

CSE 8B: Sample Midterm Solution

Winter, 2006

Name: _____

Student ID: _____

No books and no calculators are allowed. One page of handwritten notes is allowed. If you need to make an assumption to solve a problem, state the assumption.

1. 9 pts. Consider the following (incomplete) class definitions:

```
public abstract class Employee {  
    public Employee() { ... }  
  
    public abstract void print();  
}  
  
public class HourlyEmployee extends Employee {  
    public HourlyEmployee() { ... }  
  
    public void print() {  
        System.out.println("hourly employee");  
    }  
}
```

Which of the following statements does *not* cause a compile-time error? (circle all that are correct)

- (a) Employee s = new HourlyEmployee();
- (b) Employee s = new Employee();
- (c) HourlyEmployee s = new Employee();

Solution: a) is correct; you can assign a subclass to a superclass. b) is incorrect; you can't instantiate an abstract class. c) is incorrect, you can't assign a superclass to a subclass.

2. 10 pts. For a well-written equals method, which of the following should be true? (circle all that are correct)

- (a) if a.equals(b), then b.equals(a)
- (b) a.equals(a) is true
- (c) If a is null, then b.equals(a) is false
- (d) If a is null, then b.equals(a) generates a NullPointerException
- (e) If b is null, then b.equals(a) generates a NullPointerException

Solution: a) is correct; that's the symmetric requirement. b) is correct; that's the reflexive requirement. c) is correct; nothing equals null. d) is incorrect; see c). (e) is correct; trying to call *any* method on a null object should generate a NullPointerException.

For the next two questions, refer to the following Money class.

```
public class Money {
    // Money represents the amount of money
    private int dollars; // 0 <= dollars
    private int cents;   // 0 <= cents <= 99
    private String currency; // "$", for example
    public Money(int d, int c, String curr) {
        dollars = d;   cents = c;   currency = curr;
    }
    public int getDollars() { return dollars; }
    public int getCents() { return cents; }
    public String getCurrency() { return currency; }
    public void setDollars(int d) {dollars = d;}
    public void setCents(int c) {cents = c;}
}
```

3. 10 pts.

Consider the following code:

```
Money[] moneyList = new Money[4];
moneyList[0] = new Money(3, 40, "$");
moneyList[1] = moneyList[0];
moneyList[0].setDollars(5);
moneyList[2] = moneyList[0];
moneyList[3] = moneyList[0];
for (int i = 0; i < 4; i++)
    System.out.print(moneyList[i].getDollars() + " ");
```

What happens when the code executes?

- (a) It prints out 5 5 5 5.
- (b) It prints out 5 3 5 5.
- (c) It prints out 5 3 3 3.
- (d) It generates a NullPointerException.
- (e) It generates a ArrayIndexOutOfBoundsException.

Solution: a) is correct. Each array element contains a reference to the same object whose dollars instance variable is set to 5.

4. 10 pts. Given the following declaration:

```
private Money m1, m2;
```

Which of the following is the best way to test whether the money objects represented by m1 and m2 have the same currency?

- (a) `m1 == m2`
- (b) `m1.equals(m2)`
- (c) `m1.getCurrency() == m2.getCurrency()`
- (d) `m1.getCurrency().equals(m2.getCurrency())`
- (e) `m2.equals(m1)`

Solution: d) is correct. Obtain the currencies, and then use equals on the currencies since they are strings. a) would check to see whether the two objects have the same reference. b) and e) would check to see whether the two objects are equal (rather than just their currencies). c) would check whether the two currencies have the same reference.

5. 6 pts. Circle all of the following that are correct:

- (a) Methods that throw RuntimeExceptions need not declare them.
- (b) If method A calls method B, and method B throws a checked exception, method A must catch it.
- (c) The catch block parameter has a type associated with it. The catch block will execute if an exception of that type, or any subclass of that type is thrown during the associated try block.
- (d) An example of a RuntimeException is ArrayIndexOutOfBoundsException.
- (e) A try block can be followed by at most one catch block.
- (f) A finally block following a try block contains code that will execute whether or not an exception occurs.

Solution: a) is correct; a RuntimeException is an unchecked exception. b) is incorrect. the method A can declare the exception instead of handling it. c) is correct. d) is correct. e) is incorrect; there can be multiple catch blocks. f) is correct; see page 559 of Savitch.

6. 8 pts. Consider the following (incomplete) class:

```
public class ColoredShape {
    private int color;

    ColoredShape() {
        super();
        color = 3;
    }

    // ...

    public boolean equals(Object o) {
        if (o == null)
            return false;
        if (getClass() != o.getClass())
            return false;
        ColoredShape s = (ColoredShape) o;
        return color == s.color;
    }
}
```

What, if anything, is wrong with the equals method (choose one)?

- (a) It will generate a runtime exception if called with an argument of null.
- (b) It will return true for two colored shapes with the same color, even if their inherited Shape data is different.
- (c) It will generate a runtime exception if called with a string argument.
- (d) There is nothing wrong with the method.

Solution: b) is correct; all the method checks is the color, it doesn't bother calling `super.equals`. a) is incorrect; the method returns false when its parameter is null. c) is incorrect; there's a check to see whether the parameter is of the same class as this. d) is incorrect; see b).

7. 10 pts. Assume we have the following method declaration:

```
public final void MyMethod() throws IOException {  
    Method2();  
    Method3();  
}
```

Which of the following statements are correct (circle all that are correct)?

- (a) MyMethod *will* throw a IOException if it is called.
- (b) MyMethod *might* throw a IOException if it is called.
- (c) If another method calls MyMethod, it must either catch IOException (or a superclass) or declare that it throws IOException (or a superclass).
- (d) Either Method2 or Method3 *must* declare that they throw an IOException
- (e) If Method2 and Method3 both do not declare that they throw anything, then MyMethod need not declare that it throws anything either.

