

Problem Set 1

Due: 11:59pm Thursday, September 16

Revisions:

Sep. 12: In the first full paragraph of Problem 1, the occurrences of `cards1()`, `cards2()`, and `cards3()` should not have the letter `s` in them.

Sep. 15: The original specification of `completeSet` claimed there was a working solution requiring no occurrences of `if`. This is false. What it should say is “A working solution can be implemented in ≤ 20 lines of code using no more than two occurrences of `if`.”

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.

Note:

The rationale for the midnight Thursday (as opposed to a 6pm Friday deadline) is that I am leaving for a conference on the morning of Friday, September 17 and want to be able to grade your assignments while I'm on my trip.

Reading:

- Handouts #1 – #7;
- Read Chapters 1–7 of Downey's online Java book (accessible from the Resource Links section of the CS230 home page).

Submission:

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 6pm 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 `SetHand.java`.

Each team should also submit a single softcopy (consisting of your 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 commands in Linux in the account of the team member being used to store the code.

```
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 `telnet/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 three files:

1. `SetHand.java`: This file contains skeletons for each of Java methods you need to write for this assignment. It also contains code for testing these methods. This is the only one of the three files that you should edit as part of this assignment. Note that for several problems it is helpful to define your own helper methods.
2. `SetCard.java`: This file contains an implementation of the `SetCard` abstraction discussed in class. You do not need to study it, but may learn how to concisely solve certain problems if you do (such as how to implement the `equals` and `compareTo` methods of the `SetHand` class).
3. `Tester.java`: This file contains an implementation of the testing framework used in this and other assignments. You do not need to study the testing code.

e. : Compiling SetHand.java

Next you should compile the `SetHand.java` program. To do this, first make sure that your current directory is set appropriately by executing the following command in a Linux shell (you only need to do this once):

```
cd "~/cs230/ps1"
```

Now you can compile it by executing the following Linux command:

```
javac SetHand.java
```

The `SetHand.java` file you are initially provided with should compile without errors.

To do the rest of the problems on this assignment, you will need to edit `SetHand.java`. Every time you change the `SetHand.java` file, you will need to recompile it via `javac`. As you edit the file, you may introduce compile-time errors. If any errors are reported by the compiler, you will need to fix them. For more details on how to do this, see Section 7 of Handout #3.

f. : Executing SetHand

Next you should invoke the `main` method of the `SetHand` class. You do this by executing the following Linux command after compiling `SetHand.java`:

```
java SetHand
```

Note that you do *not* include the `.java` suffix when executing the program (but *must* include it when compiling the program).

The initial method implementations in the `SetHand` class are *stubs* – methods whose very simple bodies have the right type but in almost all cases compute the wrong answer. Stubs allow us to compile a file before we have figured out how to implement all the methods. On this assignment, stubs are provided for you. In future assignments, you will need to provide your own stubs.

The `main` method of the `SetHand` class performs a series of tests for each of the methods in the class. If a test ends in `OK!`, it succeeded, but if `ERROR` is reported then the test has failed. Because the initial implementations are stubs, most of the test cases will fail. That's OK – as you replace the stubs by real method bodies, more and more of the test cases will succeed.

The `main` method has been configured so that you can test a particular method by providing an argument to `java SetHand`. For instance, if you want to test just the `fromString` method, you can execute:

```
java SetHand fromString
```

The following arguments to `java SetHand` are supported: `cardConstructor`, `stringCardConstructor`, `stringConstructor`, `toString`, `fromString`, `equals`, `compareTo`, `isSet`, and `completeSet`. Executing `java SetHand` without an argument performs the tests on all the methods.

See Appendix C for a discussion of the `SetHand` testing methods.

The rest of this assignment involves the implementation of some simple aspects of the Game of Set¹, which was introduced in class. There are two Java classes you need to understand:

1. The `SetCard` class models cards in the game. A contract for `SetCard` is presented in Appendix A. In addition to the methods presented in lecture, it also includes an `equals` method and a `compareTo` method. You have been provided with a working version of this class in the file `SetCard.java`.
2. The `SetHand` class models hands consisting of three cards. A contract for `SetHand` is presented in Appendix B. The file `SetHand.java` contains stubs for each of the methods in the contract, along with appropriate testing methods. Your goal is to replace the stubs by versions of the method that work in all cases. For some methods there are simple and concise solutions as well as verbose and complex solutions. You should strive to make all your method definitions as simple as possible. It is often the case that method definitions can be simplified by using one or more helper methods. In particular, if you find yourself using the same coding idiom more than once, you should try to abstract over that idiom by introducing a helper method.

Problem 1 [25]: Constructors and Selectors

Begin by implementing the following two constructor methods² and three selector methods for `SetHand`:

