Handling errors

Objectives

Bad Things^{TM} happen, eg,

- programming logic errors,
- sloppy programming,
- an object might be used is some way unanticipated by the class developer, eg, an upcast from Square to Rectangle,

and we need to write code to somehow minimize their effect. Even well-tested programs can fail, eg, when some essential resource is missing.

Main concepts discussed in this chapter

- defensive programming
- error reporting
- exception throwing and handling
- simple file processing

Background

Resources

- Classes needed for this lab - *chapter12.jar*.

To do

The address-book project

We will start with a not particularly good address book application for storing personal contact details: name, address, phone number. Contact details are indexed by name and by phone number.

Contacts are stored in the `AddressBook` class of the *address-book-v1t* project:

```
package addressbook.v1t;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.SortedMap;
```

```
import java.util.TreeMap;
import java.util.TreeSet;

/**
 * A class to maintain an arbitrary number of contact details. Details are
 * indexed by both name and phone number.
 *
 * @author David J. Barnes and Michael Kolling.
 * @version 2008.03.30
 */
public class AddressBook {
    // Storage for an arbitrary number of details.
    private TreeMap<String, ContactDetails> book;
    private int numberOfEntries;

    /**
     * Perform any initialization for the address book.
     */
    public AddressBook() {
        book = new TreeMap<String, ContactDetails>();
        numberOfEntries = 0;
    }

    /**
     * Look up a name or phone number and return the corresponding contact
     * details.
     *
     * @param key
     *            The name or number to be looked up.
     * @return The details corresponding to the key.
     */
    public ContactDetails getDetails(String key) {
        return book.get(key);
    }

    /**
     * Return whether or not the current key is in use.
     *
     * @param key
     *            The name or number to be looked up.
     * @return true if the key is in use, false otherwise.
     */
    public boolean keyInUse(String key) {
        return book.containsKey(key);
    }

    /**
     * Add a new set of details to the notebook.
     *
     * @param details
     *            The details to associate with the person.
     */
    public void addDetails(ContactDetails details) {
        book.put(details.getName(), details);
        book.put(details.getPhone(), details);
        numberOfEntries++;
    }
```

```
/**
 * Change the details previously stored under the given key.
 *
 * @param oldKey
 *            One of the keys used to store the details.
 * @param details
 *            The replacement details.
 */
public void changeDetails(String oldKey, ContactDetails details) {
    removeDetails(oldKey);
    addDetails(details);
}

/**
 * Search for all details stored under a key that starts with the given
 * prefix.
 *
 * @param keyPrefix
 *            The key prefix to search on.
 * @return An array of those details that have been found.
 */
public ContactDetails[] search(String keyPrefix) {
    // Build a list of the matches.
    List<ContactDetails> matches = new LinkedList<ContactDetails>();
    // Find keys that are equal-to or greater-than the prefix.
    SortedMap<String, ContactDetails> tail = book.tailMap(keyPrefix);
    Iterator<String> it = tail.keySet().iterator();
    // Stop when we find a mismatch.
    boolean endOfSearch = false;
    while (!endOfSearch && it.hasNext()) {
        String key = it.next();
        if (key.startsWith(keyPrefix)) {
            matches.add(book.get(key));
        } else {
            endOfSearch = true;
        }
    }
    ContactDetails[] results = new ContactDetails[matches.size()];
    matches.toArray(results);
    return results;
}

/**
 * @return The number of entries currently in the address book.
 */
public int getNumberOfEntries() {
    return numberOfEntries;
}

/**
 * Remove an entry with the given key from the address book.
 *
 * @param key
 *            One of the keys of the entry to be removed.
 */
public void removeDetails(String key) {
    ContactDetails details = book.get(key);
    book.remove(details.getName());
```

```
        book.remove(details.getPhone());
        numberOfEntries--;
    }

    /**
     * @return All the contact details, sorted according to the sort order of
     *         the ContactDetails class.
     */
    public String listDetails() {
        // Because each entry is stored under two keys, it is
        // necessary to build a set of the ContactDetails. This
        // eliminates duplicates.
        StringBuffer allEntries = new StringBuffer();
        Set<ContactDetails> sortedDetails = new TreeSet<ContactDetails>(book
                .values());
        for (ContactDetails details : sortedDetails) {
            allEntries.append(details);
            allEntries.append('\n');
            allEntries.append('\n');
        }
        return allEntries.toString();
    }
}
```

New contact details are added by the `addDetails` method. Details can be retrieved from the address book either by indexing on the name or the phone number, or by searching.

