Steve Cooper
CS106A

Handout #28
May 1, 2013

Practice Midterm Examination

Midterm Time: Tuesday, May 7, 7:00P.M.–9:00P.M.

Portions of this handout by Eric Roberts 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 on Tuesday.

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 tomorrow (May 3rd), which means that you are responsible for the Karel material plus the appropriate chapters (as indicated in the course schedule) of the class textbook *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 on this practice exam is 140. The actual midterm will be out of 100 points.

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 or chapter number in which that definition appears.

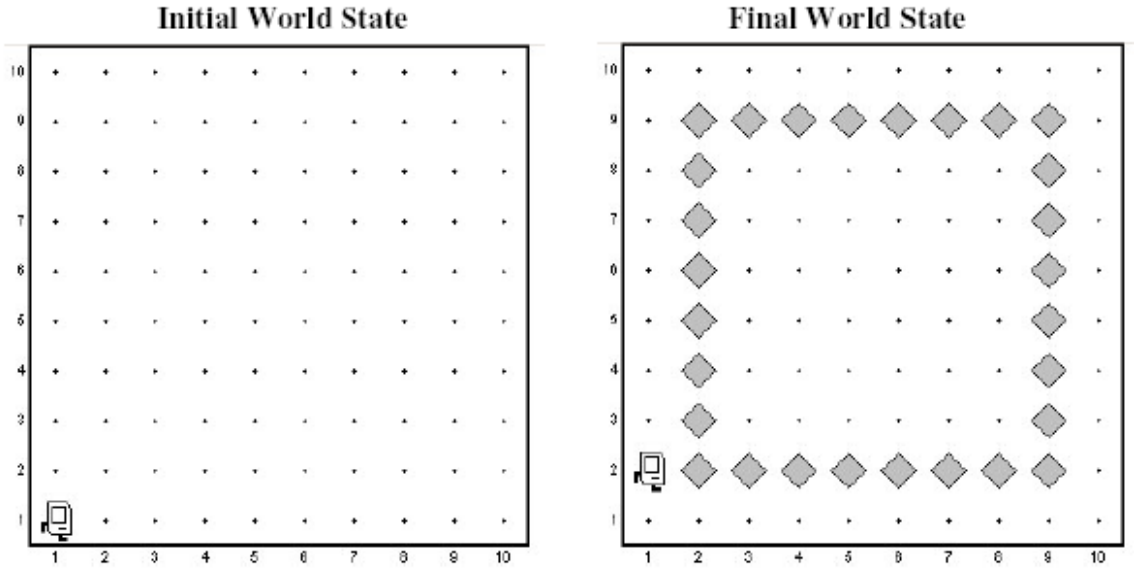
Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine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

Problem 1: Karel the Robot (20 points)

We want to write a Karel program which will create an inside border around the world. Each location that is part of the border should have **one** (and only one) beeper on it and the border should be inset by one square from the outer walls of the world like this:



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

- You may assume that the world is at least 3x3 squares. The correct solution for a 3x3 square world is to place a single beeper in the center square.
- Karel starts off facing East at the corner of 1st Street and 1st Avenue with an infinite number beepers in its beeper bag.
- We do not care about Karel's final location or heading.
- You do not need to worry about efficiency.
- You are limited to the instructions in the Karel booklet—the only variables allowed are loop control variables used within the control section of the for loop.

Write your solution on the next page (blank page omitted to save trees).

