Pset 2 Notes (CSCI E-10B)

General Information

a. Problem set 2 is due Monday, Oct 9, 2023 at 9:00 AM Eastern Time. You can resubmit your assignment as often as desired, but each submission's zip must include every file that you want to be graded.

b. We will deduct 10% for a homework assignment that is turned in up to 3 days late. 20% will be deducted if the homework is more than 3 days late. No homework will be accepted more than 7 days late. The last submission controls the late penalty for the entire assignment.

c. Don't procrastinate! Note, though, that the pset covers material from lecture 5 so you will probably need to defer some questions until you've viewed future lectures and attended or viewed future sections.

d. The course website pages from which problem sets can be retrieved and submitted are easily located via the home page's "Lecture …" buttons.

e. Your Java code must compile. Programs that do not compile will automatically have their score divided by two.

f. Your programs must behave as specified. Do everything the specs say to do. Do not do more than the specs say to do. Precisely follow the directions in the pset (e.g. use the exact filenames specified in the pset), except make adjustments specified by the staff in these notes or on the Ed discussion board.

g. Some of the work is designated as "extra credit." In this course, extra credit points are kept separate from regular credit points. They only come into play at the end of the semester when Dr. Leitner is assigning final grades. If you are on the cusp between, say, a B+ and an A-, extra credit points can influence that decision.[1]

   If the pset says "identify which problem(s) you want treated as extra credit," ignore that! We automatically allocate points to maximize your regular credit score because that contributes the most to your grade in the course.

   If your submission gets a late penalty, then you won't get any extra credit points for that submission.

h. This course emphasizes programming style. See the Ed posts titled *Reasonably-commented programs*, *Javadoc commenting* and *Java Style Guide*. Inadequately-commented or -styled programs will lose points.

Pencil-and-Paper Exercises

The Simple Exercises must be submitted as plain text files. Plain text files can be created in Visual Studio Code, or by using a programmer's editor like Notepad++, Sublime, TextPad, or by using a plain text editor such as Windows Notepad. Answers submitted in another file format such as **.doc**, **.pages**, **.rtf**, **.pdf**, ... will not be accepted.[2]

Nobody should lose points on the Simple Exercises programming questions! After you have thought through the problem and written down your answer, check your work by writing simple programs to verify your answer. Don't submit your test programs; only submit your answer for each problem.

a. Problem 1 (`Cell.txt`) needs to state whether each assignment statement is legal or illegal, given the provided classes. Explain what is wrong with the illegal statements.

[1] The decision is also influenced by your teaching assistant's feedback, so it's a good idea to get to know that person.

[2] Fyi, note that `.java` files are stored as plain text files.

b. Problem 2 (**`Extends.txt`**) needs to state whether the relationship makes sense or not. Like we discussed in section, if **`B`** extends **`A`** then there is an *is a* relationship between **`B`** and **`A`**. That is, any **`B`** *is an* **`A`**. For instance, it makes sense to say that **`Mammal`** extends **`Animal`** because any **`Mammal`** *is an* **`Animal`**. It would not make sense to say that **`Animal`** extends **`Mammal`** because not every **`Animal`** *is a* **`Mammal`**.

c. Problem 3 (**`Pet.java`**): Given the provided **`PetTest`**, **`Dog`**, and **`Cat`** classes, create an abstract **`Pet`** class such that running **`PetTest.java`** produces the specified output. You cannot modify the provided **`PetTest`**, **`Dog`**, and **`Cat`** classes except that it's ok to change the **`Dog`** and **`Cat`** classes' access modifiers if you want to consolidate the **`Pet`**, **`Dog`** and **`Cat`** classes into **`Pet.java`**.

   Put the **`Pet`** class's code in **`Pet.java`**. Do not submit the assignment's code for the **`PetTest`**, **`Dog`**, and **`Cat`** classes if you have them in separate files from **`Pet.java`**.

Programming Problems

Submit complete and correct programs that behave exactly as specified, except make adjustments specified by the course's staff in these notes or on the Ed discussion board.

Your programs' interaction with the user should match what's in the assignment, and when sample cases are shown in the problem set, your program must produce the same output when given the sample case's input.

Whenever you are asked to write a specific method that's not **`main()`**, calling that method must not cause any output to be sent to the terminal unless the instructions say otherwise.

If you are told to provide code that demonstrates your method, you must follow these rules:

1. You must not require your TA to define the test cases.
2. You must not require your TA to read your source code in order to decipher your code's output.