Exercises

1. Open the *address-book-v1t* project, create the usual `Main` class to run everything. Have its `main` method create an `AddressBookDemo` object and call its `showInterface` method.
2. Examine the implementation of the `AddressBook` class and assess whether you think it has been well written or not. Do you have any specific criticisms of it?
3. The `AddressBook` class uses quite a lot of classes from the `java.util` package; if you are not familiar with any of these, check the API documentation to fill in the gaps. Do you think the use of so many utility classes is justified? Could a `HashMap` have been used in place of the `TreeMap`?
4. Modify the `CommandWords` and the `AddressBookTextInterface` classes of the *address-book-v1t* project to provide interactive access to the `getDetails` and `removeDetails` methods of `AddressBook`.
5. The `AddressBook` class defines an attribute to record the number of entries. Do you think it would be more appropriate to calculate this value, as required, from the number of unique entries in the `TreeMap`? For instance, can you think of any circumstances in which the following calculation would not produce the same value?

```
return book.size() / 2;
```

Defensive programming

Client-server interaction

An `AddressBook` object responds to requests - it acts like a *server*. The `AddressBook` class developer can assume one of two strategies regarding any client:

- clients are well-behaved - requests sent by clients will always be reasonable; or
- the server operates in a hostile environment.

Depending on the approach, the server developer must decide:

- How much checking should a server's methods perform on client requests?
- How should a server report errors to its clients?
- How can a client anticipate failure of a request to a server?
- How should a client deal with failure of a request?

This version of `AddressBook` trusts its clients completely.

Exercises

7. Using the *address-book-v1t* project `Main` class, create a new `AddressBook` object. This will be initially completely empty of contact details. Now make a call to its `removeDetails` method with any string value for the key. What happens? Why?
8. For a programmer, the easiest response to an error situation arising is to allow the program to terminate, ie, to crash. Can you think of any situations in which just allowing a program to terminate could be very dangerous?
9. Many commercially sold programs contain errors that are not handled properly in the software and cause the program to crash. Is that unavoidable? Is it acceptable? Discuss.

The `removeDetails` method assumes that the key passed as an argument is valid:

```
ContactDetails details = book.get(key);
book.remove(details.getName());
```

If the key is not valid however, then `details` is `null` and the attempt to get its name leads to an exception and a program crash.

Who is to blame?

- The client for using an invalid key?
- The server for assuming that the key is valid without checking?

Exercises

10. Save a copy of the *address-book-v1* projects under another name to work on. Make changes to the `removeDetails` method to avoid a `NullPointerException` from arising if the key value does not have a corresponding entry in the address book. If the key is not valid then the method should do nothing.
11. Is it necessary to report the use of an invalid key in a call to `removeDetails`? Is so, how would you report it?

12. Are there any other methods in the `AddressBook` class that are vulnerable to similar errors? If so, try to correct them in your copy of the project. Is it possible in all cases for the method simply to do nothing if its arguments are inappropriate? Do the errors need reporting in some way? If so, how would you do it, and would it be the same for each error?

Argument checking

Methods are vulnerable to receiving bad arguments. If there is any possibility that an argument might be bad, then the method must check the arguments itself. Otherwise, using bad arguments can lead to crashes.

To protect itself against `NullPointerExceptions`, `removeDetails` can check its argument:

```
/**
 * Remove the entry with the given key from the address book.
 * The key should be one that is currently in use.
 * If the key does not exist, do nothing.
 * @param key One of the keys of the entry to be removed.
 */
public void removeDetails(String key)
{
      if(keyInUse(key)) {
            ContactDetails details = (ContactDetails) book.get(key);
            book.remove(details.getName());
            book.remove(details.getPhone());
            numberOfEntries--;
      }
}
```

The new behaviour of the method is documented.

Other methods that are vulnerable to bad arguments:

- `addDetails` should check for `null`.
- `changeDetails` should check that the old key exists, and that the new details are not `null`.
- `search` should check for `null`.

Exercises

13. Why do you think we have felt it unnecessary to make similar changes to the and to the `getDetailskeyInUse` methods?
14. In dealing with argument errors, we have not printed any error messages. Do you think an `AddressBook` *should* print an error message whenever it receives a bad argument to one of its methods? Are there any situations where a printed error message would be appropriate?
15. Are there any further checks you feel we should make on the arguments of other methods, to prevent an `AddressBook` object from functioning incorrectly?

Server error reporting

If the server receives a bad argument, it's often because of some error in the client. Instead of just protecting itself, the server should also inform the client of the problem so that it can be corrected before it arises again. How can the server inform the client?

Exercise

16. How many different ways can you think of to indicate that a method has received incorrect parameter values, or is otherwise unable to complete its task?

Notifying the user

An obvious approach would be to inform the application's user by printing a message or popping a graphical error-message window. However,

- What if the application runs automatically, without any human intervention? Nobody's looking!
- What if a user *is* looking, but can't do anything about the error? Eg, an ATM (*automatic teller machine*, or DAB - *distributeur automatic de billets*) printing a `NullPointerException`??!!?

Notifying the client object

There are two main ways for a server to inform the client of a problem:

- Return a specific return value to *flag* the error.
- *Throw an exception*, see below.

Throwing an exception is more radical as it forces the client to take possible errors into account.

