UNBC Laboratory Assignment
Computer Science 101—Winter 2023

Card Classes

Purpose

To explore class related features: `enums`, inheritance, overriding `Object` class methods, and recursion. Not all of these ideas are contained in this assignment. There will be a follow-on assignment that bulids on this one.

Due Date

The completed lab assignment is due by 18:00 Tuesday, 2023-02-07.

enum types

This section contains background information. The actual coding assignment starts in the Section "More Finite classed" on page 3.

Often we deal with data where the number of possible values is small and finite. For instance, `booleans` have two possible values, playing cards have four possible suits.

Programming languages have had `enum` types since long before object oriented programming. Java also enumeration types, but they are integrated with class types. `enum` types are discussed briefly in Section 5.4 of the text, and in more detail in Appendix A.

The basic syntax is;

```
public enum Suit { Club, Diamond, Heart, Spade ; }
```

This is short-hand for something like

```
public final class Suit extends Enum<Suit> {
  public static final Club = new Suit(...);
  public static final Diamond = new Suit(...);
  public static final Heart = new Suit(...);
  public static final Space = new Suit(...);
  // ...
  }
```

- You can't actually compile this code; however it does explain some important points.
- The `enum` values (like Club) are public static final constants of the Suit class.
- Addicg constructors to an `enum` class is possible (alhtough they must be private).

- enums are like other classes in that they can be public and top-level:

  lab9/Dwarf.java

  ```
  package lab9;
  public enum Dwarf {
      Thorin, Fíli, Kíli, Óin, Glóin, Balin,
      Dwalin, Ori, Dori, Nori, Bifur, Bofur, Bombur ;
  }
  ```

  In this case, they need to follow the same file name rules as other classes.

- Alternatively, (and frequently) they can be enclosed in a surrounding class.

enums and switches

Enumerated types work particularly well with switch statements. If an enumerated type is used in a switch, the case labels are simple names, regardless of the long name for the type. For instance:

```
lab8.cards.Rank x = // ...
switch(x)
  {
  case Ace: // just Ace, not lab8.cards.Rank.Ace
            // ...
  case Ten: // ...
  default: // ...
  }
```

Converting enums to and from ints

Every enum class has a non-static member function ordinal that returns an int. For instance, in the Dwarf example, Thorin.ordinal() is 0, and Ori.ordinal() is 8.

It is also possible, to convert ints to enums, but in this case you need to be careful, because only certain values will convert. Use code like

```
Dwarf.getClass().getEnumConstants()[i]
```

to find the i^{th} Dwarf (0-based indexing).

Additional functions

It is possible to add member functions. For instance, in cribbage, each card has a count, which is the face value, except that jacks, queens, and kings all count as 10.

```
public enum Rank {
    Ace, Two, Three, /* ... */ Ten, Jack, Queen, King ;

    public int count() { return Math.min(ordinal()+1,10) ; }
    // ordinal is inherited.  It starts at 0 for Ace, ...
}
```

It is even possible to add state (private member variables) to an `emum` class. See Appendix A of the book for details.

More finite classes: a `Card` class

Here is where actual code requirements start.

⇒ Create `Rank` and `Suit` `enum` types and use them below.

⇒ Create an immutable `Card` class with the following interface

```
  public Rank getRank() ;
  public Suit getSuit() ;
  public int getCribCount() ;
  public String toString() ;

  public static Card getCard(Rank r, Suit s) ;
  public static Card getCard(int i) ;
```

The `toString()` method should return a string like "`9S`" for the nine of spades. Tens can be represented either by "`T`" (`'\u0054'`) or by "⑩" (`'\u2469'`). (Instead of `C D H S` for clubs, diamonds, hearts and spaces, ♣ (`'\u2663'`) ♢ (`'\u2662'`) ♡ (`'\u2661'`) and ♠ (`'\u2660'`) are also acceptable.

Your `Card` class needs constructors, and internal private state variables. The `static getCard` mehtods should just call these constructors, as in

```
  public static Card getCard(Rank r, Suit s) { return new Card(r,s) ; }
  public static Card getCard(int i)          { return new Card(i) ;   }
```

The `Card(int` i`)` should take a number $0 \leq i < 52$.

You can make the constructors protected or private if you wish. In a following assignment we will explore ways to make the `Card` class have at most 52 objects.

Test Class I

⇒ Write a `TestCard` class that reads in lines of input and writes lines of output until there is no more input. Each input line should have one or more card values, separated by single spaces. For instance an input line might look like

```
5S TC AH JD
```

The output should be the same card values in the same format, followed by the total pegging count, for instance:

```
5S TC AH JD : total count 26.
```

Test Class II

This part is optional. You can get full marks on the assignment without completing it. However, it will also give you up to 5 marks to compensate for lost marks in the preceding parts.

⇒ Write a `Test15s` class that reads in lines of input and writes lines of output until there is no more input. Each input line should have exactly five card values separated by spaces. The output should be like `TestCard`, except that it should report the total number of card combinations whose count totals to 15.

Here is some possible output:

```
JS QS KS 5H 5D : there are 6 fifteens.
JS QS 5C 5H 5D : there are 7 fifteens.
AS 2S 3C 2H 2D : there are 0 fifteens.
4S 6S 4C 5H 5D : there are 4 fifteens.
```

Approaches

Finding a means to count the number of 15s is similar to finding a way to compute the subsets of a set. There are a number of ways of doing this. The most systematic and powerful way is to use recursion. A sketch of this approach is shown in Figure 1 on the following page.

Another approach is to use the integers $0, \ldots, 31$ to stand for subsets. A sketch of this approach is shown in Figure 2.

Yet another approach is to use five nest for-loops. A sketch of this approach is shown in Figure 3.

You may even find your own approach.

```
 private static int countXs(ArrayList<Card> cards, int goal)
     {
     if (goal==0) return 1;
     else if (goal <0 || cards.size()<1) return 0 ;
     else
         {
         ArrayList<Card> smaller = new ArrayList<Card>(cards) ;
         Card lastCard = smaller.remove(smaller.size()-1) ;
         int newGoal = goal - lastCard.getCribCount() ;
         return countXs(smaller,goal) + countXs(smaller,newGoal) ;
         }
     }
```

Figure 1: Sketch of a recursive approach

```
     for (int subset=0; subset < 1<<cards.size(); ++subset)
         {
         int total=0 ;
         for (int i=0; i<cards.size(); ++i)
             {
             if ((subset & (1<<i)) > 0)
                 total += cards.get(i).getCribCount() ;
             }
         if (total==15) { ++answer ; }
         }
```

Figure 2: Sketch of a bit-bashing approach

```
     for (int useCard0=0; useCard0<2 ; ++useCard0)
         for (int useCard1=0; useCard1<2 ; ++useCard1)
             for (int useCard2=0; useCard2<2 ; ++useCard2)
                 for (int useCard3=0; useCard3<2 ; ++useCard3)
                     for (int useCard4=0; useCard4<2 ; ++useCard4)
                         if (useCard0*cards.get(0).getCribCount() +
                             useCard0*cards.get(1).getCribCount() +
                             useCard0*cards.get(2).getCribCount() +
                             useCard0*cards.get(3).getCribCount() +
                             useCard0*cards.get(4).getCribCount() == 15)
                             ++count;
```

Figure 3: Sketch of a loopy approach