Problem 2: Simple Java expressions, statements, and methods (20 points)

(2a) 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.

```
5.0 / 4 - 4 / 5
```

_____

```
7 < 9 - 5 && 3 % 0 == 3
```

_____

```
"B" + 8 + 4
```

_____

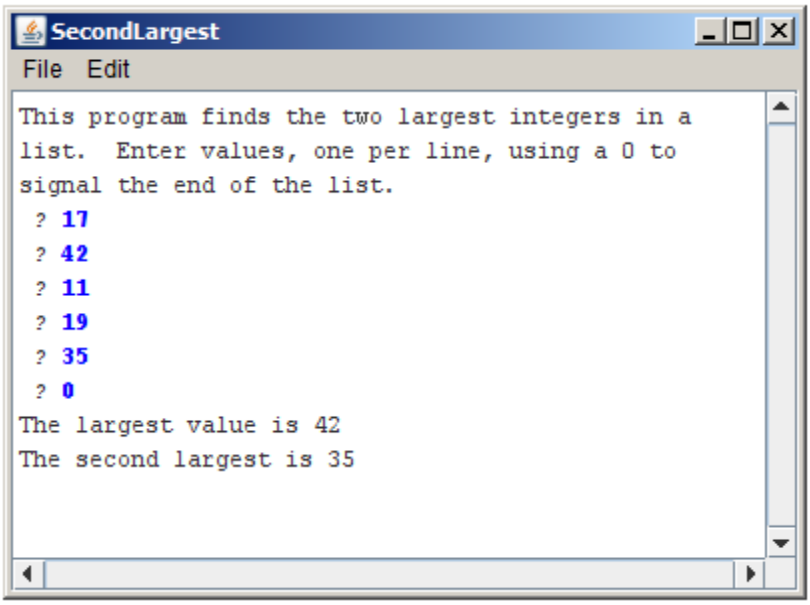
(2b) What output is printed by the following program:

```
/*
 * File: Problem2b.java
 * --------------------
 * This program doesn't do anything useful and exists only to test
 * your understanding of method calls and parameter passing.
 */

import acm.program.*;

public class Problem2b extends ConsoleProgram {

  public void run() {
     int num1 = 2;
     int num2 = 13;
     println("The 1st number is: " + Mystery(num1, 6));
     println("The 2nd number is: " + Mystery(num2 % 5, 1 + num1 * 2));
  }


  private int Mystery(int num1, int num2) {
     num1 = Unknown(num1, num2);
     num2 = Unknown(num2, num1);
     return(num2);
  }


  private int Unknown(int num1, int num2) {
     int num3 = num1 + num2;
     num2 += num3 * 2;
     return(num2);
  }

}
```

Answer:

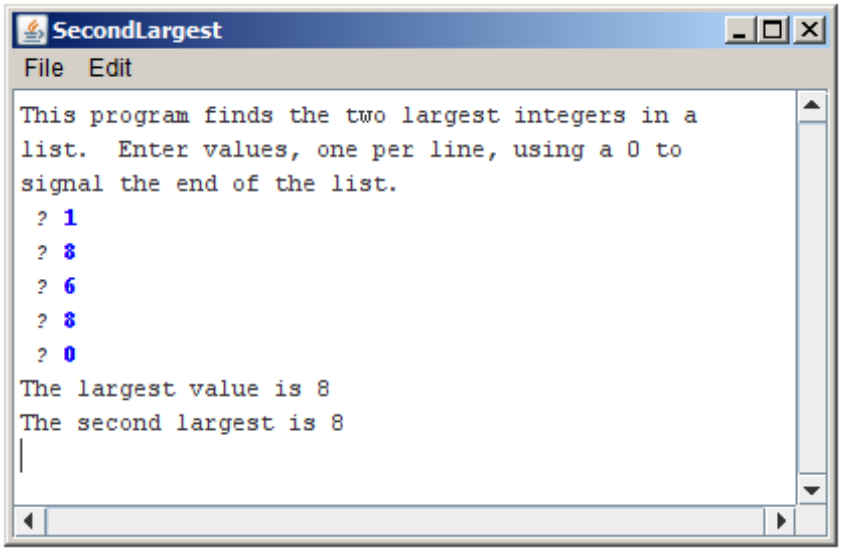
Problem 3: Simple Java programs (25 points)

In Assignment #2, you wrote a program to find the largest and smallest integers in a list entered by the user. For this problem, write a similar program that instead finds the largest and the second-largest integer. As in the homework problem, you should use 0 as a sentinel to indicate the end of the input list. Thus, a sample run of the program might look like this:



