Com S 227
Fall 2017
Assignment 4
300 points

Due Date: Friday, December 8, 11:59 pm (midnight)

***NO LATE SUBMISSIONS** - All work must be turned in Friday*

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html , for details.

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy acknowledgement* on the Homework page on Blackboard. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Please start the assignment as soon as possible and get your questions answered right away! There are likely to be more errors and ambiguities in the spec than in previous assignments, and part of your job is to seek clarification about them.

Check Piazza regularly for updates to this document.

Introduction

The purpose of this assignment is to give you some experience working with inheritance to implement a collection of related classes.

A portion of your grade on this assignment (roughly 15% to 20%) will be determined by the logical organization of classes in this hierarchy and by how effectively you have been able to use inheritance to minimize duplicated code.

Overview

In this homework you will implement a collection of classes for evaluating hands in card games in which a winner is determined by a relative *ranking*[1] of hands. A *hand* is just a combination of a fixed number of cards. The cards that are used in the game, the number of cards in a hand, and the definition of which combinations rank higher than others, all depend on the game that is being played.

One example of such a game is *poker*, in which the cards consist of a standard 52-card deck and a hand usually consists of five cards. In poker, the ranking of the hands is based on their actual mathematical probability, with least likely combinations ranked higher than more likely combinations.

However, our system is not restricted to standard poker hands, decks of 52, hands of five cards, nor the usual ranking of hands. The idea is that each type of hand (such as a straight, two pair, full house, etc., as seen in poker) will be represented by a class we will call an *evaluator*. Each evaluator can be assigned a ranking relative to others for use in a given type of game.

The purpose of an evaluator is to answer questions such as the following:

Given a list of cards, does it satisfy the criteria for the type of hand represented by this evaluator? For example, the criteria for a "two pair evaluator" would be that are the list includes four cards, two having the same rank and the two others having the same rank

Given a list of cards, is it possible to select a subset of those cards that satisfies the criteria for the evaluator? For example, given cards with ranks [2, 3, 5, 5, 3, 7, 3, 2] (we can ignore the suits for now), it is possible to select a subset such as [3, 3, 2, 2], which would satisfy the criteria for two pairs.

Given a list of cards and the total size of the hand, find the best possible hand that contains a subset satisfying the criteria for the evaluator. For example, given [2, 3, 5, 5, 3, 7, 3], the best possible five-card hand consisting of two pairs would be [5, 5, 3, 3, 7]. Note that in standard poker rules, this is not the best possible hand from these cards, since it would be beaten by the full house [3, 3, 3, 5, 5]. However, each evaluator is only concerned with one type of hand. The relative rankings between evaluators is determined separately when defining a collection of evaluators for a particular card game.

[1] Potential confusion here: the *rank* of a hand (combination of cards) should not be confused with the *rank* of a single card, which is just its numerical value.

Summary of tasks

You will create one abstract class, **`AbstractEvaluator`**, plus a minimum of nine concrete types that directly or indirectly extend it. Skeletons for these classes can be found in the sample code:

```
AllPrimesEvaluator
AllSuitsEvaluator
CatchAllEvaluator
FourOfAKindEvaluator
FullHouseEvaluator
OnePairEvaluator
StraightEvaluator
StraightFlushEvaluator
ThreeOfAKindEvaluator
```

(The list includes generalized versions of some standard poker hands, plus the types **`AllPrimesEvaluator`** and **`AllSuitsEvaluator`** which we just invented.) **`AbstractEvaluator`** implements the **`IEvaluator`** interface, and therefore *all* the classes above are subtypes of **`IEvaluator`**.

An important part of your task is to organize the classes into a hierarchy to eliminate redundant or duplicated code as much as possible. At a minimum, operations that are common to all or most of the concrete types should be factored into **`AbstractEvaluator`**. There may be other similarities between types that might be able to share code (e.g., how is ThreeOfAKind similar to FourOfAKind? How is a Straight related to a StraightFlush?) You can create additional abstract classes if you wish in order to exploit such similarities.

Please note that you are NOT being asked to implement an entire card game, just the classes listed above. These might form a useful part of a card game implementation, but that is outside the scope of this assignment.

Detailed specifications

For details on the **`IEvaluator`** interface, see the javadoc. For detailed specifications of the required concrete types, consult the javadoc for each type in the skeleton code.

