00202

THE UNIVERSITY OF BRITISH COLUMBIA
CPSC 210: MIDTERM EXAMINATION – March 8, 2017

Last Name: Zhao First Name: Steven

Signature: [illegible] UBC Student #: 18949164

Important notes about this examination

1. You have **90 minutes** to write this exam.
2. **You should have received a separate collection of pages including Java code.**
3. **All questions must be answered on this portion of the exam.**
4. **You may complete the exam in either pen or pencil.**
5. Put away books, papers, laptops, cell phones... everything but pens, pencils, erasers and this exam.
6. Good luck!

Please do not write in this space:

Question 1:	10	AR
Question 2:	16	BO
Question 3:	14	TA
Question 4:	16	JD
Question 5:	15	RC
Question 6:	16	[illegible]

0 236366 453321

Use this page if you need more space to answer a question.

Question 1 [10 marks]: Type Hierarchies

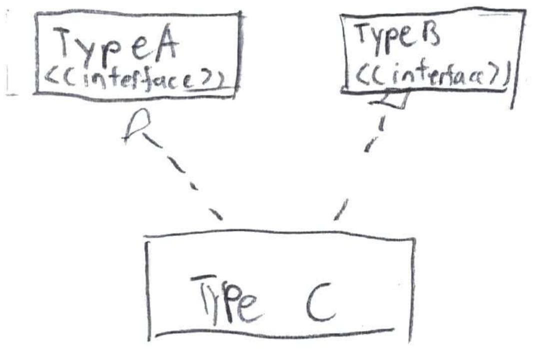
In the code segment that follows, the lines labelled (1) and (2) compile, while the remaining lines do not compile.

```
(1)   TypeA a1 = new TypeC();
(2)   TypeB b1 = new TypeC();

(3)   a1 = b1;
(4)   b1 = a1;

(5)   TypeA a2 = new TypeA();
(6)   TypeB b2 = new TypeB();
```

Draw a type hierarachy that includes TypeA, TypeB and TypeC that is consistent with the information provided above.



10

Question 2 [16 marks]: Exception Handling – Flow of Control

```
public class Main {
    public static void main(String[] args) {
        int n = 100;
        Dividend d = new Dividend(n);
        System.out.println("main: start");
        try {
            System.out.println("main: try");
            n = 50;
            d.divideBy(n);
            n = 0;
            d.divideBy(n);
            n = -50;
            d.divideBy(n);
        } catch (RangeException e) {
            System.out.println("main: catch1: n==" + n);
        } catch (ZeroValueException e) {
            System.out.println("main: catch2: n==" + n);
        } finally {
            System.out.println("main: finally: n==" + n);
        }
        System.out.println("main: end: n==" + n);
    }
}

public class Dividend {
    private Ranger ranger;
    private final int DIVIDEND; -100

    public Dividend (int dividend) {
        DIVIDEND = dividend;
        ranger = new Ranger(dividend);
    }

    public int divideBy(int divisor) throws ZeroValueException,
                                                RangeException {
        try {
            int result = DIVIDEND / ranger.check(divisor);
            System.out.println("divideBy: result==" + result);
            return result;
        } catch(ArithmeticException e) {
            System.out.println("divideBy: catch");
            throw new ZeroValueException();
        }
    }
}
```

```
public class Ranger {
    private final int MAX;

    public Ranger(int max) {
        MAX = max;
    }

    public int check(int num) throws RangeException {
        System.out.println("check(" + num + "): start");
        if (num < 0 || MAX < num) {
            throw new RangeException();
        }
        System.out.println("check(" + num + "): end");

        return num;
    }
}
```

Assuming that `RangeException` and `ZeroValueException` are checked exceptions and that each provides all the necessary constructors for the code above to compile, what is printed on the console when we run the method `Main.main`?

Note that an `ArithmeticException` is thrown if an attempt is made to divide an integer by zero.

```
Main: start
Main: try
check(50): start
Check(50): end
divideBy: result == 2
Check(0): start
Check(0): end
divideBy: catch
main: catch2: n==0
Main: finally: n==0
main: end: n==0
```

