Practice Midterm Exam

Based on a handout by Eric Roberts, Mehran Sahami, and Patrick Young

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 (i.e., you cannot use laptops on the exam).

Coverage

The midterm exam covers the material presented in class through today, Wednesday, February 6, 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. The total number of points is 120.

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.

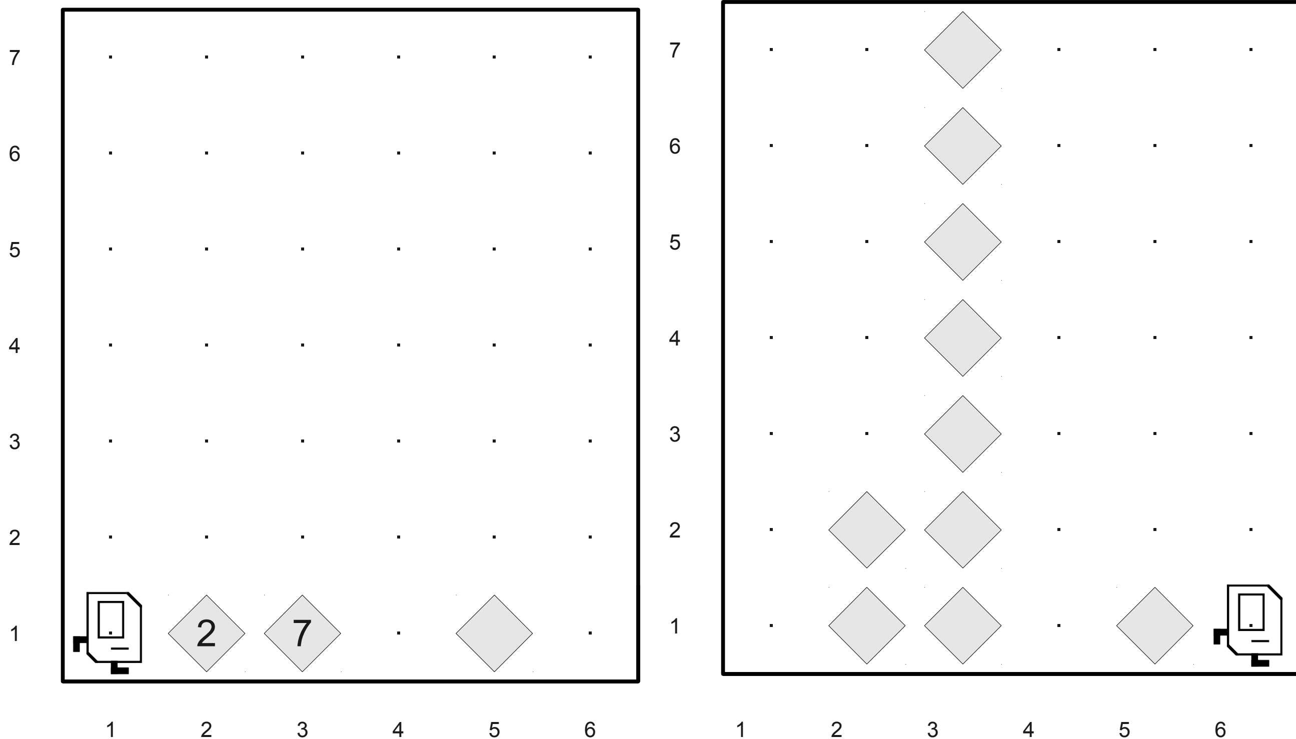
Blank pages for solutions omitted in practice exam

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: Tower-Building Karel **(20 Points)**

In this problem, Karel begins on the corner of 1st Street and 1st Avenue, facing East. Scattered along 1st Street are piles of beepers representing the raw materials needed to assemble towers. Karel's job is to take the piles of beepers and unstack them to form towers of the appropriate heights. For example, if a pile contains four beepers, Karel would unstack the pile into a tower of four beepers. If the pile contains just one beeper, Karel would build a tower of height one consisting of just that beeper. Here is a sample run of the program:



In solving this problem, you can count on the following facts about the world:

- Karel starts off facing East at the corner of 1st Street and 1st Avenue with no beepers in its beeper bag.
- The world is tall enough to ensure that Karel has enough vertical space to build all the towers. However, some towers might be exactly the height of the world. In the above picture, for example, there is just enough room to build the tower of height seven.
- Each corner on 1st Street may contain a stack of beepers, including the very first and very last corners.

