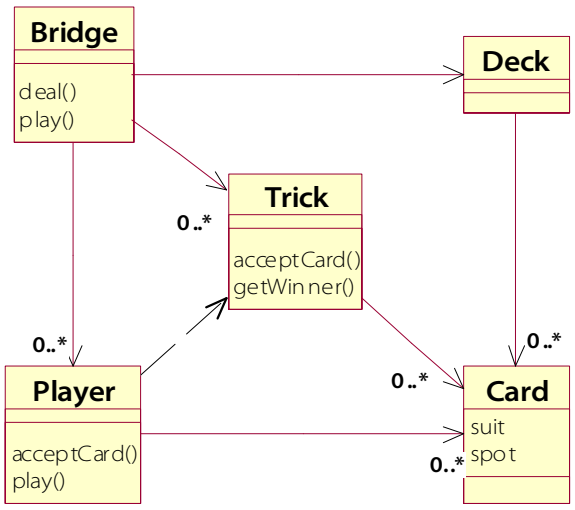
Cards3-4

Part 1: Bridge

You'll pick up with your card-games model (starter code in **Cards3**), and add support for two new games, the first of which is contract bridge. You'll use the **Card**, **CardFormatter**, and **Deck** classes – with some key enhancements to **Card**. You'll build a new class encapsulating a bridge player who holds a hand of 13 cards and makes bidding and playing decisions, and optionally an application class that can simulate game play using that player class.



Start by reviewing the **Trick** class, in package **com.amica.games.bridge**. This encapsulates one trick in the game, where each player puts a card from their hand on the table, in sequence. The cards are gathered up and "won" by the player who played the highest card in the suit that was led: so, this means the highest card out of four in many cases, but especially near the end of the game when a player can't "follow suit" they must play a card from another suit, and this card can't win the trick, even if higher than all the others in spot/rank.

Trick extends **ArrayList<Card>**, and your code doesn't need any of its specialized logic, so you can just treat objects of this class as lists.

Unlike in blackjack, we will now need to sort cards by their spot values. It will also be good practice to define equivalence for a card. So let's start with implementations of **equals**, **hashCode**, and the **Comparable<Card>** interface …

- … or, in fact, let's start with a helper method that will fuel all three of these. On the **Card** class, define a method **getOrdinal** that takes no parameters and returns an **int**. This method can be **private**. The idea is to develop a unique ordinal value for each card in the deck, grouping all cards of a suit in ascending order, followed by the next suit. You can do this by multiplying the card's **suit** ordinal by 13 (or, by the length of the **values()** array on **Spot**, to avoid the magic number), and then adding the spot ordinal. So the 2 of clubs would have ordinal 0 (since both clubs and _2 have ordinal 0); the ace of clubs would be 12, the 2 of diamonds 13, up to the ace of spades at 51.
- Once you have this method, the rest are simple, mostly just delegating to the corresponding method on the **Integer** class. Start with **equals**. First, check that the passed object is a **Card**, using **instanceof**, and return **false** if not. If it is, call **getOrdinal** on **this** and on the other object (you will have to downcast to **Card**), and return true if the values are equal, false if not.
- **hashCode** is simpler: just return the results of **getOrdinal**.
- Make the class implement **Comparable<Card>**, and implement the **compareTo** method to delegate to **Integer.compare** – passing first your own **getOrdinal** result and then the other card's ordinal.

You might want to test these out, especially **equals** and **compareTo**. Just create a few instances of **Card** and call the methods directly to see results. After this, the code in these methods will be called only indirectly, by collection classes that you use in your other classes.

Now you can build **com.amica.games.bridge.Player**, which accepts 13 cards one at a time and organizes them for bidding and playing. (We'll focus on a primitive form of playing a hand, and leave bidding alone.)

Now this class could keep a list of the cards, but for the decision-making involved in the game it's much better to group them by suit, and then sort them by spot. So ...

- Give the class a field **cards** that is a **Map** whose key is a **Suit** and whose value is a **SortedSet<Card>**.
- In a no-argument constructor, initialize this field to a new instance of a **TreeMap**, and **put** a new **TreeSet<Card>** as the value for each of the four suits. When instantiating the **TreeSet**, pass the results of a call to **Collections.reverseOrder** to the constructor. This will assure that all cards in the suit are sorted in descending order of their spot value, so aces first, down to 2's.
- Implement a method **acceptCard** that adds the given card to the list of cards in its suit: that means first getting the list from the map, and then adding to it.
- Override **toString** to list out all of the cards, by suit, using the "spot abbreviation" for each card and then showing the name of the suit just once – like this:

```
J 7 A 2 of clubs -- 5 of diamonds -- K 5 of hearts – 2 J 4 6 5 3 of spades
```

- You might build a quick main-method test at this point, just creating your player, dealing 13 cards to it from a deck, and printing it to see a result like the one above.

