COMP 9 Lab 2: Blackjack!

Out: Thursday, February 3rd, 1:15 PM

Due: Thursday, February 10th, 1:15 PM

1 Overview

In this assignment, you will write a blackjack game. You will start learning about loops and conditionals, and you will practice using methods, input and output, and conversions.

2 Some new topics in code

2.1 Conditionals

We've discussed *boolean* methods, which return `True` or `False`. In addition, various *expressions* can return boolean values. Remember that while `=` means *assignment*, `==` means *equals*, and (just as in mathematics) is a true-or-false statement.

```
4 < 3 => False
2 == 2 => True
1 + 3 == 4 => True

name = 'Fred' => nil
name == 'Bob' => False
name == 'Fred' => True
```

Given *expressions* that can be true or false, we may want to do things based upon the truth or falsehood of these expressions. The `if` statement is the simplest.

```
if weather == 'snow'
    puts "Would you like to buy a shovel?"
end
```

You can have more than one line of code inside a conditional:

```
if month > 11 and month < 4
    puts "Hey, it's winter in New England."
    puts "Why go to the gym when you can shovel snow instead?"
end
```

There is also a *postconditional* version that can be quite convenient.

Note that this first example is identical to the first `if` example above

```
puts "Would you like to buy a shovel?" if weather == snow

puts "Would you prefer a ticket to San Diego?" if weather == snow

puts "I hear Bermuda is lovely this time of year!" if temperature < 32
```

Sometimes, you want to do one thing if a condition is *true*, and a different thing if it is *false*. The `if-else` or `if-elsif` construct is your friend.

```
if month > 11 and month < 4
    puts "Let's go sledding!"
else
    puts "I'm glad I don't have to shovel!"
end


if month > 11 and month < 4
    puts "Let's go sledding!"
elsif month >= 9 and month <= 11
    puts "I hear the leaves are pretty right now."
elsif month > 6 and month < 9
    puts "Let's go to the beach!"
else
    puts "Meh. It's raining."
end
```

So, conditionals determine *whether or not* to do something.

2.2 Loops

Loops give us a way to *repeatedly* do something.

```
5.times do
    puts "How ya like me now?"
end

counter = 99
counter.times do
    puts "#{counter} bottles of beer on the wall!"
    puts "#{counter} bottles of beer on the wall!"
    puts "take one down, pass it around..."
    counter = counter - 1
    puts "#{counter} bottles of beer on the wall!"
end
```

Remember that the `#{}` construct inside a double-quoted string does *variable interpolation*, which always calls the `to_s` method on the variable inside the curly braces. In this case, it means that a `String` conversion of `counter` is put in place in those `puts` statements.

2.3 Control Flow

Combining conditionals and loops is also useful:

```
counter = 99
while counter > 0 do
    puts "#{counter} bottles of beer on the wall!"
    puts "#{counter} bottles of beer on the wall!"
    puts "take one down, pass it around..."
    counter = counter - 1
    puts "#{counter} bottles of beer on the wall!"
end
```

We will cover more about *control flow* next week; for now, this should be sufficient to help you with this lab. Let's play some cards.

3 First, some housekeeping

From the last lab, you most likely have a home directory containing a `comp9` directory, which contains the files (`report.txt`, `hypotenuse.rb`, perhaps `hypotenuse_name.rb`) that you wrote for lab 1. By the end of the semester, we'll have quite a lot of clutter if we don't clean things up a bit.

First, create a `lab1` directory *within* `comp9` and *move* (`mv`) all the lab 1 files into there.

Next, create a `lab2` directory also within `comp9` (not within `lab1`).

Finally, open Kate if you don't already have it open.

4 Getting started

In Kate, create a file and save it as `blackjack.rb` within your `lab2` directory.

Create a header for that file, just like you did in lab1. It should include the necessary 'boilerplate' for a Ruby program:

```
#!/usr/bin/env ruby
```

And it should also include some information about you and the assignment:

```
# File: blackjack.rb
# Name: The Abominable Snowman
# Date: 02/03/2011
# Lab 2
```

Unless you really are the Abominable Snowman, use a more appropriate name.

Note those lines that begin with `#`? They aren't Ruby code. They're *comments*. You can write whatever you like on a line as long as it starts with a `#`. The purpose of those is to help *humans* read your program. The exception is that `#!/usr/bin/env ruby` line, which just has to be there. I'll explain it eventually, I promise.