16

Question 3 [16 marks]: Designing and Testing Robust Methods

```java
// Represents an automatic garage door that is open or closed
// and has a sensor that is triggered if there is something
// blocking the door.
class GarageDoor {
    private boolean isOpen;
    private boolean isSensorTriggered;

    // EFFECTS: constructs door that is closed with sensor not triggered
    public GarageDoor() {
        // stub
    }

    // MODIFIES: this
    // EFFECTS:  sensor is cleared
    public void clearSensor() {
        // stub
    }

    // MODIFIES: this
    // EFFECTS:  if the door is closed, throws StateException
    //           otherwise sensor is triggered
    public void triggerSensor() throws StateException {
        // stub
    }

    // EFFECTS: returns true if door is open, false otherwise
    public boolean isOpen() {
        return false;  // stub
    }

    // EFFECTS: returns true is sensor is triggered, false otherwise
    public boolean isSensorTriggered() {
        return false;  // stub
    }

    // MODIFIES: this
    // EFFECTS:  if the door is already open, throws StateException
    //           otherwise opens the door
    public void openDoor() throws StateException {
        // stub
    }

    // MODIFIES: this
    // EFFECTS:  if the door is already closed, throws StateException
    //           otherwise, if the sensor is triggered, throws SensorException
    //           otherwise, closes the door
    public void closeDoor() throws StateException, SensorException {
        // stub
    }
}
```

a) Design a jUnit test for `GarageDoor.closeDoor` in the case where the method is expected to throw a `SensorException`. Do not rely on a method tagged with the `@Before` annotation. All the code needed must be written in your test method.

```
@Test (expected = SensorException.class)                    throws SensorException
public void testCloseDoorSensorException() ^{
        GarageDoor gd = new GarageDoor();
        assertFalse(gd.isOpen());
        try {
            gd.openDoor();
            gd.triggerSensor();
        } catch (StateException e) {
            fail("Caught unexpected StateException");
        }
        gd.closeDoor(); // throws SensorException
}
```

1
1
1
1
1
1

b) Implement the method `GarageDoor.closeDoor`. Assume that the exceptions referred to in the specification have been designed to include a constructor that takes no arguments. It is not necessary to copy the comment statements that form part of the method's specification into your answer.

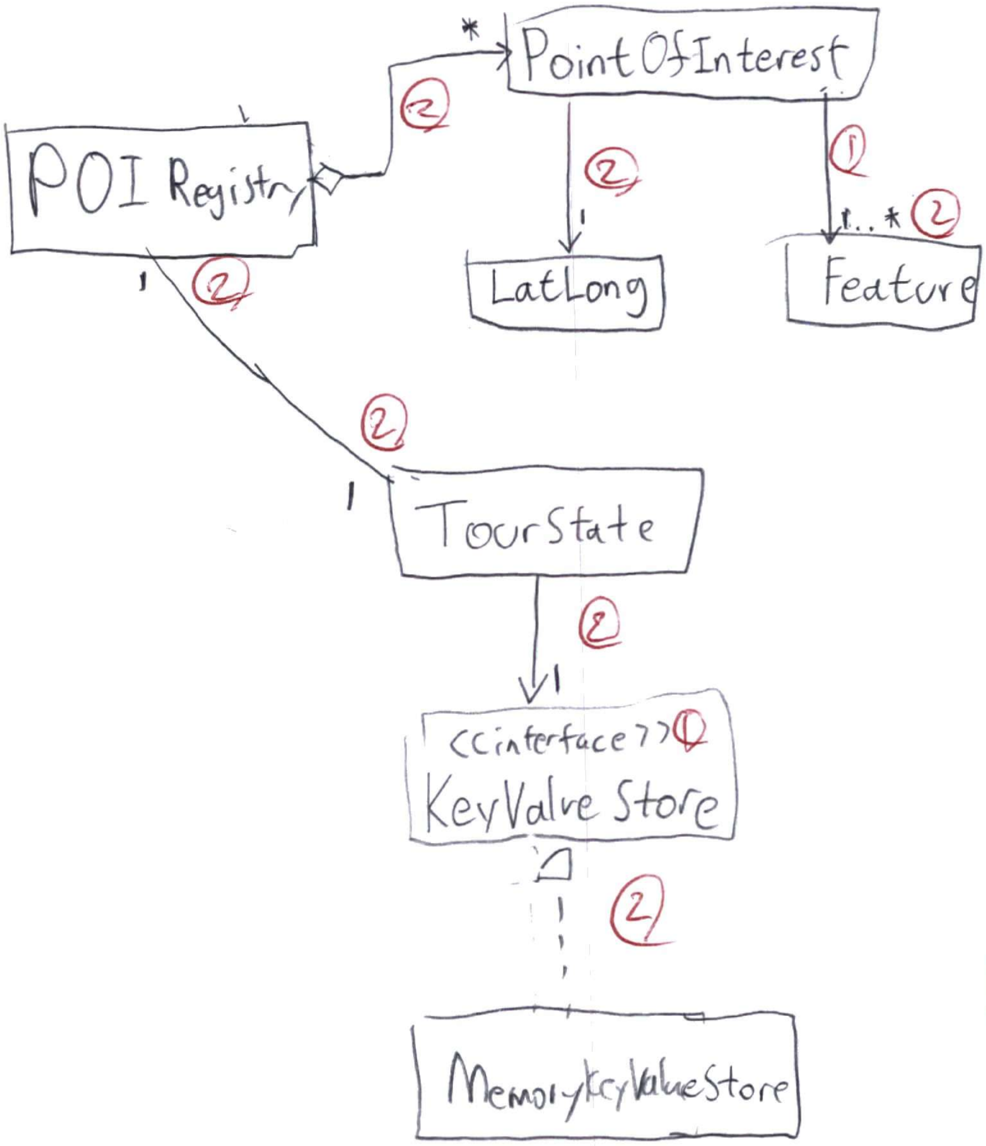
```
public void closeDoor() throws StateException, SensorException {
        if (!this.isOpen())
            throw new StateException();
        if (this.isSensorTriggered())
            throw new SensorException();
        isOpen = false;
}
```