Other classes in the API

The **`api`** package in the sample code includes the **`IEvaluator`** interface and the following additional types. *You should not modify any of these classes/interfaces*. The class **`Card`** and the enumerated type **`Suit`** are similar to the ones you saw in lab. Some methods are added for conveniently creating arrays of cards for testing purposes. In addition, **`Card`** implements the **`Comparable`** interface so that arrays of cards can be sorted using **`Arrays.sort()`**. Cards are ordered by descending rank; however, aces (cards of rank 1) come first. Within cards of the same rank, the ordering is by suit with SPADES the highest and CLUBS the lowest. Card ranks may be higher than 13.

The class **`Hand`** represents a collection of cards satisfying some evaluator's criteria. A **`Hand`** is constructed from a list of *main cards*, those that directly satisfy the evaluator's criteria as determined by the **`canSatisfy`** method, and a list (possibly empty) of *side cards*. The number of main cards is the value returned by the evaluator's **`numMainCards()`** method, and the total number of cards (main cards plus side cards) is the value of the evaluator's **`totalCards()`** method. The array of side cards is always ordered according to the Card **`compareTo`** method (i.e., sorted with highest card first). The main cards are normally ordered the same way, but there are certain exceptions, described in more detail below.

The **`Hand`** class implements the **`Comparable`** interface. The ordering of hands is defined as follows:

1. Hands with higher ranking (smaller number) come first.
2. Among hands with the same ranking, hands are ordered lexicographically by the main cards.
3. Among hands with the same ranking and same main cards, hands are ordered lexicographically by the side cards.

(Lexicographic ordering of two arrays means the first elements are compared, and if they are the same then the second elements are compared, and so on. You can take a look at the **`compareTo`** method in the **`Hand`** class to see exactly how this is done.)

Here are some examples. We will use the notation of the **`toString`** method, that is, [4s 4c : 7h 6h 5h] represents a hand with five total cards: main cards consisting of the four of spades and the four of clubs, plus side cards consisting of the six, five, and four of hearts. Suppose we are working with a TwoPairEvaluator; since the suits don't matter here, we can just write [4 4 : 7 6 5]. The hand [4 4 : 7 6 5] comes before (i.e., is better than) [3 3 : K Q J], based on comparison of main cards, while [4 4 : 8 3 2] comes before [4 4 : 7 6 5] based on comparison of side cards. You can see that for this to work, we have to have the side cards sorted as 8-3-2, with the highest card first, and not in some other order.

The **`Hand`** class is already written and you can't modify it, so your main responsibility in using the **`Hand`** class is to ensure the two arrays you supply to the constructor are correctly ordered, in order for the lexicographic ordering in (2) and (3) to be right. In most cases, just making sure they are sorted according to the `compareTo` method of `Card` is sufficient. However, there are two cases in which this won't work.

1. *A full house where the hand size is an odd number*. When a hand has an odd number of cards, there are two groups of cards with matching ranks, one larger than the other, e.g. [4 4 4 K K] (all main cards, no side cards). This hand is beaten by [5 5 5 2 2], even though the first hand has higher cards; that is, the larger group is compared first. If the first hand was sorted normally, it would be [K K 4 4 4] and the lexicographic comparison would put it first. So in order for this to work, your FullHouseEvaluator must create the hand with the main cards ordered with the larger group first in case of an odd hand size. (Typically this happens in the `createHand` method.)

2. *A straight with an ace as the low card*. [6 5 4 3 2] should defeat (come before) [5 4 3 2 A], but if the latter is sorted normally as [A 5 4 3 2], a lexicographic comparison would put it first.

Subset Finder

In order to answer questions such as "Is there a subset of the given cards that satisfies this evaluator's criteria?" it is convenient to be able to enumerate all the subsets of a given size. You are provided with a class called `SubsetFinder` in the api package of the sample code. It has a static method that, given two numbers *n* and *k*, generates a list of all *k*-element subsets of the numbers *0* through *n* - 1. Each subset is represented by an int array with the values in ascending order. To find subsets of (for example) an array of `Card` objects, you can use the given int values as indices. The `Util` class has methods for doing this and there are some usage examples in a main() method in `SubsetFinder`. The Hand class also has a constructor for creating a hand using int indices to select the main cards from an array of all the cards.