Karel's final location and heading do not matter, and you do not need to worry about efficiency. 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 find the method **`beepersInBag()`** useful, though your solution does not need to use this method.

```
import stanford.karel.*;

public class TowerBuildingKarel extends SuperKarel {
    public void run() {
```

/* There would normally be a blank page here on which to write your answer. */

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.

```
4 / 7 * (double)7 / 4                  ______________________

137 / 42 == 0 && 137 / 0 == 42         ______________________

1 + 2 + "1 + 2" + 1 + 2                ______________________

(char)('3' - '0' + 'A')                ______________________
```

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

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

```
import acm.program.*;
public class JavaLeCarre extends ConsoleProgram {
      public void run() {
            int tinker  = 36;
            int tailor  = 54;
            int soldier = smiley(tailor, tinker);
            println("soldier = " + soldier);
      }

      private int smiley(int tinker, int tailor) {
            int poorMan = guillam(tinker, tailor);
            println("poorMan = " + poorMan);

            int beggarMan = guillam(tailor + 9, tinker / 9);
            println("beggarMan = " + beggarMan);

            return poorMan + beggarMan;
      }

      private int guillam(int karla, int mundt) {
            karla %= 10;
            mundt /= 10;
            return 100 * karla + mundt;
      }
}
```

Problem Three: The Saint Petersburg Game **(25 Points)**

The *Saint Petersburg Game* or *Saint Petersburg Lottery* is a hypothetical casino game played by two players (say, you and me) sitting at a table. I begin by putting $1 on the table, and you then repeatedly flip a coin until it comes up tails. Each time the coin comes up heads, I double the amount of money on the table. As soon as the coin comes up tails, the game is over and you win all the money on the table (no strings attached – I'm just feeling really generous!)

One sample run of the game is as follows. I put $1 on the table. You then flip tails, so the game ends and you collect $1. Another run might go like this: I put $1 on the table. You flip heads, so I double the money to $2. You flip heads again, so I double the money to $4. You flip heads again, so I double the money to $8. You then flip tails, so the game ends and you collect $8.

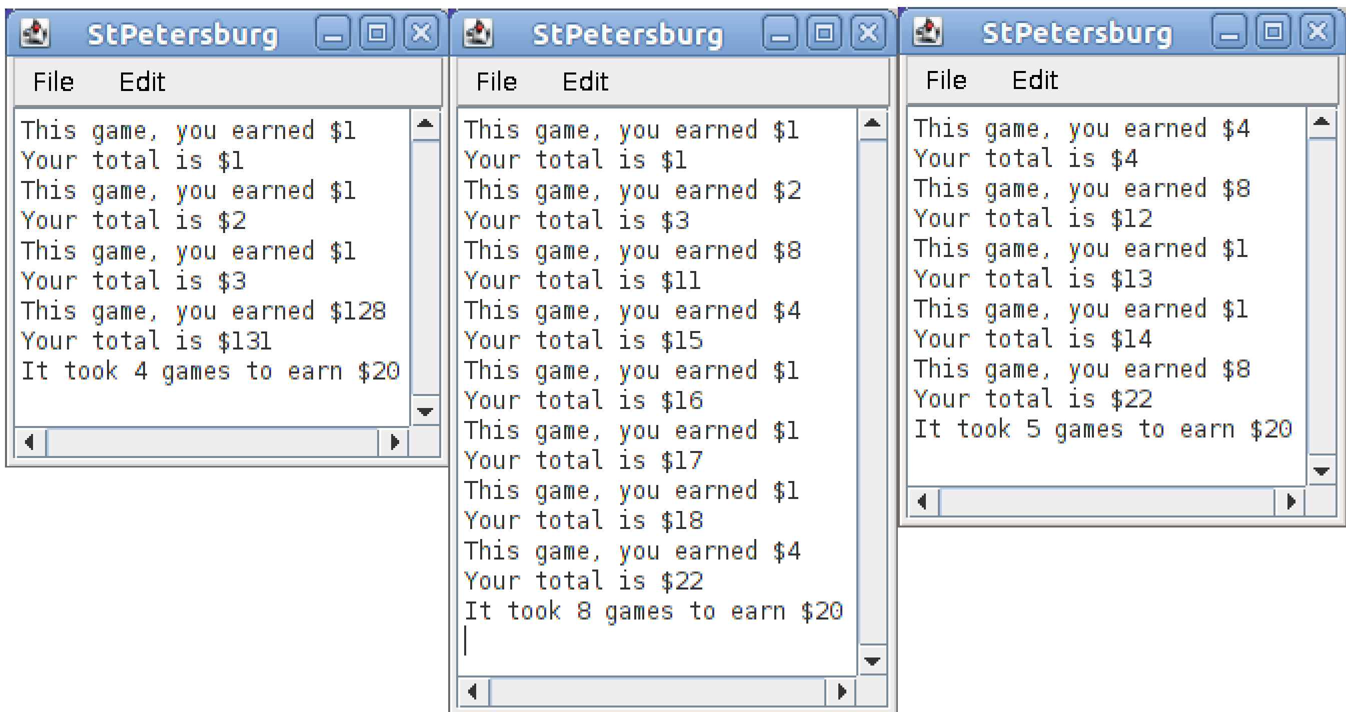
Write a program that plays the Saint Petersburg Game as many times as is necessary to earn a total of at least $20. To clarify – you aren't playing until you win $20 in a single game; instead, you're playing until your total winnings across all games you've played is $20. After each game, your program should print out how much money you won during that game, along with your total winnings. For example, if after the first game you earned $4, you would print out

```
This game, you earned $4
Your total is $4
```