Instead, you need to implement a robust set of self-documenting test cases that convincingly demonstrate your method. Here's how I would do that: For a well-chosen set of test cases, my demo would contain statements analogous to:

```
System.out.printf( "someMethod( %d, %d ) = %d\n", int1, int2,
                                                   someMethod( int1, int2 ) );
```

You will replace "**someMethod**" with the actual name of the method being demonstrated, and you'll adjust the number of arguments, their data types, the method's return type, etc.

a. Problem 4 (**`BarCode.java`**): This is the most complicated problem in the assignment, because it has lots of parts. But each part on its own is pretty straightforward. The assignment provides a lot of detailed direction to which you should pay close attention.

   Begin this problem by "blocking out" the program. For instance, the problem says "You will develop a BarCode class ...", so start with this (what should the class's access modifier be?):

```
?? class BarCode {
};
```

   Then add instance variables named **`myZipCode`** and **`myBarCode`**. What should their access modifier be? Then add the specified constructor, without actually coding that method's body. What should the constructor's signature be? Then add the getter methods, without actually coding the methods' bodies. What should the getter methods' signatures be? Then add the private helper methods, without actually coding the methods' bodies. What should the private helper methods' signatures be?

Finally, now that you have the skeleton in place, save a copy in case I ask to see it, and then go back and fill in the blanks. The assignment gives you some specific help here too. E.g.

*"The constructor takes a zip code as a parameter and stores it in **`myZipCode`**; then calls the **`encode`** method to initialize **`myBarCode`**. … Or … The constructor takes a bar code as a parameter and stores it in **`myBarCode`**; then calls **`decode`** to initialize **`myZipCode`**. Your constructor will need to figure out whether it has been passed a zip code or a bar code. That's easy to do if you consider the length of the **`String`** argument passed."* [3]

Your **`encode`** method should do its error checking via **`isValidZipCode`** and your **`decode`** method should do its error checking via **`isValidBarCode`**.

The assignment specifies that the constructor should "throw an exception" when it has been passed an invalid zip code or an invalid bar code. You learned how to throw an **`IllegalArgumentException`** in the previous problem set, and it's also discussed in this problem set. You can check the values returned by **`encode`** and **`decode`** to figure out when the constructor has been passed an invalid zip code or an invalid bar code.[4]

Your **`encode`** and **`decode`** methods need not implement the complicated bar code encoding algorithm described in the problem. It's much easier for **`encode`** and **`decode`** to leverage a **`String`** array that maps digits to bar code segments. Note that **`isValidBarCode`** can leverage that same **`String`** array.

After you get the program to produce correct output, take a break. Then return to the program and read your code critically, asking yourself "how can I make this better?" Make changes, test them, and repeat this step a few times. This process of stepwise refinement is *essential* to producing quality results.

Do not deviate from the pset's or this hints document's specifications.

b. Problem 5 (**`Ship.java`**, **`ShipTest.java`**): This problem is required for graduate students, and is extra credit for undergrad students.

   The assignment doesn't specify filenames, but I suggest that you place the abstract **`Ship`** class and its subclasses in file named **`Ship.java`**, and that you put your "convincing demonstration of those classes" in a file named **`ShipTest.java`**.

   This problem's instructions are very detailed. I suggest that you first block out the structure of the program as discussed above for **`BarCode.java`**, by precisely parsing the problem's specs. You'll be close to finished after merely blocking out what the problem asks for.

Extra Credit Problem-Solving For Everyone

The guidelines discussed in the beginning of the **Programming Problems** section also apply to this section.

a. Problem 6 (**`King.java`**, **`Chessboard.java`**): This problem is extra credit for everyone. Retrieve **`Chessboard.java`**, **`Piece.java`**, **`Bishop.java`**, and **`Knight.java`** from the Unit 5 Lecture Files section of the Java Resources page. Create **`King.java`** by editing a copy of **`Bishop.java`** or **`Knight.java`** as necessary to accommodate the **`King`**-specific rules for moving. Modify **`Chessboard.java`** to use your **`King`** class. Do not modify **`Piece.java`**, **`Bishop.java`**, or **`Knight.java`**. **`Chessboard.java`** needs to tell the user how to select their desired piece, because it might not be obvious how to specify a **`King`** vs. how to specify a **`Knight`**.

[3] The length of the argument is an approximate, close-enough, rule for making the decision.

[4] The specs say that **`encode`** and **`decode`** should return the empty **`String`** if passed an invalid argument.