We can change `removeDetails` to signal a problem

```
/**
 * 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.
 * @return true if the entry was successfully removed, false otherwise.
 */
public boolean removeDetails(String key)
      {
      if(keyInUse(key)) {
            ContactDetails details = (ContactDetails) book.get(key);
            book.remove(details.getName());
            book.remove(details.getPhone());
            numberOfEntries--;
            return true;
      } else {
            return false;
      }
}
```

This could be used as follows

```
if (addresses.removeDetails("...")) {
```

```
        // entry successfully removed, continue as normal
    } else {
        // removal failed, attempt recovery or whatever
    }
```

If the method normally returns an object, then returning `null` could signal a problem, eg,

```
    /**
     * Look up a name or phone number and return the
     * corresponding contact details.
     * @param key The name or number to be looked up.
     * @return The details corresponding to the key, or
     * null if the key is not in use
     */
    public ContactDetails getDetails(String key)
    {
        if (keyInUse(key)) {
            return (ContactDetails) book.get(key);
         } else {
            return null;
         }
    }
```

This could be used as follows

```
    ContactDetails details = addresses.getDetails("Fred");
    if (details != null)
        // send a text message

        String phone = details.getPhone();

        ...
    } else {
        // removal failed, attempt recovery or panic
    }
```

Exercises

17. Using a copy of the *address-book-v2t* project, make changes to the `AddressBook` class, where appropriate, to provide failure information to a client when a method has received incorrect parameter values, or is otherwise unable to complete its task.
18. Are there any combinations of argument values that you think would be inappropriate to pass to the constructor of the `ContactDetails` class?
19. Do you think that a call to the `search` method that finds no matches requires an error notification? Justify your answer.
20. Does a constructor have any means to signal to a client that it cannot set up the new object's state correctly? What should a constructor do if it receives inappropriate arguments?

This strategy has some problems.

- It's not always possible to use some return values as error flags, eg, when the whole range of values is normal.
- The client can completely ignore the error flags it receives.

- It might not be possible to differentiate between an unsuccessful request (which can be perfectly normal) and an incorrect request.

There is a better way to signal errors - by *throwing exceptions*.

Exception-throwing principles

There are advantages to throwing exceptions rather than returning error codes:

- They don't interfere with the method's normal return values.
- They are hard for a client to ignore.
- They lead to much cleaner code, where the *normal code* is separated from the *error-handling stuff*.

Throwing an exception

The following code shows how to throw an exception if the `key` argument is `null`

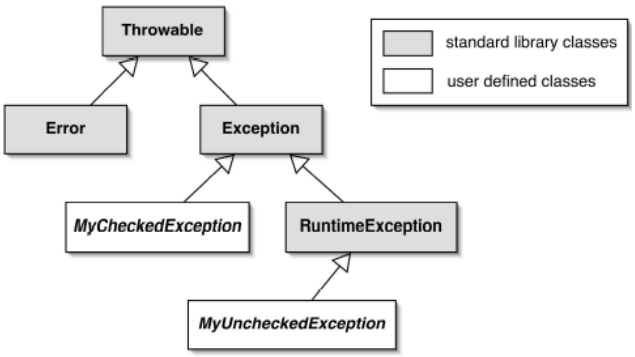
```
/**
 * Look up a name or phone number and return the
 * corresponding contact details.
 * @param key The name or number to be looked up.
 * @return The details corresponding to the key, or
 * null if there are none matching.
 * @throws NullPointerException if the key is null.
 */
public ContactDetails getDetails(String key)
{
      if (key == null) {
            throw new NullPointerException("null key in getDetails");
      }
      return (ContactDetails) book.get(key);
}
```

The exception object is first created, then thrown, usually in a single statement

```
throw new ExceptionType("optional diagnostic string");
```

Exception classes

If Java's exception hierarchy seems overly-complicated, that's because it is - for "historical reasons"!



We're mainly interested in exceptions that derive from the `Exception` class. Exceptions that derive from `RuntimeException` are called *unchecked exceptions* and generally indicate programming errors, like trying to access a `null` object. It should be the responsibility of the developer to fix these kinds of errors. Exceptions that derive from `Exception`, but not from `RuntimeException`, are *checked exceptions*, conditions that are note really the developer's fault, eg, accessing a file which no longer exists. These are not really programming or logic logic errors, but are due to foreseeable circumstances. The application might be able to recover from such exceptions. Exceptions that derive from `Errore` are an indication of some serious failure of the runtime-system, not something the developer can do anything about.

Java forces the developer to treat checked and unchecked exceptions quite differently. Checked exceptions must be anticipated, and the program generally tries to recover from these exceptions. Unchecked exceptions should not arise in a correct program and require no particular treatment.

Exercises

22. List three exception types from the `java.io` package.
23. Is `SecurityException` from the `java.lang` package a checked or an unchecked exception? What about `NoSuchMethodException`?

The effect of an exception

The exception-throwing mechanism provides a parallel exit strategy from methods. In the `getDetails` method containing

```
if (key == null) {
      throw new NullPointerException("null key in getDetails");
} else {
      return (ContactDetails) book.get(key);
}
```

throwing the exception immediately stops method execution and exits from the method, without returning anything. In code calling the `getDetails` method,

```
AddressDetails details = addresses.getDetails(null);
// the following line cannot be reached
String phone = details.getPhone();
```

throwing the exception in the first line causes the code to exit. The third line never gets executed.

