Name:_____

CSE331 Fall 2014, Final Examination
December 9, 2014
Please do not turn the page until 2:30.

Rules:

- The exam is closed-book, closed-note, etc.
- **Please stop promptly at 4:20.**
- There are **156 (not 100) points**, distributed **unevenly** among **11** questions (many with multiple parts):

Question	Max	Earned
1	16	
2	10	
3	24	
4	20	
5	8	
6	28	
7	11	
8	21	
9	4	
10	6	
11	8	

Advice:

- Read questions carefully. Understand a question before you start writing.
- **Write down thoughts and intermediate steps so you can get partial credit. But clearly indicate what is your final answer.**
- The questions are not necessarily in order of difficulty. **Skip around.** Make sure you get to all the questions.
- If you have questions, ask.
- Relax. You are here to learn.

Name:_____

1. (**16** points) Suppose we create two Java exception classes `NoSolutionExists` and `BadValueEncountered` as subclasses of the standard library's `ArithmeticException`. Also note `ArithmeticException` is a subclass of `RuntimeException`.

   Consider each of the following `@throws` specifications that could be used as part of a method specification:

   S1: `@throws NoSolutionExists`

   S2: `@throws BadValueEncountered`

   S3: `@throws ArithmeticException`

   S4: `@throws RuntimeException`

   S5: `@throws NoSolutionExists, BadValueEncountered`

   S6: `@throws NoSolutionExists, ArithmeticException`

   (a) List all the specificatons above stronger than (or equal to) S1.

   (b) List all the specificatons above stronger than (or equal to) S2.

   (c) List all the specificatons above stronger than (or equal to) S3.

   (d) List all the specificatons above stronger than (or equal to) S4.

   (e) List all the specificatons above stronger than (or equal to) S5.

   (f) List all the specificatons above stronger than (or equal to) S6.

   Solution:

   (a) S1

   (b) S2

   (c) S1, S2, S3, S5, S6

   (d) S1, S2, S3, S4, S5, S6

   (e) S1, S2, S5

   (f) S1, S2, S3, S5, S6 (note this one is "tricky" as this is the same as S3)

   Note: This is all under the interpretation of @throws E meaning "under unspecified situations, the exception might be thrown," much like Java's checked exceptions. When @throws specifications have conditions like, "E if $x < 0$" the reasoning is different. We discussed the distinction a little in class but probably should have done more. We sent a clarifying email a day or two before the exam.

Name:_____

2. (**10** points) Suppose we have a Java interface `I` with a specification.

   (a) If a class `C` implements `I`, is type `C` a Java subtype of of `I`? Answer "always," "sometimes,", or "never." No explanation required.

   (b) If a class `C` implements `I`, is type `C` a true subtype of of `I`? Answer "always," "sometimes,", or "never." No explanation required.

   Solution:

   (a) Always

   (b) Sometimes

Name:_____

3. (**24** points) (Problems continue on next page) For each problem below:

   - If the code type-checks, give the number of methods named `m` that are part of instances of `B` and briefly explain your answer.
   - Else briefly explain why the code does not type-check.

   Note `Integer` is a subtype of `Number`.

(a)
```
class A {
   Number m(Number x, Number y) { return x; }
}
class B extends A {
   Object m(Number x, Number y) { return super.m(x,y); }
}
```

(b)
```
class A {
   Number m(Number x, Number y) { return x; }
}
class B extends A {
   Integer m(Number x, Number y) { return super.m(x,y); }
}
```

(c)
```
class A {
   Number m(Number x, Number y) { return x; }
}
class B extends A {
   Integer m(Number x, Number y) { return null; }
}
```

Name:_____

(d)
```
class A {
   Number m(Number x, Number y) { return x; }
}
class B extends A {
   Object m(Number x, Number y) { return null; }
}
```

(e)
```
class A {
   Number m(Number x, Number y) { return x; }
}
class B extends A {
   Number m(Number x, Integer y) { return y; }
}
```

(f)
```
class A {
   Number m(Number x, Number y) { return x; }
}
class B extends A {
   Object m(Number x, Number y, Number z) { return y; }
}
```

Solution:

(a) Does not type-check. Because the number and types of arguments match a method in the superclass, the return type must be covariant, but `Object` is a supertype of `Number`.

(b) Does not type-check. The result of the supercall has type `Number` which is not a subtype of `Integer`, so the statement `return super.m(x,y);` does not type-check. *(This one turned out to be much trickier than we anticipated as many people did not think to check whether the method body type-checked.)*

(c) Type-checks. There is only one method `m` because this is overriding.