Once you've earned at least $20, your program should output how many times you had to play the Saint Petersburg Game to earn $20. For example, if it takes five games to win, at the end you'd print

```
It took 5 games to earn $20
```

Three sample runs of the program are shown below:



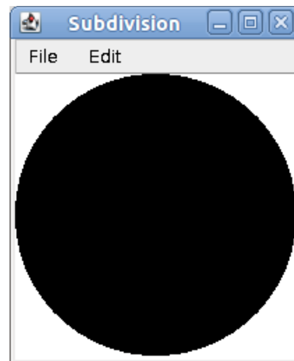

```
import acm.program.*;
import acm.util.*;

public class SaintPetersburgGame extends ConsoleProgram {
    /* Again, we would normally provide space for you to write your solution */
```

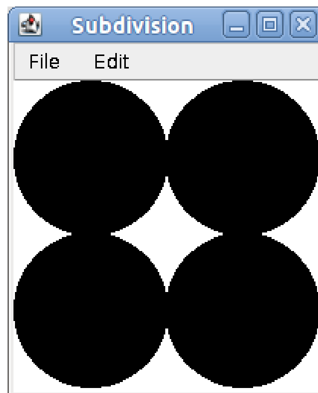
Problem Four: Subdivision **(25 Points)**

In this problem, you'll write a program that lets the user continuously cut an circle into smaller and smaller pieces, leading to aesthetically pleasing pictures.

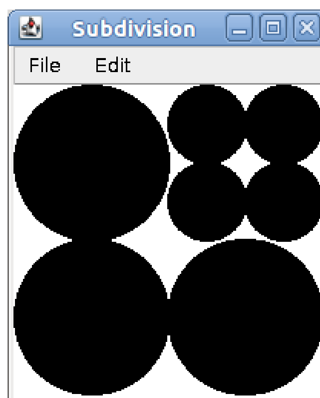
When the program starts up, the user sees a black circle that occupies the entire window, like this:



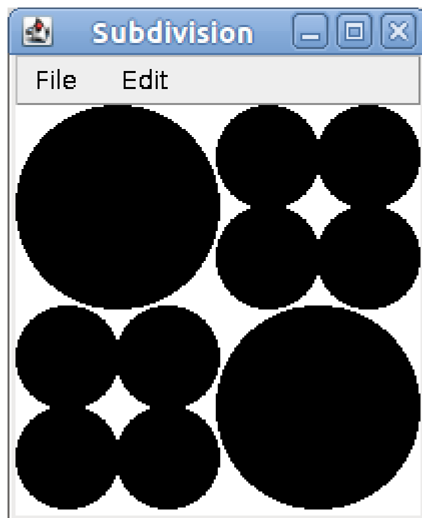
Now, if the user clicks on the circle, that circle is replaced by four smaller circles, each of which is a quarter of the size of the original circle. For example, after clicking on the circle above, the program would display



