|  |
| --- |
| **Exercise 18.4**  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 18.5**  **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 18.6**  It would be more appropriate 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 accidentally 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 18.7**  **In principle, adding an email address as an additional key is both a useful and reasonable thing to be able to do. However, careful examination of the code will be necessary in order to establish all the places where changes will be required. There could easily be hidden assumptions that the number of keys is two rather than three.** |
| **Exercise 18.8**  It throws an error: java.lang.NullPointerException in AddressBook.removeDetails(). |
| **Exercise 18.9**  **If software controlling an aircraft's systems simply crashed it may well leave the pilots with no control over the aircraft.** |
| **Exercise 18.11**  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 18.12**  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 18.13**  See Section 18.2.2. |
| **Exercise 18.14**  getDetails() and keyInUse() do 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 18.15**  **Amongst other things, if there is no user interface then printed error messages are pointless.** |
| **Exercise 18.19**  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 18.21**  No. A failed search for something should be expected behavior. |
| **Exercise 18.22**  At least one of the arguments name and phone should contain a valid, non-blank 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 18.23**  A constructor can not return any values to indicate an error. One way it could be solved is to use unchecked exceptions, which are discussed in the a later section of the book. |
| **Exercise 18.24**  CharConversionException EOFException FileNotFoundException InterruptedIOException InvalidClassException InvalidObjectException IOException NotActiveException NotSerializableException ObjectStreamException OptionalDataException StreamCorruptedException SyncFailedException UnsupportedEncodingException UTFDataFormatException WriteAbortedException |
| **Exercise 18.25**  SecurityException is an unchecked exception. |
| **Exercise 18.26**  See the project *address-book-v3t*. |
| **Exercise 18.28**  **Implementation of** Iterator **requires that a** remove() **method be included, but the body of the method should simply through the** UnsupportedOperationException **if it is called. This will prevent modification of the associated collection.**  /\*\*  \* Remove an entry.  \* This operation is not permitted.  \* @throws UnsupportedOperationException if called.  \*/  public void remove()  {  throw new UnsupportedOperationException(  "It is not permitted to remove entries.");  } |
| **Exercise 18.29**  **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 18.31**  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 18.33**  All exceptions are caught by the first catch because Exception is a super type of all other exception types, so it can never reach the error handling for unchecked exceptions (RuntimeException). |
| **Exercise 18.34**  /\*\*  \* Capture a key that is a duplicate for an entry  \* in the address book.  \*  \* @author David J. Barnes and Michael Kölling.  \* @version 2011.07.31  \*/  public class DuplicateKeyException extends RuntimeException  {  // The duplicate key.  private String key;  /\*\*  \* Store the details in error.  \* @param key The duplicate key.  \*/  public DuplicateKeyException(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 "This key is a duplicate: " + key;  }  } |
| **Exercise 18.35**  **The choice behind making** DuplicateKeyException **checked or unchecked should follow the principles we have outlined in the preceding sections. If its arising is likely to be the result of a logical error in the code, then choose unchecked. If duplicate keys are likely to arise on a regular basis, and need anticipating in normal usage then checked is acceptable. There is no absolute right answer to this question; it all depends on application context.** |
| **Exercise 18.37**  testForAdditionError() results in an assertion failure. Note that the BlueJ environment must be configured to enable assertions at runtime. See the enableassertionsoption in the Java tools documentation and Appendix A for how to configure BlueJ. |
| **Exercise 18.38**  Yes, it should have a consistency check because we might change something into a key that already exists, thereby decreasing the count. |
| **Exercise 18.39**  It fails at the consistentSize() assertion in removeDetails() because it does not remove the entry which uses the address as the key. |
| **Exercise 18.40**  If you could change the phone number of a ContactDetails object, the phone number used as a key in the book would no longer be correct. This would result in problems in the methods that uses the phone number as the key. |
| **Exercise 18.41**  **A sample of the information available is:**   * **whether the file is readable –** isReadable() * **writeable –** isWritable() * **executable –** isExecutable() * **its size in bytes –** size() |
| **Exercise 18.42** You can use the method isDirectory(). |
| **Exercise 18.43  The static** probeContentType() **method provides a way to attempt to discover information a file's contents.** |
| **Exercise 18.44**  **See:** 18-44-zuul-logging**.**  **A variation on that approach could involve the introduction of a user** save **command that writes out the current version of the log.**  **The command-by-command saving version can also be achieved by having the parser call its own** saveGameLog() **method each time it adds a new command to its log list. The advantage of this approach is that it is more robust to program failure, allowing the game state to be restored on a restart. The disadvantage is the overhead of writing the complete log each time, which may become large if the game were to last several hours. This can be mitigated by opening the file in 'append' mode and simply writing each new command as soon as it is read.**  **If the open-write-close process is too costly, keep the file open throughout the game and ensure that the** flush() **method of the** FileWriter **is called after each write, in case of failure.** |
| **Exercise 18.47**  **See** 18-47-tech-support-io**.**  **The file-handling operations have been placed in a separate class called** ResponseReader**. Note that one effect of this refactoring is that responses occupy a single line – there are no embedded newline characters. This has both pros and cons.**  **The most obvious negative aspect is that the lines output by the responder are very long. However, this actually brings the opportunity for flexibility in formatting the output according to the support system's user interface. If multiple, shorter output lines are required it is easy to split a** String **at whitespace characters and then build shorter, multi-line output via a** StringBuilder **object, for instance.** |
| **Exercise 18.49**  **Some of the** next **methods are given below.**   |  |  | | --- | --- | | BigDecimal nextBigDecimal() | Scans the next token of the input as a BigDecimal. | | BigInteger nextBigInteger() | Scans the next token of the input as a BigInteger. | | boolean nextBoolean() | Scans the next token of the input as a boolean. | | byte nextByte() | 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. | | String nextLine() | Advances this scanner past the current line and returns the input that was skipped as a String. | | long nextLong() | Scans the next token of the input as a long. | | short nextShort() | Scans the next token of the input as a short. | |
| **Exercise 18.52**  **See** 18-52-network**.**  **The reading and writing has been placed in the** FileHandler **class. The classes** NewsFeed **and** Post **must implement the** java.io.Serializable **interface. It is not necessary for the individual subclasses of** Post **to implement the interface.** |
| **Exercise 18.53**  **An** InvalidClassException **will be thrown; e.g:**  java.io.InvalidClassException: NewsFeed; local class incompatible: stream classdesc serialVersionUID = 7412497150530686252, local class serialVersionUID = 5470017242796117334 |