Project 2 (100 pts.)

Due date: Oct 26, 2019, 11:59pm

Objectives

- Apply basic principles of inheritance by implementing a subclass and its superclass.
- Demonstrate the ability to implement and use an abstract class with abstract methods.
- Use the static methods in the Collections class.
- Implement various interfaces specified in the Java API, like the Comparable and Comparator interface.
- Implement and apply a new interface.

Specifications

The project is broken into three parts:

- Part 1 is worth 70 points.
- Part 2 is worth 15 points.
- Part 3 is worth 15 points.

You are not required to complete Parts 2 and 3, although you will forfeit the corresponding points if you don't. Part 2 will be more difficult than Part 1 and, likewise, Part 3 will be more difficult than Part 2.

- Part 2 will not be graded if you do not earn 60 of the 70 points on Part 1.
- Part 3 will not be graded if you do not earn 10 of the 15 points on Part 2.

The specification of each Part is given below. Do not move onto the next part until you have successfully completed ALL of the previous part.

Notes and requirements for your submission

- Any project that does not compile will receive a zero.
- Points will be deducted for not following the naming conventions for the project name and archive file.
- You may not discuss the project with anyone within or outside the class. There are two exceptions:
    - On Part 1 you may receive minimal help from the csX lab. You may discuss or receive help from the instructor.
    - On Parts 2 and 3, the only person that you are allowed to discuss any aspect of these two parts is with the instructor.
    - You may not receive help from anyone except the instructor on Parts 2 and 3.
- Please follow all the expectations given in Lab 4, which apply to all labs and projects.
- Make sure any/all classes you modify are formatted (Ctrl+Shift+F) before you submit. Any class that is not formatted will result in a penalty of 3%.
- The penalty for one Checkstyle warning is 1% of the project grade.

Getting started

1. Download the `CS2Project2Starter.zip` project from Moodle and import it in Eclipse.
2. The project compiles and runs so you should be able to see the GUI, but it does not function correctly and not all unit tests pass. You will implement the functionality.
3. Rename the project to *`CS2`*`Project2`*`FirstnameLastname`*.
4. Activate the UWG Checkstyle for the project.The remote configuration is located here: https://www.westga.edu/~dyoder/Checkstyle/UWG_checks.xml
5. This is a GUI application using JavaFX. For your convenience, here are the vm args:
   --module-path "${eclipse_home}/javafx-sdk-11.0.2/lib" --add-modules javafx.controls,javafx.fxml
6. Setup Git for the project and create a Bitbucket repository named `CS2Project2`*`FirstnameLastname`*. Commit and then push the project to Bitbucket.
7. **Remember to commit and push your changes as you complete each step below.**

Blackjack game description

In this project, you will be implementing Blackjack, a card game equally known as Twenty-One. The popularity of this game among gamblers stems from the fact that a player who is able to count cards and play a mathematically sound game, can stack the odds in their favor. The goal of the game is to beat the dealer by getting a score (count of cards value in hand) as close as possible to 21 without **busting** *i.e.*, going over 21. Many flavors of this game are played around the world in casinos, at home, as well as online, although the general rules are consistent. Our application will implement the rules below, and will make use of one standard 52-card deck. We will have a dealer and only one player.

1. The dealer is a “permanent bank” and can never run out of money.
2. The player (human user) joins the casino table with a specified amount of money. They have to bet on each round of Blackjack they play, and can play as many rounds as they have money to bet. For one round of Blackjack, the player can bet any amount from $1 to their total amount of money. When they lose all the money, the game is over. The dealer must match the player’s bet resulting in a pot value equal to twice the bet money.
3. To start the game, the player must place a bet. A round starts by dealing two cards to each player (dealer and human player), starting with the human player, and dealing alternatively one card at a time from the deck. The first card of the dealer is placed face down on the table, and the second one is revealed face up.
4. Everyone can see all the player’s cards and the player continues the round by **standing**, *i.e.,* asking the dealer to reveal all their cards, at which point the outcome of the round is determined, or **hitting**, *i.e.*, asking the dealer for additional cards, one at a time.