```
public SetHand (SetCard sc1, SetCard sc2, SetCard sc3); // card constructor
public SetHand (String s1, String s2, String s3); // string card constructor
public SetCard card1 ();
public SetCard card2 ();
public SetCard card3 ();
```

Your implementation should use a single instance variable `cards` that holds an array of three cards.

The specification requires that `card1()`, `card2()`, and `card3()` should return the three cards from smallest to largest according to the ordering on cards (see the definition of `compareTo` in Appendix A). There are many ways to do this, but it is recommended that you store the three cards into `cards` in sorted order. If you don't know how to do this right now, you can implement simple constructors that ignore order. Later, you can return to this problem to consider order.

The constructor specifications also require that `RuntimeExceptions` be thrown in certain circumstances. You can do this as follows:

```
throw new RuntimeException (message);
```

where `message` is a string indicating the nature of the error.

You can test your methods via the following Linux commands:

```
java SetHand cardConstructor
java SetHand stringCardConstructor
```

These commands invoke testing methods that are at the end of the `SetHand.java` file. Each testing method uses its own **test suite**, a collection of cases to be tested. The test suites that are provided for you are very rudimentary, containing only a few test cases each. It is possible for your solution code to pass all the test cases in a suite but still contain bugs that weren't caught by the test cases

¹See <http://www.setgame.com/set> for more details about the game. Our version of the game differs from the "official" version in a two ways: our shapes are circles, squares, and triangles rather than diamonds, ovals, and squiggles; and our colors are blue, green, and red rather than green, purple, and red.

²There is a third constructor method, `public SetHand (String s)`, that has already been provided for you. This third constructor method is implemented in terms of the `fromString` method you will write in Problem 2.

provided. For this reason, in this problem and in the following problems, you are encouraged to extend the test suites with more test cases. See Appendix C for details on how to add new test cases to a test suite.

Problem 2 [25]: toString and fromString

Implement the `toString` and `fromString` methods for `SetHand`. A fully working version of `fromString` filters out whitespace. The class method `Character.isWhitespace` is handy for determining which characters are whitespace.

If you don't know how to filter out whitespace right now, you can implement a simple version of `fromString` that assumes that its input has no whitespace. Later, you can return to handle whitespace filtering.

You can test your methods via the following Linux commands:

```
java SetHand toString
java SetHand fromString
```

As above, you are encouraged to extend the suite of test cases for these methods.

Problem 3 [10]: equals and compareTo

Implement the `equals` and `compareTo` methods for `SetHand`. For each of these methods there is a very simple solution. *Hint*: Study the implementation of the `equals` and `compareTo` methods of the `SetCard` method.

You can test your methods via the following Linux commands:

```
java SetHand equals
java SetHand compareTo
```

As above, you are encouraged to extend the suite of test cases for these methods.

Problem 4 [30]: completeSet

Implement the `completeSet` method for `SetHand`. Aim for a simple and concise solution. If you find yourself entangled in a rat's nest of conditionals, you are going down the wrong path. A working solution can be implemented in ≤ 20 lines of code using no more than two occurrences of `if`. *Hints*: (1) each of the four card attributes can be handled (almost) uniformly, with a slight tweak for the number attribute; (2) it may help to convert between ASCII characters and integers using `(int)` and `(char)` (what is the “sum” of 'e', 'f', and 'h'?); and (3) use one or more helper methods.

You can test your methods via the following Linux command:

```
java SetHand completeSet
```

As above, you are encouraged to extend the suite of test cases for these methods.

Problem 5 [10]: isSet

Implement the `isSet` method for `SetHand`. It is possible to do this as a “one-liner” method that invokes `completeSet` in a clever way.

You can test your method via the following Linux command:

```
java SetHand isSet
```

As above, you are encouraged to extend the suite of test cases for these methods.

Appendix A: SetCard Contract

The `SetCard` class models a single card in the Game of Set. Such a card has four attributes, and three possible values for each attribute: (1) *number*: 1, 2, or 3; (2) *shading*: empty, filled, or hatched; (3) *color*: blue, green, or red; and (4) *shape*: circle, square, or triangle.

Public Constructor Methods:

```
public SetCard (int number, char shading, char color, char shape);
```

Creates a card with the specified attributes. `number` should be either 1, 2 or 3; `shading` should be either 'e' (empty), 'f' (filled) or 'h' (hatched); `color` should be either 'b' (blue), 'g' (green) or 'r' (red); and `shape` should be either 'c' (circle), 's' (square) or 't' (triangle). For example, `new SetCard(2, 'f', 'g', 'c')` creates a card with two filled green circles. If one or more parameters is not one of the allowed values for an attribute, a `RuntimeException` is thrown.

Public Instance Methods:

```
public int number ();
```

Returns the number attribute of this card (either 1, 2, or 3).

