Informatics I – EProg HS15

Exercise 7

1 Task: Variables, Overloading, and Return Values

1.1 Learning Objectives

1. Distinguish instance variables from class variables.
2. Understand method overloading.
3. Revisit return values.

1.2 Assignment

Given an instance of the following class created via `Task1 task1 = new Task1()`:

```
public class Task1 {
   private String firstname = "Johnny";
   private static String surname = "Depp";

   // ...
}
```

a) Variables

1. What is the output of `Task1.printFirstname()`[1]?

```
public static void printFirstname() {
   System.out.println(this.firstname);
}
```

2. What is the output of `task1.printName()`[1]?

```
public void printName() {
   System.out.println(this.firstname + Task1.surname);
}
```

[1]Assume that this method is part of the class from above.

b) Overloading

Assume the following methods are part of the class from above:

```java
public void prefix(String s) {
   System.out.println(s + s + firstname);
}

public void prefix(char c) {
   System.out.println(c + firstname);
}

public void prefix(double d) {
   System.out.println((d * 2) + firstname);
}

public void prefix(int i) {
   System.out.println((i * 1000) + firstname);
}
```

What is the output of the following method calls?

1. `task1.prefix('4');`

2. `task1.prefix(4);`

3. `byte b = 4; task1.prefix(b);`

   Additionally, what would happen if there is an implementation for `prefix(byte b)` available?

4. `task1.prefix("4");`

c) Return Value

What is the output and what is the return value of the following method calls?

1. `printHello1()`

```java
public void printHello1() {
   System.out.println("Hello1");
   String result = "Hello2";
}
```

2. `printHello2()`

```java
public void printHello2() {
   if (true) {
      return;
   }
   System.out.println("Hello3");
}
```

2 Task: Static Variables, Constructors, and Overloading

2.1 Learning Objectives

1. Learn how to use static variables and static methods.
2. Use multiple constructors.
3. Practice overloading.

2.2 Assignment

The university needs a system to manage their students. Therefore, you should implement a `Student` class that is able to keep track of the name and matriculation number of a student and can return a string representation in different formats.

a) Preparation

In order to simplify the implementation of the `Student` class you should first write a class called `StringUtils` that implements the static method `repeat` which follows the rules of the corresponding Javadoc:

```
/**
* <p>Returns padding using the specified delimiter repeated
* to a given length.</p>
*
* <pre>
* StringUtils.repeat('e', 0)  = ""
* StringUtils.repeat('e', 3)  = "eee"
* StringUtils.repeat('e', -2)  = ""
* </pre>
*
*
* @param ch character to repeat
* @param repeat number of times to repeat char, negative treated as zero
* @return String with repeated character
*/
```

Listing 1: Javadoc of the `repeat` method

Use the class StringBuilder from the Java API for your implementation.

b) The `Student` Class

Every student must have a name and a matriculation number in the format `XX-XXX-XXX` where `X` is a number in the range 0-9. Students that already have a matriculation number assigned can be instantiated via their name and their existing matriculation number. However, it should be possible to instantiate students with solely their name. In this case, a unique matriculation number must be created automatically. Use the following helper method to create a matriculation number based on a unique id you have to provide:

```java
private static String matriculationNumberForId(int id) {
   return String.format("%1$ty-%2$03d-%3$03d", new Date(),
                                                    id / 1000, id % 1000);
}
```

Additionally, write 5 different `toString` methods:

1. Takes no parameters and returns the matriculation number followed by the name which is separated by a single whitespace from the matriculation number.
2. Basically does the same as 1. but takes a single character as parameter to determine the separation character between the matriculation number and name.
3. Basically does the same as 1. but takes the number of whitespace characters that should separate the matriculation number from the name as parameter.
4. Combines 2. and 3. by taking the separation character and number of horizontal space (i.e. padding) as parameters. Thus, this method allows to specify how many times a specific character should be repeated to separate the matriculation number from the name.
5. Exactly the same as 4. but with inverted order of method parameters.

