capstone courseware

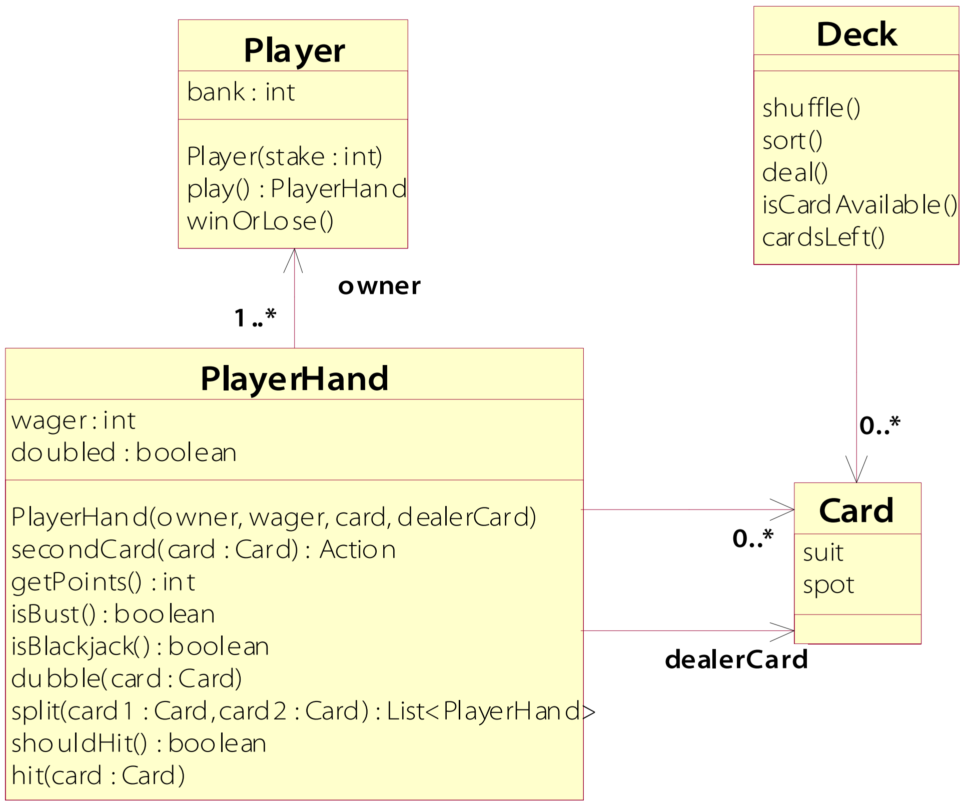
Capstone Courseware, LLC
1 Washburn Place
Brookline, MA 02446

877-227-2477
capstonecourseware.com

Cards 1-2

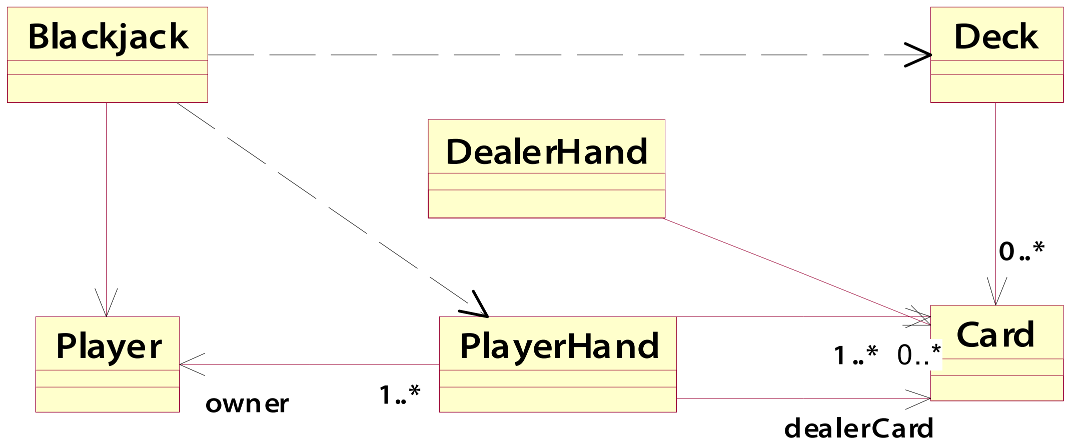
In this exercise you'll build a small system of classes to model card games played with a standard 52-card deck. You'll build this up from small pieces, starting with the cards themselves, then a deck of cards and hands of cards and players for various games.

By the end of this exercise you'll have build a system of **Deck** and **Card** classes supporting the blackjack **Player** and **PlayerHand**:



You'll also create a **CardFormatter** to support console and log output – not shown in these UML diagrams.