(d) Does not type-check for the same reason as part (a).

(e) Type-checks. There are two methods `m` because any change to number or type of arguments introduces overloading.

(f) Type-checks. There are two methods `m` because any change to number or type of arguments introduces overloading.

Name:_____

4. (**20** points) (Note this question has a part (b) on the next page.) Consider this generic class for holding two data items:

```
class Pair<T1,T2> {
   T1 x;
   T2 y;
   Pair(T1 _x, T2 _y) {
        x = _x;
        y = _y;
    }
    ...
}
```

(a) For each of the following methods, indicate whether including the method in the definition of class `Pair` would type-check. If it does not type-check, explain why not, including the first line that would not type-check and the reason it is an error. (If it does type-check, say so with no explanation necessary.)

i.
```
static <T> void swap1(Pair<T,T> p) {
   T t = p.x;
   p.x = p.y;
   p.y = t;
}
```

ii.
```
static <T1,T2> void swap2(Pair<T1,T2> p) {
    T1 t = p.x;
    p.x = p.y;
    p.y = t;
}
```

iii.
```
static void nullOut1(Pair<T1,T2> p) {
    p.x = null;
    p.y = null;
}
```

iv.
```
void swap3() {
    T1 t = x;
    x = y;
    y = t;
 }
```

v.
```
void nullOut2() {
    x = null;
    y = null;
}
```

Solution:
See next page

i. type-checks

ii. does not type-check because T1 and T2 may be different types, so the assignments p.x = p.y and p.y = t are not allowed

iii. does not type-check because the method is static so the type variables T1 and T2 are not in scope here. *(This one turned out to be much trickier than we anticipated as many people missed the omission and thought it did type-check.)*

iv. does not type-check for same reason as ii

v. type-checks

Name:_____

(b) This method does type-check:

```
static void swap4(Pair p) {
    Object t = p.x;
    p.x = p.y;
    p.y = t;
}
```

but with this warning:

```
Note: Pair.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

and these details when recompiling as directed:

```
warning: [unchecked] unchecked assignment to variable x as member
of raw type Pair
        p.x = p.y;
          ^
warning: [unchecked] unchecked assignment to variable y as member
of raw type Pair
        p.y = t;
          ^
2 warnings
```

This client of the method type-checks without warnings:

```
Pair<String,int[]> m = new Pair<String,int[]>("331 rox", new int[]{12,27,13});
Pair.swap4(m);
String s = m.x;
System.out.println(s);
```

What happens when the client code is executed? Is anything printed? If an exception occurs, what exception is it and what code causes the exception?

Solution:
Nothing is printed. A ClassCastException occurs at the assignment of `m.x` to `s` because `m.x` evaluates to an array of ints, not a String.

Name:_____

5. (**8** points) For each method below, change it so that its functionality is unchanged but it is "more generic" and therefore usable by more clients. You can describe your changes rather than rewrite the whole method, but explain *exactly* the code change(s) you would make.

(a)
```
static void addRange(List<Integer> lst, Integer low, Integer high) {
    for(int i = low; i <= high; i++) {
        lst.add(i);
    }
}
```

(b)
```
static void removeFirstInRange(List<Integer> lst, Integer low, Integer high) {
    for(int i = 0; i < lst.size(); i++) {
        int j = lst.get(i);
        if(j >= low && j <= high) {
            lst.remove(i);
            return;
        }
    }
}
```

Solution:

(a) Change `List<Integer> lst` to `List<? super Integer> lst`

(b) Change `List<Integer> lst` to `List<? extends Integer> lst`

(You can also do this without wildcards.)

This problem was fortunately worth only 8 points since many people made it much more difficult than intended. Many people tried to also change the `low` *and* `high` *variables to use bounds. This is not necessary since callers can already pass subtypes of* `Integer`*. Other people did make the code more generic by changing it to work for any* `T` *that extends* `Comparable<T>`*, modifying the bodies to use* `compareTo` *as appropriate. We didn't intend for this, but we gave full credit when done correctly.*

Name:_____

6. (**28** points) This problem has four parts as described on the next page. Put your answer to part (a) on this page (unless you need more space) and your other answers on the page after the next page (unless you need more space). All four parts refer to this code:

```
interface SalesWatcher {
    public void regularTicket(int num, int pricePerTicket);
    public void complimentaryTicket(int num);
}

public class TicketSeller {
    private int capacity;
    private int ticketsSold = 0;
    private int moneyCollected = 0;

    public TicketSeller(int capacity) {
        this.capacity = capacity;
    }
    public int getCapacity() {
        return capacity;
    }
    public int ticketsAvailable() {
        return capacity - ticketsSold;
    }
    public int totalMoneyCollected() {
        return moneyCollected;
    }
    public boolean sellTickets(int num, int pricePerTicket) {
        assert(pricePerTicket > 0);
        if(num > ticketsAvailable())
            return false;
        ticketsSold += num;
        moneyCollected += num * pricePerTicket;
        return true;
    }
    public boolean giveAwayTickets(int num) {
        if(num > ticketsAvailable())
            return false;
        ticketsSold += num;
        return true;
    }
}
```

Solution:
See same page with solutions for (b)-(d).

Name:_____

(a) Add code to the `TicketSeller` class so that it uses the Observer pattern to allow any number of implementations of `SalesWatcher` to be notified of *successful* ticket acquisitions. A successful ticket acquisition is a call to `sellTickets` or `giveAwayTickets` that returns `true`. You can add whatever code you need (fields, methods, additions to existing method bodies). Do not change the provided code. Your code should do the following:

- Have a method for adding a `SalesWatcher`.
- When tickets are successfully sold, call all observers' `regularTicket` method with the same arguments passed to `sellTickets`.
- When tickets are successfully given away, call all observers' `complimentaryTicket` method with the same arguments passed to `giveAwayTickets`.

To indicate where your additions would go you can:

- Write a circled number with a short arrow to a place in the code (between two provided lines).
- Repeat the circled number next to the code that should go in that place.

Easy to read arrows from your code to place in the provided code are also okay.

(b) Define a class `PriceChangeCounter` that:

- Implements `SalesWatcher`
- Has a method `getChangeCount` that returns how many times since the observer was added to a `TicketSeller` the price-per-ticket for a successful ticket purchase has changed (i.e., is different than the immediately previous successful ticket purchase).

Relevant details:

- Count the first observed successful ticket purchase as a change.
- Tickets given away have no effect on the price-change counting.

(c) Define a class `LargestTicketBlockTracker` that:

- Implements `SalesWatcher`
- Has a method `getMaxBlock` that returns the largest number of tickets provided at once (the first argument to `sellTickets` or `giveAwayTickets`) since the obsersver was added to a `TicketSeller`.

(d) Write client code that creates a new `TicketSeller` with a capacity of 1000 and adds two observers, one for counting price changes (part (b)) and one for tracking the largest ticket block (part (c)).

Name:_____

Place for your answers to parts (b), (c), (d)

Solution:

(a)
- We need a field for hold watchers, e.g.,
```
private List<SalesWatcher> observers = new ArrayList<SalesWatcher>();
```
and a method for adding a watcher, e.g.,
```
public void addWatcher(SalesWatcher w) {
        observers.add(w);
}
```
- `sellTickets` needs code like this anywhere after the conditional and before the return:
```
for(SalesWatcher w : observers)
     w.regularTicket(num,pricePerTicket);
```
- `giveAwayTickets` needs code like this anywhere after the conditional and before the return:
```
for(SalesWatcher w : observers)
     w.complimentaryTicket(num);
```

(b)
```
class PriceChangeCounter implements SalesWatcher {
    private int lastSalePrice = -1; // works because prices are positive
    private int changeCount = 0;
    public void regularTicket(int num, int pricePerTicket) {
        if(pricePerTicket != lastSalePrice) {
            changeCount++;
            lastSalePrice = num;
        }
    }
    public void complimentaryTicket(int num) {}
    public int getChangeCount() {
        return changeCount;
    }
}
```

(c)
```
class LargestTicketBlockTracker implements SalesWatcher {
    private int maxBlock = 0;
    public int getMaxBlock() {
        return maxBlock;
    }
    public void regularTicket(int num, int pricePerTicket) {
        if(num > maxBlock)
            maxBlock = num;
    }
    public void complimentaryTicket(int num) {
        if(num > maxBlock)
            maxBlock = num;
    }
}
```

(d)
```
TicketSeller s = new TicketSeller(1000);
s.addWatcher(new PriceChangeCounter());
s.addWatcher(new LargestTicketBlockTracker());
```

Name:_____

7. (**11** points) Using the terms *defect*, *error*, and *failure*:

   (a) In approximately 1–2 sentences, define debugging.

   (b) In approximately 2–3 sentences, explain why code with well-written assertions is easier to debug.

   Solution:

   (a) Debugging is the process that starts when a failure (a visible error) is observed and goes "backwards" to find a defect (a mistake committed by a human, often a coding error) that causes the failure.

   (b) A failed assertion is a failure. Without assertions, errors (incorrect computations) may remain invisible (not failures), leading to longer sequences of events from defect to failure. Since debugging must go from failure to defect, shorter sequences are easier to identify.

Name:_____

8. (**21** points) Short answer — *no explanations required.*

   (a) Consider defining a wrapper class. For each statement below, indicate whether the statement describes an advantage of using delegation or an advantage of using subclassing. (Just write "delegation" or "subclassing.")

      i. It is easier to compose two separately developed wrappers to get both their functionality.

      ii. You can restrict the functionality without violating Java's subtyping rules.

      iii. You do not have to (re-)implement many methods that do nothing but pass the same arguments on to a method of a different class.

   (b) For each of the following, true or false about the Visitor pattern:

      i. It is an approach to implementing the Procedure pattern.

      ii. Each operation over the data implements the same Visitor interface.

      iii. Each operation over the data implements a traversal order for itself.

      iv. A language needs overloading or the Visitor pattern is impossible to implement.

   (c) For each of the following, true or false about creational patterns:

      i. Factory patterns and the Interning pattern are different ways to overcome the same limitation of Java constructors.

      ii. The Interning pattern should be used only with immutable objects.

      iii. The purpose of the Flyweight pattern is to reduce memory usage.

      iv. The best way to implement the Factory Method pattern is to define constructors in Java interfaces instead of Java classes.

      v. Java constructors can be private.

      vi. A static field of class `C` can hold an instance of `C`.

Solution:
See next page

(a) i. delegation
ii. delegation
iii. subclassing

(b) i. true
ii. true
iii. false
iv. false

(c) i. false
ii. true
iii. true
iv. false
v. true
vi. true

Name:_____

9. (**4** points) This Java Swing code is not intended to *do* anything, but rather just to show a prototype of an interface with two buttons:

```
JFrame frame = new JFrame("CSE331 Final Exam");
frame.setLayout(new FlowLayout());
JButton button1 = new JButton("Hello");
JButton button2 = new JButton("Goodbye");
frame.add(button1);
frame.add(button2);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
```

But when you run the program, you just see this:

What line of code is missing from the code?

Solution:
`frame.pack()`, which should be either the second-to-last or third-to-last line (after adding components and before making the frame visible)

The problem was poorly worded because it did not make clear that we hoped the answer would indicate where the missing line of code should go. So we made the "where to put it" worth only 0.5 points. The most common incorrect answer was a call to `setSize`*, which is not necessary (without it, the frame is just scaled to hold the buttons, which themselves have default sizes), but the call to* `pack` *is necessary.*

Name:_____

10. (**6** points) The code below creates a Java Swing button. (Not shown is any code to place it in a container and display it.) Complete the code so that pressing the button alternates its color between purple and gold (i.e., it changes color each time the button is pressed). Here is information you may find useful:

    - `addActionListener` takes one argument of type `ActionListener`.
    - The `ActionListener` interface has one method: `public void actionPerformed(ActionEvent e)`
    - `java.awt.Component` has methods: `public void setBackground(Color c)` and `public Color getBackground()` whose purpose is hopefully intuitive.
    - `java.awt.Color` overrides `equals` appropriately.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

        final JButton button = new JButton("UW");
        final Color purple = new Color(75,0,130); // more indigo actually
        final Color gold = new Color(255,215,0);
        button.setBackground(purple);
        button.addActionListener(/* your code here */);
```

Solution:
You can also keep track of the previous color (or a count of presses or whatever) in a field of the ActionListener.

```
new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(button.getBackground().equals(purple))
            button.setBackground(gold);
        else
            button.setBackground(purple);
    }
}
```

Name:_____

11. (**8** points) For each of the following, give one of the following answers (no explanation required):

    A. It is an advantage of bottom-up implementation and not an advantage of top-down implementation.

    B. It is an advantage of top-down implementation and not an advantage of bottom-up implementation.

    C. It is an advantage of both bottom-up implementation and top-down implementation.

    D. It is an advantage of neither bottom-up implementation nor top-down implementation.

    (a) Before implementation is complete, you have a prototype you can show to customers for feedback.

    (b) You do not need to write as much "extra" code that is not part of the final product.

    (c) It often discovers performance constraints earlier in the implementation.

    (d) When testing a module Q that depends on a module R, any failing unit tests for Q must be due to a coding defect in the source code for Q.

    Solution:

    (a) B
    (b) A
    (c) A
    (d) D