CS2030 Programming Methodology
Semester 1 2020/2021

30 September 2020
Problem Set #5 Suggested Guidance

1. In Java, a `Set` is a `Collection` that does not contain duplicate elements (this is in contrast to a `List` which does allow duplicates). You are given the `Point` class below:

```
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "(" + this.x + ", " + this.y + ")";
    }
}
```

   (a) What is the output of the following program fragment executed in jshell?

```
List<Point> points = new ArrayList<>()
points.add(new Point(1, 1))
points.add(new Point(1, 1))
points.indexOf(new Point(1, 1))
```

```
jshell> List<Point> points = new ArrayList<>()
jshell> points.add(new Point(1, 1))
$3 ==> true
jshell> points.add(new Point(1, 1))
$4 ==> true
jshell> points.indexOf(new Point(1, 1))
$5 ==> -1
jshell>
```

   Notice how the `indexOf` *method makes use of* `Objects.equals(Object a, Object b)` *method which indirectly involves the use of an object's* `equals` *method when searching the list. Since the* `Object` *class* `equals` *method is invoked, each new point is a different object despite having the same* x *and* y *coordinate values.*

(b) By defining an appropriate overriding `equals` method, demonstrate how the `indexOf` method can now give the correct behaviour.
Include the following method in the `Point` *class.*

```
    @Override
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (obj instanceof Point) {
            Point point = (Point) obj;
            return this.x == point.x && this.y == point.y;
        } else {
            return false;
        }
    }
}
```

Running jshell now gives the desired output:

```
jshell> List<Point> points = new ArrayList<>()
jshell> points.add(new Point(1, 1))
$3 ==> true
jshell> points.add(new Point(1, 1))
$4 ==> true
jshell> points.indexOf(new Point(1, 1))
$5 ==> 0
jshell>
```

(c) What is the output of the following program fragment executed in jshell?

```
Point p = new Point(1, 1);
Point q = new Point(1, 1);
p.equals(q)
Set<Point> set = new HashSet<>()
set.add(p)
set.add(q)
set
```

```
jshell> /open Point.java
jshell> Point p = new Point(1, 1);
jshell> Point q = new Point(1, 1);
jshell> p.equals(q)
$4 ==> true
jshell> Set<Point> set = new HashSet<Point>()
jshell> set.add(p)
$6 ==> true
jshell> set.add(q)
$7 ==> true
jshell> set
set ==> [(1, 1), (1, 1)]
```

(d) Notice that although `p.equals(q)` returns `true`, the two points are considered distinct by `HashSet`. How do we ensure that only one point is maintained in the set?

Hint: Refer to the definition of the `equals` *method in* `Object` *class*

We need to know the "Contract between `equals()` *and* `hashCode()`*", i.e. if two objects are equal according to the* `equals(Object)` *method, then calling the* `hashCode()` *method on each of the two objects must produce the same integer result.*

Indeed, `p.hashCode()` *and* `q.hashCode()` *gives a different result. As such we can define the following* `hashCode` *method in the* `Point` *class.*

```
@Override
public int hashCode() {
    return Arrays.hashCode(new int[] {this.x, this.y});
}
```

Now `HashSet` *will behave correctly:*

```
jshell> Point p = new Point(1, 1);
jshell> Point q = new Point(1, 1);
jshell> p.equals(q)
$4 ==> true
jshell> Set<Point> set = new HashSet<Point>()
jshell> set.add(p)
$6 ==> true
jshell> set.add(q)
$7 ==> false
jshell> set
set ==> [(1, 1)]
jshell>
```

2. The Java `Collection<E>` interface extends the `Iterable<E>` interface with the following abstract method declared.

```
Iterator<E> iterator();
```

(a) Using the methods in the `Iterator` class, demonstrate how iteration is performed on a `List`, e.g.

```
List<Point> list = new ArrayList<>();
list.add(new Point(1, 1));
list.add(new Point(2, 2));

Iterator<Point> iter = list.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

(b) How is the use of an `Iterator` object, different from the following

```
for (Point p : list) {
    System.out.println(p);
}
```

From the `Iterator` *class, there is a* `remove()` *method that removes the last element returned by the iterator.*

```
jshell> List<Point> list = new ArrayList<>()
jshell> list.add(new Point(1, 1))
$3 ==> true
jshell> list.add(new Point(2, 2))
$4 ==> true
jshell>
jshell> Iterator<Point> iter = list.iterator();
jshell> while (iter.hasNext()) {
   ...>     System.out.println(iter.next());
   ...>     iter.remove();
   ...> }
(1, 1)
(2, 2)
jshell> list
list ==> []
jshell>
```

3. What is the output of the following program fragment? Explain.

