This is a CLOSED-BOOK quiz.
Before you start, write your name at the top of every sheet.
Please check your copy of the quiz before you start to make sure it is complete: you should have 7 sheets, printed on 14 sides in total.
There are 6 True/False questions, 19 multiple choice questions and 6 questions where you need to find the right answer from a list. Multiple choice questions can have more than one correct answer. Unless specified otherwise, CIRCLE ALL THAT APPLY.
You have 75 minutes and should attempt to answer all questions. Good luck!

Name: ________________________________________________

TA: __________________
Please select true or false

1) True / False  Having all public methods synchronized implies thread-safety.

2) True / False  Thread-safety implies that all public methods are synchronized.

3) True / False  Immutable objects are thread-safe.

4) True / False  Ordering locks ensures fairness.

5) True/False  Visitor pattern is extremely useful when you want to add a new subtype to the type with many methods

6) True/False  Visitors can be used as a higher order function

Alyssa P. Hacker has implemented an ATM system that has the following class:

```java
public class Account {
    private final int ID; // unique identifier
    private int balance;

    public String getID() { return ID; }
    public int getBalance() { return balance; }
    public void changeBalance(int offset) {
        ... // check that there's enough balance
        balance = balance + offset;
    }
}
```

In her ATM, she implements a method to handle transactions shown below. To deal with concurrency, Alyssa plants to use nested synchronized blocks on the two accounts:

```java
class ATM {
    ... // other stuff
    public void transaction(Account a, Account b, int amount) {
        Account acc1 = a; b;
        Account acc2 = b; a;
        synchronized(acc1) {
            synchronized(acc2) {
                ... // perform transaction
            }
        }
    }
}
```
7) Which of the following condition can she use to order her locks, i.e. to fill both blanks? Circle all that applies.
   a) a.getID() > b.getID()
   b) a.getBalance() > b.getBalance()
   c) a.hashCode() > b.hashCode().
   d) Consistently ordering locks is theoretically impossible.

8) Ordering locks is an example of which object-safety technique? (Choose one)
   a) Thread confined
   b) Single owner
   c) Shared read-only
   d) Shared thread-safe
   e) Guarded by others

Ben Bitdiddle decides to implement a stopwatch. It has two buttons: "start/stop" and "reset." The first button will cause the watch to start and pause timing. When paused, the watch displays the time so far elapsed since starting. If "reset" is pressed while the watch is not running, it should set the display to zero. Assume that the System.currentTimeMillis() is monotonically increasing and does not overflow.

```java
public class StopWatch {
    boolean isRunning = false; // keeps track if the watch is running.
    long startTime = 0;
    long stopTime = 0;

    // record time and reverses isRunning flag
    public void StartStop() {
        if (isRunning)
            stopTime = System.currentTimeMillis();
        else
            startTime = System.currentTimeMillis();
        isRunning = !isRunning;
    }

    public void reset() {
        if (!isRunning)
            startTime = stopTime;
    }

    // queries the current display
    public long display() {
        if (!isRunning)
            return stopTime - startTime;
        else
            return System.currentTimeMillis() - startTime;
    }
}
```
9) Which of the following are invariants in a single-threaded environment
   a) startTime <= stopTime
   b) startTime >= stopTime
   c) !isRunning → startTime < stopTime.
   d) System.currentTimeMillis() >= startTime
   e) System.currentTimeMillis() < startTime

10) Ben observed that in a multi-threaded environment (where multiple threads have access to one StopWatch object), calling display() sometimes returns a negative number. In order to prevent this, Ben absolutely need to synchronize (Pick the minimal set of methods):
   a) startStop()
   b) reset()
   c) display()
   d) currentTimeMillis()
   e) None. Ben was hallucinating.

11) Visitor Patterns rely on
   a) Casting.
   b) Overriding.
   c) Overloading
   d) Interning
   e) Generic Types
Ben Bitdiddle wants to write a new Java library for advanced arithmetic, for which he needs a way to represent the number system. He thinks integers and natural numbers are trivial, so he decides to tackle the more interesting case of rational numbers first. To this end, he wants to create a Java class called `RationalNumber`.

A `RationalNumber` object represents a rational number, defined as any value that can be expressed as a fraction a/b, where both a and b are integers.

