Reference information about some standard Java library classes appears on the last pages of the test. You can tear off these pages for easier reference during the exam if you like.

Question 1. (3 points) Java categorizes some exceptions as checked (like `IOException`) and others as unchecked (like `NullPointerException`). A method that might encounter checked exceptions either has to have a `try-catch` block to handle the exception or has to declare the exception in a `throws` clause in the method heading to indicate that the method might generate that exception. But neither of these are required for unchecked exceptions. Why not?

Unchecked exceptions can be generated almost anywhere in an Java program. If they were required to be declared or handled, then almost every method would have to do this, and the resulting throws clauses or exception handlers would be useless clutter, rather than providing useful information about the method's specification.

Question 2. (3 points) The `getSelectedFile` method of a `JFileChooser` object returns a `File` object. What exactly is this `File` object? A string containing a file name, a disk file, something else?

A file object is the Java library's internal representation of information about a file located on a disk, including the full path describing where it is located. It is used by the stream libraries to open streams to read and write from the actual files, and it also contains information about the file, such as its status, permissions, the directory in which it is located, and so forth.

Question 3. (6 points) Consider the following code:

```
public class Exceptional {

  public void x() {
    throw new IndexOutOfBoundsException();
  }

  public void y(int n) {
    try {
      if (n > 10) {
        x();
      } else {
        z();
      }
    } catch (IndexOutOfBoundsException e) {
      System.out.println("index out of bounds caught in y");
    } catch (Exception e) {
      System.out.println("exception caught in y");
    }
  }

  public void z() {
    throw new NullPointerException();
  }
}
```

What happens when each of the following groups of statements are executed? Indicate what output is produced or what unhandled exceptions are generated.

(a)
```
Exceptional ex = new Exceptional();
ex.z();
```

Throws and does not catch NullPointerException

(b)
```
Exceptional ex = new Exceptional();
ex.y(42);
```

Prints:
Index out of bounds caught in y

(c)
```
Exceptional ex = new Exceptional();
ex.y(5);
```

Prints:
Exception caught in y

Question 4. (16 points) One of the most important current uses of computer technology is to generate spam – messages intended to entice people to buy products. In this question, we want to explore some simple spam generating methods.

(a) (8 points) To generate "personalized" messages, we can start with a text that contains copies of a string that should be replaced by the name of the person we are sending the message to. For instance, if the message string is

```
Dear *name*, Today, *name*, you can get stuff real cheap!
```

a personalized message could be generated by replacing all occurrences of "`*name*`" with a particular name.

Complete the definition of method `personalize` below so it returns a string where all occurrences of `pattern` have been replaced by `target`. If for example, `pattern` is `*name*` and `target` is `Pat`, then replacing the pattern by the target in the above string would return the result `Dear Pat, Today, Pat, you can get stuff real cheap!`

For full credit, you must use appropriate `String` functions to search for substrings and compute the result, instead of processing the strings one character at a time. Reference information about `String` functions is included on the last pages of this test, which you can detach if that makes it easier to refer to the information.

```
/* Return a String that is a copy of text where every occurrence
 * of pattern is replaced by target */
public String personalize(String text, String pattern,
                                          String target) {


   Two versions of personalize are shown on the following page.



}
```

```
public String personalize(String text, String pattern,
                                                    String target) {
    int where;           // Position of a match to the pattern.
    String result = "";  // Personalized letter

    // Repeatedly loop looking for the pattern.  Each time it is found,
    // copy the previous text into the result along with the target, and
    // update the remaining text string to be the rest of the string
    // following the pattern.

    while ((where = text.indexOf(pattern)) != -1) {
      result = result + text.substring(0, where) + target;
      text = text.substring(where + pattern.length());
    }

    // Copy over the tail end of text.
    result = result + text;
    return result;
}

public String personalize(String text, String pattern,
                                                    String target) {
    int where;           // Position of a match to the pattern.
    int start = 0;       // Beginning of text we haven't yet looked at.
    String result = "";  // Personalized letter

    // Similar to the above solution, but does not modify the original
    // text string.  Instead, after copying each chunk of text plus the
    // target to the result, update the starting location of the next
    // search for the pattern.

    while (true) {
      where = text.indexOf(pattern, start);
      if (where == -1) break;
      result = result + text.substring(start, where) + target;
      start = where + pattern.length();
    }

    // Copy the tail end of text and return the result
    result = result + text.substring(start);
    return result;
}
```

