Chapter 12
Files and Exception Handling

Files are used by many applications for storing and retrieving data. In this chapter, sequential file access, including object serialization will be discussed. Exception handling will also be explained.

What is a File?

Up to this point, the applications created in this text have stored data in the computer's memory. However, storage in memory is available only when the computer is on and an application is running. A *file* is a collection of related data stored on a persistent medium such as a hard disk or a CD. *Persistent* simply means lasting.

persistent

Files often store data used by an application. Files are also used to store the data generated by an application. In either case, a file is separate from the application accessing it and can be read from and written to by more than one application. Most applications require access to one or more files on disk.

The File Classes

java.io

The File class, part of the java.io package, is used for creating an object that represents a file. A File object can be used to create a new file, test for the existence of a file, and delete a file. Some of the File class methods include:

Class File (java.io)

Constructor/Methods

`File(String f)` creates a File object that refers to the file `f`.

`createNewFile()`
creates a new file using the file name specified in the constructor if the file does not already exist. Returns `true` if the file is created, `false` otherwise. This method throws an IOException exception if the file cannot be created.

`delete()`
permanently deletes the file represented by the File object. Returns true if the file is deleted, false otherwise.

`exists()`
Returns `true` if the file represented by the File object exists, `false` otherwise.

The application below checks for the existence of a file:

```
import java.io.*;

public class TestFiles {

  public static void main(String[] args) {
    File textFile = new File("c:\\temp\\supplies.txt");
    if (textFile.exists()) {
      System.out.println("File already exists.");
    } else {
      System.out.println("File does not exist.");
    }
  }
}
```

The name of a file is specified as a String. If a path is included in the file name, escape sequences (\\) must be used to separate the drive, folder, and file names.

Review: MyFile – part 1 of 2

Create a MyFile application that prompts the user for the name of a file and then displays a message that indicates whether the files exists or not. Note that if the user types in a full path, any single backslashes (\) will need to be replaced with an escape sequence(\\) in order to create a new File object.

Handling Exceptions

exception

An *exception* is an error affecting program execution. If an exception is not taken care of, or *handled*, the application abruptly terminates. Although many types of exceptions may still require program termination, an exception handler can allow an application to terminate gracefully by providing the user with an informative error message.

exception handler

try-catch-finally

An *exception handler* is a block of code that performs an action when an exception occurs. The `try-catch-finally` statement can be used to write an exception handler. It takes the form:

```
try {
   <statements>
} catch (exception err_code) {
   <statements>
} …additional catch clauses
} finally (exception err_code) {
   <statements>
```

The `try` statements are the statements that could possibly generate an exception. The `catch` clause waits for the exception matching the `exception` parameter and then executes its code. If more than one type of exception is possible from the statements in the `try` clause, then a separate `catch` should be written for each type of exception. The `finally` clause is optional and executes its statements regardless of what happens in the `try-catch` portion of the error handler.

IOException
throw

An exception handler is required when calling certain methods. For example, the createNewFile() method in the File class generates an IOException exception when the specified file name cannot be used to create a file. The createNewFile() method includes code to *throw*, or generate, an exception object if it cannot complete its task.

The modified TestFiles application checks for the existence of a file before creating a new one:

```
import java.io.*;

 public class TestFiles {

   public static void main(String[] args) {
     File textFile = new File("c:\\supplies.txt");
     if (textFile.exists()) {
       System.out.println("File already exists.");
     } else {
       try {
         textFile.createNewFile();
         System.out.println("New file created.");
       } catch (IOException e) {
         System.out.println("File could not be created.");
         System.err.println("IOException: " + e.getMessage());
       }
     }
   }
}
```

The Exception Stack

When an exception is thrown, the current block of code is first checked for an exception handler. Next, the calling method is checked for a handler, and so on until the Java interpreter is reached.

err stream

An exception, such as IOException, is an object of the Throwable class. Throwable objects have a getMessage() member that returns a String containing information about the exception. The *err stream* is used for displaying error messages on the screen.

Review: MyFile – part 2 of 2

Create a MyFile application that creates a file named zzz.txt and then displays a message indicating that the file has been created. The application should prompt the user to either keep or delete the file. If the file is deleted, a message should notify the user when the file has been successfully deleted.

The File Streams

TIP The Scanner class requires an input stream.

A file must be associated with a stream in order to perform operations such as reading the contents, writing over existing contents, and adding to the existing contents. A *stream* processes characters, and in Java, streams are implemented with classes.

file position

sequential file access

The file stream keeps track of the *file position*, which is the point where reading or writing last occurred. File streams are used to perform *sequential file access*, with all the reading and writing performed one character after another or one line after another.

Data Streams

A stream applies to data input/output in general. For example, memory, information sent to a printer, and data sent and received from an Internet site can all be streamed.