Card value

5. Face (or picture) cards (*i.e.*, King, Queen, Jack) are worth 10 points, and the rest of the cards are worth their pip (or numerical) value, from 2-10. The suit and color is irrelevant. The ace is worth either 1 or 11 points, as each player desires based on their current hand.
6. Ace in combination with a non-face card is known as a "soft-hand" since the ace can be counted as 1 or 11. A hand containing one face card and one ace represents Blackjack or "Natural 21".
7. The dealer must ask for more cards if their hand score is 17 or less, but must always stand if their score is 18 or more. For example, it the dealer has an ace, a four, and a three, the total is 18 and the dealer must stand.
8. Although 17 is considered a good hand, the player always has the option to ask for one or more cards, in the hopes of a higher total. If the last drawn card creates a bust by counting the ace as 11, the player simply counts the ace as a 1 and continues to play (stand or hit).
9. The player goes first and if they score a Natural 21, the dealer pays the player twice the pot money if the dealer does not have a Natural 21 also. If dealer has a Natural 21 too, which would result in a tie, only the bet money is returned. If the player busts, they lose the bet money and the dealer does not have to show their cards, even if the dealer busts as well. If the dealer busts but the player doesn't, the dealer pays back the player's bet, or in other words, the player takes the pot.
10. If the dealer stands at 21 or less, the dealer pays the bet if the player has a higher total (but not over 21) and collects the bet if the player has a lower total. If there is a stand-off, *i.e.*, a player has the same total as the dealer but less than 21, the player takes back their bet.

Part 1
Simplifying the model - Card

Take a look at the class `Card`, which is implemented awkwardly. The `Card` is defined to consist of an `int` and a `String`. Since a playing card has very specific values for the rank and suit, we should refactor these fields into enums to restrict them to correct values:

1. In the `model` package, create an enum for `Suit`, with the following constants: ***SPADES***, ***HEARTS***, ***DIAMONDS***, ***CLUBS.***
2. Create an enum for `Rank`, where `Rank` has a private field `value`, of type `int`.
    a. Define each enumerator constant using the following names and values: ***ACE***(1), ***TWO***(2), ***THREE***(3), ***FOUR***(4), ***FIVE***(5), ***SIX***(6), ***SEVEN***(7), ***EIGHT***(8), ***NINE***(9), ***TEN***(10), ***JACK***(12), ***QUEEN***(13), ***KING***(14).
    b. Provide `Rank` with a `getValue` method that returns the constant's value.
3. Refactor `Card` to have as fields a `suit` of type `Suit` and a `rank` of type `Rank`.
4. Make sure `getRank` and `getSuit` still return an `int` and a `String`, respectively. Otherwise, the class `ViewCard` from `viewmodel` does not compile anymore, and you

should not change anything in the view and viewmodel packages. The class `Deck` however, needs to be updated. Wait until the next section to do so.

5. The tests in `edu.westga.cs1302.casino.test.deck` should pass but you can add additional tests to make sure your implementation is correct.

Simplifying the model - Iterable Deck

1. Fix the `Deck::Deck` constructor by using the enums you made for `Suit` and `Rank` to create the deck, by iterating through them.
2. The `Deck` class has two fields, uses an array of `Card`s, and explicitly implements the shuffle operation. We will simplify this class.
3. Delete the line **`import`** `java.util.Random` and remove the field `topIndex`.
4. Make `Deck` implement the `Iterable` interface and update the following members:
    a. the field `cards` to be an `ArrayList` of `Card`s
    b. the body of method `Deck::size`
    c. the method `Deck::draw`, including the specifications. Note that once drawn from the deck, the card is removed from the deck and given to the player
    d. the method `Deck::shuffle` to use the `Collections::`*`shuffle`* method
5. Make sure the tests in `edu.westga.cs1302.casino.test.deck` pass, and add your own tests to demonstrate correct functionality of the class.

Player class

Right now, the `Player` class is used for both the dealer and the human player. However, in our application, the dealer is different from the human player: the dealer is a permanent bank that never runs out of money and can always pay up. For this reason, the field `money` should be an attribute of the human player, not the dealer.

Create a new class, `HumanPlayer`, to inherit `Player`.

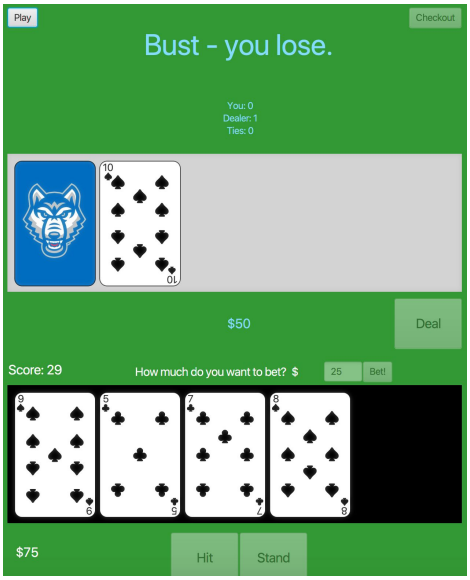
1. Move the field `money` to the `HumanPlayer` class.
2. Move all the methods pertaining to the `money` field in the `HumanPlayer` class.
3. Update the specs of the `Player` constructor, since `money` is not a field anymore.
4. Create a specialized constructor for the `HumanPlayer` class.
    a. make sure your code doesn't have the "magic number" problem
    b. add appropriate specifications
5. At this point, the class `Blackjack` does not compile anymore.
    a. Change the declaration of `humanPlayer` as `HumanPlayer`, not `Player`.
    b. Change the return type of the getter `getHumanPlayer` to `HumanPlayer`