A provided **Blackjack** class models a round of blackjack, and you can bring this code into your project once you have the above types, plus a **DealerHand** that has some of the capabilities of **PlayerHand** but is simpler, since the dealer's decision-making is more limited than the player's.



Start by creating a **Card** class, and define two enumerations within it:

- **Suit**, with values **CLUBS**, **DIAMONDS**, **HEARTS**, and **SPADES**
- **Spot**, with values running from **_2** to **_10**, and then **JACK**, **QUEEN**, **KING**, **ACE**

Build out the state and behavior of a **Card**:

- Give it fields **suit** and **spot**, each of the appropriate enumerated type.
- Give it a constructor that initializes both fields.
- Give it getters for both fields.

Implement whatever quick test code you like, in order to make sure these basic features are working.

Now create a **CardFormatter** class. All methods on this class will be **static** – making it what is known in UML as a class utility – and all will return strings. These methods will carry out some simple formatting of cards and their component values – suit and spot – to make it easier to produce readable program output and log entries.

- Build a method **abbreviationOf** that takes a **Spot** value. Convert to a string. Strip off any leading underscore. Then take the first character of the string. This covers you for all the numbers except 10 and all of the words. If your character is "1", just add the "0" back to it. Return the abbreviation. (So you'll return values such as "3", "10", and "K".)
- Build a method **abbreviationOf** that takes a **Card**. Implement it to return the abbreviation for the card's spot value, plus the first letter of the card's suit. ("4C", etc.)
- Build a method **capitalize** that takes a string. Return a version of the string with an initial capital letter and the rest lower case.
- Build a **nameOf** method for **Suit**s, returning the results of calling **capitalize** on the suit's string representation. ("Clubs", "Diamonds", "Hearts", "Spades".)
- Build a **nameOf** for spot values. Call **capitalize** here as well – and note that it's safe to pass digits through this method, as they'll be left alone – and be sure to strip off any leading underscore, either before or after that call. Return the result. ("3", "10", "King", etc.)
- Finally, build a **nameOf** for cards themselves, using the full names of spot and suit and putting them together as in "3 of Spades" or "Jack of Clubs".

Test this out briefly before moving on.

Now build a **Deck** of cards.

- This class will have a **List<Card>** as its only field.
- Give it a no-argument constructor that populates the list with all 52 cards, by looping over all suits and then all spots.
- Give it a methods **shuffle** that calls the corresponding method on the **Collections** utility, passing your list of cards.
- In the constructor, after populating the list, call **shuffle**.
- Give it a method **deal** that returns the first card in the list (we'll call that the top of the deck), and removes that card from the list.
- Define an overload of **deal** that takes a number of cards to deal out, and returns a list of cards rather than a single card.
- Implement a method **isCardAvailable**, which returns **true** if there's at least one card left in the list, and another **cardsLeft** which returns the size of the list.

Again, this is a good time for a little **main** method that exercises your class: create a new deck, shuffle it, deal a few cards, print out results as you go. Prove that you can deal out the whole deck, not by counting to 52 but by checking **isCardAvailable** and stopping when the deck is exhausted.

Okay, now we'll start to focus on one specific card game, which is blackjack. The full rules of this casino game are fairly complicated. You'll build a naïve player implementation and get it working with a prepared game model (the currently skimpy **Blackjack** class) and then start to refine your logic to take advantage of various options as the player.

Start by creating the **Player** class. This class is actually pretty simple:

- It holds a **bank** field, of type **int**, which is the value of the player's betting chips.
- Provide a constructor that takes a value for **bank**, and another that defaults to $10,000.
- Define a getter for **bank**.
- Define a method **winOrLose** that takes an **int** representing a win (positive number), loss (negative) or "push" (zero). Add the number to **bank**.

The caller is going to expect the **Player** to be able to create an object that represents the hand of cards – and the class of this object will also make decisions about how to play the hand. Once the player creates this object, the player sits back and waits to hear how it went. The "hand" object will know how to report back to the player, by calling **winOrLose**.

Create this **PlayerHand** class now. This is the biggest and most involved class in the exercise. Start with the state model:

- An **owner** field of type **Player**. This will be used to report back when the hand is over.
- A field **wager**, of type **int**.
- A field **cards**, whose type is a **List** of **Card** references, initialized to a new **ArrayList**.
- A field **dealerCard**, of type **Card**.
- An **ID**, of type **int**.
- A Boolean field **doubled**. We won't use this immediately but will come back to it.

