

Problem Set 1

Due: 1:30pm Tuesday, February 13

Note:

This assignment is due at the beginning of the *first* CS230 lab on Tuesday, February 13, regardless of which lab you happen to be in.

Overview:

The purpose of this assignment is to give you practice writing some simple Java methods using Emacs and Java on the Linux workstations. Since learning a new programming environment takes time, it is strongly recommended that you (1) start early and (2) work with a partner. Allocate time over several days to work on the problems; it is very unwise to start the assignment only a day or two before it is due. Don't hesitate to ask for help if you hit a roadblock.

Submission:

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 1:30pm on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet) indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. your final version of `DiceGame.java` from Problem 1.
3. a transcript demonstrating that your `DiceGame` program works as expected.
4. your final version of `Palindrome.java` from Problem 2.
5. a transcript demonstrating that your `Palindrome` program works as expected.

Each team should also submit a single softcopy (consisting of your entire final `ps1` directory) to the drop directory `~cs230/drop/ps1/username`, where `username` is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute the following Linux commands in the team member's account where the code is stored:

```
cd /students/username/cs230
cp -R ps1 ~cs230/drop/ps1/username/
```

Problem 0: Getting Started

a. : Puma Account

If you do not already have a Puma account (from taking CS111), you should begin this assignment by requesting a Puma account by following the directions on Handout #3. If you do have an account but have forgotten your password, contact Lyn or Scott Anderson to reset it.

b. : Linux

Once you have your Puma account, you should log in to a Linux workstation and learn some simple Linux commands, as described in Handouts #3 and #4.

c. : Emacs

Next you should learn how to use Emacs. See the information in Handouts #3 and #5 on using Emacs. Learning to execute all cursor-motion and editing commands via keystrokes (rather than via mouse and menus) is an important skill that will save you lots of time over the semester. It will also make it easier for you to work remotely via `ssh`. A good way to begin learning the keystroke commands is taking the online interactive Emacs tutorial (see Handout #3 for how to do this).

d. : CVS

To do the rest of the problems on this assignment, you will need to use several files that are in the CVS-controlled CS230 repository. Follow the directions in Handout #7 for how to install your local CVS filesystem. You only need to install it once.

Once you have installed your local CVS filesystem, you can access all CVS-controlled files by executing the following in a Linux shell:

```
cd ~/cs230
cvs update -d
```

Indeed, every time you log in to a Linux machine to work on a CS230 assignment, you should execute the above commands to ensure that you have the most up-to-date versions of the problem set materials.

On this assignment, executing the above commands will create the local directory `~/cs230/ps1` containing the following three files:

1. `DiceGame.java`: This file contains a Java skeleton for the *DicePoker* game for Problem 1.
2. `sentences.txt`: This file contains some sentences for testing the palindrome checker in Problem 2.
3. `big.txt`: This file contains a single very long palindromic line for testing the palindrome checker in Problem 2.

Problem 1 [50]: Let's Play Dice Poker!

The Rules of The Game

Dice Poker is a variation of the game of Poker that uses dice instead of cards. In this problem, you will complete the implementation of a program that simulates a Dice Poker game played between the computer and user. The game consists of multiple rounds. In each round, the computer and user each roll five dice. The combination of five dice values is ranked on a scale from 0 to 6 as follows:

- **rank 6:** *Five of a Kind:* all five dice have the same value.
- **rank 5:** *Four of a Kind:* four of the five dice have the same value and the fifth has a different value.
- **rank 4:** *Full House:* the dice include three of one value and two of another.
- **rank 3:** *Three of a Kind:* three dice have the same value and the remaining two have different values
- **rank 2:** *Two Pairs:* the dice include two pairs of the same values and a fifth die of a different value
- **rank 1:** *One Pair:* two dice have the same value and the remaining three have different values
- **rank 0:** *Nothing:* all five dice have different values

The player with the highest ranking roll of the dice wins the round. The player who wins the most rounds is the winner of the game.

Some Logistics

Figs. 1–3 present the skeleton of a program that simulates the playing of Dice Poker, with some code filled in. For simplicity, the code is all placed in a single file named `DiceGame.java` that contains the definition of a single class named `DiceGame`. This skeleton code file can be found in your `~/cs230/ps1` directory (after you perform a cvs update).