A stream can be thought of as a sequence of characters. For example, a file containing a list of names and scores may look like the following when viewed in a word processor:

```
Drew 84
Tia 92
```

However, when thinking about file operations, the file should be visualized as a stream of data:

D	r	e	w		8	4	*Cr*	*Lf*	T	i	a		9	2	*Cr*	*Lf*	–1

line terminator, end of file

The carriage return character (Cr) followed by a line feed character (Lf) is called a *line terminator*. A –1 is the *end of file*.

The FileReader and BufferedReader Classes

input file stream

The FileReader and BufferedReader classes, both from the java.io package, are used together to read the contents of an existing file. The FileReader class is used to create an *input file stream*. Next, the BufferedReader class is used to read text from the stream. The following is a summary of the FileReader and BufferedReader classes:

Class FileReader (java.io)

Constructor/Method

`FileReader(File fileName)`

FileNotFoundException

creates an input file stream for the File object. This constructor throws a FileNotFoundException exception if the file does not exist.

`close()` closes the input file stream. This method throws an IOException exception if the file cannot be closed.

TIP A buffer stores a large number of characters from the stream so that more than one character at a time can be read, such as in a readLine().

Class BufferedReader (java.io)

Constructor/Methods

`BufferedReader(Reader stream)`

creates a buffered-input stream from `stream`. Reader is the FileReader superclass.

`read()` reads a single character from the input stream. This method throws an IOException exception if the stream cannot be read.

`readLine()` reads a line of text from the input stream. This method throws an IOException exception if the stream cannot be read.

`close()` closes the input file stream. This method throws an IOException exception if the stream cannot be closed.

Reading Characters

The read() method returns an `int`, which corresponds to a Unicode value. In Unicode, a space corresponds to 32, a tab to 9, carriage return to 13, and line feed to 10.

The application on the next page reads an existing file line-by-line to show the contents of the file:

```
import java.util.Scanner;
import java.io.*;

public class ReadFile {

  public static void main(String[] args) {
    File textFile = new File("wonder.txt");
    FileReader in;
    BufferedReader readFile;
    String lineOfText;

    try {
      in = new FileReader(textFile);
      readFile = new BufferedReader(in);
      while ((lineOfText = readFile.readLine()) != null ) {
        System.out.println(lineOfText);
      }
      readFile.close();
      in.close();
    } catch (FileNotFoundException e) {
      System.out.println("File does not exist or could
                          not be found.");
      System.err.println("FileNotFoundException: "
                         + e.getMessage());
    } catch (IOException e) {
      System.out.println("Problem reading file.");
      System.err.println("IOException: " + e.getMessage());
  }
}
```

The ReadFile application reads and displays the file contents within a `try-catch` statement because both the FileReader and BufferedReader constructors and methods throw exceptions if there is a problem reading the file. A `try` can have multiple `catch` statements. The `catch` statements are in the order that they may occur. For example, the FileNotFoundException is handled first because the FileReader object is created first in the `try` statement.

The close() methods are used to close the FileReader and BufferedReader streams. It is important that the streams be closed in the reverse order that they were opened.

The ReadFile application displays the following output when run:

```
The little girl shouted "Go, Wonder Horse, go!"
Her brother stared in disbelief as his toy horse began to fly.
Together they laughed while the horse flew around them.
```

Review: Assignment

Create a Assignment application that reads and then displays the contents of a file containing instructions for this assignment. Use Notepad or some other word processor to create the file. Be sure that the file is saved as a Text file (TXT). The Assignment application will need to include the correct path to the location of the file. If a path is not specified, the file must be placed in the same folder as the Assignment executable file.

Processing Numeric Data

A file on disk is a set of characters, even when the file contains numeric data, such as test scores. An application written to process numeric data from a file must convert the data after it is read. The Double and Integer classes include class methods for converting a string to a primitive data type:

Class Double (java.lang.Double)

Method

`parseDouble(String text)`

returns the `double` value in the String `text`.

Class Integer (java.lang.Integer)

Method

`parseInteger(String text)`

returns the `int` value in the String `text`.

The AvgScore application reads tests scores that are stored one score per line in a text file and then reports the average:

DAT Files

Text files that contain numeric data often have the file name extension `.dat`, also referred to as a DAT file.

```
import java.io.*;

public class AvgScore {

  public static void main(String[] args) {
    File dataFile = new File("scores.dat");
    FileReader in;
    BufferedReader readFile;
    String score;
    double avgScore;
    double totalScores = 0;
    int numScores = 0;

    try {
      in = new FileReader(dataFile);
      readFile = new BufferedReader(in);
      while ((score = readFile.readLine()) != null ) {
        numScores += 1;
        System.out.println(score);
        totalScores += Double.parseDouble(score);
      }
      avgScore = totalScores / numScores;
      System.out.println("Average = " + avgScore);
      readFile.close();
      in.close();
    } catch (FileNotFoundException e) {
      System.out.println("File does not exist or could
                          not be found.");
      System.err.println("FileNotFoundException: "
                          + e.getMessage());
    } catch (IOException e) {
      System.out.println("Problem reading file.");
      System.err.println("IOException: " + e.getMessage());
    }
  }
}
```

The application produces the output:

```
88
92
76
95
67
82
91
Average = 84.42857142857143
```

Review: Stats – part 1 of 2

Create a Stats application that reads names and scores from a data file named test1.dat, supplied with this text. The file contains a student name on one line followed by the student's test score on the next line. The Stats application should read and display each name and score. After all the scores have been displayed, the lowest score, highest score, and average score should be displayed.

