

CSE 120

Midterm II — April 4, 2008

Name: _____

Student ID: _____
(from your PennCard)

Email _____ @seas.upenn.edu

Lab:(circle one)

201	202	203	204
W 1:00	W 2:30	R 12:00	R 1:30

- This exam text and attached code, the *Java Backpack Reference Guide*, and your brain are the only information sources you can use during this exam.
- You have 50 minutes to answer all of the questions. The entire exam is worth 50 points. The point value of each question is given to help you plan your time.
- Write your answers on the exam pages. The back side of each page may be used as a scratch pad.
- The answers to the questions require only basic Java language facilities and the code and APIs given in the appendix. No other classes or packages are needed.
- Questions during the exam should be about the wording of the exam only. If you have a question, raise your hand and we'll come to you. (This is less disruptive for others than if you come to us.)
- **Partial credit will be given where appropriate.**
- DON'T PANIC! If you find a question that you cannot solve right away, consider moving on and returning to it after you finish the other questions.

	Score	Max
1		20
2		10
3		10
4		10
Total		50

1. (20 points) Complete the following method which determines the most frequently used word in a file. On the next page, write the code that should be placed in the boxes marked (a)-(e). The reference handout includes documentation for the `Scanner` and `HashMap` classes, as well as the `Map` and `Map.Entry` interfaces. Note: you can complete this problem without documentation for the `Integer` class—Java’s autoboxing allows you to treat `Integer` values as `ints`.

```
import java.util.*;
import java.io.*;
public class MostCommon {
    public static void mostCommon(String filename) throws  {
        Scanner s = new Scanner (new File(filename));
        // Set the delimiter to be non-letters
        s.useDelimiter("[^A-Za-z]");
        Map<String, Integer> map = ;
        // skip any non-letters at the beginning of the input
        s.skip("[^A-Za-z]*");
        while () {
            // read the next word
            String word = s.next();
            // update the map
            
            // skip non-letters after the word
            s.skip("[^A-Za-z]*");
        }
        // Find the word with the largest number of occurrences
        // and store its entry in the following local variable
        Map.Entry<String, Integer> largest = null;
        
        // Print out that word
        if (largest != null) {
            System.out.println ("Most common word: " + largest.getKey() + ".");
        } else {
            System.out.println ("No words found in file!");
        }
    }
}
```

Write the code that completes the method on the previous page here. Note that some answers may take several lines.

(a) *Answer: FileNotFoundException or IOException*

(b) *Answer: new HashMap<String, Integer>()*

(c) *Answer: s.hasNext()*

(d) *Answer:*

```
if (!map.containsKey(word)) {
    map.put(word, 1);
} else {
    map.put(word, map.get(word) + 1);
}
```

(e) *Answer:*

```
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    if (largest == null)
        largest = entry;
    else if (entry.getValue() > largest.getValue())
        largest = entry;
}
```

Grading Scheme: 2 pts each for (a)-(c), 6 pts for (d), 8 pts for (e). Any exception accepted for part (a) as the reference API documentation in the handout inadvertently did not include that information.

2. (10 points)

Suppose we have defined the following class (don't worry about what these methods actually do, just concentrate on their type declarations.)

```
class P {
    void m0(Collection<Point> x) { ... }
    void m1(List<Object> x) { ... }
    void m2(List<Point> x) { ... }
    <T extends Displaceable> void m3(List<T> x) { ... }
}
```

In this problem, we will consider the addition of new methods to this class that call the above methods. In each of these new methods, you should circle all invocations that cause a *compile-time* error.

For example, the first line below is circled because the method `m0` cannot be called with argument `a`.

```
void test0 (List<Object> a, List<Point> b,
           LinkedList<Point> c, Collection<Point> d) {
    m0(a);
    m0(b);
    m0(c);
    m0(d);
}
```

Also recall that `Point` implements the `Displaceable` interface, `LinkedList<E>` implements the `List<E>` interface, and that `List<E>` is a subinterface of `Collection<E>`.

```
(a) void test1 (List<Object> a, List<Point> b,
              LinkedList<Point> c, Collection<Point> d) {
    m1(a);
    m1(b);
    m1(c);
    m1(d);
}
```

Answer: b,c,d

```
(b) void test2 (List<Object> a, List<Point> b,
              LinkedList<Point> c, Collection<Point> d) {
    m2(a);
    m2(b);
    m2(c);
    m2(d);
}
```

Answer: a,d

```
(c) void test3 (List<Object> a, List<Point> b,
              LinkedList<Point> c, Collection<Point> d) {
    m3(a);
    m3(b);
    m3(c);
    m3(d);
}
```

Answer: a,d

Grading Scheme: (a) 3 pts, (b) 3 pts, (c) 4 pts

3. (10 points) Consider the implementation of a generic linked list that was discussed in class. The beginning of this implementation is shown below.

```
import java.util.*;

class LinkedList<Element> {
    private class Node {
        public Element element;
        public Node next;

        public Node( Element element, Node n ) {
            this.element = element;
            this.next     = n;
        }
    }
    /** first element in the list */
    private Node first;

    /** Constructor */
    public LinkedList() { first = null; }
}
```

Complete the definition of the `get` method that accesses an element at a particular position in the list. If the index is equal to or greater than the size of the list, the method should throw an `IndexOutOfBoundsException`.

```
public Element get(int i) {
```

Answer:

```
    /** Returns the element at the specified position in this list. */
    public Element get(int i) {
        Node current = first;
        for (int j = 0; j < i && current != null; j++) {
            current = current.next;
        }
        if (current == null) {
            throw new IndexOutOfBoundsException();
        }
        return current.element;
    }
}
```

Grading Scheme: Looking for the following elements: check for out-of-bounds, thrown exception, initialize temporary Node reference, have a loop, loop termination, advance reference in loop, return element not Node, avoid NullPointerException

4. (10 points) Consider the class `MouseAdaptor` from the `java.awt.event` library, shown below:

```
class MouseAdaptor implements MouseListener {
    /** Invoked when the mouse has been clicked on a component. */
    public void mouseClicked(MouseEvent e) {}

    /** Invoked when the mouse enters a component. */
    public void mouseEntered(MouseEvent e) {}

    /** Invoked when the mouse exits a component. */
    public void mouseExited(MouseEvent e) {}

    /** Invoked when a mouse button has been pressed on a component. */
    public void mousePressed(MouseEvent e) {}

    /** Invoked when a mouse button has been released on a component. */
    public void mouseReleased(MouseEvent e) {}
}
```

This class trivially implements the `MouseListener` interface by defining stub methods that do nothing. Suppose you were writing a GUI paint program and you wanted to react to mouse clicks on the image to add new shapes to the drawing. In a few short sentences below, describe (i) how you would use this class in that situation and (ii) why it makes sense to use this class. Be brief and specific. However, do not worry if you do not remember the exact name of a method, we are more interested in the *process*.

Answer: (i) You would use this class by creating a subclass and overriding the `mouseClicked` method. Then to set up the mouse listener, you would create an instance of the new class and pass it as the argument to the panel's `addMouseListener` method. (ii) Creating a mouse listener in this way is convenient because you only want to react to one sort of mouse event. By using subclassing, you do not need to define the stub methods for the other events.

Grading Scheme: 10 points for mentioning subclassing and overriding and the fact that using the adaptor means that you don't have to implement all of the stubs. 8 points for getting close to the above but missing something. 5 points for discussing the event model of GUI programming coherently.