To reduce the number of special cases, you may make the following assumptions:

- The user must enter at least two values before the sentinel.
- All input values are positive integers.
- If the largest value appears more than once, that value should be listed as both the largest and second-largest value, as shown in the following sample run:

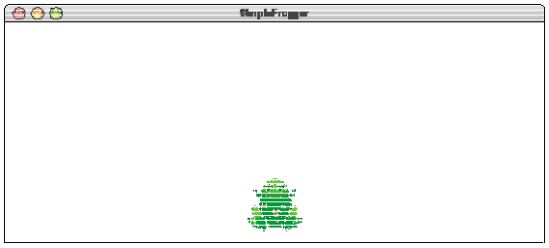


Write your solution on the next page (omitted).

Problem 4: Using the graphics and random number libraries (35 points)

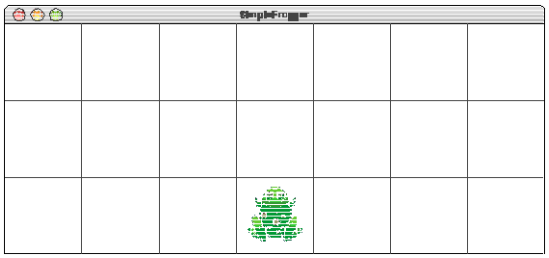
In the arcade game Frogger, this is a frog that "hops" along the screen. A full game is beyond the scope of an exam problem, but it is relatively straightforward to write the code that (1) puts an image of the frog on the screen and (2) gets the frog to jump when the user clicks the mouse.

Your first task in this problem is to place the frog at the bottom of the graphics window, as shown on the right. The frog itself is the easy part because all you need to do is create a **GImage** object with the appropriate picture (you can assume the file **frog.gif** is provided), as follows:



```
GImage frog = new GImage("frog.gif");
```

The harder part is getting the image in the appropriate place in the bottom of the window. In Frogger, the frog image cannot be just anywhere on the screen but must instead occupy a position in an imaginary grid such as the one shown on the right. The size of the grid is controlled by three named constants, which have the following values for this grid:



```
public static final int SQSIZE = 75;
public static final int NCOLS = 7;
public static final int NROWS = 3;
```

The **SQSIZE** constant indicates that each of the squares in the grid is 75 pixels in each dimension and the other two parameters give the width and height of the grid in terms of the number of squares. Remember that the squares shown in the most recent diagram do not actually exist but simply define the legal positions for the frog. In the initial position, the frog must be in the center square along the bottom row. You may assume **NCOLS** is odd so that there is a center square, and you may also assume that **APPLICATION_WIDTH** and **APPLICATION_HEIGHT** have been set so the **NCOLS** x **NROWS** squares fill the window.

The second part of the problem is getting the frog to jump when the user clicks the mouse. The goal is to get the frog to jump one square in the direction that moves it closest to the mouse. For example, if you click the mouse at the location shown in the diagram at the right, the frog should move **SQSIZE** pixels upward so that it occupies the center square in the grid. If the user then clicked the mouse at the left edge of the screen, the frog should jump **SQSIZE** pixels to the left. The frog, however, should never jump outside the window.

The following restatement of the rule may clarify the intended behavior more explicit. The frog should jump one square position in the direction—up, down, left, or right—that corresponds most closely to the direction from the center of the frog to the mouse position. Thus, in the diagram, the frog should move up rather than right because the distance to the mouse is larger in the *y* direction than it is in the *x* direction. If, however, the new position would lie outside the **NCOLS** x **NROWS** grid, the frog should stay where it is. Write your solution on next page (omitted).

Problem 5: Strings and characters (20 points)

In the early part of the 20th century, there was considerable interest in both England and the United States in simplifying the rules used for spelling English words, which has always been a difficult proposition. One suggestion advanced as part of this movement was the removal of all doubled letters from words. If this were done, no one would have to remember that the name of the Stanford student union is spelled "Tresidder," even though the incorrect spelling "Tressider" occurs at least as often. If double letters were banned, everyone could agree on "Tresider."