The FileWriter and BufferedWriter Classes

output file stream

The FileWriter and BufferedWriter classes, both from the java.io package, are used together to write data to a file. The FileWriter class is used to create an *output file stream*. A BufferedWriter class object is then used to send text to the stream. Some of the FileWriter and BufferedWriter classes methods include:

Class FileWriter (java.io)

Constructor/Method	
`FileWriter(File fileName, boolean append)`	creates an input file stream for the File object. If `append` is true, then data written to the file will be added after existing data, otherwise the file will be overwritten. This constructor throws an IOException exception if the file cannot be created or opened.
`close()`	closes the output file stream. This method throws an IOException exception if the file cannot be closed.

When a FileWriter object is created, the file referenced by the File object is automatically overwritten unless the FileWriter object is set to append. In either case, if the file does not yet exist, a new one will be created. Caution must be used so that a file is not inadvertently overwritten.

Class BufferedWriter (java.io)

Constructor/Methods

`BufferedWriter(Writer stream)`
: creates a buffered-writer stream from `stream`. Writer is the FileWriter superclass.

`newLine()`
: writes a newline character to the output stream. This method throws an IOException exception if the stream cannot be written to.

`write(String str)`
: writes the string `str` to the output stream. This method throws an IOException exception if the stream cannot be written to.

`write(char c)`
: writes the character `c` to the output stream. This method throws an IOException exception if the stream cannot be written to.

`close()`
: closes the output file stream. This method throws an IOException exception if the stream cannot be closed.

The CreateDataFile application prompts the user for names and scores and then writes them to a new file:

```
import java.io.*;
import java.util.Scanner;

public class CreateDataFile {

  public static void main(String[] args) {
    File dataFile = new File("StuScores.dat");
    FileWriter out;
    BufferedWriter writeFile;
    Scanner input = new Scanner(System.in);
    double score;
    String name;

    try {
      out = new FileWriter(dataFile);
      writeFile = new BufferedWriter(out);
      for (int i = 0; i < 5; i++) {
        System.out.print(:Enter student name: ");
        name = input.next();
        System.out.print("Enter test score: ");
        score = input.nextDouble();
        writeFile.write(name);
        writeFile.newLine();
        writeFile.write(String.valueOf(score));
        writeFile.newLine();
      }
      writeFile.close();
      out.close();
      System.out.println("Data written to file.");
    } catch (IOException e) {
      System.out.println("Problem writing to file.");
      System.err.println("IOException: " + e.getMessage());
    }
  }
}
```

Note that the application will overwrite the StuScores.dat file each time that it is run. The CreateDataFile application displays output similar to that shown on the next page:

```
Enter student name: Tyra
Enter test score: 98
Enter student name: Carmen
Enter test score: 83
Enter student name: Kaeli
Enter test score: 77
Enter student name: Mel
Enter test score: 92
Enter student name: Juan
Enter test score: 95
Data written to file.
```

Review: Stats – part 2 of 2

Modify the Stats application to allow the user to enter the names and grades of the students. The user should be prompted for the name of the file to create and for the number of student grades that will be entered. After the data has been entered and the written to a file, the file should be read and the lowest, highest, and average score displayed.

Object Serialization

A file can also be used to store object data. Writing objects to a file is called *object serialization*. In this process, class information about an object is written out to a stream. If a class uses another class, this information is also written out, and so on. When information about an object is retrieved from a file, it is called *object deserialization*.

Object serialization and deserialization is performed with object output and input streams. The FileOutputStream and ObjectOutputStream classes, both from the java.io package, are used together to write objects to a file. The FileInputStream and ObjectInputStream classes, also from the java.io package, are used together to read objects from a file. Some of the methods from the classes for writing and reading objects include:

Class FileOutputStream (java.io)

Constructor/Method	
`FileOutputStream(File fileName, boolean append)`	creates an output file stream for the File object. If `append` is true, then data written to the file will be added after existing data, otherwise the file will be overwritten or created if the file does not exist. This method throws a FileNotFoundException exception if the file cannot be opened or created.
`close()`	closes the output file stream. This method throws an IOException exception if the file cannot be closed.

In addition to methods for writing objects to the output stream, the ObjectOutputStream class also contains method for writing primitive data types.

Class ObjectOutputStream (java.io)

Constructor/Method

`ObjectOutputStream(FileOutputStream stream)`

creates an object stream from `stream`.

`writeObject(Object obj)`

writes object information to the output stream. This method throws an IOException exception if there are problems with the class or if the stream cannot be written to.

