Exercise 12.4

The use of many different utility classes is actually quite good. The alternative would have been to write our own similar functionality, and that would be a waste when it is already there.

A HashMap does not have the method *tailMap*() which is used in the *search*() method, so we can not replace the TreeMap with a HashMap.

Exercise 12.5

Add the following to the run method of AddressBookTextInterface:

```
            else if(command.equals("get")) {
                get();
            }
            else if(command.equals("remove")) {
                remove();
            }
```

and implementations of the get and remove methods:

```
    private void get()
    {
        System.out.print("Key: ");
        String key = parser.readLine();
        ContactDetails details = book.getDetails(key);

        if(details != null) {
            System.out.println(details);
        }
        else {
            System.out.println("No details matching " + key + " were
found.");
        }
    }

    private void remove()
    {
        System.out.print("Key: ");
        String key = parser.readLine();
        book.removeDetails(key);
        System.out.println("Details matching " + key + " were
removed.");
    }
```

Exercise 12.6

It would be more appropiate to calculate the value. If we calculate the value, we don't have to remember to increase and decrease the value.

One example where it fails is if we accidently put in the same contact details twice, then it would incorrectly increment the numberOfEntries twice. Another is if a set of

details is only entered under a single key.

Exercise 12.7

It throws an error:

java.lang.NullPointerException
at AddressBook.removeDetails

Exercise 12.8

If software controlling an aircraft's systems simply crashed it may well leave the pilots with no control over the aircraft.

Exercise 12.10

The modified method:

```
    public void removeDetails(String key)
    {
        ContactDetails details = book.get(key);
        if(details != null) {
            book.remove(details.getName());
            book.remove(details.getPhone());
            numberOfEntries--;
        }
    }
```

Exercise 12.11

It depends... but it might be a good idea to report the error, as it is likely that the client of the removeDetails actually thought that the key was there.

It could be reported with a return value which could be checked by the client.

Exercise 12.12

See section 12.2.2 in the book.

Exercise 12.13

getDetails and *keyInUse* does not result in any errors no matter what arguments we give them. The results returned are able to indicate unambiguously both success and failure.

Exercise 12.14

Amongst other things, if there is no user interface then printed error messages are pointless.

Exercise 12.18

Yes, the output should be different. The graphical version should pop up a window and the text based version should print out a string.

Exercise 12.19

Yes, at least one of the arguments *name* and *phone* should contain a valid string. If both of them are null or empty there is no key available to look up the address. If the name and phone are identical strings then this might result in implementation errors.

Exercise 12.20

No. A search for something should be expected not to find any matches.

Exercise 12.21

A constructor can not return any values to indicate an error. One way it could be solved is to use *unchecked exceptions* which is discussed in the next section of the book.

Exercise 12.22

CharConversionException
EOFException
FileNotFoundException
InterruptedIOException
InvalidClassException
InvalidObjectException
IOException
NotActiveException
NotSerializableException
ObjectStreamException
OptionalDataException
StreamCorruptedException
SyncFailedException
UnsupportedEncodingException
UTFDataFormatException
WriteAbortedException

Exercise 12.23

SecurityException is an unchecked exception.

Exercise 12.24

See the project address-book-v3t.

Exercise 12.26

The revised removeDetails method:

```
    /**
     * Remove the entry with the given key from the address book.
     * The key should be one that is currently in use.
     * @param key One of the keys of the entry to be removed.
     * @throws IllegalArgumentException If the key is null.
     * @throws NoMatchingDetailsException If the key does not match
     *         any details in the address book.
     */
    public void removeDetails(String key) throws
NoMatchingDetailsException
    {
        if(key == null){
            throw new IllegalArgumentException("Null key passed to
removeDetails.");
        }
        if(keyInUse(key)) {
            ContactDetails details = book.get(key);
            book.remove(details.getName());
            book.remove(details.getPhone());
            numberOfEntries--;
        }
        else {
            throw new NoMatchingDetailsException(key);
        }
    }
```

Note that the changeDetails method will also need altering because it calls removeDetails which throws a *checked* exception.

