CS180 Spring 2011 Exam 2, 6 April, 2011
Prof. Chris Clifton

Turn Off Your Cell Phone. Use of any electronic device during the test is prohibited.

Time will be tight. If you spend more than the recommended time on any question, **go on to the next one**. If you can't answer it in the recommended time, you are either going in to too much detail or the question is material you don't know well. You can skip one or two parts and still demonstrate what I believe to be an A-level understanding of the material.

For multiple choice questions, please circle the letter next to your choice.

I would hope to see an A student get at least 28 on the exam, a B student at least 21, and a C student at least 14.

1 Inheritance (8 minutes, 9 points)

Given the following classes:

```
class Person {
  public Person() {
    System.out.println("Person constructor");
  }

  public void showClassName() {
    System.out.println("Person");
  }

  public void printSomething() {
    System.out.println("In printSomething");
  }
}

class Employee extends Person {
  public Employee() {
    this("Calling this");
    System.out.println("Employee constructor");
  }

  public Employee(String s) {
    System.out.println(s);
  }

  public void showClassName() {
    System.out.println("Employee");
  }
}
```

For each of the following statements, show the output. Assume the statements are executed in the order given (note that you should be able to figure out the later ones even if you aren't sure about the earlier ones.)

Scoring: 1 point each

```
Person p1 = new Person();
```

Person constructor

```
p1.showClassName();
```

Person

```
Employee e1 = new Employee();
```

Person constructor
Calling this
Employee constructor

```
e1.showClassName();
```

Employee

```
e1.printSomething();
```

In printSomething

```
p1 = e1;
```

Nothing *(one point for stating "nothing" or for leaving it blank.)*

```
p1.showClassName();
```

Employee

```
Person e2 = new Employee("e2");
```

Person constructor
e2

```
e2.showClassName();
```

Employee

2 Generics (4 minutes, 2 points)

Given a Set class (partially shown below):

```
public class Set<E> {
  // Fields and other methods here...
  void insert(E value) {
    // Code to do insert here
  }
}
```

Using the Person and Employee classes from question 1, we do the following:

```
Set<Person> ps = new Set<Person>();
Employee e1 = new Employee();
ps.insert(e1);
```

Is the last statement (`ps.insert(e1);`) legal? (that is, will it compile and run?) Briefly explain your answer. *You may assume that Set is implemented correctly and that the first two statements compile and run.*

Yes, it will. since e1 is an Employee, and Employee extends Person, e1 can be used anyplace a Person object is needed. Since ps is a set of Person, e1 can be inserted into the set.

Scoring: 1 point for "yes", 1 for showing understanding of inheritance.

3 Recursion (4 minutes, 2 points)

Given the following class:

```
public class Recurse
{
  public static int fun(int n)
  {
    if (n <= 1)
      return 1;
    else
      return  fun(n - 1) + n ;
  }
}
```

What is returned by `Recurse.fun(3)` ?

6

Scoring: (2 points)

4 Exceptions (5 minutes, 5 points)

The sqrt function in the java Math class takes a double value as an argument, and returns the special double value "NaN" (Not a Number) if the argument is negative. We'd like to instead use a `NegativeSqrtException` exception to handle this case:

```
public class NegativeSqrtException extends Exception
{
}
```

Without using exceptions, your function looks like:

```
public static double mySqrt(double n)
{
  return Math.sqrt(n);
}
```

4.1 Proper use of exceptions (3 minutes, 2 points)

Should the `NegativeSqrtException` be thrown by `mySqrt`, or should `mySqrt` contain a try/catch block? Explain.

As a "library method", it's better to throw the exception to notify the caller of the method that he has supplied an illegal argument. Try/catching would be forced to return an ambiguous value (probably an arbitrary negative number) which still has to be interpreted.

Scoring: 1 point for correct answer, 1 point for explanation. If explanation was unclear, looked to answer to question 4.2 for clarification.

4.2 Writing the code (2 minutes, 3 points)

Write the mySqrt function that makes proper use of the `NegativeSqrtException`.

Solution:

```
public static double mySqrt(double n) throws NegativeSqrtException
{
    if (n < 0)
        throw new NegativeSqrtException();
    return Math.sqrt(n);
}
```

Scoring: -1 if the method signature does not contain "throws". -2 if the "conditional + throw" is incorrect. If previous section was incorrect, combined partial credit was given based on overall validity of the answer.

5 Gradebook (8 minutes, 5 points)

Suppose you are writing a program which implements a Gradebook class whose primary role is to hold a collection of Student objects.

5.1 Array vs. Linked List (4 minutes, 2 points)

An instructor using this class should have the ability to add students (when they sign up for the class) and remove students (if they drop the course) as necessary. Should the gradebook hold the collection of students using an array-based implementation or a linked list implementation? Justify your choice.

Either is acceptable. If you assume all students are added to the book at once and there is minimal students dropping the course, then an array-based implementation (e.g., java.util.Vector) may be preferred for easy indexing and fast iteration. If you assume the size of the gradebook is very fluid, then a linked list implementation may be preferred for fast addition/subtraction from the gradebook. It is not correct to say that the linked list implementation is resizable and the array is not (it is only easier to be resized).

Scoring: 1 point for making a choice or saying either, 1 point for a justification that correctly supports the answer given.

5.2 Superclasses/Subclasses (4 minutes, 3 points)

All students must be either an UndergradStudent or GradStudent; we would never actually create an object that was just a Student. However, we do need a Student class, since this is what the gradebook holds; UndergradStudent and GradStudent will be *subclasses* of Student.

