Lab 12
BlueJ, Classes and jUnit Revisited II

Fall 2017

1 Introduction

Grading: This lab requires the use of the grading sheet for responses that must be checked by your instructor (marked as Question) AND the submission of some programs to WebCAT (marked as Lab).

In an earlier lab, you learned how to debug a class, specifically a `PersonName` class, and how use jUnit to test the class.

In this lab you will extend your understanding of classes and objects by

- incorporating last week's `PersonName` class into a `Person` class,
- doing some testing with jUnit, and
- investigating what `static` means.

2 Exercises

Fork and clone the `cpsc150l-lab12` repository from the 150 students group on Gitlab.

2.1 Creating the `Person` Class

Later in the course, we will use a `Person` class for another lab, so here we will be begin the process of creating this class. Create a class called `Person` with a private `PersonName name` field and a private `int age` field. Implement the following methods.

1. `public Person()`

   This creates a new `Person` with `"no name"` as its first name and an age of zero.

2. `public Person(String aName, int anAge)`

   This create a new `Person` with a name of `aName` and an age of `anAge`.

3. `public String getName()`

   This getter returns a `String` containing this `Person`'s entire name.

4. `public void setName(String aName)`

   This setter sets the name of this `Person` to `aName`.

5. `public int getAge()`

   This getter returns the age of this `Person`.

6. `public void setAge(int anAge)`

   This setter sets this `Person`'s age to `anAge`.

7. `public boolean canDrive()`

   This method returns whether or not this `Person` can drive. That is, it returns `true` if this `Person`'s age is over 16 years old and `false` if not.

8. `public static boolean canDrive(int anAge)`

   This method returns whether or not a person with an age of `anAge` can drive. That is, if that age is over 16 years old, it returns `true` and `false` otherwise.

For the `canDrive` methods, checkout Section 4.3 for tips.

Exercise 1 Complete

Run:

```
git add .
git commit -m "Completed exercise 1"
git push origin master
```

2.2 Testing the Person Class with jUnit

Create a jUnit test class which minimally creates three different people, one under 16, one equal to 16, and one over 16. This type of testing does boundary condition testing as it investigates how your program functions as the data transitions from one state to another; here from being too young to drive to being eligible to drive.

> ***Question 1:*** *When you have finished coding the jUnit class for testing* `Person`*, show your instructor.*

Exercise 2 Complete

Run:

```
git add .
git commit -m "Completed exercise 2"
git push origin master
```

---

3 Common Mistakes

The solutions to some common mistakes are as follows.

1. Pay close attention to when you need to increment and decrement the name count in your setter methods.
2. Pay attention to spacing in the `getEntireName` method. Specifically, ensure that there is no extraneous whitespace in your output. That is if the first name is `"Matt"`, the middle name is empty, and the last name is `"Smith"`, the output should be `"Matt Smith"`, not `"Matt  Smith"`.
3. When writing jUnit tests, be sure to put the `@Test` annotation before each test declaration.
4. Do **not** delete the original `canDrive` method when you overload it in the last exercise.

4 Tutorial

4.1 Using `String.split(" ")`

Look up the JavaDoc for the `String` class and look at the `String[] split(String regex)` method. We note that the method processes a `String` object based on a rule called a regular expression and returns an array of `String` objects. Furthermore, if we pass `" "` to `String.split` it will create a new array of each word in the original `String` separated by at least one space. An example of its use is as follows.

Code:

```
String yolo = "You only live once";
String[] words = yolo.split(" ");
for (int i = 0 ; i < words.length ; i++)
     System.out.println(words[i]);
```

The previous example will output the following.

Code:

```
You
only
live
once
```

4.2 Using `String.trim()`

Consider the `String`, `"    you only   live  once "`. We see that it has far more whitespace than necessary. One way to get rid of some of this whitespace is to use `String.trim()`. The `String.trim()` method removes all whitespace from the beginning and end of a `String` object. Consider the following code.

Code:

```
String s = "    you only   live  once ".trim();
```

Since `trim()` removes whitespace from the beginning and end of a `String`, `s` is equal to `"you only   live  once"`. Note how `trim()` did not touch the extra whitespace inside of the string.

4.3 Overloading Methods and `static` Methods

Let's investigate two important concepts with a simple exercise. In Exercise 2.1, we need to *overload* the `canDrive` method. To do so, add the following method declaration to the `Person` class while leaving the current `canDrive` in place.

Code:

```
public static boolean canDrive(int age)
```

And now you have two `canDrive` methods in the `Person` class and we have now *overloaded* the `canDrive` method. Instead of testing an object's age, this second version of `canDrive`

grabs the age it will test from the parameter. Furthermore, we will not be able to access any fields within `Person` from this method since it is a *static* method. Static methods are independent of instances of each class and can be called without an attached object.

Another way of viewing this is to right-click the class and actually invoke `Person`'s `canDrive` method. It will prompt you with a request for the parameter value (an age) that it can use to answer the question. Note that this method does not use the age of a specific object, but can be used independent of the creation of objects.

On the other hand, right-click on the `Person` and use the constructor to create a `Person` object. Name it `bob` and give the `name` value as `"Bob"`, and choose an age to enter. You should see an object named `bob` in the lower left corner. Now right-click the object and invoke the `canDrive` method. Note that this method is going to use `bob`s age to determine if he can drive, and does not prompt you for an age.

Two conclusions to observe:

1. You are *overloading* the method, defining two different versions of the method, only differing by the interface. Same as with the constructor for this class.
2. `static` methods do not require you to create an object in order to use it. In this case, the `canDrive` method is available to use as long as you pass it the age. If you want it to respond based on the `Person` object, you must create it and then invoke as you did in the previous step.

In this case we have examined how to invoke these methods graphically, but how do you call this `static` method? A `static` method can be called without creating an object. In order to call the method, you must use the class name.

For example, the following code snippet is `static` as shown by calling `Person.canDrive(17)`.

Code:

```
if (Person.canDrive(17))
     System.out.println("This person can drive");
```

Since `Person` is a class, `Person.canDrive(17)` invokes a *static* method. Wheras the following code is *nonstatic*

Code:

```
Person myPerson = new Person("Bobby Paul Smith", 15);
if (myPerson.canDrive())
     System.out.println("This person can drive");
```

since `myPerson` is an object.