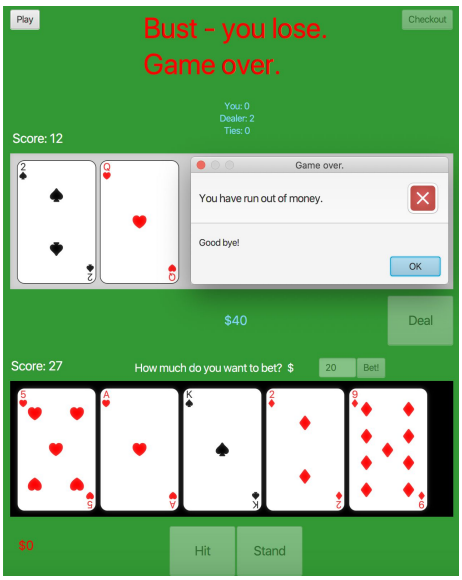
c. In the private method `initializePlayers`, instantiate `humanPlayer` as type `HumanPlayer`.
d. At this point, the class should not have any more compilation errors.

Completing the game

Complete the unfinished methods in `edu.westga.cs1302.casino.game::Blackjack`.

1. Implement `Blackjack::dealHands` to match the specifications. This method is responsible for dealing the first two cards to each player, at the beginning of a new round of Blackjack: first card goes to player, next to dealer, next player, next dealer.
    a. At this point, the tests in packages `test::blackjack` should all pass.
2. Implement `Blackjack::setPot` to match the specifications.
3. Implement `Blackjack::getHumanHandScore` and `Blackjack::getDealerScore` in accordance with the Blackjack rules specified in the **Blackjack game description - Card value** section at the beginning of the project description.
4. Code the body of the `Blackjack::hit` method. Note that the message to be set to the game's message field in case the human player busts is: "`Bust - you lose.`"
    a. Add unit tests to make sure this method works correctly.
    b. If the player loses all the money in this round, the message should also include "`Game over.`" on a new line, *i.e.*, "`Bust - you lose.`" + "`\nGame over.`". In the image on the left, the player still has money, so the message is in blue, inline. In the image on the right, the player has busted and lost all the money, so now they have to quit the game. The GUI will take care of changing the color to red and sending the alert popup window forcing the user to quit the application.





5. Code the body of the `Blackjack::stand` method. This method will essentially close the current round of the game, meaning it will set the resolution message to the game's message field, update the appropriate number of wins, and pay off the player with the corresponding amount. The following messages should be set as follows:
    a. "`Stand-off at Natural 21.`" if both the dealer and player have Natural 21.
        i. the player gets their bet money back
        ii. the number of ties is incremented
    b. If both players have 21 (but neither has Natural 21), then there is a tie and the message is simply "`Stand-off.`"
        i. the player gets their bet money back
        ii. the number of ties is incremented
    c. If only the player has a Natural 21, the message to set to the game's field is "`Natural 21 - you win 2 x pot!`" and:
        i. the player wins double the value of the pot money
        ii. the player's number of wins is incremented
    d. If the dealer scores a Natural 21, regardless of what the human player has, the message is "`Dealer wins with Natural 21.`"
        i. the pot money goes to the dealer
        ii. the dealer's number of wins is incremented
    e. If the player scores 21 (not Natural 21) and the dealer doesn't, the message is "`You win!`" and
        i. the pot goes to the player
        ii. the player's number of wins is incremented
    f. If the dealer scores 21 (not Natural 21) and the player doesn't, the message is "`Dealer wins.`"
        i. and the pot goes to the player
        ii. the dealer's number of wins is incremented
    g. If the dealer busts, the message is "`Dealer busts.`"
        i. the player gets the pot money
        ii. the player's number of wins is incremented
    h. If nobody has 21, but the player wins by scoring closer to 21 than the dealer, the message is "`You win.`"
        i. and the pot goes to the player
        ii. the dealer's number of wins is incremented
    i. If both players tie with a score less than 21, the message is "`Tie.`"
        i. the number of ties is incremented
        ii. the player gets the bet back
    j. Uncomment `edu.westga.cs1302.casino.test.Blackjack::TestStand` which tests the stand method and make sure all unit tests all pass.