Now, take a break from code!

5 Blackjack

Please write out an *outline* of your program. You may do it on paper; either use *pseudocode* (just English describing your program in detail) or diagrams of any sort. We ask you to check with a member of course staff (Noah, Sarah, or Joel) to approve your design before you begin coding. We will also ask you to submit this *design* along with your final lab submission. If you'd like us to photocopy the design, let us know; you may also just submit it with two names on the top, in hardcopy. Explain on paper:

1. where and how you will ask for input
2. where and how you will provide output

3. what loops and conditionals you will need

4. most importantly, how you will be sure you are upholding the rules of the Blackjack game.

I encourage you to work on this first step with someone sitting near you. Don't share code, but discuss how you might approach the problem of writing a blackjack game as described below. Sketch out on paper how you might approach this problem.

You are going to write a blackjack game. Now, for simplicity's sake, we're going to cheat a bit for now: we're not going to use a fair deck, as that would require keeping track of which cards have already been played. So really, don't bet against the house on this game. We're going to assume an infinite deck, for starters.

Leaving aside our simplifications, here's a refresher as to the rules of Blackjack (or, if you will, a really simplified version of Blackjack):

1. The player is dealt two cards.

2. The player's object is to get as high a score as possible, without exceeding 21 ("busting").

3. Numbered cards count as their natural value

4. Jack, Queen, and King ("face cards") count as 10

5. Aces are valued at 1 or 11 according to the player's preference.

6. If the hand value exceeds 21 points, it busts.

Simplifications:

1. For this game, we won't display or represent actual cards, but just their scores: values between 1 and 11.

2. We are not going to worry about the fact that once you've drawn all four '2' cards, the probability of drawing another '2' is zero. In our case, the dealer is cheating with an infinite deck.

3. We are not going to worry about keeping track of what hand the player already has, just what their total score is.

4. At some tables, the player is asked what value he or she wants for an ace; at others, the dealer chooses the 'best' choice for them. Our program should always ask, even if the choice would be obvious (leading to a score of 21, or leading to a 'bust')

Initially, the game should present the player with a *two-card* hand: the sum of two random values, each ranging from 1 to 10. Why 1 to 10, and not 1 to 11? Because if the player gets an ace, he or she gets to *choose* whether or not it should count as 1 or 11. If your program (as the dealer) ever deals a 1, it should *ask* the user whether or not to add 10 to the total.

The astute reader may notice that while there are 4 ways to get a card valued at 2 (one for each suit), and 4 ways to get a card valued at 4 (one for each suit), there are *16* ways to get a card valued at 10 (for each suit, a 10, Jack, Queen, or King). We will ignore this for now, because we're playing with a simplified (and infinite) deck.

Now, as mentioned, the player is dealt two cards, and should be *asked* if any 1 should be turned into an 11.

The player should then be told his or her score, and so begins a *loop*:

The player is asked whether to *hit* or *stick*. Hit means the program should deal the player one more card; stick means the player is satisfied with that score. If the player types *hit*, the program should deal another card. The program should continue asking *hit* or *stick* until the sum of the card values (including aces valued at 11) hits 21; if it equals 21, the player wins! If, however, it exceeds, 21, the player loses, and should be insulted accordingly!

As a reminder, sketch out your approach to this problem with a neighbor, and please ask for help if you're stuck. Once you feel your approach is reasonable, ask a member of the course staff (Noah, Sarah, or Joel) to look at your design. Once they approve it, you may begin coding. Your code must be your own; you should not collaborate with your neighbor beyond this point.

6 Code

Just to recap, write a Ruby program, `blackjack.rb`, that implements the simplified blackjack game described above. You will need a loop and several conditionals, and some use of `gets` and `puts`.

7 Handing in your solution

When you are satisfied with your program, submit it to be graded with the following command:

```
provide comp9 lab2 blackjack.rb
```

You must also hand in your design sketch; please submit it (by hand) with the names of all involved students at the top. If you would like a photocopy for your reference, let us know.

8 Future directions

The limitations of this program are pretty obvious: it doesn't really handle face cards (all valued at 10), and it assumes an infinite deck (it doesn't know that you've already drawn all four 'twos,' for example).

You haven't yet been taught the tools that will be needed to correct this, but we'll get there. When we talk about *containers* you will see how you could keep track of all the drawn cards, and how to change the probability based on how many cards have been drawn.