```
public char shading ();
```

Returns the shading attribute of this card (either 'e' (empty), 'f' (filled), or 'h' (hatched)).

```
public char color ();
```

Returns the color attribute of this card (either 'b' (blue), 'g' (green), or 'r' (red)).

```
public char shape ();
```

Returns the shape attribute of this card (either 'c' (circle), 's' (square), or 't' (triangle)).

```
public String toString ();
```

Returns a string representation of this card. A string representation of a card has four characters, each of which specifies one of the four attributes of the card: the first is a digit specifying the number, the second specifies the shading, the third specifies the color, and the fourth specifies the shape. For instance "2ghs" is the string for the card that has two green hatched squares. Fig. 1 shows the string representations for all 81 cards in the Game of Set.

```
public boolean equals (Object x);
```

Returns `true` if `x` is a `SetCard` with the same four attributes as this card, and `false` otherwise.

```
public int compareTo (Object x);
```

If `x` is a `SetCard` instance, returns a negative number if this card comes before `x` in the card ordering (see the definition of card ordering below), 0 if this card is equal to `x` in the card ordering, and a positive number if this card comes after `x` in the card ordering. If `x` is not a `SetCard` instance, throws a `ClassCastException`.

The ordering on cards is defined as follows. Consider the following ordering on attributes: for numbers, $1 < 2 < 3$; for shading, empty < filled < hatched; for colors, blue < green < red; and for shapes, circle < squares < triangle. Viewing the four attributes in the order number, shading, color, and shape induces a **lexicographic ordering** (i.e., dictionary ordering) on cards in which cards are first compared by number, then by shading, then by color, and

```
1ebc, 1ebs, 1ebt, 1egc, 1egs, 1egt, 1erc, 1ers, 1ert,
1fbc, 1fbs, 1fbt, 1fgc, 1fgs, 1fgt, 1frc, 1frs, 1frt,
1hbc, 1hbs, 1hbt, 1hgc, 1hgs, 1hgt, 1hrc, 1hrs, 1hrt,
2ebc, 2ebs, 2ebt, 2egc, 2egs, 2egt, 2erc, 2ers, 2ert,
2fbc, 2fbs, 2fbt, 2fgc, 2fgs, 2fgt, 2frc, 2frs, 2frt,
2hbc, 2hbs, 2hbt, 2hgc, 2hgs, 2hgt, 2hrc, 2hrs, 2hrt,
3ebc, 3ebs, 3ebt, 3egc, 3egs, 3egt, 3erc, 3ers, 3ert,
3fbc, 3fbs, 3fbt, 3fgc, 3fgs, 3fgt, 3frc, 3frs, 3frt,
3hbc, 3hbs, 3hbt, 3hgc, 3hgs, 3hgt, 3hrc, 3hrs, 3hrt
```

Figure 1: String representations of the 81 cards in the Game of Set, ordered from least to greatest.

finally by shape. Fig. 1 shows the ordering of the string representation all 81 cards in the Game of Set ordered from least to greatest according to the card ordering. The ordering has been chosen so that it coincides with the lexicographic ordering of the strings representing the cards. For example, two green filled triangles is “less than” two green hatched squares because "2gft" is less than "2ghs" in dictionary order.

Public Class Methods:

```
public static SetCard fromString (String s);
```

Returns a card whose string representation is `s`. E.g., `SetCard.fromString("2ert")` returns a card with two empty red triangles. Throws a `RuntimeException` if the string `s` is not a valid string representation of a card.

Appendix B: SetHand Contract

The `SetHand` class models a hand of three distinct cards in the Game of Set.

Public Constructor Methods:

public `SetHand (SetCard sc1, SetCard sc2, SetCard sc3);`

Creates a hand with the three given cards. If the three cards are not pairwise distinct, a `RuntimeException` is thrown.

public `SetHand (String s1, String s2, String s3);`

Creates a hand whose three cards are specified by the three string representations `s1`, `s2`, and `s3`. For example, `new SetHand("2fgs", "3hrt", "1ebc")` creates a set whose cards are: 2 filled green squares, 3 hatched red triangles, and 1 empty blue circle. If any of the cards is not a valid string representation of a card, or if the three cards are not pairwise distinct, a `RuntimeException` is thrown.

public `SetHand (String s);`

Returns a hand whose string representation is `s`, as determined by the `fromString` class method. See the `fromString` documentation below for details.

Public Instance Methods:

public `SetCard card1 ();`

Returns the least card in this hand according to the ordering on cards. For example, if `sh` is `new SetHand("2fgs", "3hrt", "1ebc")`, then `sh.card1().toString()` is `"1ebc"`.

public `SetCard card2 ();`

Returns the middle card in this hand according to the ordering on cards. For example, if `sh` is `new SetHand("2fgs", "3hrt", "1ebc")`, then `sh.card2().toString()` is `"2fgs"`.

public `SetCard card3 ();`

Returns the greatest card in this hand according to the ordering on cards. For example, if `sh` is `new SetHand("2fgs", "3hrt", "1ebc")`, then `sh.card3().toString()` is `"3hrt"`.

public `String toString ();`

Returns a string representation of this hand. A string representation of a hand consists of the string representations of the three cards, in sorted order, separated by commas and delimited by square brackets. For example, if `sh` is `SetHand("2fgs", "3hrt", "1ebc")`, then `sh.toString()` is `"[1ebc,2fgs,3hrt]"`.

public `boolean equals (Object x);`

Returns `true` if `x` is a `SetHand` with the same three cards as this hand, and `false` otherwise.

public `int compareTo (Object x);`

If `x` is a `SetHand` instance, returns a negative integer if this hand comes before `x` in the hand ordering, 0 if this hand is equal to `x` in the hand ordering, and a positive number if this hand is greater than `x` in the hand ordering. If `x` is not a `SetCard` instance, throws a `ClassCastException`.

The ordering on two hands `h1` and `h2` is the lexicographic ordering on the three cards in each hand, considered from smallest to largest. That is, `h1` and `h2` are first compared by `card1()`,

then by `card2()`, and finally by `card3()`. This ordering coincides with the dictionary ordering on the string representations of the hands. For example, the following string representations of hands are shown from smallest to largest in the hand ordering:

```
[1fgc,1fgs,1fgt], [1fgc,1fgs,2hgc], [1frc,2hrt,3ers],  
[1frc,3hrs,3hrt], [2ebs,2egt,2erc], [3hbc,3hrs,3hrt]
```

```
public boolean isSet ();
```

Returns `true` if this hand is a *set* in the Game of Set – i.e., if for each of the four card attributes, the three cards in this hand either have the same value of the attribute or have pairwise distinct values of the attribute. An alternative characterization is that the three cards in this hand are a set as long as there is no attribute for which two cards have the same value, but the other card has a different value. For example, in the above list of six hands, there are three sets: `[1fgc,1fgs,1fgt]` `[1frc,2hrt,3ers]`, and `[2ebs,2egt,2erc]`.

Public Class Methods:

```
public static SetHand fromString (String s);
```

Returns a hand whose string representation is `s`. For example,

```
SetHand.fromString("[1ebc,2fgs,3hrt]")
```

is equivalent to `SetHand("1ebc", "2fgs", "3hrt")`. The order of cards in the string `s` does not matter, and whitespace in `s` is ignored. E.g., the following strings are interpreted in the same way by `fromString`:

```
"[1ebc,2fgs,3hrt]"  
"[3hrt,1ebc,2fgs]"  
"[1ebc, 2fgs, 3hrt]"  
"[2 f gs, 3h rt,1e b c]"  
"[ 3hrt , 1 e\tb\nc, 2 f g s]"
```

```
public static SetHand completeSet (SetCard sc1, SetCard sc2);
```

If `sc1` and `sc2` are distinct cards, returns the unique hand containing `sc1` and `sc2` that is a set in the Game of Set. E.g.,

```
SetHand.complete(SetCard.fromString("2ebt"), SetCard.fromString("3ert"))
```

denotes the set whose string representation is `"[1egt,2ebt,3ert]"`. If `sc1` and `sc2` are the same card, throws a `RuntimeException`.

Appendix C: Testing Details

This appendix discusses the testing methods in the `SetHand` class. You will need to understand how the test suites for each method are constructed if you want to extend them for the problems in your assignment.

There are eight testing methods in `SetHand.java`: `testCardConstructor`, `testStringConstructor`, `testToString`, `testFromString`, `testEquals`, `testCompareTo`, `testIsSet`, and `testCompleteSet`. Each of the testing methods is an instance of the following template:

```
public static void tester-name () {
    Tester t =
        new Tester() {
            public String answer (String input) {
                answer-body
            }
        };
    t.test(name, entries);
}
```

As a concrete example, the testing method `testToString` is shown in Fig. 2.

