Practice Midterm Exam

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the midterm examination.

Exam is open book, open notes, closed computer

The examination is open-book (specifically the course textbook *The Art and Science of Java* and the Karel the Robot coursereader) and you may make use of any handouts, course notes/slides, printouts of your programs or other notes you've taken in the class. You may not, however, use a computer of any kind (e.g., you cannot use laptops on the exam).

Coverage

The midterm exam covers the material presented in class up to and including string processing, which we should finish by Friday, February 7, which means that you are responsible for the Karel material plus Chapters 1-6, 8, 9, and the use of mouse listeners from Chapter 10 (sections 10.1-10.4) from *The Art and Science of Java*.

General instructions

Answer each of the questions included in the exam. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

Each question is marked with the number of points assigned to that problem.

In all questions, you may include methods or definitions that have been developed in the course, either by writing the `import` line for the appropriate package or by giving the name of the method and the handout number, chapter number, or lecture in which that definition appears.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do.

Blank pages for solutions omitted in practice exam

In an effort to save trees, the blank pages that would be provided in a regular exam for writing your solutions have been omitted from this practice exam.

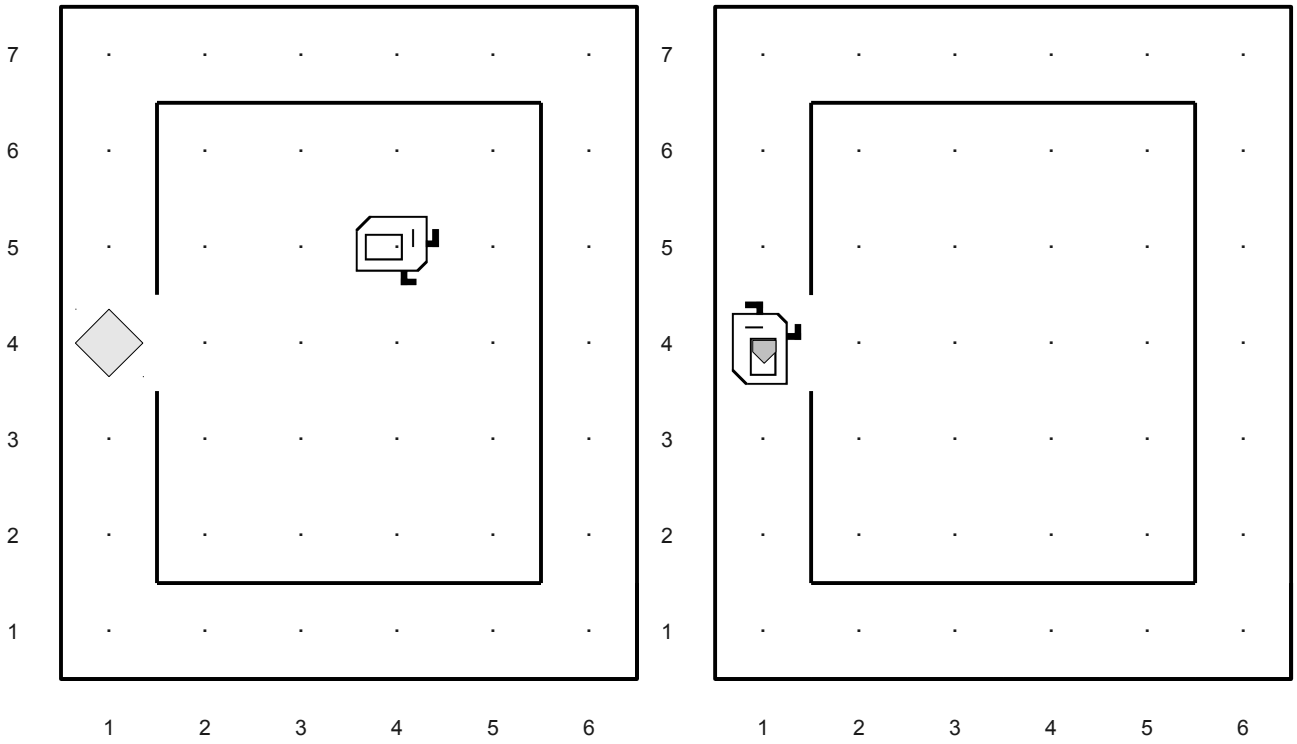
Good Luck!