Now add constructors and methods:

- The constructor will take four parameters, and use them to initialize fields: the **owner**, the **wager**, and two cards – but perhaps not the two you might think. One of these is the first of the player's cards, and the other is the dealer's "up" card – the one the player gets to see while playing their own hand. We'll get that second card via a separate method. For now, capture the values of **owner**, **wager**, and **dealerCard**, and call **cards.add** and pass the first card into the hand.
- Build getter methods for **owner**, **wager**, **ID**, and **doubled**.
- Also build a setter for **ID**. (This property has nothing to do with playing the game. It's used by the **Blackjack** class to distinguish one hand from the next, just so console output is less confusing.)
- Create a method **secondCard** that takes a **Card** and returns a **Blackjack.Action**. This enumerated type represents some more advanced playing options. For now, just **add** that second card to your hand, and return **Action.NONE**. This isn't as negative as it sounds: it just means you want to play the hand normally.
- Now, to play well, you're going to have to be able to evaluate the hand, by counting up points. This is <u>almost</u> easy: add up the values of the number cards (exactly as shown, as in a seven is worth 7 points), the face cards (all worth 10) and aces (aha, that's the tricky bit). Aces can be either 1 or 11, at the player's option. So your optimal count is going to be found by adding up the points, with one point for each ace, and separately counting your aces. Then, as long as you can afford to (that is, you won't go over 21 points), add 10 for each ace.
- Build a method **isBust** that returns **true** if your points are (unavoidably) over 21.

- Build a method **isBlackjack** that returns **true** if you have 21 points with your first two cards. This can only be an ace and a 10-valued card, in either order – so just testing the **size** of your list of cards and the point total is a reliable implementation.
- Build a method **shouldHit** that returns a Boolean: true if you want another card, false if not. For now, just return **true** if your total is under 16. This is not a strong playing strategy, but it will get us going.
- Implement a method **hit** that takes a **Card** and adds it to the hand.
- Build a method **winOrLose** that takes an **int** and just turns around and passes it to the method of the same name on the hand's **owner**.
- Define a method **dubble** that takes a **Card** and returns nothing. Leave this un-implemented for now.
- Define a method **split** that takes two **Card**s and returns a **List** of **PlayerHand**s. This too can be left alone for the moment: return **null**.
- Give the class a **toString** method to your liking, so you can print out representations later as you're developing and debugging.

Now that your **PlayerHand** class is complete, you can circle back and implement the **play** method on **Player**. This method takes the first card and the dealer card as parameters, and just returns a new **PlayerHand**, passing its own **owner** and **a wager of $10** along with the two cards to the constructor.

With this class completed, you can start to bring in some prepared code that wouldn't have compiled at the start of the exercise but now should work with your new classes. First, rename the **PlayerHandTest.java.txt** by removing the extra ".txt" from the filename. The resulting **.java** file is now part of the project. If you see compile errors at this point, fix them: this is going to be a matter of the prepared code expecting specific method names and parameter types as per the above instructions.

If you run this class as a Java application, it will create a hand object, deal it specific cards, and check that it's working correctly: counting points, reporting bust or not, blackjack or not. Many of the later tests will fail, because they expect more sophisticated behaviors that you'll implement later on. But what you have built should test out correctly – at least these specific tests:

```
Testing for 9 points ...
Testing for 13 points ...
Testing for 14 points ...
Testing for 12 points ...
Testing for 21 points ...
Testing for 12 points ...
Testing for not bust ...
Testing for bust ...
Testing for blackjack ...
Testing for blackjack ...
Testing hit up to 9 points, dealer showing a 6 ...
Testing hit up to 11 points, dealer showing a 6 ...
...
SOME TESTS FAILED
```

Create a **DealerHand** class, initially as a copy of your **PlayerHand**. The dealer doesn't have to work quite so hard! So remove a few things. Get rid of all fields except **cards**, and the corresponding getters and setters. Give the class a different constructor that takes two cards and adds them both to the list of cards.

Keep methods **getPoints**, **isBust**, **isBlackjack**, **shouldHit**, **hit**, and **toString**. Ditch the rest!

Adjust **shouldHit** so that it returns true if the point count is less than 17.

Now you should be able to run a complete round of blackjack. Open **Blackjack.java.txt**, copy the complete contents of the file, and paste them over the little bit of code in **Blackjack.java**. Again, fix any compile errors, which will be little integration issues such as method names and specific parameter or return types.

When you run this class as a Java application, it runs a round of the game, with your code as the only player at the table. Each run uses a new, shuffled deck, so output isn't totally predictable. Here are examples of what ought to happen:

```
Dealer shows a(n) 7
Dealer's hold card is a(n) 9
Hand 1 gets a(n) 4
Hand 1 gets a(n) King
Hand 1 gets a(n) 7
Hand 1 is bust.

Dealer shows a(n) 5
Dealer's hold card is a(n) 9
Hand 1 gets a(n) 2
Hand 1 gets a(n) Ace
Hand 1 gets a(n) 7
Dealer has 14 points.
Dealer hit: Ace
Dealer hit: 8
Dealer is bust.
Hand 1 has 20 points.
Hand 1 wins.
```

Run this a bunch of times and see how you do! Don't worry if you lose a lot: it's blackjack, and you are, ultimately, supposed to lose.

