Name:____

CSE331 Winter 2014, Midterm Examination
February 12, 2014

Please do not turn the page until 10:30.

Rules:

- The exam is closed-book, closed-note, etc.
- **Please stop promptly at 11:20.**
- There are **100 points** total, distributed **unevenly** among **9** questions (many with multiple parts):

Question	Max	Earned
1	11	
2	13	
3	20	
4	10	
5	20	
6	12	
7	6	
8	6	
9	2	

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 problems.
- If you have questions, ask.
- Relax. You are here to learn.

Name:____

This page does not contain a question. It has information related to many of the following questions.

A running theme on this exam is collections of U.S. coins, which we call collectively "*change*." As you probably know, U.S. coins have these values:

coin name	coin value
penny	1
nickel	5
dime	10
quarter	25

There are other obscure U.S. coins, but on this exam assume they do not exist.

In any collection of change, there is a non-negative number of each kind of coin. In general, there is no upper limit on the number of each kind of coin.

Name:____

1. (**11** points) (Don't miss that there is a part (b) below.)

   (a) The code below has a provided post-condition. For this post-condition, use backward reasoning to find the weakest precondition for each part of the code, filling in all six of the provided blanks. Assume all variables hold integers. Simplify the overall pre-condition (the top-most blank) as much as possible. Simplifying other assertions is fine but optional.

```
{                                      }
---------------------------------------

if(y > 5) {

   {                                      }
   ---------------------------------------

   x = y + 2;

   {                                      }
   ---------------------------------------

else {

   {                                      }
   ---------------------------------------

   z = z - 1;

   {                                      }
   ---------------------------------------

   x = y + z;

   {                                      }
   ---------------------------------------
}

{ x > 17 }  (This is the provided post-condition.)
----------
```

   (b) What is the largest integer n such that the following statement is true? *In* <u>*all*</u> *states satisfying the initial precondition in your answer to part (a), either* $y \geq n$ *or* $z \geq n$ *or both.*

Name:____

2. (**13** points) The code below takes an array of numbers and sums the penny and nickel values (ignoring all other values). However, it "replaces 5 pennies with a nickel" such that the final value of `p` and `n` is specified by the given post-condition.

   Fill in the provided blanks to prove this program is correct. Put the loop invariant in the blank starting "`{inv:`". You do not need to do anything other than fill in the blanks, but you might write assertions for other code points as "scratch work." As in the provided post-condition, use the notation `count X in arr[Y..Z]` to mean the number of times `X` occurs in the portion of `arr` between `Y` and `Z` that *includes* `Y` and *excludes* `Z`.

```
{ true } // (lack of) initial pre-condition
p = 0;
n = 0;
i = 0;

{                                                                      }
-----------------------------------------------------------------------


{inv:                                                                  }
----------------------------------------------------------------------

while(i != arr.length) {
   if(arr[i] == 1) {
       p = p + 1;
       if(p==5) {
           p = 0;

           {                                                                      }
           ----------------------------------------------------------------------

           n = n+1;
        }
   } else if(arr[i] == 5) {
       n = n + 1;
   } else {
       // do nothing
   }

   {                                                                        }
    ------------------------------------------------------------------------

   i = i + 1;
}
{ p < 5 /\  p + 5*n = count 1 in arr[0..arr.length] + 5 * (count 5 in arr[0..arr.length] )}
```

Name:____

3. (**20** points) Consider designing a class whose instances are collections of U.S. coins. In this problem, consider the partial implementation below where each element of the list in the `coins` field represents a single coin, so, for example, if the collection has 6 quarters and 2 dimes, then the list would have 6 elements that are an `Integer` with value 25 and 2 elements that are an `Integer` with value 10. There are no constraints on the order of elements.

```
class CoinPile {
  private List<Integer> coins;

  public CoinPile() {
      coins = new ArrayList<Integer>();
  }

  ... // many more methods for adding and removing coins, computing change, etc.
}
```

(a) Give a *class description* of abstract values implemented by `CoinPile` in terms of 4 *specification fields*. Include a simple *abstract invariant* as appropriate.

(b) Give a *representation invariant* for instances of `CoinPile`.

(c) Give an *abstraction function* for instances of `CoinPile`.

(d) Suppose the class contains this method:

```
@returns a list of coins with one coin of value n for each coin in
this with value n (i.e., the list of coins in this)
public List<Integer> getCoins() {
    return new ArrayList<Integer>(coins);
}
```

Does this method cause *representation exposure*? Explain your answer in 1–3 English sentences.

Name:____

4. (**10** points) This problem considers a *different* implementation of `CoinPile` from the previous problem. Here, the implementation simply keeps counts of each kind of coin:

```
class CoinPile {
  private int numPennies;
  private int numNickels;
  private int numDimes;
  private int numQuarters;

  public CoinPile() {
      numPennies  = 0;
      numNickels  = 0;
      numDimes    = 0;
      numQuarters = 0;
  }

  ... // many more methods for adding and removing coins, computing change, etc.
}
```

(a) Suppose we want two instances of `CoinPile` to be equal if and only if they (currently) have the same of number of pennies as each other, the same number of nickels as each other, etc. Write an appropriate `equals` method for the `CoinPile` class.

