**Solutions to Quiz 2 (March 12, 2014)**

**Problem 1** (Short Answer) (**15 points**).  
**Circle all correct answers** for the following questions.

(a) Consider the datatype definition:  
\[
\text{Geology} = \text{Core}(a: \text{int}, b: \text{int}, c: \text{int}, d: \text{int})
+ \text{Planet}(\text{core}: \text{Core}, a: \text{int}, b: \text{int}, c: \text{int}, d: \text{int})
+ \text{System}(\text{geology}: \text{Geology}, a: \text{int}, b: \text{int})
\]
Suppose you have a reference to a Geology object. How many integers could it have in its representation?

A. 1  
B. 2  
C. 4  
D. 8  
E. 10  
F. 12

**Solution.** C, D, E, F.

(b) Suppose you hear the following statements in a discussion. Which statements are sensible?

A. “The constructor’s precondition is violated, so it’s legal for it to create an object that violates the rep invariant.”  
B. “This Java interface needs to have a rep invariant written in a comment.”  
C. “The way to tell that a class is immutable is to look for the final keyword in its instance variables.”  
D. “An immutable class cannot mutate any of its instance variables once the constructor returns.”  
E. “hashCode() is a good shortcut for testing equality – if two things have the same hashCode(), you know they’re equal.”

**Solution.** Only A. B is wrong because interfaces have no rep, and hence no rep invariant. C is wrong because final variables may still point to mutable objects; the only sure way to tell if a class is immutable is to look for mutator methods. D is wrong because an immutable class can do beneficent mutation (see lecture 11). E is wrong because two equal things must have the same hash code, but not the other way around.

(c) Consider the following code, which is supposed to be an implementation of Java’s String class.

```java
public class String {
    public final List<Character> chars = new ArrayList<Character>();
}
```

Which of the following statements are true?
A. The class’s immutability is threatened because the rep is public.
B. The class’s immutability is threatened because it uses Character.
C. The class’s immutability is threatened because it uses final.
D. This rep is immutable because it uses final.

Solution. A.

Problem 2 (Abstract Data Types) (30 points).
Suppose you have an abstract datatype Vegetable with the following operations:

leafy: Vegetable -> boolean  
requires: non-null Vegetable  
returns: true if and only if the Vegetable is leafy

crunch: Vegetable -> int  
requires: non-null Vegetable  
returns: crunchiness as an integer from 1 (mushy) to 10 (teeth shattering)

crunchiest: List<Vegetable> -> Vegetable  
requires: ... see part (a) below ...  
returns: ... see part (a) below ...

We’ll implement this ADT with a Java class:

class Vegetable {
  ...
  
  public Vegetable(...) {
    ...
  }
  
  public boolean leafy() {
    ...
  }
  
  public int crunch() {
    ...
  }
  
  public static Vegetable crunchiest(List<Vegetable> veggies) {
    ...
  }
}
(a) Below, the letters A, B, and C are different specifications for the crunchiest operation, and the diagrams at the bottom show different possible relationships among these specifications, when the ovals are interpreted as sets of legal implementations for each specification.

A) requires: list of Vegetables
   effects: returns a vegetable with greatest crunchiness

B) requires: list of Vegetables
   effects: returns the vegetable with greatest crunchiness, or throws ChompingException in the case of a tie

C) requires: non-empty list of Vegetables
   effects: returns the vegetable with greatest crunchiness, or throws ChompingException in the case of a tie

Circle the number of the diagram that correctly describes the relationship among A, B, and C.
On the diagram you chose, label each oval with A, B, and C, using the blanks that stick out from each oval.

Solution. 2. B ⊂ C, because B is stronger than C: an implementation that can handle any list of Vegetables (spec B) can also handle a nonempty list of Vegetables (spec C), but not vice versa. A does not intersect the other two, because it does something completely different in the case of a tie, so no implementation can satisfy both spec A and one of the other specs.
(b) For each of the following implementations of a Java `equals` method for `Vegetable`, select the single best answer:

```java
@Override public boolean equals(Object obj) {
    return this == obj;
}
```

(circle only one)

A. not reflexive
B. not transitive
C. not symmetric
D. causes a compile-time error due to types
E. causes a runtime error, even for valid input
F. correctly implements observational equality
G. none of the above

**Solution.** None of the above. This implements reference equality, which satisfies all the conditions of an equivalence relation but doesn’t do observational equality.

```java
@Override public boolean equals(Object obj) {
    if (! (obj instanceof Vegetable)) { return false; }
    return this.leafy() == obj.leafy() && this.crunch() == obj.crunch();
}
```

(circle only one)

A. not reflexive
B. not transitive
C. not symmetric
D. causes a compile-time error due to types
E. causes a runtime error, even for valid input
F. correctly implements observational equality
G. none of the above

**Solution.** compile-time error, because `obj.leafy()` can’t call `leafy()` on a reference of type `Object`.

```java
@Override public boolean equals(Object obj) {
    if (! (obj instanceof Vegetable)) { return false; }
    Vegetable v = (Vegetable)obj;
    return this.leafy() == v.leafy() && Math.abs(this.crunch() - v.crunch()) < 3;
}
```