You can compile and execute the existing program, but it doesn't do much yet. The `main` method contained in the `DiceGame.java` file assumes that the user has entered a name and an optional number of rounds in the command line when executing the program, for example:

```
java DiceGame Dave
java DiceGame Dave 7
```

If you do not supply the correct number of arguments, the program just displays a message indicating the correct usage and then terminates. (If you do not immediately recognize the joke, you should check out the classic *2001: A Space Odyssey*)

Implementation

In the implementation, the five dice values for a player are stored in an array of five integers. The method `throw5Dice()`, whose definition is already complete, returns a five-slot array filled with random integers between 1 and 6. It uses the built-in Java method `Math.random()`, which returns a random number of type `double` in the range from 0.0 up to, but not including, 1.0. For future reference, the Java `Math` class has a number of methods for performing useful mathematical functions. You should browse the documentation on the available `Math` methods at the Sun Java website:

```
http://java.sun.com/j2se/1.5.0/docs/api/
```

In the list of Java 5 Platform Packages, select `java.lang`, and under the Class Summary list, choose `Math`.

```

/** Simulates a Dice Poker game played between the computer and user.
 * This class definition contains a main() method that assumes
 * that the user enters a name and an optional number of rounds
 * in the command line. For example:
 *
 * java PlayDice Wendy (plays 5 rounds by default);
 * java PlayDice Dave 10 (plays 10 rounds)
 *
 * @author Takis Metaxas provided the original skeleton
 * @author [01/30/07] Lyn Turbak made several changes to the skeleton
 * @author Your name here
 * @version %I%, %G%
 */
public class DiceGame {

    /* Start the homework by reading this main method. */
    /** Simulates a Dice Poker game played between the computer and user.
     * args[0] should be the name of the user;
     * args[1] is an optional number of rounds (the rounds will be 5 by
     * default if not explicitly specified).
     */
    public static void main (String args[]) {

        // If invocation is malformed, do not let user play
        if ((args.length == 0) || (args.length > 2)) {
            System.out.println("Usage: java DiceGame <yourname> [<rounds>]");
            System.exit(0);
        }

        try {
            String name = args[0];
            int rounds = (args.length == 2) ? (Integer.parseInt(args[1])) : 5;
            // 5 is the default number of rounds if not supplied by user

            playDiceGame(name, rounds);
        } catch (NumberFormatException ex) { // thrown if args[1] doesn't specify an integer
            System.out.println("Sorry, \"" + args[1] + "\" is not an integer number of rounds.");
            System.out.println("Usage: java DiceGame <yourname> [<rounds>]");
        }

    }
}

```

Figure 1: Code skeleton for DiceGame.java, part 1

```

/** Simulates the playing of numRounds of the Dice Poker game between
 * HAL and player name, and prints the winner at the end
 *
 * @param name      the player's name
 * @param numRounds the number of rounds to play
 */
public static void playDiceGame (String name, int numRounds) {

    System.out.println("Hello " + name + ".");
    System.out.println("I'm completely operational and all my circuits are functioning perfectly.");
    System.out.println("Would you like to play a game of Dice Poker? I play very well.");

    // declare variables for storing the score of all the rounds
    int cwin=0, pwin=0; // round wins of the computer (0) and the player (1)

    // for each round of the game:
    int rounds = 1;
    do{
        System.out.println("*** ROUND " + rounds);
        // play the round, determine the winner of the round
        int winner = playOneRound();
        if(winner == 0) cwin++;
        if(winner == 1) pwin++;
    } while(rounds++ < numRounds);

    // After all rounds played, determine the final winner of the game and print the results
    if(pwin>cwin) System.out.print("The game was won by " + name + " with a score of "
        + pwin + " to " + cwin);
    else if(cwin>pwin) System.out.print("The game was won by the HAL with a score of "
        + cwin + " to " + pwin);
    else System.out.print("The game was tied with a score of " + cwin + " to " + pwin);

    System.out.println(" in " + numRounds + " rounds.");
}

/** Plays one round of the game: First the computer's turn,
 * then the player's turn
 *
 * @return 0 if computer wins the round, 1 if player wins, 2 if a tie
 */
public static int playOneRound() {

    // *** FLESH OUT THIS METHOD (and replace the following stub):
    return 2;

}

/** Simulates the rolling of five dice by throwing one dice 5 times.
 *
 * @return a new five-slot array filled with random integers between 1 and 6.
 */
public static int[] throw5Dice() {
    int[] diceArray = new int[5];
    for (int i = 0; i < 5; i++)
        diceArray[i] = (int)(Math.random() * 6) + 1;
    return diceArray;
}