- Implement a **play** method that takes a **Trick** object and returns the **Card** that the player will play for that trick. There are a few possible cases to consider here:
    - If the trick is empty, you are leading, and legally you can play any card. This is the most subjective situation, and for now you might just scan your hand for the highest card you have, regardless of suit, and play it. (Note that you'll only need to look at 4 cards, at most, because you can get each **SortedSet** and just call **first** on it, which will return the highest card in that suit.)
    - If the trick is not empty, then you must look to the first card played in the trick, and follow suit if you can. If you have a higher card than the current winning card in the trick, play it; otherwise, play your lowest card in that suit.
    - If you don't have any of the suit that was led, scan your hand for your lowest card overall, in any of your suits, and play that. (This Is very similar to finding the highest card, and you can use the **last** method on **TreeSet** here.)

Whichever card you choose to play, be sure to **remove** it from the appropriate set!

At this point you can rename the **PlayerTest.java.txt** file to just **PlayerTest.java**, and it should compile and be runnable to test your class, with output like this:

```
Test leading from longest and strongest suit ...
Test following suit, winning the trick ...
Test following suit, playing low ...
Test playing low when void ...

ALL TESTS PASSED.
```


Part 2: Poker

The second of our new card games will be draw poker. This of course requires a different sort of player encapsulation: as we'll consider it, the game will involve a player being dealt an initial 5 cards, evaluating the "rank" of the hand – for example one pair, three of a kind, etc. – choosing to draw (replace) up to 3 cards with new ones from the deck, and evaluating again.

- Create a **com.amica.games.poker.Player** class. A lot of what we'll do in this class focuses on the rank of a hand: a classification based on certain combinations of cards the hand might hold, and some of these combinations are more valuable than others. So create an enumerated type inside the class, called **Rank**, with values **NO_PAIR**, **ONE_PAIR**, **TWO_PAIR**, **THREE_OF_A_KIND**, **STRAIGHT**, **FLUSH**, **FULL_HOUSE**, **FOUR_OF_A_KIND**, and **STRAIGHT_FLUSH**.
- We're going to want to sort our poker hand by spot value first, rather than suit-first as the **Card** class does by default. Define a public, static class inside the **Player** class, called **CompareBySpot**, that implements **Comparator<Card>**. Implement the **compare** method to compare spot values first, and then only compare suits if the spot values are the same.
- Make the **Player** class extend **TreeSet<Card>**. This will simplify a lot of your code, because you can just call useful methods on **TreeSet**, without referring to a field such as **cards**. The player is essentially a set of cards, with some specialized logic. (On the other hand, this exposes the set operations publicly, too.)
- Give the class a constructor that takes a **Collection** of cards. Start your constructor implementation by calling **Super** and passing an instance of your **CompareBySpot** class. Then call **addAll** and pass the given collection. Now you have the cards, and they've been sorted in ascending order by spot.
- Many of the hand ranks are based on evaluation of how many cards have the same spot value: pairs, three of a kind, etc. A good way to simplify this analysis is to create a helper method **getCounts**: this will compile a map, with spot values as the keys and counts of those spots as the values. So for example if you hold { 3, 8, 8, Q, Q } the returned map should have an entry for 3 (Spot._3 that is) with a value of 1, and entries for 8 and Q with values of 2.
- Give the class a method **getNumberToDraw**, taking no parameters and returning an **int**. For now just return zero – this is a placeholder to allow a test class to build.
- Give the class a method **draw**, taking a list of cards and returning nothing. Again, this is just a placeholder.

- Create a method **getRank**, which returns a **Rank**. This is the big analysis of the hand's contents, and you can tackle this by checking for one rank after another, as many will rule out others – for example if you have a pair you can't have a straight or a flush.
    - Start by calling **getCounts** and storing the map as a variable **spotsAndCounts**.
    - Define a second variable **counts**, a **Collection<Integer>**, and initialize it by calling **spotsAndCounts.values**.
    - If **counts.contains(4)**, return **FOUR_OF_A_KIND**.
    - If **counts** contains 3, then you have to check if it also contains 2. If it does, return **FULL_HOUSE**, and if not it's **THREE_OF_A_KIND**.
    - If **counts** contains 2, you have a pair, and you might have two pair. An easy way to check this is to call **counts.size**, because the only possible contents at this point in your method are { 2, 2, 1 } and { 2, 1, 1, 1 }. So if the size is 3, you can return **TWO_PAIR**, and if not return **ONE_PAIR**.