`writeInt(int num)`

writes an `int` to the output stream. This method throws an IOException exception if the stream cannot be written to.

`writeDouble(double num)`

writes a `double` to the output stream. This method throws an IOException exception if the stream cannot be written to.

`close()` closes the output stream. This method throws an IOException exception if the stream cannot be closed.

Objects are read from a FileInputStream stream object:

Class FileInputStream (java.io)

Constructor/Method

`FileInputStream(File fileName)`

creates an input file stream for the File object. This method throws a FileNotFoundException exception if the file cannot be read.

`close()` closes the input file stream. This method throws an IOException exception if the file cannot be closed.

The ObjectInputStream class contains method for reading both objects and primitive data types.

Class ObjectInputStream (java.io)

Constructor/Method

`ObjectInputStream(FileInputStream stream)`

creates an object stream from `stream`. This constructor throws an IOException exception if the stream cannot be read.

`readObject()` reads an object from the input stream. This method throws exceptions IOException and ClassNotFoundException if the the stream cannot be read or a class cannot be deserialized.

`readInt()` reads an `int` from the input stream. This method throws an IOException exception if the file cannot be read.

`readDouble()`	reads a `double` from the input stream. This method throws an IOException exception if the file cannot be read.
`close()`	closes the input stream. This method throws an IOException exception if the file cannot be closed.

The ObjectWriteRead application demonstrates writing and reading objects from a file:

```
import java.io.*;

public class ObjectWriteRead {

 public static void main(String[] args) {
   File stuFile = new File("students.dat");

   try {
     /* write objects */
     FileOutputStream out = new FileOutputStream(stuFile);
     ObjectOutputStream writeStu = new ObjectOutputStream(out);
     writeStu.writeObject(new Student("Drew", 87));
     writeStu.writeObject(new Student("Tia", 92));
     writeStu.close();
     System.out.println("Data written to file.");

     /* read objects */
     FileInputStream in = new FileInputStream(stuFile);
     ObjectInputStream readStu = new ObjectInputStream(in);
     Student stu1 = (Student)readStu.readObject();
     Student stu2 = (Student)readStu.readObject();
     readStu.close();

     System.out.println(stu1);
     System.out.println (stu2);

     } catch (FileNotFoundException e) {
       System.out.println("File could not be found.");
       System.err.println("FileNotFoundException: "
                          + e.getMessage());
     } catch (IOException e) {
       System.out.println("Problem with input/output.");
       System.err.println("IOException: " + e.getMessage());
     } catch (ClassNotFoundException e) {
       System.out.println("Class could not be used to
                          cast object.");
       System.err.println("ClassNotFoundException: "
                          + e.getMessage());
     }
  }
}
```

The ObjectWriteRead application writes two Student objects to the students.dat file. The Student class is shown on the next page:

```
import java.io.*;

public class Student implements Serializable {
  private String stuName;
  private double stuGrade;

  /**
   * constructor
   * pre: none
   * post: A Student object has been created.
   * Student data has been initialized with parameters.
   */
  public Student(String name, double grade) {
    stuName = name;
    stuGrade = grade;
  }


  /**
   * Creates a string representing the student object
   * pre: none
   * post: A string representing the student object
   * has been returned.
   */
  public String toString() {
    String stuString;

    stuString = stuName + " grade: " + stuGrade;
    return(stuString);
  }
}
```

Serializable interface

If the objects of a class are to be written to a file, the class must implement the Serializable interface. This interface is part of the java.io package and contains no methods to implement. It simply allows information about an instance of the class to be written out.

Note that reading an object from a file requires casting. The readObject() method reads Object data from the file. It is up to the programmer to cast the object to the appropriate type.

The ObjectWriteRead application catches several exceptions. First, a FileNotFoundException occurs when there is a problem creating a File object. IOException is a more general exception and occurs for various input/output problems. If IOException were first in the `catch` clauses, the other more specific exceptions would not be caught and the user would not be able to read their descriptive error messages. Finally, the ClassNotFoundException occurs if the class for an object written to the file cannot be found.

Review: Roster

Create a Roster application that prompts the user for the name of the file to store student names and then prompts the user for the number of students in a class. The application should then prompt the user for the first and last name of each student and write this data to a file. After all the data is written to a file, the application display the class roster with one name after the other in a list. Create a StuName class that has member variables `firstName` and `lastName` and a toString() member method.

Chapter 12 Case Study

In this case study, a LocalBank2 application will be created. LocalBank2 is the Chapter 10 case study modified to read and write account information from a file.

LocalBank2 Specification

The LocalBank2 application has the same specification as LocalBank. It allows accounts to be opened, modified, and closed. Each account has a unique account number, which is required for all transactions. Transactions include deposits and withdrawals. An account balance can also be checked.