```

Figure 2: Code skeleton for DiceGame.java, part 2

```

/** Counts how many distinct values appear in the input array.
 * PRE-CONDITION: assumes all slots in the input array are between 1 and 6.
 *
 * @param input an input array whose slots hold values between 1 and 6.
 * @return a 7-slot integer array whose 0th slot is 0 and whose ith slot (i>0)
 *         contains the frequency of i in the input array.
 */
public static int[] accumulateValues(int[] input) {

    // *** FLESH OUT THIS METHOD (and replace the following stub):
    return new int[] {};

}

/** Given an input array storing five dice values,
 * determines the rank of the Dice Poker hand represented by those values.
 *
 * @param input an array with five dice values
 * @return the rank: an integer between 0 and 6
 */
public static int getRank (int[] input) {

    // First determine how many of each dice value are stored in input
    // by calling the accumulateValues() method
    int[] diceResults = accumulateValues(input);

    // *** FLESH OUT THE REST OF THIS METHOD (and replace the following stub):
    return 0;

}

/** Returns a string representing the contents of the input array.
 * @param arr the input array
 * @return a string with space-separated representations of the array slots
 */
public static String arrayToString (int[] arr) {
    String result = "";
    for (int i = 0; i < arr.length; i++) {
        result = result + arr[i] + " ";
    }
    return result;
}

private static String[] ranks = {"Nothing",           // rank 0
    "One Pair",           // rank 1
    "Two Pair",           // rank 2
    "Three of a Kind",    // rank 3
    "Full House",         // rank 4
    "Four of a Kind",     // rank 5
    "Five of a Kind" };  // rank 6

/** Returns a string representing a given rank
 * @param rank the input rank
 * @return a string indicating the meaning of the rank (6="Five of a Kind",
 *         5="Four of a Kind", 4="Full House, 3="Three of a Kind", 2="Two Pair",
 *         1="One Pair", 0="Nothing".)
 */
public static String rankToString (int rank) {
    return ranks[rank];
}
}

```

Figure 3: Code skeleton for DiceGame.java, part 3

To understand the skeleton we have given you, you should start by reading the `main()` method, and then read the rest of the program as needed. The program is complete except for three methods. Your task is to complete the definitions of the following three methods:

- `public static int[] accumulateValues(int[] input)`
Given an input array `input` containing five dice values, this method should determine how many of each of the possible six dice values appear in `input`, and return this information in the output array, which we will call `accumResult`. For simplicity, assume that `accumResult` is an array of 7 integers such that `accumResult[0]` is necessarily 0 and `accumResult[i]` contains the number of times `i` appears in `input`. That is, `accumResult[1]` should contain the number of 1s that appear in `input`, `accumResult[2]` should contain the number of 2s, and so on. E.g., for the input array `{1,4,4,5,1}`, `accumResult` should be `{0,2,0,0,2,1,0}`. Think carefully about how to implement this method compactly – its definition can be very concise. The purpose of defining this method is to facilitate the implementation of the next method, `getRank()`.
- `public static int getRank (int[] input)`
Given an input array `input` containing five dice values, this method should return the rank of the combination of values (an integer between 0 and 6, as described earlier). In the program skeleton, the body of the `getRank()` method already contains a statement that declares an array `diceResults` that stores how many of each dice value are contained in `input`. There are many strategies for implementing the rest of this method — think carefully about different options before writing any code.
- `public static int playOneRound()`
This method simulates one round of the game by first simulating the computer’s turn (throws the 5 dice and getting its rank), then simulating the player’s turn, and then determining the winner. It returns an integer that signifies the winner: 0 if the computer wins, 1 if the user wins, and 2 if there is a tie.

