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. (2 points) When a method is called we often want to check that its precondition is met before doing anything else. One way to handle this is to use an `assert` statement; another way is to throw an exception. When would it be appropriate to throw an exception instead of using `assert`?

Throwing an exception would be appropriate if we're checking the precondition of a method that is part of the external interface used by client code that we did not write. An `assert` is appropriate when we're checking for internal errors.

Question 2. (2 points) After we are done using a stream, the stream should be closed, either automatically when the program finishes (i.e., method `main` terminates), or explicitly by executing the `close()` method on the stream. What is the purpose of closing a stream – i.e., what does the close operation do?

Closing a stream breaks the connection between the external stream and the program, and frees up any resources that have been allocated to maintain the connection. (It also means that any other programs that need exclusive access to the stream, say to write to it, can proceed.)

The next question involves the following code, which you can remove from the test for reference if you wish.

```
public class TestQuestion {

  public void a(int n) {
    System.out.println("start a");
    b(n);
    System.out.println("end a");
  }

  public void b(int n) {
    System.out.println("start b");
    try {
      c(n);
    } catch (NullPointerException e) {
      System.out.println("Caught NullPointerException in b");
      if (n%2 == 0) {
        throw new IndexOutOfBoundsException("from b");
      }
    }
    System.out.println("end b");
  }

  public void c(int n) {
    System.out.println("start c");
    if (n > 10) {
      throw new NullPointerException("from c");
    }
    System.out.println("end c");
  }

}
```

Question 3. (8 points) What happens when each of the following sets of statements are executed using the code in the class on the previous page? You should indicate what output is produced and show where exceptions are thrown and caught, if this happens.

(a)
```
TestQuestion q = new TestQuestion();
q.a(10);
```

```
start a
start b
start c
end c
end b
end a
```

(b)
```
TestQuestion q = new TestQuestion();
q.a(12);
```

```
start a
start b
start c
Caught NullPointerException in b
IndexOutOfBoundsException: from b
```

Question 4. (2 points) The `iterator()` method in collection classes like our `SimpleArrayList` returns a new instance of a class that implements `Iterator`, which includes methods `hasNext()` and `next()`. Why is an iterator usually implemented as a separate class, instead of just including methods `hasNext()` and `next()` as methods in the collection itself? In other words, what advantage, if any, is there to having an iterator as a separate object?

By having iterators be separate objects that are instances of a class, it is possible to have more than one active iteration processing a collection at the same time.

Question 5. (10 points) (a) (8 points) We would like to add a method to our `SimpleArrayList` class to remove an item at a given position from the list. More specifically, complete the definition of the method `remove`, below, so that

```
   remove(int pos)
```

removes the list element at the given position `pos`, or throws an `IndexOutOfBoundsException` if `pos` is negative or greater than or equal to the current size of the list. Recall that the state of a `SimpleArrayList` is represented as follows:

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

After removing the requested item, the list should be properly adjusted to reflect the class invariant in the comments above, and the `items` array should not contain any extra references to objects other than the ones necessary to represent the contents of the list.

```
   public void remove(int pos) {
      if (pos < 0 || pos >= size) {
         throw new IndexOutOfBoundsException();
      }
      // move items[pos+1..size-1] one position to the left
      for (int k = pos; k < size-1; k++) {
         items[k] = items[k+1];
      }
      // null out extra reference at the end and adjust size
      items[size-1] = null;
      size--;
   }
```

Note: It's also ok to use the `size()` function to reference the list size instead of the `size` instance variable.

(b) (2 points) Do we need to add "`throws IndexOutOfBoundsException`" to the heading of this method to get it to compile? Why or why not?

`IndexOutOfBoundsException` is an unchecked exception, so it does not need to be declared in a `throws` clause.

Question 6. (18 points) In this question we would like to read a file containing definitions of words and then be able to find all the words whose definitions *contain* a particular word. The format of the input file consists of sequences of words followed by their definitions, separated by blank lines. Here is an example of a sample input file containing three definitions.

```
frog
a frog is a smooth or
slimy creature with two
bulging eyes and webbed
hind feet

toad
a kind of frog with a stubby body and
short hind legs; generally with warts
and dry skin

newt
an amphibian that is sometimes interested
in politics
```

(a) (10 points) Complete the definition of method `readDictionary` on the next page so that it reads the contents of the `BufferedReader` stream `in` one line at a time (using the `readLine()` method), and stores the words and their definitions in the HashMap `dictionary`. Your code should store each definition as a single long `String` with one extra blank between each line from the original input file, and with no other characters like newlines separating the input lines.

You should make the following assumptions to simplify the task:

- There are no blank or empty lines before the first word/definition pair in the file,
- Every definition contains at least one line,
- Every definition, including the last one, is followed by exactly one line containing a zero-character string (“”), and
- No word is defined more than once in the input file.

If an exception occurs while the file is being read, you should print an appropriate message and stop.

[Turn to the next page to write your answer. Also, remember there is some reference information on the last pages of the test that you might find useful.]

Question 6(a). (10 points)

```
/** Read words and definitions from stream in and store
 *  the definitions in the map dictionary using the
 *  words as the keys */
public void readDictionary(BufferedReader in,
                           HashMap dictionary) {

   try {
      String word = in.readLine();
      while (word != null) {
         // process next word and definition
         String definition = in.readLine();
         String nextLine = in.readLine();
         while (nextLine.length() > 0 {
            definition = definition + " " + nextLine;
            nextLine = in.readLine();
         }
         // add to dictionary and skip past "" after definition
         dictionary.put(word, definition);
         word = in.readLine();
      }
   } catch (IOException e) {
      System.out.println("Exception while reading file");
   }
}
```

In the inner while loop, another way to check for the end of a definition would be to compare the input to a zero-length string:

```
while (!nextLine.equals("")) { … }
```

Question 6 (b) (8 points) Complete the definition of the following method so it prints to `System.out` the words (keys) in the `HashMap dictionary` whose *definition* (not key) contains the string `target`. So, for example, if we execute `printWords("frog", dictionary)`, where dictionary has been initialized with the definitions given at the beginning of this question, the method should print the words "`frog`" and "`toad`", since the word "`frog`" occurs in the definitions of those two words. Your solution only needs to detect exact matches, i.e., if the word is "frog", you only need to find definitions that contain "frog". You can ignore definitions that contain "Frog", "FROG", and similar words as long if they don't contain the string "frog".

```
/** Print on System.out all words (keys) in dictionary
 *  whose definitions contain the string target */
public void printWords(String target, HashMap dictionary) {

   Set keys = dictionary.keySet();

   Iterator it = keys.iterator();

   while (it.hasNext()) {

      String word = (String)it.next();

      String definition = (String)dictionary.get(word);

      if (definition.indexOf(target) != -1) {

         System.out.println(word);

      }

   }

}
```

Note that the `(String)` cast preceding `it.next()` is not needed if you use `Object` as the type of the item returned by the iterator. The second `(String)` cast on the result of `get()` is necessary in order to process the result as a string.