12) Ben decided to represent his `RationalNumber` as an immutable type. Which of the following correctly describes why he should implement `RationalNumber` as an immutable type?

   a) He can pass `RationalNumber` objects around, without danger of them being misused or becoming inconsistent.

   b) State comparison is much easier.

   c) Building an immutable class gives Ben the opportunity to test the synchronized access techniques he learned in 6.005.

   d) Rational numbers are good candidates for an immutable data type since they are built from primitive types.

Ben decides to choose a representation where he keeps track of the fraction's numerator and denominator

```java
public class RationalNumber {
    public final int num;
    public final int den;

    // Constructors
    ...

    // Observers
    public double doubleValue() { return ((double) num) / den; }
    ...
}
```

13) Ben’s friend Allyssa did a code review and came up with the following statements about the current implementation and other design choices. Which of the following is true?

   a) If it is good that num and den are `final`. This satisfies the client’s need to understand the underlying representation when creating new `RationalNumber` objects.

   b) It is ok to declare num and den as `public` but not `final`. This poses no problems in the case where the client is a trusted party. They can look at the representation but not change it.

   c) It is better to make num and den `private`. This will prevent the need for modification in the client code if the ADT representation changes.

   d) This code has a problem. Not making the fields `private` allows the client to break the rep invariant.
Initial implementation of equals() is

```java
@Override
public boolean equals(Object o) {
    if (!(o instanceof RationalNumber)) return false;
    RationalNumber other = (RationalNumber) o;
    return this.num == other.num && this.den == other.den;
}
```

14) Allyssa also discovers that the equals() method is incorrect
   a) because it does not properly override Object.equals().
   b) because multiple instances of the same concrete value will be unequal.
   c) because multiple instances of the same abstract value will be unequal.
   d) because equals should have used == to test the equality as RationalNumbers is an immutable type

Ben remembers from 6.005 that representation invariants can be very useful and decides on refining his implementation using three rep invariants:
   1. denominator can never be zero
   2. denominator can never be negative
   3. fraction must always be in its simplified form

15) In order to impose the above rep invariant, Ben has to do the following minimal code changes to the code:
   Modify the checkRep method as well as:
   a) the members
   b) the constructors
   c) the observers
   d) the producers
   e) only the checkRep

   He adds the following code for the CheckRep method:

   ```java
   private void checkRep() {
       assert den > 0;
       assert gcd(num, den) == 1;
   }
   ```

   where gcd(int a, int b) is a helper function he implements to get the greatest common divisor of two numbers. (A simplified fraction means the GCD of the numerator and the denominator is one.)
Despite this rep invariant, Allyssa still finds a bug by running a set of unit tests comparing two RationalNumber objects. Allyssa find that they really should specify some of the edge cases. For simplicity, we'll ignore the infinity cases, but we'll examine the zero case. Ben decides extend the rep invariant such that an abstract value of zero should have a numerator of 0 and a denominator of 1.

16) Alissaa created two different RationalNumber objects to have the same abstract value, where each of the objects passes the above checkRep() but equals() method returns false. Identify the test Alissaa ran to discover the bug.

   a) (new RationalNumber(3, 2)).equals(new RationalNumber(3, 1))
   b) (new RationalNumber(6, 2)).equals(new RationalNumber(3, 1))
   c) (new RationalNumber(0, 1)).equals(new RationalNumber(0, 2))
   d) (new RationalNumber(-3, 4)).equals(new RationalNumber(3, -4))

17) Ben needs to add the following assertion to the checkRep() method to check for the extension to the rep invariant.

   a) assert num == 0 ? den == 1;
   b) assert num == 0 || den == 1;
   c) assert num*den >= 0 || num*den <= 0;
   d) assert num > 0;

18) After fixing the bug In the CheckRep Ben has to

   a) Leave the constructor unchanged as

   ```java
   public RationalNumber(int num, int den) {
       this.num = num;
       this.den = den;
   }
   ```

   b) Change the constructor to

   ```java
   public RationalNumber(int num, int den) {
       this.num = num;
       this.den = den;
       checkRep();
   }
   ```

   c) Change the constructor to:

   ```java
   public RationalNumber(int num, int den) {
       if((den > 0) ||
           (gcd(num, den) == 1) ||
           //check for the bug fix omitted
           ) then return;
       this.num = num;
       this.den = den;
       checkRep();
   }
   ```
d) Change the constructor to:

```java
public RationalNumber(int num, int den) {
    if (num == 0) {
        den = 1;
    }
    if (den < 0) {
        num = 0 - num; // negate both
        den = 0 - den;
    }
    ... // code to simplify num and den omitted
    this.num = num;
    this.den = den;
    checkRep();
}
```

Ben mistakenly implemented the producer negate() as follows.

```java
public RationalNumber negate() {
    return new RationalNumber(num, -den);
}
```

19) The above code for negate

a) Is incorrect because it violated the Rep Invariant that den > 0
b) Is incorrect because this method exposes the representation
c) Is buggy because executing this code fails CheckRep()
d) While not elegant, this code produces a valid RationalNumber that is the negation.

