

Massachusetts Institute of Technology  
6.170 Laboratory in Software Engineering

Spring 2007

# Quiz 1

Friday, March 2, 2007

Name:     SOLUTIONS    

Athena username: \_\_\_\_\_

Section (circle one):

1: Lucy Mendel

2: Vikki Chou

3: David Glasser

4: Kah Seng Tay

5: Omair Malik

6: Clayton Sims

This quiz is closed book, closed notes. You have 50 minutes to complete it. It contains 24 questions in 15 pages (including this one), totaling 100 points. The last three pages contain duplicate material from multi-page questions. You may tear-off those pages for your reference. Before you start, please check your copy to make sure it is complete. Turn in pages 1-15, together, when you are finished. Separately turn-in pages 16-18.

**Write your initials and section number on the top of ALL pages.**

Please write neatly; we cannot give credit for what we cannot read.

*Good luck!*

Question(s)	Grading Scheme	Total	Max												
1 to 7	<input style="width: 50px; height: 20px;" type="text"/> X 2 =	<input style="width: 50px; height: 20px;" type="text"/>	of 14												
8 to 18	<table border="1" style="display: inline-table; border-collapse: collapse;"> <tr> <td style="padding: 2px;">8,9</td> <td style="padding: 2px;">10, 11</td> <td style="padding: 2px;">12, 13</td> <td style="padding: 2px;">14, 15</td> <td style="padding: 2px;">16</td> <td style="padding: 2px;">17, 18</td> </tr> <tr> <td style="height: 20px;"></td> <td></td> <td></td> <td></td> <td></td> <td></td> </tr> </table> <span style="margin-left: 20px;">X 5 =</span>	8,9	10, 11	12, 13	14, 15	16	17, 18							<input style="width: 50px; height: 20px;" type="text"/>	of 55
8,9	10, 11	12, 13	14, 15	16	17, 18										
19		<input style="width: 50px; height: 20px;" type="text"/>	of 7												
20		<input style="width: 50px; height: 20px;" type="text"/>	of 6												
21	<input style="width: 50px; height: 20px;" type="text"/> X 4 =	<input style="width: 50px; height: 20px;" type="text"/>	of 12												
22		<input style="width: 50px; height: 20px;" type="text"/>	of 6												
		<input style="width: 50px; height: 20px;" type="text"/>	of 100												

## True/False Questions

### Question 1.

**T/F** Unlike interfaces, abstract classes are partially implemented and can be instantiated.

F - Abstract classes cannot be instantiated

### Question 2.

**T/F** Unlike interfaces, abstract classes may implement constructors.

T - Although abstract classes cannot be instantiated, subclasses may call the abstract class's constructor to initialize the super class's state

### Question 3.

**T/F** In Java, both implements and extends indicate the "is-a" relationship between the class and what that class implements or extends.

T - A class can be substituted anywhere the class that it extends or implements is expected (this is also the "true subtype" concept, not just what the compiler permits)

### Question 4.

**T/F** Abstract Data Types are immutable.

[F; the concepts aren't really related]

### Question 5.

**T/F** In Java, Abstract Data Types are expressed as interfaces.

[F; they can be classes too]

### Question 6.

**T/F** According to the specification for `Object.equals(Object)`, `y.equals(x)` implies `x==y`

[F]

### Question 7.

**T/F** According to the specification for `Object.equals(Object)`, `x.equals(y)` implies `y.equals(x)`

[T]

## Multiple Choice Questions:

### Question 8. (circle all that apply)

The specifications for an Abstract Data Type can refer to which of the following:

- (A) Specification fields
- (B) Java fields
- (C) The Abstraction Function
- (D) Observers
- (E) Creators

[A, D, and E]

### Question 9. (circle all that apply)

Which of the following statements are true?

- (A) A factory method can act as a substitute for a constructor.
- (B) Using interfaces eliminates the need to rewrite fields and methods that were already implemented in the interface.
- (C) A Java class can have multiple superclasses.
- (D) An interface can extend multiple interfaces.
- (E) Java supports multiple inheritance

[A, D]

**Question 10.** (circle all that apply)

Which of the following statements are true?