Exercise 12.27

The remove method from AddressBookTextInterface:

```
    /**
     * Remove an entry matching a key.
     */
    private void remove()
    {
```

```
        System.out.println("Type the key of the entry.");
        String key = parser.readLine();
        try {
            book.removeDetails(key);
        }
        catch(NoMatchingDetailsException e) {
            System.out.println("No details were found matching " +
e.getKey());
        }
    }
```

Exercise 12.28

This is bad for at least three reasons:

1. It is catching all Exceptions (even unchecked exceptions are caught). It should be more specific.
2. It does nothing in the exception handling block to either alert to the error or correct it.
3. The actions after the try statement assume that the lookup worked, regardless of whether it did or not.

Exercise 12.30

All exceptions are caught by the first catch because Exception is a supertype of all other exception types, so it can never reach the error handling for unchecked exceptions (RuntimeException).

Exercise 12.34

`testForAdditionError()` results in an assertion failure. Note that the BlueJ environment must be configured to enable assertions at runtime. See the **enableassertions** option in the Java tools documentation and Appendix A for how to configure BlueJ.

Exercise 12.35

Yes, it should have a consistency check because we might change something into a key that already exists, and thereby decreasing the count.

Exercise 12.36

It fails at the consistentSize assertion in the removeDetails because it does not remove entry which uses the address as the key.

Exercise 12.37

If you could change the phone number of a ContactDetails the phone number used as key in the book is no longer correct. This could result in problems in the methods that uses the phone number as the key.

Exercise 12.38

```
    /**
     * Write the notebook to the given file.
     * @param filename Where to write the notes.
     * @return true on success, false on failure.
     */
    public boolean writeToFile(String filename)
    {
        // Whether the write succeeded or not.
        boolean ok;
        try {
            FileWriter writer = new FileWriter(filename);
            for(String note : notes) {
                writer.write(note);
                writer.write('\n');
            }
            writer.close();
            ok = true;
        }
        catch(IOException e) {
            ok = false;
        }
        return ok;
    }
```

Exercise 12.40

```
    /**
     * Read notes from the given file.
     * @param filename Where to read the notes from.
     * @return true on success, false on failure.
     */
    public boolean readFromFile(String filename)
    {
        // Whether the read succeeded or not.
        boolean ok;
        try {
            BufferedReader reader =
                new BufferedReader(new FileReader(filename));
            String line = reader.readLine();
            while(line != null) {
                notes.add(line);
                line = reader.readLine();
            }
            reader.close();
            ok = true;
        }
        catch(FileNotFoundException e) {
            ok = false;
        }
```

```
        catch(IOException e) {
            ok = false;
        }
        return ok;
    }
```

Exercise 12.43

You can use the method *isDirectory()* on the File object.

Exercise 12.44

Not much is available about the contents of a file through the File obejct. However, you can get the size of the file in bytes via the method *length().*

Exercise 12.45

Add the following:

```
        File fileDetails = new File(filename);
        if(fileDetails.exists() && fileDetails.canRead() &&
fileDetails.length() > 0) {
```

Passing these checks makes no difference to the need to deal with exceptions.

Exercise 12.47

```
 BigDecimal    nextBigDecimal()
          Scans the next token of the input as a BigDecimal.
 BigInteger    nextBigInteger()
          Scans the next token of the input as a BigInteger.
 BigInteger    nextBigInteger(int radix)
          Scans the next token of the input as a BigInteger.
 boolean       nextBoolean()
          Scans the next token of the input into a boolean value and
returns that value.
 byte   nextByte()
          Scans the next token of the input as a byte.
 byte   nextByte(int radix)
          Scans the next token of the input as a byte.
 double nextDouble()
          Scans the next token of the input as a double.
 float  nextFloat()
          Scans the next token of the input as a float.
 int    nextInt()
          Scans the next token of the input as an int.
 int    nextInt(int radix)
          Scans the next token of the input as an int.
 String nextLine()
```

```
          Advances this scanner past the current line and returns the
input that was skipped.
 long   nextLong()
          Scans the next token of the input as a long.
 long   nextLong(int radix)
          Scans the next token of the input as a long.
 short  nextShort()
          Scans the next token of the input as a short.
 short  nextShort(int radix)
```

Exercise 12.49

If you get the input from a source that does not give you back a stream, channel or file, a String will in most cases be available. For instance if you get the text out of a graphical Swing component (like JTextField and JTextField). A Scanner is able to parse this text in the same way that it can the contents of a file.

Exercise 12.50

Both String and ArrayList are already serializable, so it is only necessary to add `implements Serializable` to the class header of Notebook.

Exercise 12.52

As serialized objects may be inconsistent with a new version of a class an InvalidClassException may be thrown.