3

3
2

Question 4 [16 marks]: Extracting a UML Class Diagram
Extract a UML class diagram that includes the following types presented in the code package: `Feature`, `PointOfInterest`, `LatLong`, `KeyValueStore`, `MemoryKeyValueStore`, `POIRegistry` and `TourState`. You must include all implements, extends, association and aggregation relationships as well as multiplicities where appropriate.

On your diagram it is necessary to include only the name of the type in the box that represents that type – do not add fields or methods to those boxes.



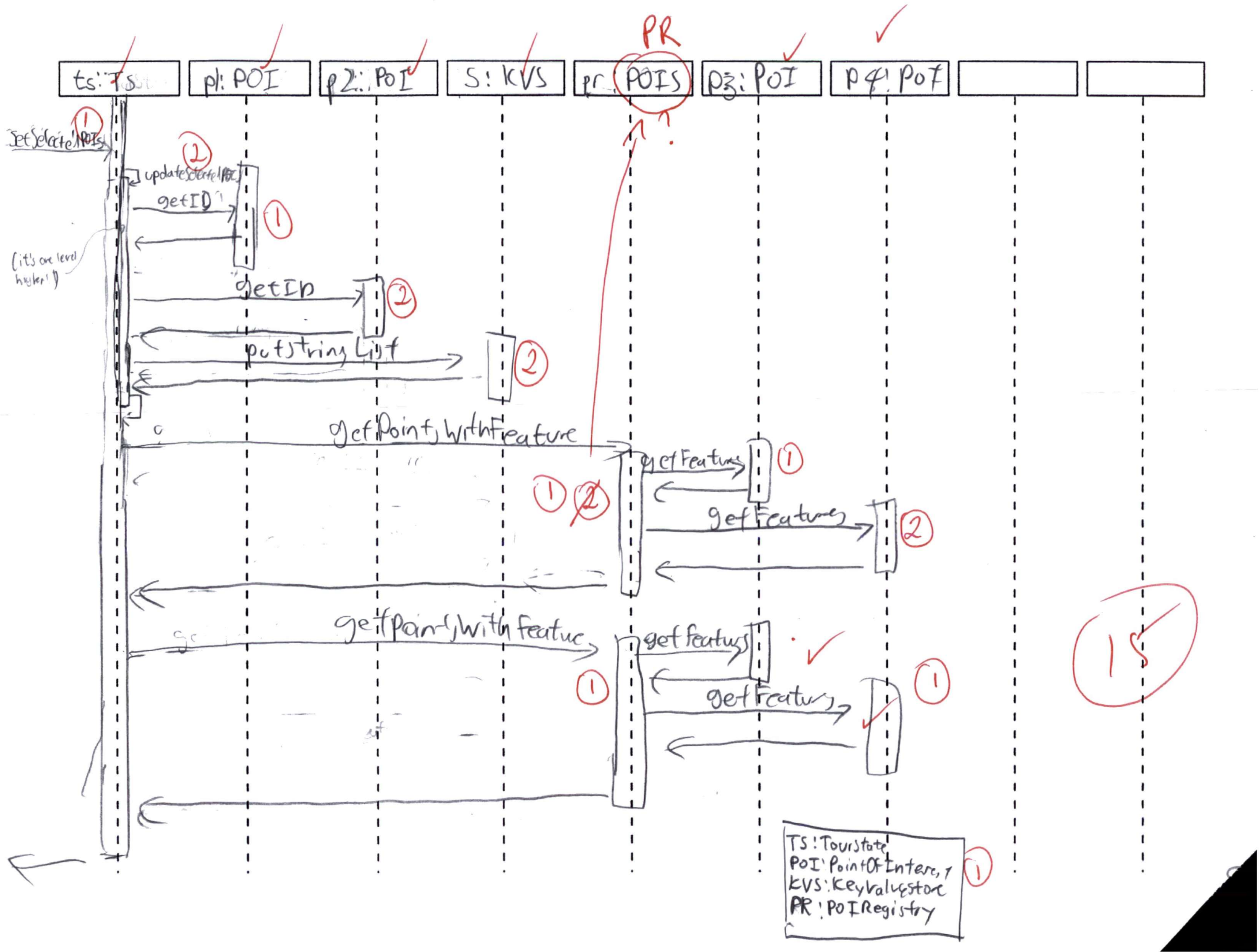
16 ☺

Question 5 [16 marks]: Extracting a UML Sequence Diagram
Extract a UML sequence diagram for the method `TourState.setSelectedPOIs` presented in the code package. Include only calls that are specified in the code package (so don't include calls to any methods in the Java library). If you encounter a for-each loop in the code, you must assume that the collection over which you are iterating contains exactly two values.

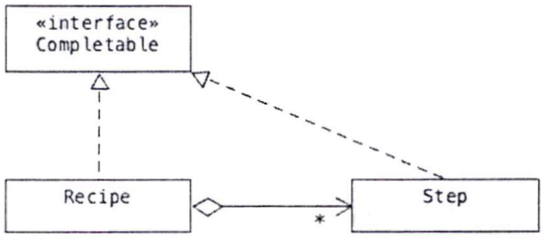
Note that the method `Feature.values()`, used in the `TourState.setSelectedPOIs` method, returns a collection of all the values in the Feature enumeration. Further note that there are exactly two such values: `LEED_CERTIFICATION` and `SOLAR_ENERGY`.

Draw your diagram onto the skeleton UML sequence diagram provided on the following page. Be sure to fill in all necessary details. Note that you may not need all of the objects provided but you will not need more than those provided. If necessary, abbreviate the names of types and functions and provide a legend to specify the correspondence between the abbreviation and the full name.

Do not attempt to draw your diagram on this page!


ts: TS
p1: POI
p2: POI
S: KVS
pr: POIS
PR
p3: POI
p4: POI
setSelectedPOIs
updateSelectedPOIs
getID
(it's one level higher!)
getID
putStringList
getPointsWithFeature
getFeatures
getFeatures
getPointsWithFeature
getFeatures
getFeatures
TS: TourState
POI: PointOfInterest
KVS: KeyValueStore
PR: POIRegistry
1
2
15

Question 6 [16 marks]: Implementing an Object-Oriented Design
