CS1812 Exercises 1

Goals

The goal of these exercises is to gain practice using enumerated types, and to write some recursive programs in Java.

Before you start

☞ Download the archive `cs1812-week1-assets.zip` from Moodle. It should contain the following files.

- `Rank.java`
- `Suit.java`
- `TestSuit.java`
- `Agent.java`
- `StringRec.java`

1 Enumerated Types

An `enum` type is a special data type that ensures the value of a variable can only be one from a set of predefined constants. You can find a full description here: `http://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html`

1.1 Playing Cards

Each card from a traditional deck of 52 cards has a rank (Ace, King, Queen, etc.) and a suit (Spades, Hearts, Clubs, or Diamonds). We will represent both ranks and suits as enumerated types.

In this week's assets you will find the file `Suit.java` where the `enum` type `Suit` is defined. The contents of the file are as follows.

```java
public enum Suit {
  SPADES, HEARTS, CLUBS, DIAMONDS;
}
```

The following conventions are used for `enum` types.

- ℹ The name of the `enum` type is capitalised.
- ℹ The members of the `enum` type are all written in upper case.

In fact, `enum` types are classes and, as such, the `enum` class body can include methods and other fields. Some methods are added by default to an `enum` class. The following are examples of such methods.

- ℹ The method `name()` returns the name of the `enum` constant it is applied to.
- ℹ The method `values()` returns an array containing all of the values in the order they are declared.

Let's try them out. In this week's assets you will find the file `TestSuit.java` with the following contents.

```java

public class TestSuit {

  public static void main(String[] args) {
    Suit a = Suit.HEARTS;
    Suit b = Suit.HEARTS;
    Suit c = Suit.CLUBS;

    System.out.println(a.name() + " = " + b.name() + " ? " + a.equals(b));
    System.out.println(a.name() + " = " + c.name() + " ? " + a.equals(c));
    System.out.println(a.equals(c));

    for (Suit s : Suit.values()) {
      System.out.println( s );
    }
  }
}
```

Compile both classes `Suit` and `TestSuit`. Now, run `TestSuit` and check the output. The program creates three variables of type `Suit`.

- ℹ Notice how values of `enum` are referred to: this is the standard for `enum` types.
- ℹ Observe that equality the variables is checked using the `equals()` method.
- ℹ The names of the suits are printed with the `name()` method.

The program enters a loop in which it prints the four suits to the command line, in the order they were declared. As mentioned previously, `Suit.values()` is an array of type `Suit[]`. To iterate over that array we are using a `for each` loop; this is similar to a `for` loop, but does away with the need for an index variable.

☞ Modify the `TestSuit` class so that it uses a `for` loop instead of the `for each` loop.

Once you have compiled and run the program, observe that the `for each` loop delivers the array elements in the ascending order of their index.

The `toString()` method (created by default for all the Java classes) returns, in the case of an `enum` type, the name of the `enum` constant it is applied to. As it can be overridden (the `name()` method cannot), its output may be more user friendly and should be used instead.

☞ Modify `TestSuit` so that it uses `toString()` instead of `name()`.

☞ Compile and run the program; observe that the output is the same.

Since the `toString()` method of an object is automatically called when the object is printed, we may omit it.

☞ In your new version of `TestSuit` replace, in the loop, `s.toString()` by `s`; observe that the output is still the same.

We will also represent ranks as enumerated types, but we want to assign a value to each rank. There are several games in which cards have values, we will use the following scoring system: aces are scored at 20 points, face cards at 15 points, and all other cards at their numerical values.

2 ... 10	Jack	Queen	King	Ace
2 ... 10	15	15	15	20

⚠ From a conceptual point of view, since cards may be used for any game the values should not be directly associated with the cards: all the game specificities would be included in some other class. Bear in mind that we are doing this to get acquainted with `enum` types and that if we were building a real card deck we would separate cards from points.

In this week's assets you will find the file `Rank.java` with the following contents.

```java
public enum Rank {
  ACE(20), KING(15), QUEEN(15), JACK(15), TEN(10), NINE(9), EIGHT(8), SEVEN(7),
  SIX(6), FIVE(5), FOUR(4), THREE(3), TWO(2);

  private int points;

  private Rank(int points) {
    this.points = points;
  }
  public int getPoints() {
    return this.points;
  }
}
```

To add a value to each card we need the constructor `Rank(int points)`, which is private. This is always the case with constructors of `enum` types, as the elements of the set are constants and may never be modified (this is why there is no setter for the field `points`). In fact, if you declare the constructor as public you will get a compilation error.

☞ Compile `Rank`.

☞ Make the constructor public, compile `Rank` again, and read the compilation error.

☞ Make the constructor private again, as we will need the correct version later on.

Exercise 1. Write a program `TestRank.java` that uses a `for each` loop to print all the ranks.

a) Start by using the method `name()` to print just the name of each rank.

b) Override the method `toString()` in the `enum` type `Rank` so that it prints both the name and the value of the rank.

c) Modify `TestRank` to print both the name and the value of all the ranks using the method `toString()`.

Exercise 2. There are three kinds of ranks: aces, numbers and faces.

a) Add to the `enum` type `Rank` a private field of type `String` and associate a kind to each rank. For example, the King rank should now become `KING(10,"face")`.

b) Modify the constructor of `Rank` accordingly.

c) Update the `toString()` method of `Rank` so that it now also prints the kind of the rank.

d) Test these modifications. Do you need to modify the `TestRank` program to do that?