- (A) The abstraction function can refer to the concrete (Java) fields of a class.
- (B) The representation invariant specifies the valid outputs of the abstraction function.
- (C) The representation invariant should be part of a class's public Javadocs.
- (D) An abstraction function for a class can be written by someone with no knowledge of or access to the implementation of that class.
- (E) Checking the validity of rep invariant at the end of a public method is good defensive programming

[A, E]

**Question 11.** (circle all that apply)

The tests that run every night during the project development or before a programmer checks in any changes to the global repository are normally characterized as \_\_\_\_\_.

- (A) Regression tests
- (B) Integration tests
- (C) Validation tests
- (D) System tests
- (E) None of the above

[A.]

**Question 12.** (circle all that apply)

Glass box tests \_\_\_\_\_.

- (A) should always be written before you write the code for a class.
- (B) have their quality measured by what percentage of the code they cover
- (C) may require their writer to think about control-flow details
- (D) ensure that clients fulfill the preconditions of a method
- (E) can be reused even if the implementation of a class changes, as long as the specification does not change

[B, C]

**Question 13.** (circle the most relevant)

A a = new B()

- (A) Will compile in Java only if A is a java subtype of, or equal to, B
- (B) Will compile in Java only if B is a java subtype of, or equal to, A
- (C) Will always compile in Java
- (D) Will not compile in Java if A and B are not the same type.
- (E) None of the above

[B]

```
/** represents a mutable solid ball */
public class SolidBall {
    // radius : integer
    // RI – radius >= 0
    // AF – this.radius represents the actual radius of the physical Ball
    private int radius; // ball radius
    SolidBall(int radius) {
        this.radius = radius;
    }
    /** @returns weight */
    public int getWeight() {
        return (int)(4.0/3.0*Math.PI*Math.pow(radius, 3));
    }
    /** @returns volume */
    public double getVolume() {
        return getWeight();
    }
}

/** represents a SolidBall filled with low-density filler with air bubbles making the ball bouncy */
public class BouncySolidBall extends SolidBall {
    BouncySolidBall(int radius) {
        super(radius);
    }
    /** @returns weight */
    public int getWeight() {
        return (int)(getVolume()/2);
    }
}

SolidBall ball = new BouncySolidBall(6); // note: 63 = 216
System.out.print(ball.getWeight());
```

**Question 14.** (circle the most relevant)

In the class `SolidBall`, the method `getWeight()` is invoked in the code for `getVolume()`. What is the compile-time type of the receiver of `getWeight()`?

- (A) The answer varies depending on how `getVolume()` is invoked
- (B) `BouncySolidBall`
- (C) `SolidBall`
- (D) `Object`
- (E) None of the above

[C - `getWeight` is invoked on this, which in the context of `SolidBall` is `SolidBall`. There is only one compile-time type, thus only `Account` is correct (and not also `Object`).]

**Question 15.** (circle the most relevant)

The method call `ball.getWeight()`

- (A) prints the integer  $216 \cdot \pi$
- (B) prints the integer  $288 \cdot \pi$
- (C) prints the integer  $144 \cdot \pi$
- (D) prints an arbitrary integer due to ambiguous dispatch.
- (E) throws an exception due to infinite recursion.

[E]

Ben Bitdiddle joins a local software company in Kendall Square that deals in telecommunications. Ben's first assignment is to investigate problems that have arisen from merging code in the software's code base. Ben finds 2 different implementations for a function, `checkHashes`. Abstractly, `checkHashes` checks whether a numerical hash of each string in `entries` is equal to its predetermined value in `correctHashes`.

```
public boolean checkHashes(List<String> entries, List<Integer> correctHashes)
```

Fortunately for Ben, all the implementations have specifications written in 6.170 style.

