# Com S 227
# Spring 2022
# Assignment 4
# 300 points

Due Date: Friday, May 6, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm May 5)

**NO LATE SUBMISSIONS! No extensions. All work must be in Friday night.**

## General Information

**This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, for details. You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas.** 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. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!*

Read this specification carefully. It is not quite like reading a story from beginning to end, and you will probably have to reread some parts many times. See the section "If you have questions" if you find something unclear in this document.

## Introduction

In this homework you will implement a collection of classes for evaluating hands in card games such as poker, in which a winner is determined by a *ranking*[1] of hands. A hand is just a combination of a fixed number of cards. The cards that are used in the game, and the definition of which combinations rank higher than others, depend on the game that is being played. In many versions of poker, the cards consist of a standard 52-card deck and a hand consists of five cards. The ranking of the hands is usually based on their actual mathematical probability, with least likely combinations ranked higher than more likely combinations.

There is a good explanation of standard poker hands on the Wikipedia page,

http://en.wikipedia.org/wiki/List_of_poker_hands

[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.

However, our system will not be restricted to standard poker hands, decks of 52, hands of 5, nor the rankings shown on the Wikipedia page. The idea is that each category 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 there four cards in the list, two each of two different ranks.

**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]. (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 a particular game is configured.)

### Using inheritance

Part of your grade for this assignment (e.g. 20-25%) will be based on how effectively you are able to implement the eight required concrete types with a minimum of code duplication. Here are some specific restrictions. More detailed requirements are provided in the section "Requirements and guidelines regarding inheritance."

- You may not use **`public`**, **`protected`** or package-private variables. (You can provide protected accessor methods or constructors when needed.)
- You may not modify the **`IEvaluator`** interface itself (or anything else in the **`api`** package).

## Summary of tasks

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

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

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

*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.*

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.

## 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, where aces (cards of rank 1) are always considered highest. 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*. (In some references the side cards are referred to as a *kicker*.) The number of main cards is the value returned by the evaluator's **`cardsRequired()`** method, and the total number of main cards plus side cards is the value of the evaluator's **`handSize()`** method. The side cards are always ordered according to the Card **`compareTo`** method (i.e., sorted with highest card first). The constructor allows you to supply more side cards than necessary, and it will take the first ones. The main cards are usually 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 main cards, hands are ordered lexicographically by the side cards.

Note that for purposes of ordering hands, the suits are ignored.

*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 main cards consisting of the four of spades and the four of clubs, with side cards consisting of the six, five, and four of hearts. Since the suits don't matter we can just write [4 4 : 7 6 5]. Assuming we are working with a TwoPairEvaluator, [4 4 : 7 6 5] defeats (comes before) [3 3 : K Q J] (based on comparison of main cards), while [4 4 : 8 3 2] defeats [4 4 : 7 6 5] (based on comparison of side cards). So 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. The general solution to this problem requires a technique called *recursion*, which we have not covered yet. In the meantime, you are provided with a class called **`SubsetFinder`** in the util 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, just use the given int values as indices. There is a main method in the **`SubsetFinder`** class that has some usage examples.

## Some Usage Examples for `Card`, `Hand`, and `IEvaluator`

The following is some example usage code for **`Card`**, **`Hand`** and **`IEvaluator`** (available in **`TryEvaluators.java`**).

```
// 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.canSatisfy(cards));  // true
```

```
// This one should not satisfy the evaluator
cards = Card.createArray("Kc, Qd");
System.out.println(Arrays.toString(cards));
System.out.println(eval.canSatisfy(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.canSatisfy(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, Jh, Ah, 10h, 6h, Js, 6c, Kh, Qh");
Arrays.sort(cards); // now [Ah, Kh, Qh, Js, Jh, 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
int[] subset2 = {0, 3};
hand = eval.createHand(cards, subset2);
System.out.println(hand); // null

// Finds the best hand from these cards (for this evaluator)
hand = eval.getBestHand(cards);
System.out.println(hand); // One Pair (3) [Js Jh : 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.getBestHand(cards);
  if (best == null || h != null && h.compareTo(best) < 0)
  {
    best = h;
  }
}

// Full House (1) [6s 6h 6c Js Jh]
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. 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 code). *You may not modify the parameters.* You can implement a constructor for **`AbstractEvaluator`** any way that you see fit.

5. Note also that the 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()) {...}`**.

## Testing and the SpecChecker

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 specifications, ask questions when things require clarification, and write your own unit tests.

Since the test code 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.**

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.
```

This SpecChecker will also offer to create a zip file for you that will package up everything in your hw4 package. **Please check this carefully. In this assignment you may be including one or more abstract classes of your own, in addition to the 8 required classes, so make sure they have been included in the zip file.**

## Importing the starter code

The starter code includes a complete skeleton of the four classes you are writing. It is distributed as a complete Eclipse project that you can import. It should compile without errors out of the box.