Unchecked exceptions

These are often thrown when some method's arguments are incorrect.

```
/**
 * Look up a name or phone number and return the
 * corresponding contact details.
 * @param key The name or number to be looked up.
 * @return The details corresponding to the key, or
 * null if there are none matching.
```

```
     * @throws NullPointerException if the key is null.
     * @throws IllegalArgumentException if the key is blank.
     */
    public ContactDetails getDetails(String key) {
          if(key == null) {
                throw new NullPointerException( "null key in getDetails");
          }
          if(key.trim().length() == 0) {
                throw new IllegalArgumentException( "Empty key passed to
                getDetails");
          }
          return (ContactDetails) book.get(key);
    }
```

Exercises

24. Review all the methods in the `AddressBook` class and decide whether any of them should throw an `IllegalArgumentException`. If so, add the necessary checks and throw statements.
25. If you have not already done so, add *javadoc* documentation to describe any exceptions thrown by methods int the `AddressBook` class.

Preventing object creation

Constructors can also throw exceptions when things go wrong.

```
    public ContactDetails(String name, String phone, String address) {
          if(name == null) {
                name = "";
          }
          if(phone == null) {
                phone = "";
          }
          if(address == null) {
                address = "";
          }

          this.name = name.trim();
          this.phone = phone.trim();
          this.address = address.trim();

          if(this.name.length() == 0 && this.phone.length() == 0) {
                throw new IllegalStateException( "Either the name or phone must
                not be blank.");
          }
    }
```

The following fails to create a proper object

```
    ContactDetails badDetails = new ContactDetails("", "", "");
```

Exception handling

The Java compiler forces special handling for checked exceptions.

Checked exceptions: the throws clause

A method which may throw a checked exception must contain a *throws clause* in its signature, eg,

```
public void saveToFile(String destination) throws IOException
```

Catching exceptions: the try statement

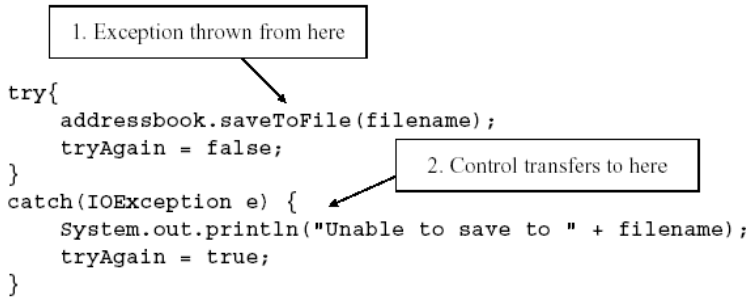
Code which invokes a method with a throws clause must take special measures to handle the exception, using the structures, *try block* and *catch block:*

```
try {
      // statements which might throw an exception
} catch (Exception e) {
      // statements which handle the exception
}
```

A try block may contain any number of statements and the catch block will catch exceptions from the try block, eg,

```
String filename = null;
try {
      filename = getAFilenameFromUser();
      addressbook.saveToFile(filename);
} catch (IOException e) {
      System.out.err("Unable to save to " + filename);
}
```

When an exception is thrown in a try block, the normal sequence of code execution is interrupted. Execution instead passes to the appropriate catch block.



Any statements in the try block following the statement where the exception is thrown are skipped, and the catch block statements are executed instead. If there is no exception thrown, all the try block statements are executed and the catch block is skipped. Execution continues after the catch block. Statments inside the try block are called *protected statements*. The catch block takes an argument which indicates what type of exception it can handle.

Exercises

26. The *address-book-v3t* project includes some throwing of unchecked exceptions if argument values are `null`. The project also includes the checked exception class `NoMatchingDetailsException`, which is currently unused. Modify the method of `removeDetailsAddressBook` so that it throws this exception if its key argument is not

a key that is in use. Add an exception handler to the `remove` method of `AddressBookTextInterface` to catch and report occurrences of this exception.

27. Make use of `NoMatchingDetailsException` in the `changeDetails` method of `AddressBook`. Enhance the user interface so that details of an existing entry may be changed. Catch and report exceptions in `AddressBookTextInterface` that arise from use of a key that does not match any existing entry.
28. Why is the following not a sensible way to use an exception handler?

```
Person p;
try {
      p = database.lookup(details);
} catch (Exception e) {
}
System.out.println("The details belong to: " + p);
```

Checked exceptions considered dangerous

Checked exceptions were an innovation introduced by Java which seemed like a good idea at the time. However, question 28 above points out a serious problem with checked exceptions - they force a developer to write extra code in the exception handler. Developers are always rushing to meet deadlines, and the easiest thing is to say, "I'll write a proper handler tomorrow, I just want to get the damn code to work..." Since tomorrow is just as busy, the code gets left with empty exception handlers. This is bad bad bad. To avoid empty handlers, the trend is to use unchecked `RuntimeExceptions` rather than checked `Exceptions`, see Bruce Eckel's article, *Does Java need checked exceptions?*

Throwing and catching multiple exceptions

A method may have to throw more than one type of exception, eg,

```
try {
      ...
      ref.process();
      ...
} catch(EOFException e) {
      // Take action on an end-of-file exception.
      ...
} catch(FileNotFoundException e) {
      // Take action on a file-not-found exception.
      ...
}
```

When an exception is thrown, catch blocks are checked in order for a matching type, so that if an `EOFException` is thrown the first block will match, and a `FileNotFoundException` matches the second block. Only one catch block can match an exception. Once a catch block is finished, control passes to the first statement after all the catch blocks. Polymorphism applies to matching exception types of course. So for example, in the following the catch block matches *any* exception

```
try {
      ...
      ref.process()
      ...
} catch(Exception e) {
```

```
        // Take action appropriate to all exceptions.
        ...
    }
```

This has the disadvantage of losing exception-specific information, all exceptions are treated the same.