Note that while undergrads and grad students have much in common, there are some differences. For example, a GradStudent should have a "thesisTitle" field; and UnderGrad should have a "classYear". However, both have common attributes and methods such as "transcript", "addCourseGrade()", and "computeGPA()".

Should the Student class be (choose one):

A an Interface,

B **an Abstract class,** or

C a regular class .

Briefly justify your answer

Student should be an abstract superclass. There's no point in a Student object being made, so it should be abstract. Student should be a class instead of an interface in order to declare similar fields and accessors/mutators which can be inherited.

Scoring: 0 points for interface, 1 point for regular class and noting that this allows you to share common methods, 3 points for saying abstract class and noting that this allows you share common methods (we assumed by selecting abstract class that you understand an instantiation won't be possible.

6 Linked Data Structures (20 minutes, 11 points)

```
class OrderedList {
  private int value;
  private OrderedList next;
  // Invariant:  value < next's value. That is, !next.lessThan(value);

  public OrderedList(int data, OrderedList next) {
```

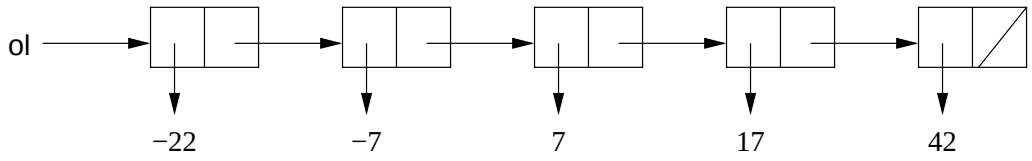
```
    value = data;
    this.next = next;
  }

  private boolean lessThan (int val) {
    // True if val < my value
    // Note that another object of the same class may use an object's private method.
    if (val < value) return true;
    else return false;
  }

  public void insert(int val) {
    // Put val at it's proper place in the list
          /* Line A (for parts 3 and 4) */
    if ( next.lessThan(val) ) {
          /* Line B (for parts 3 and 4) */
      OrderedList temp = new OrderedList(val, next);
      next = temp;
    } else {
          /* Line C (for parts 3 and 4) */
      next.insert(val);
          /* Line D (for parts 3 and 4) */
    }
          /* Line E (for parts 3 and 4) */
  }
}
```

6.1 Successful Insert (2 minutes, 5 points)

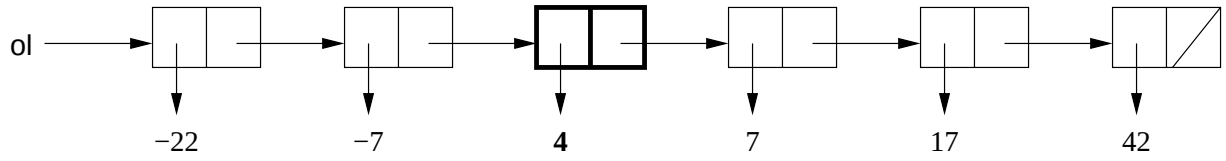
Assume `OrderedList ol;` is the following list:



Show what the above list will look like after running the statement:

```
ol.insert(4);
```

Solution:



Scoring: Minus one point for each of not having a new node, not having 4, not correctly pointing to the next node, not correctly pointing the previous to the next, not having `ol` *pointing to the right place, and not null-terminating the list. But at least one point for having one of these things right.*

6.2 Failed Insert (3 minutes, 2 points)

If you do an `ol.insert(78);` the code will fail to do what it should. This is because:

A There is no base case for `val > value`.

B **There is no base case for** `next == null`.

C There is no recursive case for `val > value`.

D There is no recursive case for `next == null`.

Scoring: 1 point for A or D, 2 points for B.

6.3 Correcting insert of large items (5 minutes, 2 points)

Fix the code so that an insert of a large item (e.g., `ol.insert(78);` will work correctly. You can do this by writing two or three lines of code that go at *A, B, C, D,* or *E* above - just give the code and state where it goes. (You can also fix it by changing one line of the code - either answer is acceptable.)

One solution (at A):

```
if ( next == null )
  next = new OrderedList(val, null);
else
```

Scoring: One point for null check, one for creating new node. Putting this at a location other than A will result in a null pointer exception in the first if (next.lessThan(val)), although you still received credit as long as everything else was right.

Another common answer was to replace the if with `if ( next == null || next.lessThan(val) ) {` . This is correct. If you put them the other way around, it would try next.lessThan(val) before checking if next==null, resulting in a null pointer exception (again, you still received full credit for getting this the wrong way, but you should remember that the order matters.)

6.4 Remaining problem (5 minutes, 2 points)

Insert will still fail to work properly in some cases. Explain briefly or give an example where it fails (1 point) and provide code to fix the problem (1 point).

If an item belongs at the beginning of the list (e.g., ol.insert(-30)), it will instead be placed after the first item. *(1 point)* **This will result in an out-of-order list.**

Fixing this is a bit tricky, as we can't change the node ol points to. Instead, we have to change the value in that node. Put the following at line A, at the beginning.

```
if ( lessThan(val) ) {  // If the new value goes in front of me,
  next = new OrderedList(value, next); // put my value in a new node,
  value = val; // And put the new value in me.
} else
```

1 point. Minor mistakes allowed, as long as you have it basically correct.

6.5 Non-Bonus question (0 points)

I claim the OrderedList class, even after fixing the insert method, is useless. Why? *This won't count for anything - don't answer this unless you are done with the exam and want to show off...*

Since we can't see what is in the list, or even check if an item is there, it won't have any value. The only public method or field is insert, which doesn't return a value or change the value of its arguments, so no matter what it does, we don't see anything. Assuming, of course, you've fixed things so it doesn't end up giving a null pointer exception.