Solution: a) is incorrect; just because an exception is declared doesn't mean it'll be thrown (it only happens on an exceptional basis). b) is correct. c) is correct; that's the catch or declare rule. d) is incorrect. It's OK for a method to declare that it throws an exception of a particular type even though it doesn't actually throw it nor call any method that throws it. e) is correct; since all the method does is call Method2 and Method3, if neither of those throw a checked exception, there's no way for MyMethod to throw a checked exception. Therefore, it need not be declared.

8. 7 pts. Given the following code:

```
int n;  
try {  
    n = 42;  
    if (n > 0)  
        throw new Exception();  
    else if (n < 0)  
        throw new NegativeNumberException();  
    else  
        System.out.println("Bingo!");  
} catch (NegativeNumberException e) {  
    System.out.println("First catch.");  
} catch (Exception e) {  
    System.out.println("Second catch.");  
} finally {  
    System.out.println("finally");  
}  
System.out.println("End of exercise");
```

What output is produced?

- (a) First catch.
End of Exercise
- (b) Second catch.
End of Exercise
- (c) First catch.
finally
End of Exercise
- (d) Second catch.
finally
End of Exercise
- (e) First catch.
- (f) Second catch.
- (g) Bingo!
End of Exercise

Solution: This problem is very similar to self-test exercise 9-19 in Savitch. d) is correct. A Exception is thrown, and caught by the second catch block. Then, the finally block is executed. Since the catch block didn't throw another exception (or rethrow the same one), execution continues after the finally block to print out End of exercise.

9. 12 pts. Which of the following are benefits of unit tests and Test-Driven Development? (circle all that are correct)

- (a) If the code passes unit tests, it's guaranteed to be correct.
- (b) Writing tests isn't put off until some later date because tests are written before the code.
- (c) Having unit tests makes it safer to refactor because if the tests still pass after refactoring, the refactoring likely didn't break anything.
- (d) Having unit tests removes the need for additional acceptance tests.
- (e) Unnecessary code isn't written because only enough code is written to pass the unit tests.
- (f) There's never a very long period with non-working code; the code is always just recently having been passing all its unit tests.

Solution: b), c), e), and f) are correct. a) is incorrect; there may be omissions in the unit tests, or bugs the unit tests didn't find. d) is incorrect; a unit test tests a particular unit (often a class), but not how units are put together into larger pieces of code. Acceptance tests verify the program as a whole works, similar to the checks we did of the output of the Simulation class.

10. 20 pts.

You're to fill in two classes that will be used for a restaurant. The Check class represents the check for a normal party at a restaurant. The LargePartyCheck represents the check for a party of 8 or more customers. The difference between the two is that large parties must pay a 15% service charge on the bill (because often large parties under-tip).

Fill in the four empty methods:

```
public class Check {
    private double totalOfItems; // stores total of items ordered
    private int checkNum; // the check number
    public Check(double total, int ckNum) {
        totalOfItems = total;
        checkNum = ckNum;
    }
    // returns the total amount of the final bill
    public double getTotal() {
        return totalOfItems;
    }
    public double getCheckNum() { return checkNum;}
}

public class LargePartyCheck extends Check {
    public LargePartyCheck(double total, int ckNum) {
        super(total, ckNum);
    }
    // returns the total amount of the final bill
    public double getTotal() {
        return 1.15 * super.getTotal();
    }
}
```

11. 18 pts. You're to fill in the following class that is to read its input from a text file named "foo" and write its output into a text file named "bar". The contents of "bar" should be the contents of "foo" with each line prepended with a line number and a colon.

For example, if "foo" contains:

```
abc
123
def
```

Then, once the program runs, "bar" should contain:

```
1: abc
2: 123
3: def
```

```
import java.io.*;
import java.util.*;
public class LineNumber {
    public static void main(String args) {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("foo");
            out = new FileOutputStream("bar");
        } catch (FileNotFoundException e) {
            System.err.println("Can't open: " + e.getMessage());
            System.exit(1);
        }
        doCopy(in, out);
    }

    static void doCopy(InputStream in, OutputStream out) {
        Scanner s = new Scanner(in);
        PrintWriter p = new PrintWriter(out);
        int count = 0;
        while (s.hasNextLine())
            p.println(++count + ": " + s.nextLine());
        s.close();
        p.close();
    }
}
```

12. 20 pts. Extra Credit. Assume you have the classes Check and LargePartyCheck from the previous exercise. Write a class CheckLog that keeps track of all checks for a day. The following behavior is needed:

- A method named addCheck that takes a check and adds it to the log object.
- A method named printSummary that prints out one line for each check in the log: the check number and the total amount of the final bill. The last line should contain the total of all the bills.
- A method named total that returns the total of all the bills

Note: you may assume that there are no more than 1,000 checks per day.

Solution:

```
public class CheckLog {
    private Check[] checks;
    private int numChecks;
    private static final int MAX_CHECKS=1000;

    public CheckLog() {
        checks = new Check[MAX_CHECKS];
        numChecks = 0;
    }

    public void addCheck(Check c) {
        checks[numChecks++] = c;
    }

    public void printSummary() {
        for (int i = 0; i < numChecks; i++)
            System.out.println(checks[i].getCheckNum() + ": "
                               + checks[i].getTotal());
        System.out.println("Total: " + total());
    }

    public double total() {
        double total = 0.0;
        for (int i = 0; i < numChecks; i++)
            total += checks[i].getTotal();
        return total;
    }
}
```