typecast. Reset the number value to `-1` if an overflow occurs.

Solution:

[1] *Don't do this* in real world until you *really* know that you need it and you *really* know what you are doing. The Java memory layout and other internal intricacies might let your "optimized" code perform even worse than before.

```java
public class Number {
   private byte number = 0;


   public void setNumber(int number) {
      if (number < -128) {
         System.out.println(number + " is too small to store!");
         this.number = -1;
      } else if (number > 127) {
         System.out.println(number + " is too large to store!");
         this.number = -1;
      } else {
         this.number = (byte) number;
      }
   }

   ...

}
```

Listing 4: `Number.java` modifications

e) Verify Interface Integrity

Check whether your `NumberTestDriver` from above still works as expected.

Remember the *Number* class from exercise 2 within task 1.2.b) in Listing 2. Write a test driver called `NumberTestDriverOld` and check whether this code works with your `Number` implementation.

Solution:

```
1  public class NumberTestDriverOld {
2
3     public static void main(String[] args) {
4        Number x = new Number();
5        Number y = new Number();
6        x.setNumber(1);
7        y.setNumber(3);
8
9        x = y;
10
11       x.printNumber();
12       y.printNumber();
13
14       x.setNumber(5);
15       y.setNumber(7);
16
17       x.printNumber();
18       y.printNumber();
19    }
20
21 }
```

Listing 5: `NumberTestDriverOld.java`

2 Task: 1D Array & Random

2.1 Learning Objectives

1. Learn how to use arrays and operate with them.
2. Practice `while` loops.
3. Use the Java API.

2.2 Assignment

a) Dice Game

Alice likes playing a dice game wherein you count the number of dice rolls until the sum reaches 30 points. She wants to simulate this game with a Java program.

1. Create a class called `DiceGame` which simulates this game with the `play()` method. The result of each dice roll must be stored in an array for later reference. Use a private helper method called `rollDice()` that returns the result of a single dice roll. Look up in the Java API how the class `Random` can be used. Ensure that you can play the game multiple times.
2. The method `printSummary()` should summarize the last game. Example:

```
[1] = 6 (6)
[2] = 5 (11)
[3] = 3 (14)
[4] = 6 (20)
[5] = 4 (24)
[6] = 6 (30)
Won after 6 dice rolls.
```

Solution:
Find the solutions in the files:

- `DiceGame.java`
- `DiceGameTestDriver.java`

b) Gambling Game (optional)

Tom offers a deal to Alice: "You pay me 2 bitcoins for each dice roll. If you win the game with 6 or less dice rolls I give you 1000 bitcoins. Do you play?" Alice decides to extend the dice game simulation instead of doing the maths manually.

Create a class `GamblingGame` with setters for `costsPerRoll`, `bonusOnSuccess`, and `maxRollsForSuccess`. The `simulate(int numberOfGames)` method should instantiate a `DiceGame`, play it `numberOfGames` times, and finally print the average profit to the console. Also create a `GamblingGameTestDriver`, configure our scenario, and try the simulation with different parameters.

Solution:
Find the solutions in the files:

- `GamblingGame.java`
- `GamblingGameTestDriver.java`

3 Task: 2D Array & Random

3.1 Learning Objectives

1. Properly use (i.e. create, initialize, iterate over) multi-dimensional arrays.
2. Use the `Random` class to produce random letters.
3. Recognize the benefits of (private) helper methods.

3.2 Assignment

a) Crossword Puzzle

In this task you'll program a small crossword puzzle similar to the ones in puzzle books where you can search for words on a 2-dimensional playground with random letters.

```
1 S K F D I M C B Z G L U
2 W P R O F G A L L W E W
3 Q P E S I X F R O E P V
4 P N E F C S P T K I F X
5 D V S V R E T L P U O D
6 Y H W X E P R O G M S R
```

1. Implement the method `nextRandomLetter()` that returns a random character from A-Z. Exploit the fact that there exists a standard mapping of decimal numbers to so called ASCII characters (A-Z are part of the ASCII characters). The website ascii-code.com lists all mappings from decimal numbers (DEC) to symbols (Since ASCII is a subset of Unicode you can find these mappings also in any Unicode table[2]). According to this table, the relevant mappings are:
   - `65` maps to `A`
   - `66` maps to `B`
   - ...
   - `90` maps to `Z`