(circle only one)

A. not reflexive
B. not transitive
C. not symmetric
D. causes a compile-time error due to types
E. causes a runtime error, even for valid input
F. correctly implements observational equality
G. none of the above

Solution. not transitive, because crunchiness 1 will be equal to crunchiness 3, and crunchiness 3 will be equal to crunchiness 5, but crunchiness 1 won’t be equal to crunchiness 5.
(e) Below are three possible representations for Vegetable. For each representation, the left column shows the rep and rep invariant, the middle column shows how two key methods are implemented with that rep, and the right column shows part of the abstraction function.

Each column has some blanks for you to fill in. Fill in the blanks so that all three representations are correct and consistent.

<table>
<thead>
<tr>
<th>Rep</th>
<th>Operation implementations</th>
<th>Abstraction function</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>--------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>private boolean leafy;</td>
<td>public boolean leafy() { return leafy; }</td>
<td>(true, 5) -&gt; (true, 5)</td>
</tr>
<tr>
<td>private int crunch;</td>
<td>public int crunch() { return Math.min(crunch,10); }</td>
<td>(false, 15) -&gt; (false, 10)</td>
</tr>
<tr>
<td>// rep invariant:</td>
<td>}</td>
<td>(true, -5) -&gt; not mapped</td>
</tr>
<tr>
<td>// 1 &lt;= crunch &lt;= 10</td>
<td>}</td>
<td>(false, 15) -&gt; not mapped</td>
</tr>
<tr>
<td>// _______________</td>
<td>}</td>
<td>(true, -5) -&gt; not mapped</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>(true, -5) -&gt; not mapped</td>
</tr>
</tbody>
</table>

Solution. rep invariant: 1 <= crunch

leafy() { return leafy; }
crunch() { return crunch; }

(5) -> (false, 6)
(15) -> (false, 6)
(-5) -> (true, 6)

Problem 3 (Rep Invariants) (25 points).
Consider the following abstract data type, which has some placeholders shown in all caps.

/**
Room is an immutable datatype representing a room with a width and a height, and a set of obstacles that a robot might have to maneuver around. It makes use of Point, an immutable datatype representing 2D positions (x,y).

```java
public class Room {

    private final double width, height;
    private final String roomNumber;
    private Set<Point> obstacles;
    /* REP INVARIANT */

    /**
     * requires: PRECONDITION
     * effects: POSTCONDITION
     */
    public List<Point> findPath(Point initial, Point goal) throws PathNotFoundException {
        ...
        List<Point> path = new ArrayList<Point>();
        ...
        return path;
    }
    ...
}
```

From the following list, choose the parts that belong in REP INVARIANT, PRECONDITION, and POSTCONDITION. Write RI, PRE, or POST next to each statement according to whether it belongs in the rep invariant, precondition, or postcondition comment. Leave a statement blank if it is redundant with or contradicts information that’s already present in the code above.

_____ Point objects in obstacles satisfy 0<=x<=width and 0<=y<=height
_____ initial and goal both satisfy 0<=x<=width, 0<=y<=height
_____ width > 0 and height > 0
_____ throws PathNotFoundException if no obstacle-free path exists from initial to goal
_____ returns a list of points that form a path from initial to goal without colliding into any obstacles.
_____ roomNumber consists only of letters, digits, dashes, or spaces
_____ roomNumber is a mutable string of characters

Solution. RI, PRE, RI, POST, POST, RI, blank.
Problem 4 (Inheritance) (30 points).
Consider these two classes:

```java
public class Shape {
    public void match(Shape s) {
        System.out.println(name() + " has a " + s.name());
    }
    public String name() { return "Shape"; }
}

public class Triangle extends Shape {
    public void match(Triangle t) {
        System.out.println(name() + " got a " + t.name());
    }
    public String name() { return "Triangle"; }
}
```

For each code fragment below, determine what happens when it executes. Either show what the code prints by circling the right words or phrases from each column, or circle Error if the code produces a compile-time error or runtime exception. Assume the code fragments appear in sequence in a Java file, so that a fragment can use the variables declared in the earlier fragments.

(a) Shape s = new Triangle();
    s.match(s);

```
Shape has a Shape
Triangle got a Triangle
```

Error

Solution. Triangle has a Triangle. Note that Triangle.match() is overloaded not overridden.

(b) Triangle t = new Triangle();
    t.match(t);

```
Shape has a Shape
Triangle got a Triangle
```

Error

Solution. Triangle got a Triangle.

(c) t.match(s);

```
Shape has a Shape
Triangle got a Triangle
```

Error

Solution. Triangle has a Triangle.

(d) Triangle u = new Shape();
    u.match(u);
Shape has a Shape
Triangle got a Triangle

Error

Solution. Error (compile-time in fact), because a Shape isn't a Triangle.