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?

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?

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();
```

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

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

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Complete the definition of method `personalize` below so it returns a string where all occurrences of `pattern` have been replaced by `target`.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) {









}
```

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) {









}
```

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









}
```

Java Reference Information

Feel free to detach these pages and use them for reference as you work on the exam.

`class` *BufferedReader*

`String readLine()`	Return next line from input stream, or `null` if no more input. Can throw `IOException`.

`class` *PrintWriter*

`void print(arg)`	Print arg to the `PrintWriter` stream. The parameter can be any type
`void println()`	Terminate the current output line and move to the beginning of the next. line
`void println(arg)`	Print `arg`, then advance to the beginning of the next line

`class` *String*

All of the search methods in class `String` return -1 if the item is not found

`int length()`	length of this string
`int indexOf(char ch)`	first position of `ch`
`int indexOf(char ch, int start)`	first position of `ch` starting from `start`
`int indexOf(String str)`	first position of `str`
`int indexOf(String str, int start)`	first position of `str` starting from `start`
`int lastIndexOf(char ch)`	last position of `ch`
`int lastIndexOf(char ch, int start)`	last position of `ch` searching backward from `start`
`int lastIndexOf(String str)`	last position of `str`
`int lastIndexOf(String str, int start)`	last position of `str` searching backward from `start`
`String substring(int start)`	substring of this string from position `start` to end
`String substring(int start, end)`	substring of this string from `start` to `end-1`
`String trim()`	copy of this string with leading and trailing whitespace deleted

All ***Collection*** interfaces (***List***, ***Set***) and classes (***ArrayList***, ***LinkedList***, ***HashSet***, ***TreeSet***)

```
boolean add(Object obj)
boolean addAll(Collection other)
void clear()
boolean contains(Object obj)
Iterator iterator()
boolean remove(Object obj)
int size()
Object[] toArray()   // return an array containing all the
                     // elements in this collection
```

In addition, all `Collection` classes provide a constructor that takes another `Collection` as a parameter and creates a new collection whose initial contents are copied from that parameter. (i.e., `public ArrayList(Collection c)`, and similarly for the other classes.)

Additional methods in ***List***, ***ArrayList***, ***LinkedList***

```
add(int position, Object obj)
remove(int position)
```

Map, ***HashMap***, ***TreeMap***

```
Object put(Object key, Object value)
Object get(Object key)
Object remove(Object key)
Set keySet()
Collection values()
int size()
```

arrays

If `a` is a Java array, `a.length` is the number of elements in that array.

If `m` is a 2-dimensional Java array, `m[k]` refers to row `k` of the array, and `m[k].length` is the length of that row (which is the same for all rows in a normal, rectangular array).

Exceptions

Some standard exceptions that might be useful: `IllegalArgumentException`, `IndexOutOfBoundsException`, `NoSuchElementException`, `NullPointerException`