But you can lose less! Read on …

Rename **Simulation.java.txt** to **Simulation.java**, and run it as a Java application. This program runs 1000 simulations, each starting the player with $10,000 and playing 1000 hands. It reports on game outcomes and the player's average win or loss over 1000 games. With your kiddie implementation, of course you don't do too well:

```
After 1,000 simulations, player averaged $-489 gain or loss over 1,000 hands.
There were 0.00 splits per 1000 rounds.
Outcome frequency:
    DOUBLE_BUST       0.00%
    DOUBLE_LOSS       0.00%
    BUST             19.30%
    LOSS             30.04%
    SURRENDER         0.00%
    PUSH              8.54%
    DOUBLE_PUSH       0.00%
    WIN              37.29%
    BLACKJACK_TIE     0.18%
    BLACKJACK         4.65%
    DOUBLE_WIN        0.00%
```

You can refine your play, in a number of ways, and each will bring some benefit that's measurable in the simulation. There are test cases for these in the **PlayerHandTest**, too, so if you implement all of them you should see "ALL TESTS PASSED" in that application's output.

- Pay attention to the dealer's up card when deciding if you should hit. The so-called "basic strategy" calls for a hit on a total under 12, standing on a total over 16 – and then (more or less) a hit on 12-16 if the dealer is showing a seven or better. Tweak **shouldHit** and see how you do. Typical output with this enhancement:

```
After 1,000 simulations, player averaged $-272 gain or loss over 1,000 hands.
There were 0.00 splits per 1000 rounds.
Outcome frequency:
    DOUBLE_BUST       0.00%
    DOUBLE_LOSS       0.00%
    BUST             16.12%
    LOSS             32.29%
    SURRENDER         0.00%
    PUSH              8.22%
    DOUBLE_PUSH       0.00%
    WIN              38.56%
    BLACKJACK_TIE     0.18%
    BLACKJACK         4.63%
    DOUBLE_WIN        0.00%
```

- You're allowed to double your bet after the initial deal – and in the process say that you want exactly one more card. So, you can't double on a 20 and then choose to stand – too easy! But try doubling on a total of 10 or 11. From there your chances of getting a 10 or 11 are good, so it's worth playing the odds. How to do this? Check the total in **secondCard**, and if it's 10 or 11, set your **doubled** field, and return **Action.DOUBLE**. The **Blackjack** class will then call your **dubble** method, and you'll add the card to your list. When the hand is evaluated, your win or loss will be doubled. The numbers shift in your favor again:

```
After 1,000 simulations, player averaged $-140 gain or loss over 1,000 hands.
There were 0.00 splits per 1000 rounds.
Outcome frequency:
    DOUBLE_BUST       0.00%
    DOUBLE_LOSS       3.12%
    BUST             14.99%
    LOSS             30.45%
    SURRENDER         0.00%
    PUSH              7.36%
    DOUBLE_PUSH       0.71%
    WIN              33.99%
    BLACKJACK_TIE     0.17%
    BLACKJACK         4.62%
    DOUBLE_WIN        4.60%
```

- You can "surrender" your hand immediately on getting the second card. Try implementing this if you get a 16, and the dealer shows a nine or better, by returning **Action.SURRENDER**. The advantage here is slight, but you should see a little better results.

- You can split an initial pair of cards, resulting in two hands that you play separately. This isn't always a good idea, but it can be a winner. In **secondCard**, determine if the two cards have the same spot value, and if they are any of { 2, 3, 6, 7, 8, 9, A } then return **Action.SPLIT**. Then you have to implement the **split** method, and this is a little trickier than the others. You will create two new **PlayerHand**s, each with your **owner**, **wager**, and **dealerCard**, and with one of your own cards. Put these in a list and return the list. These will replace **this** as the active hands on the table. (This is the reason for the odd lifecycle choice where we get the first card in the constructor and the second one in a method call: it supports this method, and code in **Blackjack** that recursively allows splits.)

With all of these tricks now in your repertoire … well, you still lose money. Again, that's blackjack, and if it were possible to automate winning, casinos wouldn't offer it. But you've tuned things up considerably:

```
After 1,000 simulations, player averaged $-95 gain or loss over 1,000 hands.
There were 29.88 splits per 1000 rounds.
Outcome frequency:
    DOUBLE_BUST       0.00%
    DOUBLE_LOSS       3.34%
    BUST             12.91%
    LOSS             29.95%
    SURRENDER         3.28%
    PUSH              7.16%
    DOUBLE_PUSH       0.73%
    WIN              33.00%
    BLACKJACK_TIE     0.17%
    BLACKJACK         4.82%
    DOUBLE_WIN        4.88%
```