Note: if some of the method names don't match, you can refactor them. The actual testing -- the spelled-out messages and the number of wins -- should NOT be changed. This unit test will also verify that your score calculation is correct, although you should write your own tests to validate its correct functionality.

6. Make sure your code adheres to the DRY and SRP principles, as there is some repetition involved in this method.
7. Work on the `Blackjack::getStats` method and modify it to return the statistics of the game formatted as a string, as follows:

```
You: <number of human wins>
Dealer: <number of dealer wins>
Ties: <number of ties>
```

Note: Do not include in the returned string the angular brackets. They only suggest that the content inside them varies, and it should be replaced with the corresponding number. This is how it should look: ---->

8. At this point, you should be able to play the game. The GUI is already implemented and you should not change anything in the `view` and `viewmodel` packages.

End of Part 1 Requirements

Finalizing Part 1

1. Tag the submission in Bitbucket as `Part1Completed`.
2. As a backup, export the completed Part 1 project using the following filename: CS2Project2Part1*FirstnameLastname*.zip
3. If you successfully completed **all** of Part 1 and you so desire, you may proceed onto Part 2 of the project. If this is all you are going to do for the project and this is your submission, then tag this final commit as `Project2Submission` and upload the zip file to Moodle.
4. Part 2 begins on the next page.

Part 2

Interface Playable

Create an interface called `Playable` in the `game` package, that contains the method signatures and specifications below. The class `Playable` is also in Moodle, ready to download.

```
/**
 * Returns the human player of this game.
 *
 * @return the human player
 */
HumanPlayer getHumanPlayer();

/**
 * Returns the dealer of this game.
 *
 * @return the human player
 */
Player getDealer();

/**
 * The human player asks for a card.
 *
 * @precondition none
 * @postcondition the player asks and is dealt a new card; stats are updated if
 *                they bust, and only score is updated otherwise
 * @return true if player can continue game, false if bust
 */
boolean hit();

/**
 * The human played the round and now it's the dealer's turn to play their
 * round and finish the game.
 *
 * @precondition player is done with their turn and score is updated
 * @postcondition the game is over and stats are updated
 */
void stand();

/**
 * Returns a message after the game's state has changed in a meaningful way,
 * e.g., when a player draws a card or when the game ends.
 *
 * @return the message from the game
 */
String getMessage();

/**
 * Returns the pot of this game.
 *
 * @return the pot
 */
int getPot();

/**
 * Sets the pot of the game.
 *
 * @param pot the pot to be set for this game
 * @precondition none
 * @postcondition getPot() == pot
 */
void setPot(int pot);

/**
 * Returns a String representation of the game's statistics to be displayed
 * when a new round has ended.
 *
```

```
     * @return the stats
     */
    String getStats();

    /**
     * Returns a String representation of player's current score. It is usually
     * called after the player has drawn a new card, which will modify their score.
     *
     * @return the human player's score
     */
    int getHumanHandScore();

    /**
     * Returns a String representation of player's current score. It is usually
     * called after the player has drawn a new card, which will modify their score.
     *
     * @return the dealer's score
     */
    int getDealerHandScore();

    /**
     * Returns the money of the human player.
     *
     * @precondition humanPlayer != null
     * @postcondition none
     * @param humanPlayer the specified human player
     * @return the money of the human player
     */
    int getMoneyOf(HumanPlayer humanPlayer);

    /**
     * This method deals the initial hands to the players.
     */
    void dealHands();

    /**
     * This method starts a new round of this game.
     */
    void startNewRound();
```

Create an abstract class `Game` in the `game` package that implements the `Playable` interface. The `Blackjack` class must inherit `Game` and implement `Playable`. The `Game` class must have the "general" methods, such as the ones that `Blackjack` can **re**use. But `Blackjack` class must also override/specialize the methods from `Game` as follows:

1. `Blackjack` partially overrides the `Game` constructor by calling its own `StartNewRound`.
2. `Blackjack` partially overrides the `startNewRound` method by assigning "`BLACKJACK`" to the `message` field, and `setPot`, by assigning the pot to be twice the human's bet.
3. The methods `getHumanHandScore` and `getDealerHandScore` should be overridden to be specific to the `Blackjack` rules.