Problem 1: Karel is Lost! (24 Points)

Karel the Robot has gotten lost in an empty room – can you help guide him out safely?

In this problem, Karel's world consists of a single room, a rectangle with exactly one wall removed as the exit. Immediately outside the exit of the room is a single beeper. Karel is guaranteed to start somewhere within the room, but other than that Karel's initial position and orientation are *unknown*. Your job is to guide Karel out of the room and safely onto the exit beeper.

For example, here is one possible starting world for Karel, along with the desired result:



You should assume the following:

- The dimensions of the room Karel starts in are completely arbitrary.
- The door can be in any of the four walls of the room and anywhere along a wall.
- Karel's position and initial heading within the room are completely arbitrary.
- The only beeper in the world is the one beeper immediately outside the door.
- Karel's final heading does not matter.
- There will always be at least one row or column of space outside each wall of the room, though there may be more.

You are limited to the instructions in the Karel booklet. For example, the only variables allowed are loop control variables used within the control section of a **`for`** loop, and you must not use the **`break`** or **`return`** statements. You may, however, use the **`&&`**, **`||`**, and **`!`** operators in the conditions of **`if`** statements or **`while`** loops. You do not need to worry about efficiency.

Problem Two: Jumbled Java hiJinks (20 Points Total)

(i) Expression Tracing (6 Points)

Compute the value of each of the following Java expressions. If an error occurs during any of these evaluations, write "Error" on that line and explain briefly why the error occurs.

```
1 - 2 - 3 - 4
```
_____

```
(13 / 7) / (7 / 13)
```
_____

```
1 == 2 || 2 / 0 == 3
```
_____

```
"1" + (1 + 1) + 1 + 1 + "1"
```
_____

(ii) Program Tracing **(14 Points)**

The following program is complex and exists solely to test your understanding of parameter passing. What does it print out?

```
import acm.program.*;
public class AbrahamJava extends ConsoleProgram {
	public void run() {
		int abraham = 12;
		int maryTodd = 56;
		thirteenthAmendment(maryTodd, abraham);

		println("abraham = " + abraham);
		println("maryTodd = " + maryTodd);
	}
	private int thirteenthAmendment(int abraham, int maryTodd) {
		int robert = abraham % 10 + maryTodd / 10;
		println("robert = " + robert);
		println("abraham = " + abraham);

		edwinStanton(robert, abraham + maryTodd);
		edwinStanton(robert, abraham + maryTodd);

		abraham = thaddeusStevens(maryTodd, robert);
		println("abraham = " + abraham);
		return abraham;
	}
	private void edwinStanton(int maryTodd, int abraham) {
		maryTodd = abraham - maryTodd;
		println("maryTodd = " + maryTodd);
	}
	private int thaddeusStevens(int maryTodd, int abraham) {
		println("maryTodd = " + maryTodd);
		return abraham;
	}
}
```

Write the output of this program in the box below:

Problem Three: Nim (32 Points)

Nim is a game played by two players. The game begins with two piles of stones that are shared by the two players. Players alternate taking turns removing any nonzero number of stones from any single pile of their choice. If at the start of a player's turn both of the piles are empty, that player wins the game.

Your job is to write a Java program that lets two players play a game of Nim. When the game starts, your program should pick a random number of stones to put into each pile (ranging from the constant **`MIN_STONES`** to the constant **`MAX_STONES`**). Your program should then let the two players alternate turns, where on each turn the player will choose which pile to pick from (either Pile 1 or Pile 2), then choose a number of stones to remove from that pile. At the start of each turn, you should print out a message noting whose turn it is and how many stones are left in each pile. For example:

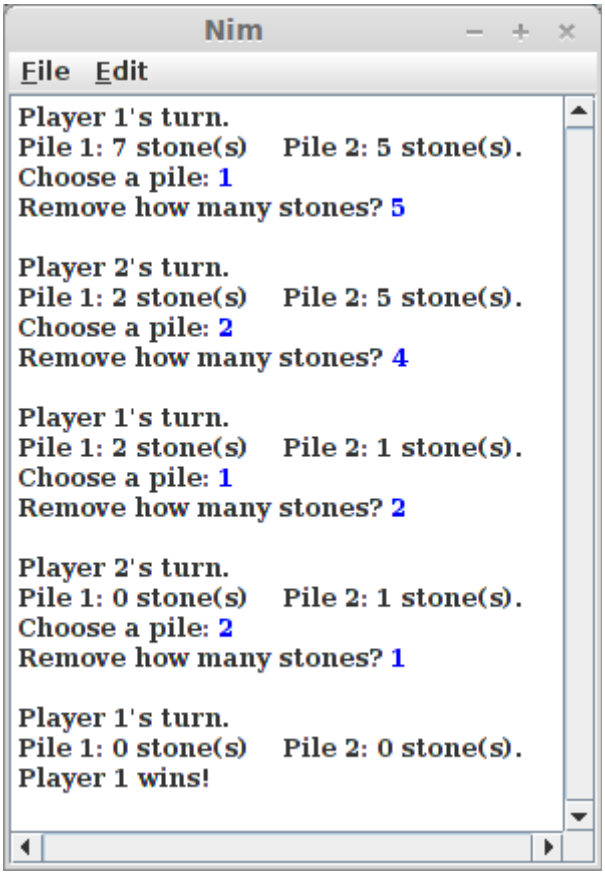
```
Player 1's turn.

Pile 1: 7 stone(s)  Pile 2: 5 stone(s)
```

At the end of the game, you should report who won the game with a message like this one:

```
Player 2 wins!
```

A sample run of this program, in which the first player wins, is shown to the right.



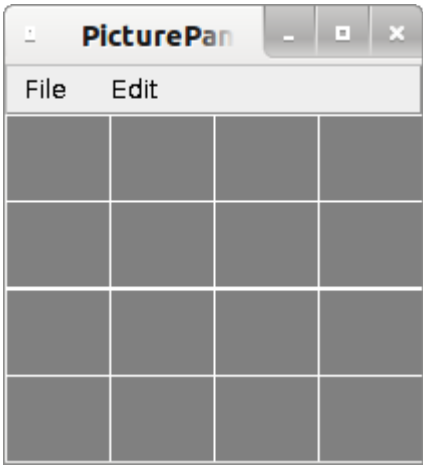
Your program needs to be able to handle invalid user input correctly. If a player wants to choose a pile other than Pile 1 or Pile 2, or tries to choose a pile containing no stones, or tries to remove an invalid number of stones from a pile (zero stones, or a negative number of stones, or more stones than are in the pile), then your program should display a message indicating that the player has made an invalid choice and then ask the user to make a different choice. You should continuously ask the player for a valid input until the player provides it.

Problem Four: Picture Panel Programs (22 Points)

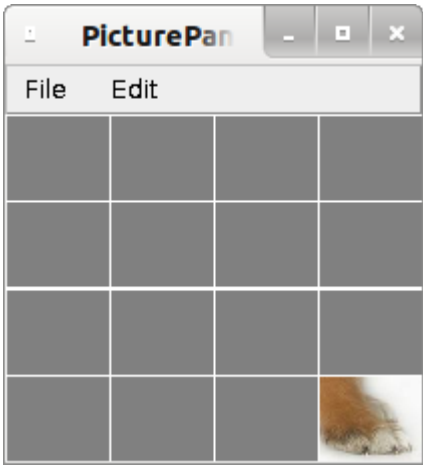
A ***picture panel program*** is a program in which an image is hidden behind a grid of rectangular blocks. The user can then remove the blocks to reveal more and more of the hidden image.

For example, suppose that the hidden image is this adorable picture of a puppy:

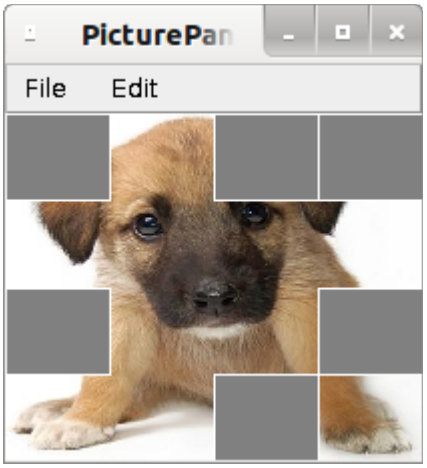
When the program starts up, the puppy is hidden behind a 4 × 4 grid of blocks:



The user can then click on the blocks to reveal more and more of the picture. For example, after clicking on the bottom-right block, that block is removed to reveal a part of the picture:



After clicking many more blocks, the majority of the picture is revealed:



Your job in this assignment is to implement a picture panel program. You can assume the following:

- The name of the file containing the image to hide is given by the constant **`IMAGE_FILENAME`**.
- The image will fit perfectly within the application window; you do not need to resize the image or set the size of the window.