Write a method **`removeDoubledLetters`** that takes a string as its argument and returns a new string with all doubled letters in the string replaced by a single letter. For example, if you call

```
removeDoubledLetters("tresidder")
```

your method should return the string **`"tresider"`**. Similarly, if you call

```
removeDoubledLetters("bookkeeper")
```

your method should return **`"bokeper"`**.

In writing your solution, you should keep in mind the following:

- You do not need to write a complete program. All you need is the definition of the method **`removeDoubledLetters`** that returns the desired result.

- You may assume that all letters in the string are lower case so that you don't have to worry about changes in capitalization.

- You may assume that no letter appears more than twice in a row. (It is likely that your program will work even if this restriction were not included; we've included it explicitly only so that you don't even have to think about this case.)

Write your solution on the next page (omitted).

Problem 6: Classes (20 points)

Suppose you work for an old-fashioned, pen-and-paper bank, and they want to make the leap to digital. An important structural component of any bank is the bank account, which you model in your code as the class BankAccount. Each BankAccount represents one account, and has certain operations that can be performed on it.

Please implement the following class (i.e. write the methods for the class). See below for an example of the methods in use.

```
/*Constructor: The ownerName is the name of the account owner,
initialBalance is the initial amount of money in the account*/

public BankAccount (String ownerName, double initialBalance)
```

```
/*Constructor: If no initialBalance is passed in, the balance starts at
zero*/

public BankAccount (String ownerName)

/* returns the balance of the account as a double*/

public double getBalance()

/* adds amountToDeposit to the current balance*/

public void deposit(double amountToDeposit)

/*withdraw(…) returns true/false depending on whether there is
enough money in the bank account to make a withdrawal. If there isn't
enough money, no money should be withdrawn, and you should return false.
If there is, the amountToWithdraw should be deducted from the balance,
and you should return true. You can assume for this problem that the
amountToWithdraw will be greater than 0.*/

public boolean withdraw(double amountToWithdraw)

/* toString() should return a string of this format: "owner_name:
balance" with the italized words replaced with the appropriate values */

public String toString()

/* transferTo(…) should withdraw "amount" money from this BankAccount
object, then deposit it in the passed in BankAccount, named
accountToTransferTo. It should return true if there was enough money to
make the transfer, and false otherwise (if there isn't enough money, no
transfer should happen). Remember that you can call other methods in the
interface from this method! */

public boolean transferTo(BankAccount accountToTransferTo, double amount)
```

Example Usage:

```
public void run(){
      BankAccount steveAccount = new BankAccount("Steve", 1000.02);
      BankAccount lynnAccount = new BankAccount("Lynn");

      /*will print: Steve has 1000.02 dollars in his account and Lynn has
      0.0 dollars in her account. */
```

```
        println("Steve has " + steveAccount.getBalance() +
          " dollars in his account and Lynn has " + lynnAccount.getBalance() +
          " dollars in her account.");

        boolean success = steveAccount.withdraw(20.0);
        if(success){
              println(“Steve successfully withdrew money.”);
        } else {
              println(“Steve didn’t have enough money.”);
        }

        steveAccount.deposit(50);

        //will print: Steve: 1030.02
        println(steveAccount.toString());

        println("Lynn has gotten ahold of Steve’s credit card." +
                " Time for a vacation!");

        //ignoring the return to save space for this example! It actually
        returns boolean just like withdraw does
        steveAccount.transferTo(lynnAccount, 1000.00);

        //will print: Steve: 30.02 and Lynn: 1000.0
        println(steveAccount.toString() + " and " + lynnAccount.toString());

        success = steveAccount.withdraw(100.0);
        if(success){
              println("Steve successfully withdrew money!");
        } else {
              println("Steve is broke now.");
        }

}
```