The LocalBank2 interface will not change from LocalBank. It will provide a menu of options:

```
Deposit\Withdrawal\Check balance
Add an account\Remove an account
Quit

Enter choice: a
First name: Adriana
Last name: Lee
Beginning balance: 100
Account created. Account ID is: ALee

Deposit\Withdrawal\Check balance
Add an account\Remove an account
Quit

Enter choice: D
Enter account ID: ALee
Deposit amount: 22
ALee
Adriana Lee
Current balance is $122

Deposit\Withdrawal\Check balance
Add an account\Remove an account
Quit

Enter choice:
```

The LocalBank2 algorithm will not change from the LocalBank application algorithm.

LocalBank2 Code Design

The LocalBank2 application will be modeled with a Bank object, Account objects, and Customer objects, just as LocalBank. However, the Bank object will store and retrieve accounts from a file. To do this, the Bank constructor should create a file stream for the File object specified when the Bank object is instantiated. The constructor should also load accounts from the file stream into an ArrayList. An updateAccounts() method will need to be added to the Bank class so that any account changes can be written back to the file.

Because account objects will be read from and written to a file, the Account class and any classes it uses must implement the Serializable interface. This includes the Customer class.

The LocalBank2 client code must be modified to create a File object that stores the account information. Before ending the LocalBank application, the updateAccounts() method is called to write account information back to the file. The pseudocode for the LocalBank2 client code follows:

```
File accountsFile =  new File("LBAccounts.dat");
Bank easySave = new Bank(accountsFile);

do {
    prompt user for transaction type

    if (add account) {
        easySave.addAccount();
    } else if (not Quit) {
        prompt user for account ID;
        if (deposit) {
          prompt user for deposit amount
          easySave.transaction(make deposit, acctID, amt);
        } else if (withdrawal) {
          prompt user for withdrawal amount
          easySave.transaction(make withdrawal, acctID, amt);
        } else if (check balance) {
          easySave.checkBalance(acctID);
        } else if (remove account) {
          easySave.deleteAccount(acctID);
        }
    }
} while (not quit);
easySave.updateAccounts(accountsFile);
```

LocalBank Implementation

The LocaBank2 application is shown below:

```
/**
 * LocalBank2 client code.
 */

 import java.io.*;
 import java.util.Scanner;

 public class LocalBank {

    public static void main(String[] args) {
        File accountsFile = new File("LBAccounts.dat");
        Bank easySave = new Bank(accountsFile);
```

```
        Scanner input = new Scanner(System.in);
        String action, acctID;
        Double amt;

        /* display menu of choices */
        do {
            System.out.println("\nDeposit\\Withdrawal\\Check balance");
            System.out.println("Add an account\\Remove an account");
            System.out.println("Quit\n");
            System.out.print("Enter choice: ");
            action = input.next();

            if (action.equalsIgnoreCase("A")) {
                easySave.addAccount();
            } else if (!action.equalsIgnoreCase("Q")) {
                System.out.print("Enter account ID: ");
                acctID = input.next();
                if (action.equalsIgnoreCase("D")) {
                    System.out.print("Enter deposit amount: ");
                    amt = input.nextDouble();
                    easySave.transaction(1, acctID, amt);
                } else if (action.equalsIgnoreCase("W")) {
                    System.out.print("Enter withdrawal amount: ");
                    amt = input.nextDouble();
                    easySave.transaction(2, acctID, amt);
                } else if (action.equalsIgnoreCase("C")) {
                    easySave.checkBalance(acctID);
                } else if (action.equalsIgnoreCase("R")) {
                    easySave.deleteAccount(acctID);
                }
            }
        } while (!action.equalsIgnoreCase("Q"));

        easySave.updateAccounts(accountsFile);      //write accounts to file
    }
}
```