  That accounts for all of the sets of 2/3/4 cards of the same spot. Check your work at this milestone by returning **NO_PAIR** at the bottom of the method; rename **PlayerTest.java.txt** to **PlayerTest.java**, and run it. You should pass the tests for the ranks you've implemented, and of course you'll fail the others:

```
Testing ranking of 8H 5H 6D KH 7S
Testing ranking of 2H 5H 5D KH 7S
Testing ranking of 2H 5H 5D KH 2S
Testing ranking of 2H 5H 5D KH 5S
Testing ranking of 8H 5H 6D 4H 7S
    Initial rank should be STRAIGHT but is NO_PAIR.
Testing ranking of 2S 5S QS 10S 7S
    Initial rank should be FLUSH but is NO_PAIR.
Testing ranking of AC 5H 5D AH 5S
Testing ranking of 5C 5H 5D KH 5S
Testing ranking of 8S 5S 6S 4S 7S
    Initial rank should be STRAIGHT_FLUSH but is NO_PAIR.
(remaining tests fail as they involve drawing cards.)
```

- To evaluate straights and flushes you'll need some different tactics. Also, you don't want to return as soon as you see a result, because it's possible to get both a straight and a flush, and you don't want to miss that one! Start by setting a Boolean **straight** to true if the hand's high card is four ordinals higher than the low card. (This is reliable, because you've ruled out any pairs, and you sort in spot order.)
- Set a Boolean **flush** to **true** – that is, we're going to start out assuming that all the cards are of the same suit. Get the first card's suit as a local variable **suit**. Then look through the whole hand, and if a card's suit isn't the same as **suit**, set **flush** to **false** and break out of the loop.
- Now return either **STRAIGHT**, **FLUSH**, or **STRAIGHT_FLUSH**, depending on the truth/falsehood of **straight** and **flush**.

Run **PlayerTest** again, and now all of the ranking tests should pass.

The remaining work on this game is optional:

- Implement logic to decide what cards you would draw to improve your hand, as a helper **getCardsToDraw** that returns a set of cards. This is more work than just ranking the hand: you need to identify the cards that make the hand what it is (or what it could be, as in a four-card straight or flush that could get filled in with a new card) and keep those while discarding the others. Start by getting the hand's rank, and based on that apply the right logic. For one pair, two pair, or three of a kind, get the cards with counts of 1. Any other ranking except no-pair, return an empty list: you can't improve your hand by drawing. The challenge, left to you, is how to handle no-pair hands: can you see a straight or flush in the making? Or should you just drop your three lowest cards?
- Based on this helper, implement the two placeholder methods. **getNumberToDraw** is just the **size** of the list your helper method returns. **Draw** actually removes all of those cards, and adds those it's given.

The full **PlayerTest** should succeed now. You can also rename **Poker.java.txt** to **Poker.java**, and run that class. It simulates a single game of draw poker, with just one **Player**, and print the results. Where the **PlayerTest** is deterministic, always feeding specific cards to the player, the game class will use a random **Deck**, so you'll see all sorts of outcomes if you run it repeatedly.

Extensions

Create a **com.amica.games.bridge.Bridge** class. This will coordinate the movement of cards between a deck, players' hands, and then into 13 tricks, and will evaluate which team won how many tricks. Some of the biggest challenges in this are tracking who is currently playing and who leads out each trick, as play keeps rotating around the table.

- Give your class a list of **Player**s, called **players**.
- Give it a field **leader**, of type **int**. This will always be a number from 0 to 3, and will work as an index into the list of players.
- Write a no-argument constructor that populates **players** with four **Player** objects.
- Create a helper method **next** that takes a player index and returns the next player index – which is just the given index plus one, but wrapping around to 0 instead of returning 4.
- Write a method **deal** that creates a **Deck** and deals out all cards in a rotation to each of the four players in turn, until the deck is empty. Print out the four hands – taking advantage of the **toString** formatting already in place for players and hands, so you should see something like this:

```
Player 0: 6 5 2 of clubs, A 8 5 2 of diamonds, Q 6 of hearts, K Q J 3 of spades
Player 1: K Q J of clubs, 10 7 6 of diamonds, 8 7 3 of hearts, A 9 7 5 of spades
Player 2: A 7 of clubs, K Q J 9 3 of diamonds, A 10 5 2 of hearts, 8 6 of spades
Player 3: 10 9 8 4 3 of clubs, 4 of diamonds, K J 9 4 of hearts, 10 4 2 of spades
```

- Write a method **play** that runs the hand. For each of 13 tricks ...
    - Create a **Trick** object.
    - Call **play** on each of the four players, starting with **leader** and proceeding to the **next()** player.
    - Print the play of the trick, in a line that looks like this:

```
Player 0 leads ... King of Spades, Ace of Spades, 6 of Spades, 2 of Spades
```

    - Check which player won the trick, and make that player the **leader** for the next trick. (Careful! Don't just set **leader** to the winning index as reported by the trick; remember, the one is relative to the other ...)
- In your **main** method, create an instance of your **Bridge** class, and call **deal** and then **play**.