An important piece of advice that we will reinforce throughout the semester is to construct your implementation incrementally — build and test one method at a time! Begin by completing the definition of `accumulateValues()` and adding test code to the `playOneRound()` method to test your code. Next complete the definition and testing of the `getRank()` method. And so on. Larger methods can also be created and tested in small pieces. As you build your implementation, also keep in mind the comments about programming style at the beginning of this handout. In the comments at the beginning of the file, be sure to add your name and the date of completion of your code.

The Output

Fig. 4 presents two sample runs of the complete `DiceGame` program. Note that five rounds are played by default if the number of rounds is not explicitly specified.

Notes on Programming Style

Programs that you compose for your CS230 assignments and labs will be evaluated not only on whether your programs work correctly, but also on programming style. This includes the use of informative names, good comments, easily readable formatting, and a good balance between compactness and clarity (e.g., do not define unnecessary variables or write multiple statements to perform a task that can be expressed clearly in a single statement.) Follow the guidelines in Takis Metaxas’s *On Programming Style*, which is available on the CS230 *Documentation* page.

In large software development projects involving a team of programmers, it is customary to include comments at the beginning of each code file that include information such as the file name,

```

[fturbak@puma ps1] java DiceGame Dave 7
Hello Dave.
I'm completely operational and all my circuits are functioning perfectly.
Would you like to play a game of Dice Poker? I play very well.
*** ROUND 1
HAL: 3 5 4 4 5 : Two Pair
You: 3 3 6 1 2 : One Pair
*** ROUND 2
HAL: 2 1 1 1 5 : Three of a Kind
You: 4 6 1 3 6 : One Pair
*** ROUND 3
HAL: 4 3 4 5 2 : One Pair
You: 1 2 5 1 3 : One Pair
*** ROUND 4
HAL: 3 6 1 3 5 : One Pair
You: 6 2 6 4 2 : Two Pair
*** ROUND 5
HAL: 5 5 2 6 3 : One Pair
You: 2 5 2 6 5 : Two Pair
*** ROUND 6
HAL: 5 3 6 2 2 : One Pair
You: 2 2 6 2 2 : Four of a Kind
*** ROUND 7
HAL: 4 3 5 5 2 : One Pair
You: 5 3 5 1 1 : Two Pair
The game was won by Dave with a score of 4 to 2 in 7 rounds.

[fturbak@puma ps1] java DiceGame Lyn
Hello Lyn.
I'm completely operational and all my circuits are functioning perfectly.
Would you like to play a game of Dice Poker? I play very well.
*** ROUND 1
HAL: 6 5 5 2 3 : One Pair
You: 6 1 2 3 1 : One Pair
*** ROUND 2
HAL: 4 3 3 1 6 : One Pair
You: 2 3 3 3 1 : Three of a Kind
*** ROUND 3
HAL: 1 2 5 1 3 : One Pair
You: 4 4 4 5 4 : Four of a Kind
*** ROUND 4
HAL: 6 6 6 3 4 : Three of a Kind
You: 2 1 2 6 4 : One Pair
*** ROUND 5
HAL: 3 6 3 2 4 : One Pair
You: 1 2 3 1 5 : One Pair
The game was won by Lyn with a score of 2 to 1 in 5 rounds.\end{code}%

```

Figure 4: Sample output from two runs of the DiceGame program.

the name of the primary developer, purpose for the code, and history of major modifications. These comments may also include the names of collaborators or other sources for the ideas or code in the file. Starting with this first assignment, we would like you to include this information at the beginning of each code file that you create or modify. The comments included in the provided code skeleton for this assignment follow the Javadoc Documentation Format. (See the CS230 *Documentation* page for a link to a discussion of this format.) Please follow these guidelines. View <http://cs.wellesley.edu/~cs230/fall06/DiceGame> to see the result of the documentation web pages automatically generated via javadoc `DiceGame.html`.

What to turn in

Your hardcopy should include your final version of `DiceGame.java` and a transcript with a few sample runs (like the ones above) demonstrating that your program works correctly. Also submit a softcopy as specified on the first page of this assignment.

Problem 2 [50]: Palindrome Checking

A *palindrome* is a word or phrase that reads the same backwards as forwards. Here are some sample palindromes:

```
radar
redivider
Madam, I'm adam.
A man, a plan, a canal: Panama.
Straw? No, too stupid a fad. I put soot on warts.
1,234,543.21
```

As indicated by the above examples, when checking palindromes, we will ignore all characters that are not letters or digits and will also ignore the case of letters. So “Madam, I’m adam.” is treated as if it were “madamimadam”.