The Bank class is shown below. The accounts are expected to be loaded from a file. Additionally, the number of accounts is also kept in the file. This number is the first data item in the file. It is read first and then used to determine how many accounts to read from the file. Throughout the class, the number of accounts is updated when a new account is added and when an account is deleted. When updateAccounts() is called, the number of accounts is written to the file first, followed by the accounts:

```
/**
 * Bank class.
 */

import java.util.ArrayList;
import java.io.*;
import java.util.Scanner;

public class Bank {
    private ArrayList accounts;
    private int numAccts;
```

```
/**
 * constructor
 * pre: none
 * post: accounts have been loaded from acctFile.
 */
public Bank(File acctsFile) {
    accounts = new ArrayList();
    Account acct;

    /* Create a new file for accounts if one does not exist */
    if (!acctsFile.exists()) {
        try {
            acctsFile.createNewFile();
            System.out.println("There are no existing accounts.");
        } catch (IOException e) {
            System.out.println("File could not be created.");
            System.err.println("IOException: " + e.getMessage());
        }
        numAccts = 0;
    } else {    /* load existing accounts */
        try {
            FileInputStream in = new FileInputStream(acctsFile);
            ObjectInputStream readAccts = new ObjectInputStream(in);
            numAccts = (int)readAccts.readInt();
            if (numAccts == 0) {
                System.out.println("There are no existing accounts.");
            } else {
                for (int i = 0; i < numAccts; i++) {
                    acct = (Account)readAccts.readObject();
                    accounts.add(acct);
                }
            }
            readAccts.close();
        } catch (FileNotFoundException e) {
            System.out.println("File could not be found.");
            System.err.println("FileNotFoundException: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("Problem with input/output.");
            System.err.println("IOException: " + e.getMessage());
        } catch (ClassNotFoundException e) {
            System.out.println("Class could not be used to cast object.");
            System.err.println("ClassNotFoundException: " + e.getMessage());
        }
    }
}


/**
 * Adds a new account to the bank accounts.
 * pre: none
 * post: An account has been added to the bank's accounts.
 */
public void addAccount() {
    Account newAcct;
    double bal;
    String fName, lName;
    Scanner input = new Scanner(System.in);

    System.out.print("First name: ");
    fName = input.nextLine();
    System.out.print("Last name: ");
    lName = input.nextLine();
    System.out.print("Beginning balance: ");
    bal = input.nextDouble();
```

```
        newAcct = new Account(bal, fName, lName);   //create account object
        accounts.add(newAcct);                      //add account to bank accounts
        numAccts += 1;                              //increment number of accounts

        System.out.println("Account created. Account ID is: " + newAcct.getID());
    }


    /**
     * Deletes an existing account.
     * pre: none
     * post: An existing account has been deleted.
     */
    public void deleteAccount(String acctID) {
        int acctIndex;
        Account acctToMatch;

        acctToMatch = new Account(acctID);
        acctIndex = accounts.indexOf(acctToMatch);      //retrieve location of account
        if (acctIndex > -1) {
            accounts.remove(acctIndex);                 //remove account
            System.out.println("Account removed.");
            numAccts -= 1;                              //decrement number of accounts
        } else {
            System.out.println("Account does not exist.");
        }
    }
```

The Account class need only be modified to implement the Serializable class:

```
/**
 * Account class.
 */

import java.io.*;
import java.text.NumberFormat;

public class Account implements Serializable {
```

…rest of Account class (refer to Chapter 10 case study)

The Customer class need only be modified to implement the Serializable class:

```
/**
 * Customer class.
 */

import java.io.*;

public class Customer implements Serializable {
```

…rest of Customer class (refer to Chapter 10 case study)

Running the LocalBank2 application displays the same output as the LocalBank application:

```
Deposit\Withdrawal\Check balance
Add an account\Remove an account
Quit

Enter choice: a
First name: Sam
Last name: Iam
Beginning balance: 100
Account created. Account ID is: SIam

Deposit\Withdrawal\Check balance
Add an account\Remove an account
Quit

Enter choice: D
Enter account ID: SIam
Enter deposit amount: 22
SIam
Sam Iam
Current balance is $122

Deposit\Withdrawal\Check balance
Add an account\Remove an account
Quit

Enter choice:
```

LocalBank2 Testing and Debugging

The LocalBank2 application should be tested to be sure that it works appropriately when no file exists, when a file with no accounts exists, and when a file with accounts exists.

Review: LocalBank2

Modify the Bank class to keep track of accounts with low balances. A low balance is an account with less than $20.00. The number of low balance accounts should be stored after the number of accounts, but before the account objects in the account file. Have the number of low balance accounts displayed when a Bank object is created and again when the updateAccounts() method is called.

Chapter Summary

This chapter discussed files and exception handling. A file is a data stored on a persistent medium such as a hard disk or CD. In Java, a File object is associated with a file name. A File object can be used to create or delete a file.

An exception is an error affecting program execution. An exception handler is a block of code that performs an action when an exception occurs. This chapter introduced the `try-catch-finally` statement for writing exception handlers.

A file must be associated with a stream in order to read and write to the file. A file stream processes characters and is used to perform sequential file access. The FileReader, BufferedReader, FileWriter, and BufferWriter classes are used to read and write to a file.

An application written to process numeric data from a file must convert the file data from a strings to numerics. The Double and Integer classes contain methods for converting numeric characters in a string to an `int` or a `double`.

Objects can be written to a file in a process called object serialization. Reading objects from a file is called object deserialization. Serialization and deserialization are performed with an object stream. The FileOutputStream, ObjectOutputStream, FileInputStream, and ObjectInputStream classes are used to write and read objects to a file. The ObjectOutputStream and ObjectInputStream classes can also be used to write and read primitve data to a file.

Vocabulary

ClassNotFoundException An exception thrown if a stream cannot be read or a class cannot be deserialized.

End of file A –1 in the file stream.

err stream The stream used for displaying error messages to the user.

Exception An error affecting program execution.

Exception handler A block of code that performs an action when an exception occurs.

File A collection of related data stored on a persistent medium.

FileNotFoundException An exception thrown when a file does not exist.

File position The point at which reading or writing in the stream last occurred.

Handle Take care of.

Input file stream A file stream for reading a file.

IOException An exception thrown when a file cannot be created, or when there is a general input/output problem.

Line terminator A carriage return followed by a line feed in the file stream.

