University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2003 Instructors: Dan Garcia and Kathy Yelick 2003-09-22

☺ CS61B Quiz ☺

Personal Information

Last name	
First Name	
Student ID Number	
The name of your TA (please circle)	David Igor Ram Rishi
Name of the person to your Left	
Name of the person to your Right	
All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61B who have not taken it yet. ***(please sign)***	

Instructions

- Question 0 (-1 points if done incorrecty) involves filling in the front of this page and putting your name on every following page.
- We'll refer to the `Account` class on the back page of your exam.
- In the case of all TRUE & FALSE questions, you will be graded #right – #wrong (i.e., it may be better to leave a question blank than to circle an incorrect answer).
- You have 50 minutes to complete this quiz. The quiz is open book and open notes, no computers.
- Partial credit may be given for incomplete answers, so please write down as much of the solution as you can.
- Please turn off all pagers, cell phones and beepers. Remove all hats & headphones.

Grading Results

Question	*Max. Pts*	*Points Earned*	*Difficulty (0=easy 5=hard)*	*Fairness (0=fair 5=unfair)*
0	**0/-1**			
1	**7**			
2	**8**			
3	**10**			
Total	**25**			

Please comment above & below:

Write the difficulty and fairness ratings above and please add additional comments ← on the left here.

Question 1 : Easy Quickies… (7 points, -1 for each wrong answer… min=0)

Fill in the blanks below with the value that would be printed by the corresponding `println` statment. If the program will produce a *compile-time* (CT) or *run-time* (RT) error, fill in CT or RT. Assume previous errors have been corrected when looking at later ones (i.e., we're not intending any CT or RT errors to cascade). We've staggered the answer blanks below to give you more writing room.

a)
```
int i = 1;
int j = i;
i = 2;
System.out.println(j);                                  ➔ _____________________
```

b)
```
String s = "one";
String t = s;
s = "two";
System.out.println(t);                                  ➔ _____________________
```

c)
```
String s = "ABCDEF";
System.out.println (s.substring(1,5).substring(1,3)); ➔ _________
System.out.println ("61".concat(s.substring(s.length()-4)));    ➔ ________
```

d)
```
System.out.println("Perfect quiz: "); ➔ Perfect quiz:
System.out.println(7+8+10);                          ➔ _____________
System.out.println("Realistically: "+6+1+0);     ➔ Realistically: ________
```

e)
```
public static boolean betterEquals(String w1, String w2) {
  for (int i=0; i < w1.length(); i++) {
    if (w1.charAt(i) != w2.charAt(i)) return false;
  }
  return true;
}

// in main
String s1 = new String("61B");
String s2 = new String("61B");
System.out.println(s1 == s2);                          ➔ ________
System.out.println(s1.equals(s2));                                 ➔ ________
System.out.println(betterEquals("61B","CS61B"));     ➔ ________
System.out.println(betterEquals("CS61B","61B"));                   ➔ ________
System.out.println(betterEquals("61B","61B rocks")); ➔ ________
System.out.println(betterEquals("61B rocks","61B"));               ➔ ________
```

f)
```
private static void changeValues(int i, String s, Account a) {
   i++;
   s = "B";
   a.deposit(5);
   a = new Account(88);
}

// in main
int     score   = 9;
String  grade   = "A";
Account account = new Account(100); // …from Account class on the last page
changeValues(score, grade, account);
System.out.println(score);                             ➔ ________
System.out.println(grade);                                         ➔ ________
System.out.println(account.balance());                 ➔ ________
```

Name: _____

Question 2 : Medium quickies (8 points)

These ask you to either circle the correct answer, fill in blanks, or both.

a) You have implemented the following `Bank` class:

```
public class Bank {
  private Vector myAccounts;
  private int maxIndex;  // index of the account with the most money
  /** Creates a new Bank with the given Accounts.
   *  @arg accts                              (1)
   *  @requires accts is a Vector.            (2)
   *  @return a new Bank Object.              (3)
   */

  public Bank (Vector accts) {
    myAccounts = accts;
    maxIndex = 0;   // find richest account and save index
    for (int i = 0; i < myAccounts.size(); i++) {
      if (((Account) myAccounts.get(i)).balance() >
          ((Account) myAccounts.get(maxIndex)).balance()) {
        maxIndex = i;
      }
    }
  }

  /** Find the account with the most money in this Bank.
   *  @requires accts is a Vector             (4)
   *  @modifies myAccounts                    (5)
   *  @return myAccounts.get(maxIndex)        (6)
   */
  public Account richest() { return (Account) myAccounts.get(maxIndex); }
}
```

Your intention was to write the *strongest possible specification* for this code, to have *as few requirements on the caller* as possible. There are some problems with your specification. The table below shows each specification tag above with a list of possible problems. For 1-6, list the letters of *all problems that apply*. In the last two lines, give the names (not full specs) of any tags that are missing from the two methods. In all cases, you may have 0, 1, or more answers per blank.