```
public static void testToString () {
    Tester t =
        new Tester() {
            public String answer (String input) {
                SetCard sc1 = SetCard.fromString(input.substring(0,4));
                SetCard sc2 = SetCard.fromString(input.substring(5,9));
                SetCard sc3 = SetCard.fromString(input.substring(10,14));
                SetHand sh = new SetHand(sc1,sc2,sc3);
                return sh.toString();
            }
        };
    t.test("Testing toString() method:",
        new String [] [] {
            // Each test entry is of form input, expected output
            { "1ebc,2fgs,3hrt", "[1ebc,2fgs,3hrt]" },
            { "3hrt,1ebc,2fgs", "[1ebc,2fgs,3hrt]" },
            { "1ebc,1ers,2frs", "[1ebc,1ers,2frs]" }
            // Add more entries here.
        });
}
```

Figure 2: Implementation of the testing method `testToString`.

The invocation `new Tester() {...}` is a so-called *anonymous inner class constructor invocation* that creates an instance of an anonymous subclass of the `Tester` class having the given `answer` method. This instance is named `t`, and the `Tester` `test` method is invoked on `name` and `entries`. The `name` argument is a string indicating the test being performed. For example, in `testToString`, `name` is `"Testing toString() method:"`.

The `entries` argument is an array of test suite entries, where each entry is a two-element string array consisting of an *input string* and an *expected output string*. For example, in `testToString`, `entries` is the array created by the notation

```

new String [] [] {
    { "1ebc,2fgs,3hrt", "[1ebc,2fgs,3hrt]" },
    { "3hrt,1ebc,2fgs", "[1ebc,2fgs,3hrt]" },
    { "1ebc,1ers,2frs", "[1ebc,1ers,2frs]" }
}.

```

Each entry describes a test case for the method being tested. For instance, the entry

```
{ "1ebc,2fgs,3hrt", "[1ebc,2fgs,3hrt]" }
```

indicates that for input string "1ebc,2fgs,3hrt" it is expected that the output string will be "[1ebc,2fgs,3hrt]".

When the `test` method is invoked (on zero arguments), it first displays a dotted line, followed by *name*, followed by a sequence of lines showing each test case, and ends with a dotted line. For example, here is the testing output displayed when `testToString()` is invoked on the initial implementation of `SetHand`:

```

[lyn@jaguar ps1] java SetHand toString
-----
Testing toString() method:
1ebc,2fgs,3hrt => [1ebc,2fgs,3hrt] : OK!
3hrt,1ebc,2fgs => [1ebc,2fgs,3hrt] : OK!
1ebc,1ers,2frs =>
*****ERROR*****
Expected:
[1ebc,1ers,2frs]
Actual:
[1ebc,2fgs,3hrt]
*****

```

Each entry is processed by invoking the `answer` method of the tester instance `t` on the input string to produce an *actual output string*. If the actual output string is the same as the expected output string, the line

```
input-string => actual-output-string : OK!
```

is displayed. This means that the test represented by the entry has succeeded. However, if the actual output string is different from the expected output string, the test represented by the entry has failed, and the following output is displayed:

```

 =>
*****ERROR*****
Expected:
expected-output-string
Actual:
actual-output-string
*****

```

This output highlights the mismatch between the expected output string and the actual output string.

On this assignment, you have been provided with testing methods for each of the methods you are supposed to write. In future assignments, you will be expected to write your own testing methods from scratch.

Each of the testing methods provided in `SetHand.java` contains only a few test case entries. As part of doing this assignment, you are encouraged to add more entries to each tester to better test the method for that tester.

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

CS230 Problem Set 1
Due 11:59pm Thursday September 16

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 [25]		
Problem 2 [25]		
Problem 3 [10]		
Problem 4 [30]		
Problem 5 [10]		
Total		