Hint: Try to reuse as much code (i.e. methods) as possible.

c) The `StudentTestDriver`

Write a test driver for the `Student` class that instantiates 4 students. Only one of them should be created based on an existing matriculation number. For the first student, call each `toString` method at least once and print the output to the console. For the other students, one `toString` method is sufficient. Before creating the 4th student, instantiate `1000` students in a loop (you don't have to store them) to demonstrate the generation of unique ids.

d) Questions

1. Why is it more useful to provide the `repeat` method in the `StringUtils` class instead of including it into the `Student` class?

2. What are the advantages of reusing existing methods? (Mention at least 2 of them)

3. Why does the `toString` method 4. and 5. (see above) with inverted parameters work? Is it a good idea to use this approach here? Explain your decision!

3 Task: An Address Book Application - Part 1

3.1 Assginment

In this assignment you should implement an application that provides the functionality of a classical address book application, such as adding and deleting contacts or searching for existing contacts. In the coming weeks we will successively extend the application. In the end, your application will provide a graphical user interface and will be able to persistently store the contacts.

In this first step we start with the basic modules for the implementation of the address book.

a) Contacts

1. Map the subsequent statements onto classes, attributes, and methods: A contact has a firstname and lastname, a date of birth, a phone number, an email address and a postal address. The postal address consists of street, postal code, and city. Firstname and lastname have to be initialized as soon as a contact is created. All the other information is facultative and can be added later. If an address is added or edited, the whole address (street, postal code, city) is specified. It is not possible, for example, to only change the postal code.
2. The date of birth should internally be stored in a `java.util.Date` object. However, this object should not be part of the public interface of a contact. Provide a method `void setBirthday(int day, int month, int year)` instead to specify a date of birth.
3. For the internal representation each contact should have a unique number (ID). This number is defined when the contact is created and does NEVER change.[2].
4. Provide a `toString()` method for the class `Contact`. This method should be flexible such that it always returns a reasonable textual representation of the object even if not all attributes are set (i.e. no birth of date). Use the class `java.lang.StringBuilder` to compose the return value. `StringBuilder` allows, in contrast to the class `String`, to subsequently change characterbands. Indeed, this is not relevant for this exercise, but the point is to train the application of the Java API. Hint: The date of birth can be formatted using the functionality of the class `java.text.SimpleDateFormat`.
5. Test the above functionality with a rudimental TestDriver.

b) Address Book

1. When creating a new address book, a name (e.g. "Michael's address book") should be specified as well as the capacity (the number of contacts that can be stored). Internally, the address book should use an array to store the contacts.
2. The address book should provide two different methods to add contacts: A method `void addContact(Contact contact))` to add an already existing contact and a second method `Contact addContact(String firstName, String lastName)` that creates a new contact itself, stores it and returns a reference to it. This allows to easily concatenate method calls, for example:

```
// without concatenation:
Contact contact = addressBook.addContact("Anton", "Muster");
contact.setBirthday(3, 10, 1980);

// with concatenation:
addressBook.addContact("Hans", "Muster").setBirthday(3, 10, 1980);
```

[2]Hint: The method `System.nanoTime()` typically returns a different number for each call (see also Java API)

3. It should also be possible to delete contacts with a method `void deleteContact(Contact contact)`. When deleting a contact, gaps in the contact-`Array` should be automatically eliminated. The class `java.util.Arrays` provides adjuvant functionality to do that.
4. Write a few `Contact findBy*(String value)` methods that allow to search for contacts. For example, the method `Contact findByLastName(String lastName)` should return the first contact that has the given last name.
5. Write a few `void sortBy*()` methods that allows to sort the address book. For example, the method `void sortByLastName()` should sort the entries of the address book by their last name.
6. Write multiple TestDrivers to test the implemented functionality!