Design Example

Requirements

- We will create a program that plays blackjack.
- Dealing:
  - The dealer gives each player two cards from the deck (face-up), and gives himself/herself two (one face-up, one face-down).
  - The total for a hand is counted by summing the face-value of each card: face cards count 10; aces count 1 or 11.
  - The object is to have card that total to as close to 21 as possible without going over.

Requirements

- Drawing cards:
  - Each player gets a turn to draw more cards.
  - Each player may take more cards. If their total goes over 21, they bust and may not take more.
  - The dealer draws last.
  - The dealer must take another card if his/her total is 16 or less, and must not if the total is 17 or more.

Requirements

- Winning:
  - If a player’s total exceeds 21, they lose.
  - If the dealer’s total exceeds 21, all remaining players win.
  - Players whose totals exceed the dealer win.
  - Players whose totals are equal to the dealer “push” (tie).
- Other rules of blackjack will be used to fill in any gaps in the specified requirements.

Requirements

- The program:
  - Users should be able to play a hand of blackjack by interacting with a text-mode user interface.
  - Details of the user interface are left for the designer/ programmer to work out.
- Note: we aren't worrying about betting, splitting, "natural" blackjacks, etc.

Design

Basic design of our
classes for blackjack

Nouns

- Notable nouns from the specification:
  - dealer
  - player
  - card
  - deck
  - hand
  - total (of the hand)
  - turn

list of cards: part of the player/dealer

represent as integer

treat as "take a turn" and create a method in player

Card Class

- Each card has its own rank (A, 2, 3, … 9, J, Q, K) and suit (♠, ♣, ♥, ♦, or spades, clubs, hearts, diamonds).
- Probably no setters: construct a single card & doesn't change after that.
- Constructor will take a suit & rank (do we need classes for those? Probably just characters.)
- A `toString` will be nice, so we can print it.

Deck Class

- Basically a list of card objects
- Constructor should build a fresh deck of 52 cards and “shuffle” it.
- Need to be able to “deal” the next card out of the deck.
  - need a method (nextCard?) that returns a card object, and **removes** it from the deck.

Player

- Has some cards has he/she has been dealt.
  - Store as `ArrayList` in the player object
- Method to “take turn” where they can draw some cards from the deck.
  - probably don’t need a “dealer” object for this: just call the deck’s nextCard method.
- Need a method to calculate the total for the current hand.

Dealer

- Like a player, has some cards in its "hand".
- Rules for drawing cards more strict than for a player: 16 yes, 17 no.
- Also should be able to calculate the total.
  - Uh-oh: code duplication.
  - Dealing with aces (1 or 11) isn't obvious. Having the same code in two places would be a maintenance nightmare.

Duplicated Logic

- Our player and dealer classes both need to calculate the "value" of a hand.
  - Not trivial; we don't want to copy-and-paste code.
- Possible solutions:
  - create a static `handValue` function: not very OO
  - $\boxed{\text{create a hand class that holds the cards \& this code. Both players and dealers would have a ``hand''.}}$
  - inherit from a common ancestor (later)

Hand Class

- Contains a list of card objects.
- Create a method to “draw” a new card into the hand.
- Method currentValue will calculate the total for this hand.
- Players and dealers no longer contain a list of cards: they each contain one hand.

Game Class

- Something we didn't notice in the description: we need a class to actually play the game.
- Players join the game; the deal starts; players get their turns; dealer draws; winners & losers.
  - no "turn" object
- Instantiate a "game" and call a play method.
- The `main()` function will then be very simple.
  - good: designing with objects, so keep the logic there.

Class Summary

- Card: simply store rank & suit.
- Deck: collection of (unused) cards.
- Hand: collection of cards; methods to draw and calculate total
- Player: a hand, and method to take his/her turn drawing (will include some user interface).
- Dealer: a hand, and method to take his/her turn.
- Game: put it all together.

Now…

- We could still have a little more detail in the design.
  - Some key methods of each class, with arguments.
- The implementation seems much less intimidating now.
  - Problem broken into several relatively small parts.
  - Each part is roughly independent (or would be if details of interacting methods were worked out).