(b) Suppose:

- You also implement `hashCode` correctly with respect to the definition of `equals` in part (a).
- You then modify your program to change how `equals` behaves: now two instances are equal if and only if the total value of all the coins (i.e., the amount of money represented) is equal.

Is it possible that `hashCode` also needs to change now? If so, explain why including an example situation where not changing `hashCode` would be wrong. If not, explain why including a brief informal proof. Note you are *not* asked to implement `hashCode`.

Name:____

5. (**20** points) This problem considers a method `makeChange` that takes an `int x` and returns a `CoinPile` (as described in either of the previous two problems) where the total amount of money in the `CoinPile` equals `x`. For example, if `x` is 3, the `CoinPile` will have 3 pennies and no other coins.

   Here are several possible specifications for `makeChange`:

   A. @requires x is non-negative
   @returns a CoinPile with total amount of money equal to x

   B. @throws IllegalArgumentException if x is negative
   @returns a CoinPile with total amount of money equal to x

   C. @requires x is non-negative
   @returns a CoinPile with total amount of money equal to x and containing as few coins as possible (e.g., 1 dime instead of 2 nickels and never more than 5 pennies)

   D. @throws IllegalArgumentException if x is negative
   @returns a CoinPile with total amount of money equal to x and containing as few coins as possible (e.g., 1 dime instead of 2 nickels and never more than 5 pennies)

   E. @requires x is a non-negative multiple of 5
   @returns a CoinPile with total amount of money equal to x and containing no pennies

   (a) List all specifications above that are stronger than A.

   (b) List all specifications above that are stronger than B.

   (c) List all specifications above that are stronger than C.

   (d) List all specifications above that are stronger than D.

   (e) List all specifications above that are stronger than E.

Name:____

6. (**12** points) Suppose the `makeChange` method from the previous problem should satisfy specification D. Consider *testing* `makeChange`. Suppose the implementor expects clients to often pass arguments less than 100 (i.e., one dollar), so he/she uses a 100-element array of precomputed answers for such arguments and a slower algorithm for larger arguments.

   (a) Come up with a test-suite of five tests using a black-box methodology. Include a *brief* (probably a few words) justification of how each test is likely to test something different. You can just write the input, not the expected output. Note: there is nothing special about the number five — a real test-suite would probably be larger.

   (b) Come up with an additional three tests using a clear-box (white-box) methodology. Again briefly justify choices and again you can just write the input, not the expected output.

   Here is an example of a brief justification: *"correct answer requires two dimes"*

Name:____

7. (**6** points) Identify two things wrong with this method specification:

```
/** Rounds its argument down to the nearest multiple of 5 (so that making
change for this amount will not require pennies).  The code uses integer
division in a clever way.
@requires x is not negative
@returns the greatest multiple of 5 less than or equal to x
@throws IllegalArgumentException if x is negative
*/
int roundOutPennies(int x) {
    if(x < 0)
       throw new IllegalArgumentException();
    return 5 * (x / 5);
}
```

Name:____

8. (**6** points)

   Multiple choice related to assigned readings: Circle one answer for each question.

   (a) Which is ***not*** a good situation for using Java for-each loops:

   i. Mutating each element in an array (e.g., adding one to each element)
   ii. Nested iteration, such as iterating over all the elements in an array of lists
   iii. Printing each element of a standard-library Collection
   iv. Summing all the elements of a very large array

   (b) What ***is*** a goal of Design By Contract?

   i. To make sure you do not start writing code until all methods have full specifications that have been approved by the customer
   ii. To make sure every method has at least one assert statement for each parameter
   iii. To be able to blame a method's caller if a pre-condition is violated for some method call
   iv. To create legal responsibilities for software developers

   (c) Making defensive copies of parameters is important for:

   i. Constructors but no other methods
   ii. Some methods but no constructors
   iii. Both constructors and some other methods
   iv. Methods that take no arguments

   (d) Which is true about naming conventions (for variables, methods, classes, etc.)?

   i. The standard Java compiler can check that you obey the well-known naming conventions.
   ii. The same conventions hold for local variables, fields, and methods.
   iii. Single-letter names are good style for (generic) type parameters.
   iv. Inner classes should often contain the word "Inner" in their name.

Name:____

9. (**2** points) (Notice this is worth only 2 points, so skip it unless you have extra time.)

   Consider a class `OptimalCoinPile` for piles of change that have *as few coins as possible* for the total amount of money represented. For example, to represent 15 cents requires 1 dime and 1 nickel.

```
class OptimalCoinPile {
    private int numPennies;
    private int numNickels;
    private int numDimes;
    private int numQuarters;

    ...
}
```

   Write a `checkRep` method that checks all the necessary properties *without* using a loop.