(line #) TAGs	PROBLEMS
(1) `@arg` _____ (2) `@requires` _____ (3) `@return` _____ (4) `@requires` _____ (5) `@modifies` _____ (6) `@return` _____	A) The tag is named incorrectly. B) The tag contains unnecessary information. C) The tag contains information that reveals the implementation. D) The tag is missing some information. E) The tag should not be present.
(7) `Bank` is missing tags: _____ (8) `richest` is missing tags: _____	

b) The implementation has a representation invariant that `myAccounts.get(maxIndex)` has at least as much money as any other account in `myAccounts`. Your lab partner claims that your constructor provides a hole in your abstraction that a user could exploit to break the invariant. You propose to fix the problem by replacing `Vector` throughout the code by the following `FixedVector` class:

```
public class FixedVector {
    private Vector myVec;
    public FixedVector (Vector v) { myVec = v; }
    public Object get(int i) { return myVec.get(i); }
}
```

Your partner says there are still problems. Is she right? Answer Yes/No and list as many of A-F as support it. Answer: _____ Reasons: _____

No (no problems)	Yes (still problems)
A) `FixedVector` is *immutable*.	D) `richest` modifies private variables.
B) `myAccounts` is never returned.	E) `Vector` (passed to `FixedVector`) is mutable.
C) `FixedVector` makes a copy of the `Vector`.	F) `Account` is mutable

c) We would like to be able to create `Account`s starting with an initial deposit in Euros € (for this example let's say 1 Euro € = 2 US $). We could modify the `Account` class (code on the last page) by adding another constructor as so:

```
public Account (int euro) { this(2 * euro); } // 1 Euro=2$, so 2$ per euro
```

What happens when we add this constructor into our class? Circle one answer.

1) CT error because _____.
2) RT error because _____.
3) Infinite loop because the units are still `euro`, so it'll call itself (with each call doubling the input) forever.
4) It will compile and run, but it doesn't make sense to mix dollars this way.
5) It will work fine; bring on the Euros!

d) `v` is a `Vector` whose elements, if any, are all `Integer`s. Given the following code, choose the answer(s) that best fits an analysis of it. *Circle all that apply*, and fill in the blank(s) if appropriate.

```
Enumeration e = v.elements();
for (Integer i = (Integer) e.nextElement() ;
     e.hasMoreElements() ;
     i = (Integer) e.nextElement())
  System.out.println(i);
```

The program will…

1) crash for any input.
2) never crash.
3) crash only when the input is _____.
4) print all of the elements.
5) print none of the elements.
6) print all but _____ of the input.

Name: _____

Question 3 : How much money does my family have?... (10 points)

In Lab 2 you created a *parent* account (code on back page), which may itself have a *parent* account, and so on. You would like to find out the **total amount in all of these accounts** (including yours). You augment the `Account` class (on back) to add two almost-identical *recursive* methods: the no-argument, non-static `familyFortune` and the one-argument, static `familyFortuneStatic` which each return the total amount in all of your linked family's accounts added together. You also must show how we would find out (from *outside the class*) the family fortune starting with an `Account` called `me`. You may only fill in 1 statement per blank (some might be empty). Each method must be an individual solution; your non-static method may not call your static method and vice-versa.

```
public _________________ familyFortune () {                    // non-static

    if ( ____________________________________________________ ) { // base case test

        return ( ____________________________________________ );  // base case

    } else {

        return ( ____________________________________________ );  // recursive case
    }

}


// Now, show a call to this method using the Account me (from outside Account)

    _____________________________me_____________________________
```

```
public static ____________ familyFortuneStatic ( _________________ ) { // static

    if ( ____________________________________________________ ) { // base case test

        return ( ____________________________________________ );  // base case

    } else {

        return ( ____________________________________________ );  // recursive case
    }

}


// Now, show a call to this method using the Account me (from outside Account)

    _____________________________me_____________________________
```

```
/* Account.java – from the Lab 2 solution with some modifications.
 * You may detach this page from your exam. */

/** This class represents a bank account whose current balance is a
 *  non-negative amount in US dollars ($). */
public class Account {

    /** Construct an account with the given initial balance.
     */
    public Account (int balance) {
        this(balance, null);
    }

    /** Construct and account with the given balance and parent account.
     *  If the balance is negative, print and error.
     */
    public Account (int balance, Account parent) {
        if (amount < 0) {
            System.err.println("Your initial balance cannot be negative!");
        } else {
            myBalance = balance;
            myParent = parent;
        }
    }

    /** Deposit into this account. If the amount is negative, print an
     *  error and leave the balance unchanged.
     */
    public void deposit (int amount) {
        if (amount < 0) {
            System.err.println("You may only deposit non-negative sums!");
        } else {
            myBalance = myBalance + amount;
        }
    }

    /** Subtract the amount from the account, if possible. If it would leave
     *  a negative balance, print an error and leave the balance unchanged.
     */
    public void withdraw (int amount) {
        if (myBalance < amount) {
            System.err.println("Not enough funds");
        } else {
            myBalance = myBalance - amount;
        }
    }

    /** Get the balance.
     */
    public int balance ( ) {
        return myBalance;
    }

    private int myBalance;
    private Account myParent;
}
```