Object deserialization The process used to read objects from a file.

Object serialization The process used to write objects to a file.

Output file stream A file stream for writing to a file.

Persistent Lasting.

Sequential file access Reading and writing one character after another.

Stream The construct used for processing characters.

Throw Generate.

Java

BufferedReader A java.io class used for creating a buffered file stream for reading a file.

BufferedWriter A java.io class used for creating a buffered file stream for writing to a file.

Double A java.lang class for converting numeric text in a string to a `double`.

File The java.io class used for creating an object that refers to a file.

FileInputStream A java.io class used for creating an object input stream.

FileOutputStream A java.io class used for creating an object output stream.

FileReader The java.io class used for creating a file stream for reading a file.

FileWriter The java.io class used for creating a file stream for writing to a file.

Integer A java.lang class for converting numeric text in a string to an `int`.

`try-catch-finally` Statement used to write an exception handler.

ObjectInputStream A java.io class used for creating an object for reading objects from a file.

ObjectOutputStream A java.io class used for creating an object for writing objects to a file.

Critical Thinking

1. Can data in memory be called a file? Explain.

2. Write the `import` statement required to access the File Class in an application.

3. Identify the error in the following statement:
   ```
   File textFile = new File("c:\inventory.txt");
   ```

4. a) Which statement is used to write an exception handler?
   b) Write an exception handler to handle an IOException if a specified file name cannot be used to create a file. The exception handler should display appropriate messages to the user.

5. a) What is the name of the stream for displaying error messages.
   b) Where are these messages displayed?

6. a) What does the file stream keep track of?
   b) What characters together make up a line terminator?

7. What two classes are used together to write data to a file?

8. Write a statement to convert account balances that have been read from a text file to a `double` value and add them to `totalBalance`.

9. Explain the difference between object serialization and object deserialization.

10. What interface must be implemented if objects of a class are to be written to a file?

11. Describe two situations where an IOException exception could be thrown, and write an example exception handler for each situation to output an appropriate message if the exception occurs.

True/False

12. Determine if each of the following are true or false. If false, explain why.
    a) An exception always results in program termination.
    b) Sequential file access reads and writes data one character after another or one line after another.
    c) `Z-1` is the end of file.
    d) A file on disk is a set of numbers ranging from 0 to 9.
    e) Numeric data in a file must be converted to a primitive data type before it can be processed numerically.
    f) A FileNotFoundException exception is thrown if a file cannot be closed.
    g) The output file stream is a file stream for writing to the screen.
    h) Reading an object from a file requires casting.

Exercises

Exercise 1 — WordCount

Create a WordCount application that displays the number of words and the average word length in a text file named source.txt. Consider a word to be any sequence of letters terminated by nonletters. For example, forty-nine is two words.

Exercise 2 — WordStats

a) Create a WordStats application that lists all the unique words in a file and how many times they occurred. WordStats should ignore capitalization. The application should provide a listing similar to:

```
WORD     OCCURENCES
the      57
and      12
zoo      3
```

b) Modify the WordStats application to list the words in alphabetical order. This can be done by either using the sorting algorithm presented in Chapter 10, Exercise 11 or by keeping the words in order as they are read.

Exercise 3 — TestProcessor

Test results for a multiple choice test can be stored in a text file as follows:

Line 1: The correct answers, one character per answer
Line 2: Name of the first student (length <= 30 chars)
Line 3: Answers for the student in line 2
The remaining lines: student names and answers on separate lines

For example:

```
BADED
Smithgall
BADDD
DeSalvo
CAEED
Darji
BADED
```

Create a TestProcessor application that processes the test results file for any number of students. The application should provide statistics similar to:

```
Smithgall   80%
DeSalvo     60%
Darji       100%
```

Exercise 4 — MadLib

A Mad-Lib story is a story where nouns and verbs in a paragraph are randomly replaced with other nouns and verbs, usually with humorous results. Create a MadLib application that displays a Mad-Lib story. The application will require three files:

- story.txt which contains a story with # signs as noun placeholders, and % signs as verb placeholders. For example:

```
Bugsy Kludge is a # with our company.
His job is to % all of the #s.
```

- verbs.txt which contains verbs, one per line. For example:

```
run
display
eat
```

- nouns.txt which contains nouns, one per line. For example:

```
banana
soprano
elephant
vegetable
```

Application output should display the story with apporpriate replacements made. A possible output would produce a MadLib similar to:

```
Bugsy Kludge is a vegetable with our company.
His job is to display all of the elephants.
```

Exercise 5 — MergeFiles

The idea of merging two or more files is an important one in programming. One approach is to merge the ordered data of two files into a third file, keeping the data in order.