Exercises

29. Enhance the try statements you wrote as solutions to exercises 12.26 and 12.27, to that they handle checked and unchecked exceptions in different catch blocks.
30. What is wrong with the following try statement?

```
try {
      Person p = database.lookup(details);
      System.out.println("The details belong to: " + p);
} catch (Exception e) {
      // handle any checked exceptions
      ...
} catch (RuntimeException e) {
      // handle any unchecked exceptions
      ...
}
```

Propagating an exception

A method doesn't always know what to do with an exception. In this case, it may have no appropriate try-catch block exists in the method, and the method must declare in its signature that it too throws the exception. From outside the method, it makes no difference if the method actually thows the exception itself, or whether it *propagates* the exception.

The finally clause

An optionally finally block in the exception handler contains code which is *always* executed, whether an exception is thrown or not.

```
try {
      // Protect one or more statements here.
} catch(Exception e) {
      // Report and recover from the exception here.
} finally {
      // Perform any actions here common to whether or not an exception is
      thrown.
      // These statements are always executed.
}
```

So why not just write

```
try {
      // Protect one or more statements here.
} catch (Exception e) {
      // Report and recover from the exception here.
}
// Perform any actions here common to whether or not an exception is thrown.
```

There are two situations where the code following the catch block above would fail to be executed:

- If there is a `return` statement in either the try or catch blocks.
- If an exception thrown in the try block is not caught.

A finally block is *always* executed.

It is possible to write try-finally blocks with no catch.

Defining new exception classes

Sometimes the standard Java exception classes are not sufficient. For example, in the address book it might be useful to have an exception include information about a key which caused an exception.

```
package addressbook.v3t;

/**
 * Capture a key that failed to match an entry in the address book.
 *
 * @author David J. Barnes and Michael Kolling.
 * @version 2008.03.30
 */
public class NoMatchingDetailsException extends Exception {
    // The key with no match.
    private String key;

    /**
     * Store the details in error.
     *
     * @param key
     *             The key with no match.
     */
    public NoMatchingDetailsException(String key) {
        this.key = key;
    }

    /**
     * @return The key in error.
     */
    public String getKey() {
        return key;
    }

    /**
     * @return A diagnostic string containing the key in error.
     */
    public String toString() {
        return "No details matching: " + key + " were found.";
    }
}
```

Exercises

31. In the *address-book-v3t* project, define a new checked exception class: `DuplicateKeyException`. This should be thrown by the `addDetails` method if either

of the non-blank key fields of an argument is already currently in use. The exception class should store details of the offending key(s). Make any further changes to the user interface class that are necessary to catch and report the exception.

32. Do you feel that `DuplicateKeyException` should be a checked or unchecked exception? Give reasons for your answer.

Using assertions

Internal consistency checks

Testing can help to ensure from the outside that a class is behaving according to specifications. Argument-checking and exception-throwing as above generally serves top protect server classes from abuse by their clients passing bad arguments to methods. The latter however, can be somewhat costly in terms of runtime checks and may even be removed once development is finished.

The assert statement

A more flexible solution is the *assert statement*, code which can be easily turned on or off.

```
/**
* 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.
*/
public void removeDetails(String key)
{
      if(key == null){
            throw new IllegalArgumentException("Null key passed to
            removeDetails.");
      }
      if(keyInUse(key)) {
            ContactDetails details = (ContactDetails) book.get(key);
            book.remove(details.getName());
            book.remove(details.getPhone());
            numberOfEntries--;
      }
      assert !keyInUse(key);
      assert consistentSize() : "Inconsistent book size in removeDetails";
}
```

The method above uses assert statments to assert things which ought to be true, like that a key should no longer be in use after it's been removed. If the boolean condition after the `assert` keyword evaluates to `true`, execution continues normally. If it evaluates to `false`, then an `AssertionError` is thrown. Note that this is a subclass of `Error` and is not expected to have an associated handler. An assertion error is probably going to be fatal anyway.

The second assertion shows a message string associated with the assertion.