In this question you are to provide an implementation for the `Recipe` class according to the design presented in the following UML class diagram



Note that the `Completable` interface includes only one method, specified as follows:

```
// EFFECTS: returns true if this is complete, false otherwise
boolean isComplete();
```

In addition to conforming to the design presented in the UML class diagram above, your `Recipe` class must meet the following requirements:

- there must be only one constructor that takes no arguments
- a recipe is considered complete if all of its individual steps are complete; a recipe having no steps is considered complete
- you can add steps to a recipe one at a time but, once added, you cannot remove them
- you can get a list of all the steps in a recipe in the order in which those steps were added to the recipe

You **must not** provide any functionality other than that specified above. Write the implementation of your `Recipe` class on the following page. Be sure to provide appropriate Requires/Modifies/Effects style documentation. Note that jUnit tests are not required.

Write your answer to this question on the following page.

ASSUMING getSteps returns an unmodifiable ArrayList of steps

This space provided for your answer to Question 6.

```
public class Recipe implements Completable {     2
    private List<Step> steps;                     1
    //MODIFIES: this
    //EFFECT: instantiates new Recipe object
    public Recipe() {
        this.steps = new ArrayList<>();          1
    }
    //MODIFIES: this
    //EFFECT: adds a step to the Recipe object
    public void addStep(Step step) {  ✓
        steps.add(step);
    }
    //EFFECT: returns a list of the steps in order
    public List<Step> getSteps() {                   ✓
        return Collections.unmodifiableList(steps);
    }
    //EFFECT: returns whether is all steps in Recipe is completed
    public boolean isComplete() {
        for (Step step : steps) {
            if (!step.isComplete())
                return false;
        }
        return true;      6                     D: 2
    }
}
```