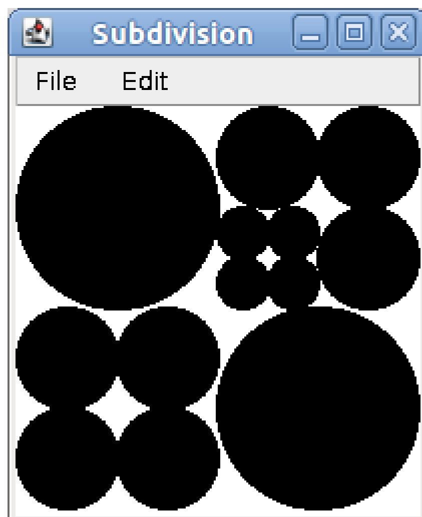
If the user then clicks on any of these four circles, that circle is removed and replaced by four more circles, each of which are one-quarter the size of the chosen circle. For example, if the user clicks on the upper-right circle, the result would be



If the user now clicks the bottom-left circle, the result would be



And finally, if the user clicked the lower-left of the small circles in the upper-right corner, the result would be



Write a program that begins by drawing a filled, black circle whose width and height are the width and height of the window (which you may assume are the same). Whenever the user clicks on an circle, you should remove that circle and add in four new circles such that

- each circle is one-quarter the size of the original circle (half as tall and half as wide),
- each circle is filled black, and
- the four circles are positioned as shown in the above diagrams.

```
import acm.program.*;
import acm.graphics.*;
import java.awt.*;
import java.awt.event.*;

public class Subdivision extends GraphicsProgram {
```

/ As before, we would normally give you more space to write your answer. */*

Problem Five: Grade-School Arithmetic **(30 Points)**

As mentioned in Chapter 3 of *The Art and Science of Java*, Java's primitive data types (**`int`**, **`double`**, etc.) have ranges on what values they can store. **`int`**, for example, can't hold values greater than 2,147,483,647. Since we may want the computer to work with values larger than this (say, for example, the US national debt or the number of atoms in one gram of carbon), programmers sometimes use other types (such as **`String`**) for large integers. For example:

```
String worldPopulation  = "6993309969";
String rubiksCubeStates = "43252003274489856000";
```

In order to make use of numbers encoded this way, we have to have some way to add them. In grade school, you probably learned how to add two large numbers one digit at a time (a technique appropriately called *grade-school addition*). You write the two numbers out, one on top of the other, then work from the right to the left adding the individual digits of the numbers. When the digits sum up to a value greater than ten, you would write out just the ones digit of their sum, then carry a 1 into the next column. For example, here's 137 + 864:

```
  1  1  1
     1  3  7
  +  8  6  4
  ----------
  1  0  0  1
```

Using your knowledge of Java, you will teach the computer how to add values this way. Your job in this problem is to write a method

```
private String addIntegerStrings(String firstNum, String secondNum)
```

that accepts as input two **`String`**s encoding integers <u>with the same number of digits</u>, then uses grade-school addition to add the two integers represented by those strings. For example:

`addIntegerStrings("44", "99")`	returns **`"143"`**
`addIntegerStrings("1234", "4321")`	returns **`"5555"`**
`addIntegerStrings("137", "42")`	cannot happen, since 137 and 42 don't have the same number of digits. You do not need to handle this case in your solution.
`addIntegerStrings("137", "042")`	returns **`"179"`**
`addIntegerStrings("123123123123123123123", "119119119119119119119")`	returns **`"242242242242242242242"`**

You may assume the following:

- The strings **`firstNum`** and **`secondNum`** are the same length, meaning that the numbers have the same number of digits.
- The integers encoded by **`firstNum`** and **`secondNum`** are nonnegative, so every encoded integer will be at least 0.
- All the characters in the two input strings are digits, so there are no commas, minus signs, plus signs, etc. Thus you will never get `"1,000,000"` or `"-3"` as inputs.

Your solution must add the numeric strings using the grade-school algorithm. You cannot assume that the numbers represented by the strings are small enough to be stored as an **`int`**, **`long`**, or **`double`**.

As a hint to make your task a bit easier, the carry from one column to the next can never be anything other than 0 or 1.

```
private String addIntegerStrings(String firstNum, String secondNum) {
```