1. Download the zip file. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.

If you have an older version of Java or if for some reason you have problems with this process, or if the project does not build correctly, you can construct the project manually as follows:

1. Extract the zip file containing the sample code (*Note: on Windows, you have to right click and select "Extract All" – it will not work to just double-click it*)
2. In Windows Explorer or Finder, browse to the **`src`** directory of the zip file contents
3. Create a new empty project in Eclipse
4. In the Eclipse Package Explorer, navigate to the src folder of the new project.
5. Drag the **`hw4`**, **`util`**, and **`api`** folders from Explorer/Finder into the **`src`** folder in Eclipse.
6. Copy the remaining top-level files (**`SubsetFinder.java`**, etc) into the **`src`** folder.

## Suggestions for getting started

1. You can make significant progress without using inheritance, so don't let that stop you.

2. Study the usage examples given above and try out the main method in the subset finder.

3. Start with **`OnePairEvaluator`** and directly implement the **`IEvaluator`** interface. (That is, temporarily forget about the requirement to extend **`AbstractEvaluator`**.) Implement **`getName`**, **`getRanking`**, **`cardsRequired`**, and **`handSize`**. These are pretty easy.

4. Experiment with the **`Hand`** class and make sure it makes sense to you. You have to supply an evaluator to construct a **`Hand`**, but it only depends on the four methods above.

5. Start looking at the other methods of **`OnePairEvaluator`**. The **`canSatisfy`** method is a good place to start. You can also implement **`createHand`** pretty easily at this point. Then use **`canSatisfy`** to implement **`canSubsetSatisfy`**.

6. Next you could start implementing another one, say **`StraightEvaluator`**, in the same way. You will quickly discover that much of the code is exactly the same as for **`OnePairEvaluator`**. That gives you a **big clue** as to what code is common to both types and should be moved into the superclass **`AbstractEvaluator`**.

## Requirements and guidelines regarding inheritance

A generous portion of your grade (maybe 20%) will be based on how well you have used inheritance, and possibly abstract classes, to create a clean design with a minimum of duplicated code. Please note that there is no one, absolute, correct answer – you have design choices to make.

*You will explain your design choices in the class Javadoc for* **`AbstractEvaluator`**.

### Specific requirements

**You are not allowed to use non-private variables**. Call the superclass constructor to initialize its attributes, and use superclass accessor methods to access them. If your subclass needs some kind of access that isn't provided by public methods, you are allowed to define your own **`protected`** methods or constructors.

### Other general guidelines

You should not use **`instanceof`** or **`getClass()`** in your code, or do any other type of runtime type-checking, to implement correct behavior. Rely on polymorphism.

- No class should contain extra attributes or methods it doesn't need to satisfy its own specification.

- Do not ever write code like this:

```
public void foo()
{
  super.foo()
}
```

  There is almost never any reason to declare and implement a method that is already implemented in the superclass unless you need to override it to change its behavior. *That is the whole point of inheritance!*

## Style and documentation

**Special documentation requirement: y**ou 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.

Roughly 10 to 15% of the points will be for documentation and code style. Some restrictions and guidelines for using inheritance are described above.

When you are overriding a superclass or interface method, **it is usually NOT necessary to re-write the Javadoc**, unless you are really, really changing its behavior. Just include the @Override annotation. (The Javadoc tool automatically copies the superclass Javadoc where needed.)

The other general guidelines are the same as in homework 3. Remember the following:

- You must add an @author tag with your name to the javadoc at the top of each of the classes you write.
- You must javadoc each instance variable and helper method that you add. Anything you add must be **`private`** or **`protected`**.
- Keep your formatting clean and consistent.
- Avoid redundant instance variables
- Accessor methods should not modify instance variables

## If you have questions

It is extremely likely that the specification will not be 100% clear to you. Part of your job as a developer is to determine what still needs to be clarified and to formulate appropriate questions.

In particular, since you are not required to turn in any test code for this assignment, your tests and test cases can be freely shared and discussed on Piazza. *It is your responsibility to make sure that you correctly understand the specification and get clarifications when needed.*
For questions, please see the Piazza Q & A pages and click on the folder **`hw4`**. If you don't find your question answered already, 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 **`hw4`**. *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. (In the Piazza editor, use the buttons labeled "code" or "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.)

## What to turn in

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

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw4.zip`** and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw4`**, which in turn contains the eight required classes along with any others you defined in the **`hw4`** package. **Please LOOK at the file you upload and make sure it is the right one and contains everything needed!**

Submit the zip file to Canvas using the Assignment 4 submission link and verify that your submission was successful.

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 everything in your hw4 directory. You can accomplish this by zipping up the **hw4** directory of your project. **Do not zip up the entire project**. 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.