Sometimes the assertion uses a method which actually does the testing and then returns the result of its consistency checks, eg,

```
/**
* Check that the numberOfEntries field is consistent with
* the number of entries actually stored in the address book.
* @return true if the field is consistent, false otherwise.
*/
private boolean consistentSize()
{
      Collection<ContactDetails> allEntries = book.values();
      // Eliminate duplicates as we are using multiple keys.
      Set<ContactDetails> uniqueEntries = new
      HashSet<ContactDetails>(allEntries);
      int actualCount = uniqueEntries.size();
      return numberOfEntries == actualCount;
}
```

Guidelines for using assertions

Assertions should be used during code development and removed in delivered code. So the following mix of assertion and normal code should be avoided.

```
// Error: don't use assert with normal processing!
assert book.remove(details.getName()) != null;
assert book.remove(details.getPhone()) != null;
```

Exercises

33. Open the *address-book-assert* project. Look through the `AddressBook` class and identify all the assert statements to be sure that you understand what is being checked and why.
34. The `AddressBookDemo` class contains several test methods that call methods of the `AddressBook` class that contain assert statements. Look through the source of `AddressBookDemo` to check that you understand the tests, and then try out each of the test methods. Are any assertion errors generated? If so, do you understand why?
35. The `changeDetails` method of `AddressBook` currently has no assert statements. One assertion we could make about it is that the address book should contain the same number of entries at the end of the method as it did at the start. Add an assert statement (and any other statements you might need) to check this. Run the `testChange` method of `AddressBookDemo` after doing so. Do you think this method should also include the check for a consistent size?
36. Suppose that we decide to allow the address book to be indexed by address as well as by name and phone number. If we simply add the following statement to the `addDetails` method `book.put(details.getAddress(), details);` Do you anticipate that any assertions will now fail? Try it. Make any further necessary changes to `AddressBook` to ensure that all of the assertions are now successful.
37. `ContactDetails` are immutable objects - that is, that they have no mutator methods. How important is this fact to the internal consistency of an `AddressBook`? Suppose the `ContactDetails` class had a `setPhone` method, for instance? Can you devise some tests to illustrate the problems this could cause?

Error recovery and avoidance

Reporting errors back to the client is good during development, but what can a production server do to recover from errors that do arise?

Error recovery

The following code is not particularly useful as it just prints an error message and then carries on regardless after an exception.

```
AddressDetails details = null;
try {
      details = addresses.getDetails(...);
} catch (Exception e) {
      System.err.println("Error: " + e);
}
String phone = details.getPhone();
```

This can actually be worse than failing immediately. The code will probably fail later anyway, but now it'll be harder to determine just what caused the problem.

When code fails, the only solution may be to try again, for example by looping to get a new filename to try.

```
// Try to save the address book.
boolean successful = false;
int attempts = 0;
do {
      try {
            addressbook.saveToFile(filename);
            successful = true;
      } catch(IOException e) {
            System.out.println("Unable to save to " + filename);
            attempts++;
            if(attempts < MAX_ATTEMPTS) {
                  filename = an alternative file name;
            }
      }
} while (!successful && attempts < MAX_ATTEMPTS);
if(!successful) {
      Report the problem and give up;
}
```

Note that:

- Anticipating and recovering from an error complicates the flow of control.
- Error-recovery code is in the catch block.
- Recovery will often involve having to try again.
- Successful recovery cannot be guaranteed.
- Hopeless recovery should be abandoned.

Error avoidance

Instead of a client getting and immediately sending an object to the server

```
String key = postCodeDatabase.search(postCode);
ContactDetails university = book.getDetails(key);
```

with the risk of sending a `null` or empty key, the client might first do some checking itself:

```
String key = postCodeDatabase.search(postCode);
```

```
if (key != null && key.length() > 0) {
      ContactDetails university = book.getDetails(key);
      ...
} else {
      // deal with postcode error
}
```

If a client can have access to a server's verification methods, then it can check data validity before using it as arguments:

```
// Add what should be a new set of details to the address book
if (book.keyInUse(details.getName()) {
      book.changeDetails(details.getName(), details);
} else if (book.keyInUse(details.getPhone())) {
      book.changeDetails(details.getPhone(), details);
} else {
      // add the details...
}
```

Using the above, `addDetails` will never throw a `DuplicateKeyException`, and the exception could be downgraded from checked to unchecked.

This shows:

- If a server's verification methods are accessible to a client, the client can avoid causing the server to throw an exception.
- Any exception that can only be thrown as the result of a programming error should be treated as an unchecked exception.
- Unchecked exceptions don't need try-catch exception handlers.

However:

- Making a server's verification methods accessible to a client increases coupling between server and client.

Case study: text input/output

Errors are endemic to programming where input/output (IO) is involved because there are any number of things that can go wrong that are no fault of the application developer, eg, a non-existent data file, a data file with inappropriate protection, a full disk, a broken network connection... Java IO is handled by the `java.io` package which contains various platform-independent classes, as well as IO error classes such as `IOException`, `EOFException`, `FileNotFoundException`, etc. In the *address-book-io* project, we'll:

- use `FileWriter` to write text output to a file;
- read text input from a file using `FileReader` and `BufferedReader`;
- anticipate possible IO errors.

Readers, writers, and streams

IO deals with two types of data, text files and binary files. Text files contain alphanumeric human-readable data (they can be opened in a text editor). Text files are handled by classes called *readers* and *writers*. Binary files may contain image data or even executable programs and are handled by *stream handlers*.