Specification A

@requires: `entries.length = correctHashes.length`

@modifies: nothing

@throws: nothing

@effects: nothing

@returns: true iff for all  $i$  s.t.  $0 \leq i < \text{entries.length}$ , `hash(entries [i]) = correctHashes[i]`

Specification B

@requires: nothing

@modifies: `entries`

@throws: `ArrayOutOfBoundsException` if `entries.length < correctHashes.length`

@effects: for all  $i$  where `hash(entriesPRE[i]) != correctHashes [i]`, `entriesPOST[i] = ""`

@returns: true iff for all  $i$  s.t.  $0 \leq i < \text{entries.length}$ , `hash(entriesPOST[i]) = correctHashes[i]`

**Question 16.** (circle the most relevant)

What following statement(s) represent the relationship between the above two specifications

- (A) Specification A and specification B are equivalent
- (B) Specification A is stronger than the specification B
- (C) Specification B is stronger than the specification A
- (D) Neither specification is stronger than the other
- (E) Without analyzing the implementation, it is impossible to say

D Neither specification is stronger. Specification A requires more, but modifies less than specification

After more investigation, Ben discovers another implementation with this specification:

Specification C

@requires: `entries.length >= correctHashes.length`

@modifies: nothing

@throws: nothing

@effects: nothing

@returns: true iff for all  $i$  s.t.  $0 \leq i < \text{correctHashes.length}$ , `hash(entries[i]) = correctHashes[i]`

**Question 17.** (circle all that apply)

We would like to reduce the code bloat by eliminating redundant methods.

- (A) Method A can be replaced by the implementation of method B.
- (B) Method A can be replaced by the implementation of method C.
- (C) Method B can be replaced by the implementation of method A.
- (D) Method C can be replaced by the implementation of method A.
- (E) No methods can be replaced.

[B]

**Question 18.** (circle all that apply)

We know class A is a Java subtype of Class B if

- (A) Every situation that requires a class of type B will work correctly when given a class of type A
- (B) Every situation that requires a class of type A will work correctly when given a class of type B
- (C) There exists a declared relationship A extends B or A implements B
- (D) There exists a declared relationship B extends A or B implements A
- (E) B's specification is stronger than A's specification

[C]

Consider another implementation of the RatNum class which you saw in Problem Set 1 that keeps to the same specifications:

**/\*\* RatNum** represents an **immutable** rational number.

It includes all of the elements in the set of rationals, as well as the special "NaN" (not-a-number) element that results from division by zero.

The "NaN" element is special in many ways. Any arithmetic operation (such as addition) involving "NaN" will return "NaN". With respect to comparison operations, such as less-than, "NaN" is considered equal to itself, and larger than all other rationals.

Examples of RatNums include "-1/13", "53/7", "4", "NaN", and "0". *\*/*

```
public class RatNum extends Number implements Comparable<RatNum> {
    int whole;
    int numer;
    int denom;

    /** A constant holding a Not-a-Number (NaN) value of type RatNum */
    public static final RatNum NaN = new RatNum(1, 0);

    /** A constant holding a zero value of type RatNum */
    public static final RatNum ZERO = new RatNum(0);

    /** @effects Constructs a new RatNum = n. */
    public RatNum(int n) {
        whole = n;    // ADDED IN THIS IMPLEMENTATION
        numer = 0;   // CHANGED IN THIS IMPLEMENTATION
        denom = 1;
    }

    /** @effects If d = 0, constructs a new RatNum = NaN. Else constructs a new
        RatNum = (n / d) */
    public RatNum(int n, int d) {
        if (d == 0) { // special case for zero denominator; gcd(n,d) requires d != 0
            whole = 0; // ADDED IN THIS IMPLEMENTATION
            numer = n;
            denom = 0;
        } else {
            // reduce ratio to lowest terms
            int g = gcd(n,d);
            n = n / g;
            d = d / g;
            if (d < 0) {
                numer = - n;
                denom = - d;
            } else {
                numer = n;
                denom = d;
            }
        }
    }
}
```



**Question 19.**

Write a representation invariant based on this description. You may use math, clear English, Java code or any mixture.