Here is the implementation of increment method.

```java
public RationalNumber increment() {
    num = num + den;
    return this;
}
```

20) This code is

a) Perfectly correct
b) Produces an incorrect because the calculation of increment is wrong
c) Wrong because it violates the rep invariant
d) It buggy because it violates immutability
e) Will be correct if return new RationalNumber(this) to avoid rep exposure.
21) Allyssa writes up a theorem solver that uses this class. Because she uses a parallel algorithm, she has multiple threads accessing the same rational numbers. She keeps getting inconsistent results and thinks the RationalNumber class is to blame. Is she right, and if so, what is the cause?

a) Yes; the fields aren't declared final.
b) Yes; the methods aren't synchronizing or locking the fields when they read them.
c) Yes; multiple instances of the same abstract value are allowed.
d) No; the class is completely thread-safe.

Allyssa also notices that his program is taking up a lot of memory, and suggests to Ben that the class save some space by using one instance for the same abstract value.

To enforce this, Ben decides to make the constructors private and provide a static method which returns instances of RationalNumber. This method can then check whether an instance representing the same abstract value already exists.

He declares a data structure to keep track of instances for abstract values and implements this method:

```java
private static Map<RationalNumber, RationalNumber> alloc =
    new HashMap<RationalNumber, RationalNumber>();

public static RationalNumber create(int num, int den) {
    RationalNumber key = new RationalNumber(num, den);
    if (!alloc.containsKey(key)) {
        alloc.put(key, key);
    }
    return alloc.get(key);
}
```

He remembers from 6.005 that an advantage to this approach is that it allows him to simplify his equals() method, since he can now assume that any two objects having the same abstract value must be the same instance.

22) He removes his equals() function, so the default Object.equals() behavior applies. This is

a) OK because now all the objects are interned and only a single copy exists
b) Incorrect because, members may have different values but still representing the same abstract number
c) Incorrect because containsKey will not find the matching object even if it is in the table
d) Incorrect because create is not synchronized thus two threads can insert identical objects at the same time
A 6.005 student implemented the Lecture exercise for Lecture 14 as follows:

```java
public interface BinaryFunction<T,U,V> {
    V apply(T t, U u);
}

public class Functions {
    public static <L,N> N reduce(List<L> lst, BinaryFunction<L,N,N> f, N zero) {
        int s = lst.size();
        if(s==0) return zero;
        return f.apply(lst.get(0), reduce(lst.subList(1, s), f, zero));
    }
}
```

She got so excited about the reduce, she decided to try it out using the following code.

```java
ArrayList<String> inList = new ArrayList<String>();
```

```java
ArrayList<String> outList = functions.Functions.reduce(lst, new BinaryFunction<String,ArrayList<String>,ArrayList<String>>() {
    public ArrayList<String> apply(String str, ArrayList<String> lst) {
        ArrayList<String> newlst = new ArrayList<String>(lst);
        newlst.add(str);
        return newlst;
    }
}, new ArrayList<String>());
```

23) **At the end of the execution, the list outList will contain**

   a) A list identical to inList
   b) A single string which is the concatenation of each strings in inList
   c) A list with the same strings as inList but in the reverse order
   d) An empty list
   e) A list of higher order functions

24) **A programmer can reduce the time spend in debugging by**

   a) Rapidly iterating through code-compile-run-debug cycle
   b) Inserting a lot of assertions to check pre and post conditions
   c) If the precondition is violated, returning null instead of throwing an exception
   d) Reducing the number of bugs discovered by reducing the number of times regression tests are run
   e) Writing rep invariants and checkRep methods for each class
Suppose we're maintaining a multiset of integers. We'd like to be able to:
- Add an integer to the multiset (remember we allow multiple instances of the same integer -- e.g. 1, 5, 3, 3, 3)
- Determine whether or not a given integer is in the multiset