Sample Usage

```
// Create a one pair evaluator that has ranking 3
// and hand size of four cards
IEvaluator eval = new OnePairEvaluator(3, 4);
System.out.println(eval.getName());  // "One Pair"

// Create an array of Cards to test.  This is equivalent to
// Card[] cards = {new Card(2, Suit.CLUBS), new Card(2, Suit.DIAMONDS)};
// (see the Card class documentation)
// This array should satisfy the One Pair evaluator.
Card[] cards = Card.createArray("2c, 2d");
System.out.println(Arrays.toString(cards));
System.out.println(eval.satisfiedBy(cards));  // true
```

```
// This one should not satisfy the one pair evaluator
cards = Card.createArray("Kc, Qd");
System.out.println(Arrays.toString(cards));
System.out.println(eval.satisfiedBy(cards));  // false

// This one won't either, since it has more than the
// required number of cards
cards = Card.createArray("2c, 2d, 3h");
System.out.println(Arrays.toString(cards));
System.out.println(eval.satisfiedBy(cards));  // false

// However, it contains a subset that does
System.out.println(eval.canSubsetSatisfy(cards)); // true

// Try a bigger array.  We'll use Arrays.sort to get them
// in order, as required by the IEvaluator API.  This
// illustrates the ordering of the Card compareTo() method
cards = Card.createArray("6s, Jd, Ah, 10h, 6h, Js, 6c, Kh, Qh");
Arrays.sort(cards); // now [Ah, Kh, Qh, Js, Jd, 10h, 6s, 6h, 6c]
System.out.println(Arrays.toString(cards));
System.out.println(eval.canSubsetSatisfy(cards)); // true

// Define a subset consisting of indices 6 and 8
// and have the evaluator create a Hand from those cards
int[] subset = {6, 8};
Hand hand = eval.createHand(cards, subset);
System.out.println(hand); // One Pair (3) [6s 6c : Ah Kh]

// Subset at indices 0 and 3 doesn't satisfy evaluator, so
// createHand returns null
int[] subset2 = {0, 3};
hand = eval.createHand(cards, subset2);
System.out.println(hand); // null

// Finds the best hand from these cards (for this evaluator)
// which will be the pair of jacks plus ace and king
hand = eval.createBestHand(cards);
System.out.println(hand); // One Pair (3) [Js Jd : Ah Kh]

// Create a list of some evaluators for 5-card hands
ArrayList<IEvaluator> evaluators = new ArrayList<IEvaluator>();
evaluators.add(new OnePairEvaluator(3, 5));
evaluators.add(new FullHouseEvaluator(1, 5));
evaluators.add(new StraightEvaluator(2, 5, 13));

// Now find the best hand we can get from these cards
Hand best = null;
for (IEvaluator e : evaluators)
{
  Hand h = e.createBestHand(cards);
  if (best == null || h != null && h.compareTo(best) < 0)
  {
    best = h;
  }
}
// Full House (1) [6s 6h 6c Js Jd]
System.out.println("Best hand: " + best);
```

Notes

1. All of your code should be in the package **`hw4`**.
2. Don't write your own sorting algorithm. If you need to sort cards or hands, use **`Arrays.sort()`** or **`Collections.sort()`**.
3. A list of cards may satisfy more than one evaluator. For example, the hand [5 5 5 7 11] would satisfy Three of a Kind, One Pair, Catch All, and AllPrimes.
4. Note that the constructor for each of the concrete evaluator types is given in the skeleton code. *You may not modify the constructor parameters.* You can implement a constructor for **`AbstractEvaluator`** any way that you see fit.
5. Note also that the required string name for each of the evaluator types is given in the class javadoc comment.
6. The **`Suit`** class is an **`enum`**, or enumeration type. These are just like symbolic constants. You can't create them with the **`new`** keyword, just use them by name (as in the usage example above). Most significantly, to check whether two **`Card`** instances have the same suit, you can just use **`==`**, as in: **`if (oneCard.getSuit() == otherCard.getSuit()) {...}`**.

Importing the sample code

See page 5 of the hw3 pdf if you aren't sure how to do this.

Testing and the SpecChecker

As always, you should try to work incrementally and write simple tests for your code as you develop it.

Since test code that you write is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specfications, ask questions when things require clarification, and write your own unit tests.