   Thus, generate a random integer number in the appropriate range and typecast it to a `char`. Verify the correct behavior of this method with a test driver.
2. Use a multi-dimensional array to represent the game board. Implement a method called `initializeGame(int width, int height)` that sets the dimensions (size) for the board and initializes the 2-dimensional array. Keep in mind that the game should support arbitrary board dimensions.
3. Implement a method called `fillBoard()` that fills the board with random letters.
4. Implement a method called `drawBoard()` that prints the board to the console in a readable manner.
5. Test your code with a test driver.
6. Move the code which necessary to play the game from the test driver into a method called `play(int width, int height)` located in the game class. The test driver should solely call the `play(...)` method. Make all other methods private. Why is this advantageous?

   Solution:

[2] Appendix 7 *Unicode Character Codes* on page 932 in Savitch 6th Edition

It contributes to *Information Hiding* by hiding (i.e. making inaccessible from outside the class) functionality that is only required internally (within the class). This allows to easily change (e.g. rename, restructure) the private methods later. Additionally, the public interface of the class is kept small and thus easier to use for clients (compare `myGame.play(...)` to a situation where the client has to know `initializeGame(...)`, `fillBoardWithMoreVowels()`, `drawBoard()`, and the order of these method calls).

7. You can now play the game by hunting for words in horizontal, vertical, or diagonal direction[3].

b) Improved Letter Generator (optional)

In order to improve the chances that you find a word, vowels should appear more often. An easy way to do so is to produce only vowels in $1/3$ of all cases and fill in an arbitrary letter in the remaining $2/3$ cases[4]. Try out different ratios.

- Consider creating a helper method `nextRandomVowel()` that returns one of the following characters `AEIOU`. A method from the `String` class might be helpful.
- Consider creating a helper method `shouldTakeVowel()` that implements the vowel ratio rule. Simply return true in $1/3$ of the cases and false otherwise.

Solution:
Find the solutions in the files:

- `Game.java`
- `GameTestDriver.java`

[3] You can search with pen and paper on screen. There is nothing to implement.
[4] Ignore the fact that the latter case may also generate vowels in addition to non-vowels

4 Task: Static Variables & Methods

4.1 Learning Objectives

1. Understand the difference between static and non-static variables and methods.
2. Implement and understand a simple usage of such static variables and methods.

4.2 Assignment

a) Theoretical Questions

1. Does a static variable belong to an object or the whole class?
2. How does an instance variable differ from a static variable?
3. How is a static method invoked? On the other hand, how is a non-static (instance) method invoked?

Solution:

1. A static variable is shared between all objects of the same class.
2. An instance variable only belongs to one object and not to each object of a class. An object may not share an instance variable with another object.
3. A static method is invoked on the class using the class name whereas an instance method may only be invoked by an object of the class.

b) Implementation

1. Implement a `Post` class representing a post in e.g. a forum or a social network. Each `Post` is assigned a unique identifier (`id`), which is derived from a shared counter within `Post`. Each `Post`, furthermore, contains a `userName` of the user that posted said `Post` and a `message` representing the posted contribution. Both `userName` and `message` have corresponding setter and getter methods. `id` only provides a getter method as it is set at the time of the `Post's` creation.
2. Implement a static method in `Post` `public static void printPosts(Post[] posts)` that takes a number of Posts within an array and prints them according to the following scheme:
   `id`: `userName` has written: "`message`".
   Example:
   1: Ron has written: "I don't like mondays.".
3. To efficiently search through posts, search engines like to know how many times a word occurred in the whole document collection. To achieve this, implement a static method in `Post` `public static int calculateDocumentFrequency(Post[] posts, String word)` that counts and returns the number of times `word` occurred within the `messages` of `posts`. (To keep it simple, we assume there are no punctuation marks in the post. If you like to, you may also use `String.replace()` or `String.replaceAll()` methods to remove such punctuation marks but this is not required. Use `String.equalsIgnoreCase(String)` to compare the words of the message with `word` or instead, convert both `message` and `word` to lower case and use `String.equals(String)`).

4. Write a test driver that generates an array of 3 `Posts`. Use `printPosts(Post[] posts)` to print all posts in the array. Then, choose a word you would like to count in all posts, calculate it using `calculateDocumentFrequency(Post[] posts, String word)`, and print the number to the console.

Solution:
Find the solutions in the files:

- `Post.java`
- `PostTestDriver.java`