The game should still run, and it is up to you to decide the best organization for the `Game` and `Blackjack` classes. Even if you don't implement Part 3, you should still read the description to better understand the role of `Game` and `Playable`, in order to make informed design decisions.

Implementing `Playable` is like "signing a contract" where `Blackjack` agrees to provide to the `viewmodel` the functionality it needs, which is specified by the `Playable` interface. In order to "activate" the GUI, `Viewmodel` needs `Blackjack` to be "`playable`".

Sorting cards for Blackjack

We want the dealer's hand of cards to be displayed in increasing order of their rank (as given by the enum Rank, and where the ace is worth 1 point).

1. Add sorting capabilities to your code to compare Cards.
2. Modify the `Blackjack::getDealerHand` to return the cards in order of their rank.
   Note: All your changes should take place in the `model` and `game` packages. `Blackjack` sends the cards to the `viewmodel` and the GUI will simply display the cards in the order they are returned by this method. There are no necessary modifications elsewhere.

Advanced sorting

Add two-level sorting capabilities to your code.

The human player's hand should be displayed in the traditional suit ranking of playing cards, with the better cards first, as follows:

1. The first-level sort is by suit: spades (highest), hearts, diamonds, clubs (lowest) and the second-level sort is by rank in descending order. The ace is worth one.
2. Modify the functionality to use this two-level sort when displaying the player's cards.
3. Again, do not change anything in the `viewmodel` and `view` packages.

Take some time to figure out what the current code does and what type of changes will need to be made to accomplish this requirement and *where*.

End of Part 2 Requirements

Finalizing Part 2

1. Tag the submission in Bitbucket as `Part2Completed`.
2. As a backup, export the completed Part 2 project using the following filename: CS2Project2Part2*FirstnameLastname*.zip
3. If you successfully completed **all** of Part 2 and you so desire, you may proceed onto Part 3 of the project. If this is all you are going to do for the project and this is your final submission, then tag this final commit as `Project2Submission` and upload the zip file to Moodle.
4. Part 3 begins on the next page.

Part 3

Adding a new game to the casino

The `view`, `viewmodel` and GUI can be easily reused for other two-player games similar to Blackjack. For example in Baccarat, the goal is to score as close as possible to 9. We will implement the following variation of Baccarat:

1. The first three rules from the **Blackjack game description** section (at the beginning of the project description) apply to Baccarat as well.
2. Cards ace through NINE have pip value, while 10, J, Q, and K have a value of zero.
3. If a hand score is more than 10, the first digit is dropped and the second digit becomes the value of the hand. For example, NINE and FOUR total 13 but the hand score is 3.
4. A "Natural" win is a hand of exactly *two* cards totaling 8 or 9.
5. The player goes first and if their hand score is 6 through 9, the player will not get any more cards and must wait for the dealer's turn to play out and wait for the game resolution.
    a. if the player has 1-5, they can ask for one more card (*i.e.*, hit)
    b. the player cannot have more than 3 cards in hand; once they draw the third card, they must stand. Our GUI will let you click "hit" but will not deal any more cards if you already have 3, and will display the message that you `"Must stand."`
6. The dealer's turn comes after the player's turn, so the dealer can see all the player's cards. The dealer decides to hit or stand based on the following rules, which are dependent on the player's third card:
    a. if the player's third card is 9, 10, face card or ace, the dealer hits if they have 0-3, and stand with 4-7
    b. if the player's third card is 8, the dealer draws another card when they have a 0-2, and stands with a 3-7
    c. if the player's third card is a 6 or a 7, the dealer hits if they have 0-6 and stands with a 7
    d. if the player's third card is 4 or 5, the dealer hits if they have a 0-5 and stands with a 6 or a 7
    e. if the player's third card is 2 or 3, the dealer draws another card if they have 0-4, and stands at 5-7.
7. After both players play their turn, the game resolution is calculated, and the winner is the hand closest to 9.The winner collects the pot. A stand-off returns the bets to players.
8. To play Baccarat, you have to make one modification to the `viewmodel::ViewModel` class. In the `ViewModel` constructor, instantiate the `game` as a `Baccarat`-type game.

End of Part 3 Requirements

Finalizing Part 3

1. Tag the submission in Bitbucket as `Part3Completed`.
2. As a backup, export the completed Part 3 project using the following filename: CS2Project2Part3*FirstnameLastname*.zip

Submission

1. Please go over the section **Notes about your submission** one more time and make sure your submission follows the course requirements.
2. If you haven't done so, tag your submission as `Project2Submission`.
3. Also, submit your completed project to Moodle.