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

Remember that your instance variables should always be declared `private`, and if you want to add any additional "helper" methods that are not specified, they must be declared `private` (or possibly `protected`) as well.

This SpecChecker will also offer to create a zip file for you that will include all the classes in your hw4 package.

Special documentation and style requirements

- You must add a comment to the top of the `AbstractEvaluator` class with a couple of sentences explaining how you decided to organize the class hierarchy for the evaluators.

- You may not use `public`, `protected` or package-private instance variables. Normally, instance variables in a superclass should be initialized by an appropriately defined superclass constructor. You can create additional `protected` getter/setter methods if you really need them.

Style and documentation

See the special requirements above.

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
    - You will lose points for having lots of unnecessary instance variables
    - All instance variables should be `private`.
- **Accessor methods should not modify instance variables**.
- Each class, method, constructor and instance variable, whether public or private, must have a meaningful and complete Javadoc comment. Class javadoc must include the `@author` tag, and method javadoc must include `@param` and `@return` tags as appropriate.
    - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.

  - **When a class implements or overrides a method that is already documented in the supertype (interface or class) you normally do not need to provide additional Javadoc,** unless you are significantly changing the behavior from the description in the supertype. You should include the `@Override` annotation to make it clear that the method was specified in the supertype.

- All variable names must be meaningful (i.e., named for the value they store).
- Your code should not be producing console output. You may add `println` statements when debugging, but you need to remove them before submitting the code.
- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. (A good rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include a comment explaining how it works.)
  - Internal comments always *precede* the code they describe and are indented to the same level.
- Use a consistent style for indentation and formatting.
  - Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to Window->Preferences->Java->Code Style->Formatter and click the New button to create your own "profile" for formatting.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder `assignment4`. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag `assignment4`. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "tt" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

Suggestions for getting started

General advice about using inheritance: It may not be obvious how to go about defining an inheritance hierarchy for the required subtypes of **`AbstractEvaluator`**. A really good way to get started is to forget about inheritance. That is, pick one of the required concrete types, such as **`OnePairEvaluator`**, add the clause **`implements IEvaluator`** to its declaration, and just write the code for all the specified methods. (That is, temporarily forget about the requirement to extend **`AbstractEvaluator`**.) Then, choose another required type, and write all the code for that one. At this point you'll notice that you had to write some of the same code twice. Move the duplicated code into **`AbstractEvaluator`**, and change the declaration for **`OnePairEvaluator`** so it says **`extends AbstractEvaluator`** instead of **`implements IEvaluator`**.

That's a good start, and you can use a similar strategy as you implement the other evaluator types: Write the code first, then look for places where you've got similar code in multiple classes. Does it make sense to define one of them as a subclass of the other? Should the duplicated code be in a common superclass?

There is no one, "perfect" way to make these decisions. All "good" solutions will involve some tradeoffs. Part of the assignment (see "Special Documentation Requirement", below) is for you to provide a brief statement at the top of **`AbstractEvaluator`** that explains your design decisions.

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Homework page on Blackboard, before the submission link will be visible to you.

Please submit, on Blackboard, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw4.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`hw4`**, which in turn contains **at least** the following eleven files:

```
AbstractEvaluator.java
AllPrimesEvaluator.java
AllSuitsEvaluator.java
CatchAllEvaluator.java
FourOfAKindEvaluator.java
FullHouseEvaluator.java
OnePairEvaluator.java
StraightEvaluator.java
StraightFlushEvaluator.java
ThreeOfAKindEvaluator.java
```

The specchecker should automatically zip up everything in your hw4 package. *If you have defined additional files as part of your inheritance hierarchy,* ***check*** *to be sure those additional files are included too*. Please do not include unrelated or unnecessary files. (E.g. your example or test code should not be in the hw4 package).

Please LOOK at the archive you upload and make sure it is the right one!

Submit the zip file to Blackboard using the Assignment 4 submission link and **verify** that your submission was successful by checking your submission history page. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw4**, which in turn should contain the files listed above. Make sure you are turning in **`.java`** files, not the **`.class`** files. You can accomplish this easily by zipping up the **src** directory of your project. (The zip file will include the other **`.java`** files in the project, but that is ok.) The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and not a third-party installation of WinRAR, 7-zip, or Winzip.