Exercise 3. Write a `Card` class to represent a single playing card. It should have three private fields: `rank` of type `Rank`, `suit` of type `Suit`, and `points` of type `int`.

a) Write getters for each field. You will not write any setters as, once a card is created, we will not need to change the fields.

b) To reinforce that fact, mark the three fields as final. What does this mean? (Hint: once you have marked the fields as final, write a setter for one of them and try to compile the class).

c) Write a constructor that takes a `Rank r` and a `Suit s` as arguments, and initialises the three fields. (Hint: the field `points` is retrieved from the rank of the card).

d) Override the `equals(Card other)` method of `Card` so that it returns `true` if, and only if, both cards have the same rank and suit. (Hint: use the corresponding methods in the `enum` types `Rank` and `Suit`).

e) Override the `toString()` method of `Card` so that it returns the card rank and suit. For example, the King of Hearts should return `"KING of HEARTS"`.

f) Test your `Card` class by writing a `TestCard` class. In its `main` method create the King of Hearts and the Ace of Spades, and print both to the command line.

1.2 Agents

In this week's assets you will find the file `Agent.java`, whose contents begins with the following code.

```java
class Agent {

  private int pos_x;
  private int pos_y;
  private int direction; // 0 = north, 1 = east, 2 = south, 3 = west

  public Agent() {
    pos_x = 0;
    pos_y = 0;
    direction = 0;
  }
```

The class defines an "agent" who finds itself at a certain position, indicated by the coordinates `pos_x` and `pos_y`, and is facing a certain direction: 0 for North, 1 for East, 2 for South, and 3 for West. Any other integer is not a valid direction. It contains methods to change the direction that the agent is facing, as well as causing the agent to take a step in the direction that it is currently facing. It also contains a `main` method to test the class.

☞ Compile and run the `Agent` class.

Exercise 4. Modify the code in the `Agent` class so that it uses an `enum` type instead of `int` for the direction. It is up to you to decide how to modify the implementations of the `turnRight()`, `turnLeft()`, and `move()` methods so that they use the new `enum` type. However, you might consider the following.

- ☑ Add fields/methods to the new `enum` type to return the change in the x and y coordinates when the agent moves in that direction.
- ☑ Add fields/methods to the new `enum` type to return the new direction that results from turning (anti-)clockwise.

☞ Test your solution by compiling and running your code: the output produced by the javainlinemain method should be identical to before.

- ❓ What are the advantages of using an `enum` in this example?
- ❓ Are there any advantages to using integers, as you were originally given?

2 Recursion

You may remember some exercises on recursion from Exercise Sheet 5 in CS1811, and from Checkpoint Sheet 5 in CS1822 last term. Here are some more exercises on recursion using Java.

Exercise 5. Write a program `SquareSum.java` that, given a positive integer n, recursively computes the sum $1^2 + 2^2 + \cdots + n^2$.

a) Write a method with the signature `static int sqSum(int n)` that recursively computes the sum given above.

b) Write the `main` method so that it reads an integer `i` from the command line arguments, and then calls `sqSum(i)` and prints the result to the screen.

If you try the program with the input 6 (by running `java SquareSum 6` on the command line) you should get the result 91.

In this week's assets you will find the file `StringRec.java`. The method `isLowerCase` provides an example of recursion on a string. The base case is when `s` is empty, in other words when `s.isEmpty()` returns true. In the recursive case a substring is taken from `s` to make it smaller. It uses `s.charAt(0)` to obtain the first character of the string `s`, and `s.substring(1)` to get the substring of `s` starting at index 1: the string `s` without its first character.

```java
class StringRec {

  static public void main(String[] args) {
    if (args.length < 1) {
      System.out.println("Please provide a command line argument");
      return;
    }
    String s = args[0];
    System.out.println(isLowerCase(s));
  }

  static public boolean isLowerCase(String s) {
    if (s.isEmpty()) {
      // all the characters in the empty string are uppercase
      return true;
    } else {
      if (Character.isUpperCase(s.charAt(0))) {
        // first character is uppercase
        // therefore not all characters are lowercase
        return false;
      } else {
        // may still find uppercase characters in the rest of the string
        return isLowerCase(s.substring(1));
      }
    }
  }
}
```

Exercise 6. Using recursion, add to the `StringRec` class a `static` method with the signature `String removeUpper(String s)` that returns the string `s` with its uppercase characters removed. For example, `removeUpper("TeSt")` should return `"et"` and `removeUpper("DONkey")` should return `"key"`.

Hint. Use the definition of the `isLowerCase` method you have been provided with as inspiration. If the current character is upper case, simply return the result of recursively calling the method on the rest of the string. If the current character is lower case, you need to return the string obtained by concatenating this character with the result of recursively calling the method on the remainder of the string.

> ☞ Test your solution by checking whether `isLowerCase(removeUpper(args[0]))` returns `true`.

Exercise 7. Using recursion, add to the `StringRec` class a `static` method with the signature `String toLowerCase(String s)` that returns the string `s` with any uppercase characters converted to lowercase.

Hint. This requires just a small modification of your solution to Exercise 6, so that in the case the current character is uppercase, the method converts it to lowercase (using the method `toLowerCase(Char c)` in the `Char` class), and returns the result of concatenating this to the result of recursively calling the method on the remainder of the string.

> ☞ Test your solution by checking whether `isLowerCase(toLowerCase(args[0]))` returns `true`.
>
> ☞ Test your solution further by checking whether the following code returns `true`: `args[0].equalsIgnoreCase(toLowerCase(args[0]))`.

Exercise 8. Using recursion, add to the `StringRec` class a `static` method with the signature `String revRec(String s)` that reverses `s`. For example, `revRec("Java")` should return `"avaJ"`.