If we represented our multiset as a list, it would be very easy to add an integer, but searching for an integer would be an \( O(n) \) operation. One way to speed up the search process is to instead store the multiset as a binary search tree (BST). Let's assume the following things:
- A BST consists of zero or more nodes (connected in a tree-like fashion)
- Each node has a VALUE (an integer) and exactly two children, LEFT and RIGHT, which are also nodes
- One or both children of a given node may be NULL
- The BST rooted at the child of node \( N \) is a SUBTREE of the BST rooted at node \( N \)
- In a BST, all the values in the left subtree of a node are less than or equal to that node's value, and all the values in the right subtree of a node are greater than that node's value
- We can represent a BST as nothing more than a reference to its root node (in the same way that we could represent a linked list as a reference to its first node)

### 25) Which of the following are valid recursive data type definitions for an integer BST?

a) \( \text{Tree} = \text{null} + \text{Node}(\text{val}: \text{int}, \text{left}: \text{Tree}, \text{right}: \text{Tree}) \)

b) \( \text{Tree} = \text{Cons}(\text{first}: \text{int}, \text{rest}: \text{Cons}(\text{first}: \text{Tree}, \text{rest}: \text{Tree})) \)

c) \( \text{Tree} = \text{null} + \text{Cons}(\text{first}: \text{int}, \text{rest}: \text{Tree}) + \text{Tree} \)

d) \( \text{Tree} = \text{null} + \text{Cons}(\text{first}: \text{Tree}, \text{rest}: \text{Cons}(\text{val}: \text{int}, \text{rest}: \text{Tree})) \)

e) \( \text{Tree} = \text{Cons}(\text{first}: \text{Tree}, \text{rest}: \text{int}) + \text{Cons}(\text{first}: \text{Tree}, \text{rest}: \text{null}) \)

f) \( \text{Tree} = \text{null} + \text{Int} + \text{Node}(\text{left}: \text{Tree}, \text{right}: \text{Tree}) \)

Consider the following minimal implementation of a BST in Java. Note that since we can treat the root node as the tree itself, we don't bother defining a Tree class here.

```java
class Node {
    public final int val;
    public final Node left;
    public final Node right;

    Node(int v, Node l, Node r) {
        val = v;
        left = l;
        right = r;
    }
}
```

We could construct the following BST \( T \):

```
2
/ \
1   3
```

```
Node T = new Node(2, new Node(1, null, null), new Node(3, null, null));
```
For each of the following functions

```java
__?__ A(Node t, int x) {
    if (x == t.val) {
        return true;
    } else if (x < t.val) {
        return (t.left == null) ? false : A(t.left, val);
    } else {
        return (t.right == null) ? false : A(t.right, val);
    }
}

__?__ B(Node t) {
    return t.val +
    (t.left == null) ? 0 : B(t.left) +
    (t.right == null) ? 0 : B(t.right);
}

__?__ C(Node t, int x) {
    if (t == null) {
        return new Node(x, null, null);
    } else if (x <= t.val) {
        return new Node(t.val, C(t.left, x), t.right);
    } else {
        return new Node(t.val, t.left, C(t.right, x));
    }
}
```

options for return types:
1. void,
2. bool,
3. int,
4. Node,
5. List<Node>

options for what the function does:
1. returns a BST with one instance of the int x removed (if the BST contains x)
2. returns a BST with one instance of the int x added to the BST
3. returns the value of the smallest int in the BST
4. returns the value of the largest int in the BST
5. returns the difference between the largest and smallest ints in the BST
6. returns the sum of all Node's values in the BST
7. swaps the left and right subtrees of every Node in the BST
8. returns true if the BST contains the int x
9. returns true if the BST contains more than one instance of the int x
<table>
<thead>
<tr>
<th>Function</th>
<th>What is the return type?</th>
<th>What does it do?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(select the correct number from the 1st list)</td>
<td>(select the correct number from the 2nd list)</td>
</tr>
<tr>
<td>Function A</td>
<td>26)</td>
<td>27)</td>
</tr>
<tr>
<td>Function B</td>
<td>28)</td>
<td>29)</td>
</tr>
<tr>
<td>Function C</td>
<td>30)</td>
<td>31)</td>
</tr>
</tbody>
</table>

Ans: A. bool, 9, observer  
B. int, 6, observer  
C. Node, 2, producer