CS3101-2 Scala, Fall 2014: Problem Set 4

Daniel Bauer

Total points: 20
Due date: Nov 18, 11:59pm EST

Submission instructions:
Place the files for all problems in a directory named `[your_uni]_week[`X`]`, where X is the number of the problem set. For instance if your uni is xy1234 and you are submitting the problem set for the first week, the directory should be called `xy1234_week1`. Either zip or tar and gzip the directory (using `tar -c xy1234_week1 | gzip > xy1234_week1.tgz`) and upload it to your directory in the drop box for this class on Courseworks.

Part 1 - Call-by-Value vs. Call-by-Name

(a - 3 pts) Consider the following scala programs. For each program state what the program outputs assuming all parameters are call-by-value. Write a short explanation for the output.

Program 1:

```
def bob(x: Int): Int = { println("Bob"); x + 1 }

def joe(x: Int,y: Int): Int = {
    println("Joe");
    val a = x; val b = y; println(a+b);
    a+b
}
def ron(x: Int, y: Int, z: Int) {
    println("Ron")
    println(x + y)
    println(x + y + z)
}
ron(bob(joe(bob(1),2)),3,4)
```

Program 2:

```
def buggy(x: Int): Int = {
     println(x);
     buggy(x - 1)
}
def foo(x: Int, y: Int): Int = {
    println(x)
    x + 2
}
println(foo(1,buggy(10)))
```

(b - 4 pts) Implement a version of each program in which all parameters are call-by-name. What does each program print? Write a short explanation for the output of each program.

Part 2 - Packages, Case Classes, and Basic Pattern Matching

In this problem you will create and use a simple library for playing cards.

(a - 0pt) Download and install sbt[1]. Unpack the archive `cards.tgz` which contains an sbt project. You should be able to `cd` into the `cards` directory from the archive and run `$ sbt compile` (although there will be errors).

(b - 6pt) A standard deck of playing cards consists of thirteen cards for each of the four suits: Clubs ♣, Diamonds ♢, Hearts ♡, and Spades ♠ (52 cards total). The thirteen cards for each suit have the values *Ace, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King*.

The file `src/main/scala/cards/playingCards.scala` defines a package `cards`, which contains a case class `PlayingCard(suit: CardSuit, value: CardValue)`, representing a card.

The package `cards.suits` (defined as a nested package) contains four case **object**s (one for each card suit) that extend the abstract class `CardSuit`. A case object is a singleton object with all the functionality of a case class.

Create a package `cards.values` that contains a similar type hierarchy to represent card values, using a common base class `CardValue`. Make sure you can represent both face values (*Ace, Jack, Queen, King*) and number values (as a class `NumberValue(value: Int)`).

It should now be possible to define cards like this:

```
val card1 = PlayingCard(Hearts, Queen)
val card2 = PlayingCard(Diamonds, NumberValue(9))
val card3 = PlayingCard(Spades, Ace)
```

(c - 7pt) The file `Blackjack.scala` in the `blackjack` package defines the object `Blackjack`.

Add a method `cardValue(card: PlayingCard, current_sum: Int): Int` to this object. The function should computes a single integer value for a card. The suit of a card is irrelevant to determine its value. The value of a card with a number value is its number value. The value for *Jack*, *Queen*, or *King* is 10. The value of an *Ace* is either 1 or 11, depending on the `current_sum` parameter. If `(current_sum + 11)` $<=$ 21 the value of an *Ace* is 11, otherwise it is 1.

Use pattern matching (using a single `match` expression) to compute the card value. Do not use `if...else` statements.

Make sure `Blackjack.scala` includes the necessary imports.

The `cardValue` method should work like this:

```
val card1 = PlayingCard(Hearts, Queen)
val card2 = PlayingCard(Diamonds, NumberValue(9))
val card3 = PlayingCard(Spades, Ace)

scala> cardValue(card1, 0)
res0: Int = 10
scala> cardValue(card2, 10)
res1: Int = 9
scala> cardValue(card3, 19)
```

[1] `http://www.scala-sbt.org/download.html` and `http://www.scala-sbt.org/0.13/tutorial/Setup.html`

```
res2: Int = 1
scala> cardValue(card3, 2)
res3: Int = 11
```

Part 3 - Building a Blackjack game (extra credit, 8pts max.)

Extend the Blackjack object to allow a single player to play Blackjack according to the following rules. The objective of the game is to collect cards whose summed points (values according to the `cardValue` method defined in Part 2) is close to 21 but does not exceed 21.

A round consists of one or two turns (one for the player, one for the dealer).

- **Player's turn**:
  1. The player is dealt a card.
  2. If the sum of all cards the player was dealt in this round is 21 the player wins and the next round begins.
  3. If the sum of cards is greater than 21 the player loses and the next round begins.
  4. Otherwise the player The player can decide to *hit* (ask for another card, back to step 1) or *stand*, in which case it is the Dealer's turn.
- **Dealer's turn**: The dealer turn is the same as for the player except that the dealer cannot make a decision. The dealer will always *hit* until his sum of cards is 17[2] or greater. If the resulting sum is greater than the player's sum but does not exceed 21, the dealer wins. If the sums are the same, the round is tied. Otherwise the dealer loses.

Assume the player starts with 100 credits and in each round the player bets 1 credit (i.e. if he loses the round his credits decrease by 1, if he wins his credits increse by 1). There are infinitely may rounds.

You can use the class `CardDeck` in the package `cards.deck`. When a new `CardDeck` is initialized it will contain a full set of 52 cards in random order. `CardDeck` has a method `nextCard: PlayingCard` that will take a card from the deck and return it. If the deck is empty, the next call to `nextCard` will re-populate the deck with 52 shuffled cards.

For instance, the output of the game could look like this:

```
PLAYER TURN. Current Credits: 100
PlayingCard(Diamonds,Ace)
Current sum: 11
Hit or Stand? [H/S]
h
PlayingCard(Hearts,NumberValue(5))
Current sum: 16
Hit or Stand? [H/S]
h
PlayingCard(Hearts,NumberValue(9))
```

[2] In real casino blackjack this rule is slightly different. The dealer stands on a *soft* 17, where an Ace is assumed to have value 1, unless interpreting it as 11 would bring his sum of points to 21. This rule gives an advantage to the dealer.

```
Current sum: 25
Dealer wins!

PLAYER TURN. Current Credits: 99
PlayingCard(Diamonds,NumberValue(5))
Current sum: 5
Hit or Stand? [H/S]
h
PlayingCard(Clubs,NumberValue(9))
Current sum: 14
Hit or Stand? [H/S]
h
PlayingCard(Spades,NumberValue(4))
Current sum: 18
Hit or Stand? [H/S]
s
DEALER TURN
PlayingCard(Spades,NumberValue(7))
Dealer sum: 7
PlayingCard(Clubs,Jack)
Dealer sum: 17
```