In implementing your solution, you should adhere to the following:

- There are exactly four rows and four columns of blocks.
- Each block should use **`Color.WHITE`** for its border and **`Color.GRAY`** for its interior.
- The blocks should be sized so that they completely cover the window. You are not guaranteed that the width and the height of the window are the same.

As a reminder, you can use the **`GImage`** subclass of **`GObject`** to represent images.

Problem Five: Damaged DNA Diagnoses (22 Points)

DNA molecules consist of two paired strands of *nucleotides*, molecules which encode genetic information. Computational biologists typically represent each DNA strand as a string made from four different letters – A, C, T, and G – each of which represents one of the four possible nucleotides.

The two paired strands in a DNA molecule are not arbitrary. In normal DNA, the two strands always have the same length, and each nucleotide (letter) from one strand is paired with a corresponding nucleotide (latter) from the second strand. In normal DNA strands, the letters are paired as follows:

A is paired with **T** and vice-versa.

C is paired with **G** and vice-versa.

Below are two matching DNA strands. Note how the letters are paired up according to the above rules:

```
GCATGGATTAATATGAGACGACTAATAGGATAGTTACAACCCTTACGTCACCGCCTTGA
↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕↕
CGTACCTAATTATACTCTGCTGATTATCCTATCAATGTTGGGAATGCAGTGGCGGAACT
```

In some cases, errors occur within DNA molecules. This problem considers two types of DNA errors:

- **Point mutations**, in which a letter from one strand is matched against the wrong letter in the other strand. For example, **A** might accidentally pair with **C**, or **G** might pair with **G**.
- **Unmatched nucleotides**, in which a letter from one strand is not matched with a letter from the other. We represent this by pairing a letter from one strand with the – character in the other.

For example, consider these two DNA strands:

GGGA-GAAT**C**TCTGG**A**C**T**

CCCT**A**CTTA-AGACC**G**G**T**

Here, there are two unmatched nucleotides (represented by one DNA strand containing a dash character instead of a letter) and two point mutations (**A** paired with **G** and **T** paired with **T**).

Although both of these types of errors are problematic, we will consider point mutations to be a less "costly" error than an unmatched nucleotide. Let's assign a point mutation (a letter matched with the wrong letter) a cost of 1 and an unmatched nucleotide (a letter matched with a dash) a cost of 2.

Your job is to write a method

```
private int costOfDNAErrorsIn(String one, String two)
```

that accepts as input two strings representing DNA strands, then returns the total cost of all of the errors in those two strands. You may assume the following:

- The two strings have the same length.
- Each string consists purely of the characters **A**, **C**, **T**, **G**, and – (the dash character). All letters will be upper-case.
- A – character in one string is never paired with a – character in the other.

Here are some example inputs to the method, along with the expected result. For simplicity, all errors have been highlighted in bold:

Strands	Result
ACGT TGCA	**0** *(No errors)*
A**-**C**-**G**-**T**-**ACGT T**T**G**G**C**C**A**A**TGCA	**8** *(Four unmatched nucleotides)*
AAA**A**AAAA TTT**A**TTTT	**1** *(One point mutation)*
GAT**T**A**C**A CTA**T**T**-**T	**3** *(One point mutation, one unmatched nucleotide)*
CAT**-**TAG**-AC**T GTA**T**ATC**CAA**A	**6** *(Two point mutations, two unmatched nucleotides)*
-------- **ACGTACGT**	**16** *(Eight unmatched nucleotides)*
TAATAA ATTATT	**0** *(No errors)*
GGGA**-**GAAT**A**TCTGG**A**C**T** CCCT**A**CTTA**-**AGACC**G**G**T**	**6** *(Two point mutations, two unmatched nucleotides)*

```
private int costOfDNAErrorsIn(String one, String two) {
```