One bug we overlooked when grading is that if you compute the result by scanning for the pattern, replacing it with the target, then rescanning the resulting string from the beginning, you will get an infinite loop if the pattern appears as part of the target string. (When the string is rescanned to look for the pattern, it is found in the part of the string that has already been processed.) This is subtle enough that we decided to let it go, and just graded solutions on how they used the string methods and the overall processing logic.

Question 4. (b) (8 points) Now we'd like to use the `personalize` method from part (a) to send messages to everyone whose name and email address appears in an input file. The input file contains lines that each have a name and an email address in the following format:

```
Sam ssmith@msn.com
Janet jj@earthlink.com
```

You can assume that there are no leading or trailing blanks on the lines, and that there is exactly one blank between the name and email address on each line.

You should also assume that there is a method you can call to send email messages with the following specification.

```
/** send the message named text to the person with the
 *   given email address */
public void sendmail(String text, String address) { ... }
```

And recall, for reference, that the `personalize` method has this specification:

```
/* Return a String that is a copy of text where every occurrence
 * of pattern is replaced by target */
public String personalize(String text, String pattern,
                                                String target) {
```

Complete the method `sendMessages` below.

```
/** Read the name/email information from stream names and
 *  send personalized versions of message to each person on
 *  that list where pattern is replaced by the person's name */
public void sendMessages(String message, String pattern,
                                          BufferedReader names) {
```

See following page.

```
}
```

```
public void sendMessages(String message, String pattern,
                                              BufferedReader names) {
    String line;
    String name;
    String addr;
    String result;
    int split;        // Dividing point between name and address

    while (true) {

      // Read a line from names.
      try {
        line = names.readLine();
      }
      catch (IOException e) {
        System.out.println("Got error while reading names file");
        return;
      }

      // Quit if no more.
      if (line == null) return;

      // Split the name and address.
      split = line.indexOf(" ");
      name = line.substring(0, split);
      addr = line.substring(split+1);

      // Personalize the message and send it
      result = personalize(message, pattern, name);
      sendmail(result, addr);
    }
}
```

Note: In a real program, we'd want to check, for instance, that the names file was properly formatted with blanks between the names and addresses. But for an exam question, it's perfectly fine to assume properly formed data.

Question 5. (8 points) One of the operations we never got around to implementing for the `SimpleArrayList` class was inserting an object at a specified position – which implies creating an opening by sliding all later elements in the list to the right. For example, if a list contains the strings

"huey" "dewey" "louie"

and we insert the string "donald" at position 1, then the resulting list should contain

"huey" "donald" "dewey" "louie"

Reminder: The instance variables for a `SimpleArrayList` are the following:

```
private Object[] items;          // items in this list are stored in
private int size;                // items[0..size-1]
```

Complete the method `add`, below, so it adds the given object at the specified position.

```
/** Add obj to the list at position pos, sliding later items
 *  to the right as needed. */
public void add(Object obj, int pos) {
    // ensure that space for a new entry is available
    ensureSpareCapacity(1);

    // add the new item at the specified location

    // Shift all items from pos to the end of the list 1 location
    // to the right.  Must be shifted from right to left.
    for(int i = size - 1; i >= pos; i--) {
      items[i+1] = items[i];
    }

    // Insert the new item and adjust the size of the list.
    items[pos] = obj;
    size++;

}
```

Note: A robust implementation of add would verify that the position was in bounds, but because the question did not specifically require this, solutions that did not verify that the position was in bounds were not penalized.