In this problem, your goal is to write a palindrome checker that determines whether a string is a palindrome. Your palindrome checker should be embodied in a `Palindrome` class in a file named `Palindrome.java` that you should create (from scratch) in the directory `~/cs230/ps1`. Your `Palindrome` program should work in two modes:

1. *String Mode*: In string mode, which is invoked via `java Palindrome <string>`, your palindrome checker should determine whether the single string argument `<string>` is a palindrome. For example:

```
[fturbak@puma ps1] java Palindrome radar
"radar" is a palindrome
[fturbak@puma ps1] java Palindrome computer
"computer" is not a palindrome
[fturbak@puma ps1] java Palindrome "A man, a plan, a canal: Panama."
"A man, a plan, a canal: Panama." is a palindrome
[fturbak@puma ps1] java Palindrome "A man, a plan, a cabal: Panama."
"A man, a plan, a cabal: Panama." is not a palindrome
[fturbak@puma ps1] java Palindrome 1,234,543.21
"1,234,543.21" is a palindrome
[fturbak@puma ps1] java Palindrome 1,234,5463.21
"1,234,5463.21" is not a palindrome
```

Because the Linux shell treats spaces as argument separators, a string containing spaces must be delimited by double quotes. This tells Linux to treat all the characters between the double quotes as a single string argument. Linux will strip the double quotes from this string before passing it to Java.

The Linux shell also treats certain characters specially, such as `'!'`. It is necessary to escape these with a backslash. For example:

```
[fturbak@puma ps1] java Palindrome "Golf? No sir -- prefer prison flog!"
bash: !": event not found
[fturbak@puma ps1] java Palindrome "Golf? No sir -- prefer prison flog\!"
"Golf? No sir -- prefer prison flog\!" is a palindrome
```

2. *File Mode* In file mode, which is invoked via `java Palindrome -file <filename>`, your palindrome checker should check each line in the file named `<filename>`. Each line in the file should be echoed to the console, separated by a blank line. Lines that are palindromes should be prefixed with the characters `*`, while lines that are not palindromes should be prefixed with `o`. For example, suppose the file `test.txt` contains the following lines:

```
Madam, I'm adam.
Mister, I'm adam.
Madam, I'm eve.
No lemon, no melon.
No lemons, no melons.
```

Then here is a sample use of the Palindrome program in file mode:

```
[fturbak@puma ps1] java Palindrome -file test.txt
*:Madam, I'm adam.

o:Mister, I'm adam.

o:Madam, I'm eve.

*:No lemon, no melon.

o:No lemons, no melons.
```

If there is no file with the given filename, the program should indicate this fact, as shown below:

```
[fturbak@puma ps1] java Palindrome -file foo.txt
There is no file named foo.txt
```

Any attempt to use the Palindrome program that does not match one of the above two modes should display a message that suggests the correct usage:

```
[fturbak@puma ps1] java Palindrome
usage: java Palindrome <string> OR java Palindrome -file <filename>
[fturbak@puma ps1] java Palindrome foo bar
usage: java Palindrome <string> OR java Palindrome -file <filename>
[fturbak@puma ps1] java Palindrome foo bar baz
usage: java Palindrome <string> OR java Palindrome -file <filename>
```

There are many ways to implement this program. Choose a palindrome-checking strategy that is easy to understand. Make liberal use of auxiliary methods to clarify your program. Your goal is not only to develop a working program, but to create a program whose structure is exceptionally clear.

In your hardcopy, you should submit your final `Palindrome.java` program, along with a transcript demonstrating that your program works as specified. To test file mode, you must try your program on the files `~/cs230/ps1/sentences.txt` and `~/cs230/ps1/big.txt` (the latter file contains a *very* long palindrome). However, to save paper, do *not* include `big.txt` in your transcript. Also submit a softcopy as specified on the first page of this assignment.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS230 Problem Set 1

Due 1:30pm Tuesday February 13

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with*):

By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.

Signature(s):

*In the **Time** column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	Time	Score
General Reading		
Problem 1 [50]		
Problem 2 [50]		
Total		