The address-book-io project

The *address-book-io* project has no user interface, but includes the `AddressBookFileHandler` class to provide file-handling operations. These include loading contents from a file, saving the contents back, and saving the results of a search operation.

```
package addressbook.io;

import java.io.*;
import java.net.URISyntaxException;
import java.net.URL;

/**
 * Provide a range of file-handling operations on an AddressBook. These methods
 * demonstrate a range of basic features of the java.io package.
 *
 * @author David J. Barnes and Michael Kolling.
 * @version 2008.03.30
 */
public class AddressBookFileHandler {
    // The address book on which i/o operations are performed.
    private AddressBook book;
    // The name of a file used to store search results.
    private static final String RESULTS_FILE = "results.txt";

    /**
     * Constructor for objects of class FileHandler.
     *
     * @param book
     *            The address book to use.
     */
    public AddressBookFileHandler(AddressBook book) {
        this.book = book;
    }

    /**
     * Save the results of an address-book search to the file "results.txt" in
     * the project folder.
     *
     * @param keyPrefix
     *            The key prefix to search on.
     */
    public void saveSearchResults(String keyPrefix) throws IOException {
        File resultsFile = makeAbsoluteFilename(RESULTS_FILE);
        ContactDetails[] results = book.search(keyPrefix);
        FileWriter writer = new FileWriter(resultsFile);
        for (ContactDetails details : results) {
            writer.write(details.toString());
            writer.write('\n');
```

```
        writer.write('\n');
    }
    writer.close();
}

/**
 * Show the results from the most-recent call to saveSearchResults. As
 * output is to the console, any problems are reported directly by this
 * method.
 */
public void showSearchResults() {
    BufferedReader reader = null;
    try {
        File resultsFile = makeAbsoluteFilename(RESULTS_FILE);
        reader = new BufferedReader(new FileReader(resultsFile));
        System.out.println("Results ...");
        String line;
        line = reader.readLine();
        while (line != null) {
            System.out.println(line);
            line = reader.readLine();
        }
        System.out.println();
    } catch (FileNotFoundException e) {
        System.out.println("Unable to find the file: " + RESULTS_FILE);
    } catch (IOException e) {
        System.out.println("Error encountered reading the file: "
                + RESULTS_FILE);
    } finally {
        if (reader != null) {
            // Catch any exception, but nothing can be done
            // about it.
            try {
                reader.close();
            } catch (IOException e) {
                System.out.println("Error on closing: " + RESULTS_FILE);
            }
        }
    }
}

/**
 * Add further entries to the address book, from a text file. The file is
 * assumed to contain one element per line, plus a blank line, for each
 * entry: name \n phone \n address \n \n A line may be blank if that part of
 * the details is missing.
 *
 * @param filename
 *            The text file containing the details.
 * @throws IOException
 *             On input failure.
 */
public void addEntriesFromFile(String filename) throws IOException {
    // Make sure the file can be found.
    URL resource = getClass().getResource(filename);
    if (resource == null) {
        throw new FileNotFoundException(filename);
    }
```

```
    filename = resource.getFile();
    BufferedReader reader = new BufferedReader(new FileReader(filename));
    String name;
    name = reader.readLine();
    while (name != null) {
        String phone = reader.readLine();
        String address = reader.readLine();
        // Discard the separating blank line.
        reader.readLine();
        book.addDetails(new ContactDetails(name, phone, address));
        name = reader.readLine();
    }
    reader.close();
}

/**
 * Read the binary version of an address book from the given file. If the
 * file name is not an absolute path, then it is assumed to be relative to
 * the current project folder.
 *
 * @param sourceFile
 *            The file from where the details are to be read.
 * @return The address book object.
 * @throws IOException
 *             If the reading process fails for any reason.
 */
public AddressBook readFromFile(String sourceFile) throws IOException,
        ClassNotFoundException {
    // Make sure the file can be found.
    URL resource = getClass().getResource(sourceFile);
    if (resource == null) {
        throw new FileNotFoundException(sourceFile);
    }
    try {
        File source = new File(resource.toURI());
        ObjectInputStream is = new ObjectInputStream(new FileInputStream(
                source));
        AddressBook savedBook = (AddressBook) is.readObject();
        is.close();
        return savedBook;
    } catch (URISyntaxException e) {
        throw new IOException("Unable to make a valid filename for "
                + sourceFile);
    }
}

/**
 * Save a binary version of the address book to the given file. If the file
 * name is not an absolute path, then it is assumed to be relative to the
 * current project folder.
 *
 * @param destinationFile
 *            The file where the details are to be saved.
 * @throws IOException
 *             If the saving process fails for any reason.
 */
public void saveToFile(String destinationFile) throws IOException {
    File destination = makeAbsoluteFilename(destinationFile);
```

```
        ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream(
                destination));
        os.writeObject(book);
        os.close();
    }

    /**
     * Create an absolute file from the given file name. If the filename is an
     * absolute one already, then use it unchanged, otherwise assume it is
     * relative to the current project folder.
     *
     * @throws IOException
     *             If a valid filename cannot be made.
     */
    private File makeAbsoluteFilename(String filename) throws IOException {
        try {
            File file = new File(filename);
            if (!file.isAbsolute()) {
                file = new File(getProjectFolder(), filename);
            }
            return file;
        } catch (URISyntaxException e) {
            throw new IOException("Unable to make a valid filename for "
                    + filename);
        }
    }

    /**
     * Try to determine the name of the current project folder. This process
     * involves locating the path of the .class file for this class, and then
     * extracting the name of the folder containing it.
     *
     * @throws URISyntaxException
     *             If the URL is not formatted correctly.
     * @return The current project folder.
     */
    private File getProjectFolder() throws URISyntaxException {
        String myClassFile = getClass().getName() + ".class";
        URL url = getClass().getResource(myClassFile);
        return new File(url.toURI()).getParentFile();
    }
}
```

The `AddressBookFileHandler` class may be closely related to the `AddressBook` class, but the code is made more cohesive by keeping the IO operations out of this latter class. This also makes it easier to add different IO handling if necessary.