<p>possible answer 1:</p> <p>(GCD(whole,denom)=1, reduced form, lowest terms, relatively prime)</p> <p>AND</p> <p>( whole=0, numer=1, denom=0  OR  whole!=0, 0&lt;=numer&lt;denom  OR  whole=0, 0&lt;=abs(numer)&lt;denom)</p> <p>=====</p> <p>possible answer 2:</p> <p>GCD(whole,denom) = 1, reduced form, relatively prime  AND  whole=0 =&gt; ((-denom&lt;numer&lt;denom) or (numer=1 &amp; denom=0))  AND  whole!=0 =&gt; 0 &lt; numer &lt; denom</p>	<p>possible answer 3:</p> <p>GCD, reduced form  AND ( (whole = 0 &amp; !numer&lt;denom (denom&gt;0)  OR (whole !=0 &amp; 0 &lt;= numer &lt; denom )  OR (0,1,0) )</p> <p>=====</p> <p>possible answer 4:</p> <p>denom &gt;= 0  denom = 0 =&gt; whole = 0 and numer = 1  (denom &gt; 0 =&gt;) gcd(whole, denom) = 1 (or numer=0)  denom &gt; 0 =&gt; abs(numer) &lt; denom  whole != 0 =&gt; numer &gt;= 0</p>
--	---

**Question 20.**

Write an abstraction function for RatNum.

<p>RatNum r such that:</p> <p>if this.denom = 0,    r = NaN  if this.whole &gt;= 0,    r = this.whole + this.numer/this.denom  if this.whole &lt; 0,    r = this.whole - this.numer/this.denom</p>
--

You realize that there are very large rational numbers that cannot be represented using RatNum. For example the whole part of RatNum can only handle numbers between Integer.MIN\_VALUE and Integer.MAX\_VALUE, thus a number such as  $17179869184 \frac{1}{3}$  (i.e.  $2^{34} \frac{1}{3}$ ) cannot be represented. You need to create a new class BigRatNum that is able to represent very large rational numbers. Thus, you decided to use a long to represent whole. You figure you can be smart about it by reusing the code of RatNum and just change whole from int to long. Clients of BigRatNum do not need to know how you implemented this class.

```
/** Immutable arbitrary-precision rational numbers. */
public class BigRatNum extends RatNum implements Comparable<RatNum>
{
    ...
    long whole;
    ...

    public BigRatNum(long n, int d) {
        if((n <= Integer.MAX_VALUE) && (n >= Integer.MIN_VALUE)) {
            numer = (int)n;
            denom = d;
            this.whole = super.whole;
        } else {
            long excess = n / d;
            numer = (int)(n - excess*d);
            denom = d;
            this.whole = super.whole + excess;
        }
    }
    ...
}
```

**Question 21.**

Write one sentence on 3 things that are wrong with this implementation.

There were many acceptable answers. Some of these include:

Implementation does not match the specification

Can only handle 64 bit integers for whole, does not support arbitrary precision.

this.whole = super.whole is incorrect as super.whole is uninitialized

The implementation of RatNum is exposed to BigRatNum

**Question 22.**

Briefly describe (in less than 2 sentences) a better way to implement BigRatNum.

Composition is better than using inheritance. Use RatNum and whole as members in the BigRatNum class.

```
/** represents a mutable solid ball */
public class SolidBall {
    // radius : integer
    // RI – radius >= 0
    // AF – this.radius represents the actual radius of the physical Ball
    private int radius; // ball radius
    SolidBall(int radius) {
        this.radius = radius;
    }
    /** @returns weight */
    public int getWeight() {
        return (int)(4.0/3.0*Math.PI*Math.pow(radius, 3));
    }
    /** @returns volume */
    public double getVolume() {
        return getWeight();
    }
}
```

```
/** represents a SolidBall filled with low-density filler with air bubbles making the ball bouncy */
public class BouncySolidBall extends SolidBall {
    BouncySolidBall(int radius) {
        super(radius);
    }
    /** @returns weight */
    public int getWeight() {
        return (int)(getVolume()/2);
    }
}
```

```
SolidBall ball = new BouncySolidBall(6); // note:  $6^3 = 216$ 
System.out.print(ball.getWeight());
```