Create a MergeFiles application that merges the integers ordered from low to high in two files into a third file, keeping the order from low to high. For example, two files of integers could contain:

```
File 1:   12 23 34 45 56 67 69 123 133
File 2:   4 5 10 20 35 44 100 130 150 160 180
```

The application should not use an array to temporarily store the numbers, but should merge the two files by taking one element at a time from each. After MergeFiles has been run, the third file should contain:

```
4 5 10 12 20 23 34 35 44 45 56 67 69 100 123 130 133 150 160 180
```

Exercise 6 — MergeLarge

The algorithm used implemented in Exercise 5 is sometimes used as part of a sorting algorithm with data that is too large to be stored in memory at once. For example, to sort a large file large.dat that is twice as large as will fit in memory, half can be read into an array, the array sorted, and then written to a file numbers1.dat. Next, the second half of large.dat can be read into the array, sorted, and then written to numbers2.dat. Finally, the two files can be merged in order back into large.dat. Create a MergeLarge application that implements this algorithm. Test the application with a file that contains 30 integers sorted from low to high.

Exercise 7 — HTMLViewer

The early success of the Web is largely due to the simplicity of HTML (HyperText Markup Language), for designing a Web page. HTML documents are text documents containing content and tags that describe the format of the content. An HTML document could look similar to:

```
JUNE BUGS<p>June bugs RAM in megabytes of fleshy fruit,<br>downsize fine fat
figs<br>chip away at melon bits.<br>They monitor windows for open screens,<br>seeking
felicity in electricity.<br>Built-in memory warns of websites<br>hiding spiders
with sly designs.<br>Scrolling the scene of leaves and trees,<br>they network the
neighborhood in flashy green jackets,<br>each bug a browser, scanner, looter-
<br>not even knowing the word "computer."<p>by Avis Harley<p><hr>
```

Tags are enclosed in angle brackets, < >. The tag
 means to start a new line. The tag <p> means to start a new paragraph (a blank line). The tag <hr> means to draw a horizontal rule. When these tags are interpreted, the HTML document above is displayed as:

```
JUNE BUGS

June bugs RAM in megabytes of fleshy fruit,
downsize fine fat figs
chip away at melon bits.
They monitor windows for open screens,
seeking felicity in electricity.
Built-in memory warns of websites
hiding spiders with sly designs.
Scrolling the scene of leaves and trees,
they network the neighborhood in flashy green jackets,
each bug a browser, scanner, looter-
not even knowing the word "computer."

by Avis Harley

------------------------------------------------
```

a) HTML documents are interpreted by browser software. Create an HTMLViewer application that interprets an HTML file to display the Web content as intended, similar to the way a browser decides how to display an HTML document.

b) Modify the HTMLViewer application to allow the user to specify the display line width. For example, a width of 35 should display the HTML document as:

```
JUNE BUGS

June bugs RAM in megabytes of
fleshy fruit,
downsize fine fat figs
chip away at melon bits.
They monitor windows for open
screens,
seeking felicity in electricity.
Built-in memory warns of websites
hiding spiders with sly designs.
Scrolling the scene of leaves and
trees,
they network the neighborhood in
flashy green jackets,
each bug a browser, scanner,
looter-
not even knowing the word
"computer."

by Avis Harley

-----------------------------------
```

Exercise 8 — CarRecall

Modify the CarRecall application created in Chapter 6, Exercise 4 to load the defective car model numbers from a file.

Exercise 9 — WordGuess

Modify the WordGuess case study from Chapter 6 to use a word from a file as the secret word. The file should contain a list of words, with one word per line. The WordGuess application should determine which word to use, by generating a random number that corresponds to one of the words in the file.

Exercise 10 — FindAndReplace

Create a FindAndReplace application that prompts the user for a file name, a search word or phrase, and a replacement word or phrase. After entering the replacement word or phrase, FindAndReplace finds all occurrences of the search word or phrase in a file and replaces them with the specified replacement word or phrase.

Exercise 11 — ApplicationDoc

Create an ApplicationDoc application that prompts the user for the file name of a Java source code file (the file name extension should be `java`) and then copies all the documentation comments (/** */) to a separate file.

Exercise 12 — MySavings

Modify the MySavings application from Chapter 8, Exercise 1 to store and load the PiggyBank object from a file.

Exercise 13 — Adder

Modify the Adder application from Chapter 8, Exercise 8 to keep track of player scores in a file. The application should prompt the player for his or her name and then create a file based on the player name. For example, if the player's name is Jo, then a file named JoScores.txt should be created. The application should write the player's score to their file at the end of the game and then display the player's previous scores.

Exercise 14 — CountVowels

Modify the CountVowels application created in Chapter 6, Exercise 20 to count the number of vowels in a text file. The application should prompt the user for the file name.

Exercise 15 — Coder

Modify the Coder application created in Chapter 10, Exercise 10 to encode the text in one file and write it to a new file.

Exercise 16 — CourseGrades

Modify the CourseGrades application created in Chapter 10, Exercise 12 to write the GradeBook object to a file.

Exercise 17 — Life

Modify the Life application created in Chapter 10, Exercise 14 to retrieve the initial life grid from a file.

Exercise 18 — Inventory

Modify the Inventory application created in Chapter 10, Exercise 16 to write the inventory items to a file.