Run that main method now, and output should be something like this:

```
The deal:
Player 0: 6 5 2 of clubs, A 8 5 2 of diamonds, Q 6 of hearts, K Q J 3 of spades
Player 1: K Q J of clubs, 10 7 6 of diamonds, 8 7 3 of hearts, A 9 7 5 of spades
Player 2: A 7 of clubs, K Q J 9 3 of diamonds, A 10 5 2 of hearts, 8 6 of spades
Player 3: 10 9 8 4 3 of clubs, 4 of diamonds, K J 9 4 of hearts, 10 4 2 of spades

Player 0 leads ... King of Spades, Ace of Spades, 6 of Spades, 2 of Spades
Player 1 leads ... King of Clubs, Ace of Clubs, 3 of Clubs, 2 of Clubs
Player 2 leads ... King of Diamonds, 4 of Diamonds, Ace of Diamonds,
    6 of Diamonds
Player 0 leads ... Queen of Spades, 5 of Spades, 8 of Spades, 4 of Spades
Player 0 leads ... 8 of Diamonds, 10 of Diamonds, Queen of Diamonds, 4 of Clubs
Player 2 leads ... Ace of Hearts, 4 of Hearts, 6 of Hearts, 3 of Hearts
Player 2 leads ... Jack of Diamonds, 8 of Clubs, 2 of Diamonds, 7 of Diamonds
Player 2 leads ... 10 of Hearts, King of Hearts, Queen of Hearts, 7 of Hearts
Player 3 leads ... 10 of Clubs, 5 of Clubs, Queen of Clubs, 7 of Clubs
Player 1 leads ... 9 of Spades, 2 of Hearts, 10 of Spades, Jack of Spades
Player 0 leads ... 6 of Clubs, Jack of Clubs, 3 of Diamonds, 9 of Clubs
Player 1 leads ... 8 of Hearts, 5 of Hearts, Jack of Hearts, 3 of Spades
Player 3 leads ... 9 of Hearts, 5 of Diamonds, 7 of Spades, 9 of Diamonds
```

What remains is to score the hand. The players play in teams, so players 0 and 2 are a team and 1 and 3 are a team. Add logic to your **Bridge** class to count up who won how many tricks, and print it out after showing the deal and play:

```
Team 1 won 7 tricks.
Team 2 won 6 tricks.
```


If you have time available, try implementing any or all of these features on **Bridge** and **Player**:

1. Refine your playing logic, beginning with your choice of lead. This is when you have the best chance to steer the play to your advantage, and leading out your highest card isn't actually the best strategy. Look for a "long" suit that you might develop, because after a couple or three tricks, other players will be out of that suit, and even a 3 can win a trick.

Also, think about what you "throw off" when you can't follow suit. Try to preserve long suits for later.

You might also experiment with "ducking," which is refusing to win a trick even though you have a higher card than what's been played. For one thing, playing high as the second player on a trick can be a way to get your king or queen taken by an ace, when it might have won a later trick. For another, if your long suit is led, you can play to take control of a suit after a trick or two has gone by, and then maybe run some easy tricks at that point.

Implement these different playing strategies as subclasses of **Player**. Most interesting would be to have two or three different styles of play, and pit them against one another in a game, by setting up two instances of one vs. two instances of another in the **Bridge** constructor.

2. Refine the rules of the game, and the playing strategy, to allow for a "trump" suit, which would always be spades. (This is less like Bridge and more like a different game, known as ... wait for it ... Spades.) Trump changes things considerably: the trick is won by the highest card in the trump suit, or the highest card in the suit led, in that order. So you can "trump" a trick by playing a spade – but only if you can't follow the suit that was led. This affects the **Trick** class, of course, and the **Player**, because now spades are important, and it's a good tactic to get rid of all cards in an initially "short" suit, so you can win tricks in that suit later by trumping them.

3. You've really got to want this one, but you might try implementing bidding. A player scores their hand by counting high-card points, 4 for an ace, 3 for a king, 2 for a queen, and 1 for a jack – and if trump is supported in your game then short suits are valuable too. Each player bids a number of tricks over 6 that they then need to make, paired with a proposed trump suit; there is also the option of a no-trump bid. Not for the faint of heart! and you'd want to look up the rules online to get a fuller understanding.