**RatNum** represents an **immutable** rational number. It includes all of the elements in the set of rationals, as well as the special "NaN" (not-a-number) element that results from division by zero. The "NaN" element is special in many ways. Any arithmetic operation (such as addition) involving "NaN" will return "NaN". With respect to comparison operations, such as less-than, "NaN" is considered equal to itself, and larger than all other rationals. Examples of RatNums include "-1/13", "53/7", "4", "NaN", and "0". \*/

```
public class RatNum extends Number implements Comparable<RatNum> {
    int whole;
    int numer;
    int denom;
    /** A constant holding a Not-a-Number (NaN) value of type RatNum */
    public static final RatNum NaN = new RatNum(1, 0);

    /** A constant holding a zero value of type RatNum */
    public static final RatNum ZERO = new RatNum(0);

    /** @effects Constructs a new RatNum = n. */
    public RatNum(int n) {
        whole = n; // ADDED IN THIS IMPLEMENTATION
        numer = 0; // CHANGED IN THIS IMPLEMENTATION
        denom = 1;
    }

    /** @effects If d = 0, constructs a new RatNum = NaN. Else constructs a new RatNum = (n / d) */
    public RatNum(int n, int d) {
        if (d == 0) { // special case for zero denominator; gcd(n,d) requires d != 0
            whole = 0; // ADDED IN THIS IMPLEMENTATION
            numer = n;
            denom = 0;
        } else {
            // reduce ratio to lowest terms
            int g = gcd(n,d);
            n = n / g;
            d = d / g;
            if (d < 0) {
                numer = - n;
                denom = - d;
            } else {
                numer = n;
                denom = d;
            }
            whole = (int) (numer / denom); // ADDED IN THIS IMPLEMENTATION
            numer = numer - whole * denom; // ADDED IN THIS IMPLEMENTATION
            if (whole < 0) { // ADDED IN THIS IMPLEMENTATION
                numer = - numer; // ADDED IN THIS IMPLEMENTATION
            }
        }
    }

    /** Makes a RatNum from a string describing it.
    @requires 'ratStr' is an instance of a string, with no spaces, of the form:
        "NaN"
        "N/M", where N and M are both integers in decimal notation, and M != 0, or
        "N", where N is an integer in decimal notation.
    @returns NaN if ratStr = "NaN". Else returns RatNum r = ( N / M ), letting M be 1 in the case where only "N" is passed in. */
    public static RatNum valueOf(String ratStr) {
        int slashLoc = ratStr.indexOf('/');
        if (ratStr.equals("NaN")) {
            return NaN;
        } else if (slashLoc == -1) { // not NaN, and no slash, must be an Integer
            return new RatNum( Integer.parseInt( ratStr ) );
        } else { // slash, need to parse the two parts separately
            int n = Integer.parseInt(ratStr.substring(0, slashLoc));
            int d = Integer.parseInt( ratStr.substring(slashLoc+1, ratStr.length()));
            return new RatNum(n, d);
        }
    }
}
...
}
```

```
/** Immutable arbitrary-precision rational numbers. */
public class BigRatNum extends RatNum implements Comparable<RatNum>
{
    ...
    long whole;
    ...

    public BigRatNum(long n, int d) {

        if((n <= Integer.MAX_VALUE) && (n >= Integer.MIN_VALUE)) {
            numer = (int)n;
            denom = d;
            this.whole = super.whole;
        } else {
            long excess = n / d;
            numer = (int)(n - excess*d);
            denom = d;
            this.whole = super.whole + excess;
        }
    }
    ...
}
```