Text output with `FileWriter`

There are three steps to storing data in a file:

1. The file is opened.
2. The data is written.
3. The file is closed.

Since any one of these steps can fail through no fault of the developer, error handling must be included.

```
try {
      FileWriter writer = new FileWriter(/* name of the file */);
      while (/* there is more text to write */) {
            ...
            writer.write(/* next piece of text */);
            ...
      }
      writer.close();
} catch (IOException e) {
      // something went wrong with processing the file
}
```

Trying to recover from an IO error here looks difficult (what do you do anyway if you can't open the file?). It may just be best to propagate the error to the method's caller.

Text input with FileReader

Reading a file is similar to writing:

1. The file is opened.
2. The data is read.
3. The file is closed.

A `FileReader` just reads character-by-character, whereas reading a whole line at a time is more practical. This can be done by wrapping a `FileReader` object in a `BufferedReader` object.

```
try {
      BufferedReader reader = new BufferedReader(new FileReader(/* name of
      file */));
      String line = reader.readLine();
      while (line != null) {
            // do something with line
            line = reader.readLine();
      }
      reader.close();
} catch (FileNotFoundException e) {
      // the specified file could not be found
} catch (IOException e) {
      // something went wrong with reading or closing
}
```

Exercises

38. Read the API documentation for the `File` class from the `java.io` package. What sort of information is available on files?
39. How can you tell whether a file name represents an ordinary file or a directory (folder)?
40. Is it possible to determine anything about the contents of an particular file from the information stored in a `File` object?

Scanner: reading input from the terminal

Since the beginning of the course, we have been using the *standard output*, `System.out` of type `java.io.PrintStream`, for doing our output to the terminal. There is a corresponding *standard input*, `System.in` of type `java.io.InputStream`, for getting

input from the terminal. This latter is a bit awkward to use directly as it only delivers input one character at a time. It is easier to use the `java.util.Scanner` class as follows

```
Scanner reader = new Scanner(System.in);
...
String question = reader.nextLine(); // returns a complete line
```

Exercises

41. Review the `InputReader` class of *tech-support-complete* to check that you understand how it uses the `Scanner` class.
42. Read the API documentation for the `Scanner` class in the `java.util` package. What *next* methods does it have in addition to `nextLine`?
43. Why do you think it might be useful for `Scanner` to have a constructor that takes a `String` parameter?

Object serialization

Serialzation allows reading or writing an object from/to a file system or across a network in a single operation. To be serializable, a class must implement the `java.io.Serializable` interface.

Exercises

45. Modify the `Responder` class of the *tech-support* project so that it reads its key words and responses from a text file. That would permit external enhancement and configuration of the system without having to modify the sources.

Testing for exceptions

When Bad Stuff is likely to happen to your code, you write exception handles to take care of it. How do you know that your exception handlers work as expected? By writing the appropriate unit tests of course!
Suppose you have a method `SimpleMath.divide` that might raise an exception, for instance it's got a possible divide-by-zero that you want to make sure raises an exception. Then you would write a test something like

```
@Test(expected=ArithmeticException.class)
public void divisionWithException() {
      //division by zero
      SimpleMath.divide(1, 0);
}
```

where you expect the `SimpleMath.divide` method to raise an exception of type `ArithmeticException`.

Exercise

46. Add some exception-testing methods to your address book code.

Summary

Things can go wrong.

- A client may not understand the limitations of a server.
- A server may be unable to satisfy a client through no fault of its own.
- A client may send incorrect arguments to a server.

Java error-throwing at least provides a clear mechanism for a server to inform a client that something has gone wrong. At the least, the client cannot ignore any programming errors that may be sending faulty information to the server.

Concept summary

exception
: An exception is an object representing details of a program failure. An exception is thrown to indicate that a failure has occurred.

unchecked exception
: Unchecked exceptions are a type of exception whose use will not require checks from the compiler.

checked exception
: Checked exceptions are a type of exception whose use will require checks from the compiler. In particular, checked exceptions in Java require the use of throws clauses and try statments.

exception handler
: Program code that protects statements in which an exception might be thrown is called an exception handler. It provides reporting and/or recovery code should one arise.

assertion
: An assertion is a statement of a fact that should be true in normal program execution. We can use assertions to state our assumptions explicitly, and to detect programming errors more easily.

serialization
: Serialization allows whole objects, and object hierarchies, to be read and written in a single operation. Every object involved must be from a class that implements the `Serializable` interface.