```
class A {
    static void f() throws Exception {
        try {
            throw new Exception();
        } finally {
            System.out.print("1");
        }
    }

    static void g() throws Exception {
        System.out.print("2");
        f();
        System.out.print("3");
    }

    public static void main(String[] args) {
        try {
            g();
        } catch (Exception e) {
            System.out.print("4");
        }
    }
}
```

214

- *Since* `main()` *calls* `g()`, `2` *will be printed first.*
- `f()` *is then executed, which leads to an exception being thrown.*
- *Before* `f()` *returns, it executes the* `finally` *block, leading to* `1` *being printed.*
- *After returning to* `g()`*, since an exception was thrown,* `3` *will NOT be printed, with control returning to* `main()`*, which catches the exception.*
- *The* `catch` *block is executed and* `4` *is then printed.*

4. You are given two classes `MCQ` and `TFQ` that implements a question-answer system:

   - MCQ: multiple-choice questions comprising answers: `A B C D E`
   - TFQ: true/false questions comprising answers: `T F`

```
class MCQ {
    String question;
    char answer;

    public MCQ(String question) {
        this.question = question;
    }

    void getAnswer() {
        System.out.print(question + " ");
        answer = (new Scanner(System.in)).next().charAt(0);
        if (answer < 'A' || answer > 'E') {
            throw new InvalidMCQException("Invalid MCQ answer");
        }
    }
}

class TFQ {
    String question;
    char answer;

    public TFQ(String question) {
        this.question = question;
    }

    void getAnswer() {
        System.out.print(question + " ");
        answer = (new Scanner(System.in)).next().charAt(0);
        if (answer != 'T' && answer != 'F') {
            throw new InvalidTFQException("Invalid TFQ answer");
        }
    }
}
```

   In particular, an invalid answer to any of the questions will cause an exception (either `InvalidMCQException` or `InvalidTFQException`) to be thrown.

```
class InvalidMCQException extends IllegalArgumentException {
    public InvalidMCQException(String mesg) {
        super(mesg);
    }
}
```

```
class InvalidTFQException extends IllegalArgumentException {
    public InvalidTFQException(String mesg) {
        super(mesg);
    }
}
```

By employing the various object-oriented design principles, design a *more general* question-answer class `QA` that can take the place of both MCQ and TFQ types of questions (and possibly more in future, each with their own type of exceptions).

Here are some design issues that needs to be addressed:

- *Abstract out common code in* `TFQ` *and* `MCQ` *to a common place* `QA`
- `MCQ` *and* `TFQ` *inherits from* `QA`
- `QA`*'s abstract method* `getAnswer` *throws a more general exception than* `MCQ` *and* `TFQ`
- *Possibly creating a more general exception class* `InvalidQuestionException`

```
import java.util.*;

abstract class QA {
    String question;
    char answer;

    public QA(String question) {
        this.question = question;
    }

    abstract void getAnswer();

    char askQuestion() {
        System.out.print(question + " ");
        return new Scanner(System.in).next().charAt(0);
    }
}

class MCQ extends QA {
    public MCQ(String question) {
        super(question);
    }

    void getAnswer() {
        answer = askQuestion();
        if (answer < 'A' || answer > 'E') {
            throw new InvalidMCQException("Invalid MCQ answer");
        }
    }
}
```

```
class TFQ extends QA {
    public TFQ(String question) {
        super(question);
    }

    void getAnswer() {
        answer = askQuestion();
        if (answer != 'T' && answer != 'F') {
            throw new InvalidTFQException("Invalid TFQ answer");
        }
    }
}

class InvalidQuestionException extends IllegalArgumentException {
    public InvalidQuestionException(String mesg) {
        super(mesg);
    }
}

class InvalidMCQException extends InvalidQuestionException {
    public InvalidMCQException(String mesg) {
        super(mesg);
    }
}

class InvalidTFQException extends InvalidQuestionException {
    public InvalidTFQException(String mesg) {
        super(mesg);
    }
}
```

because of SRP

Sample client class:

```
class Main {
    static void processQuestion(QA question) {
        try {
            question.getAnswer();
        } catch (InvalidQuestionException ex) {
            System.err.println(ex);
        }
    }

    public static void main(String[] args) {
        QA mcq = new MCQ("What is the answer to this MCQ?");
        QA tfq = new TFQ("What is the answer to this TFQ?");
        processQuestion(mcq);
        processQuestion(tfq);
    }
}
```