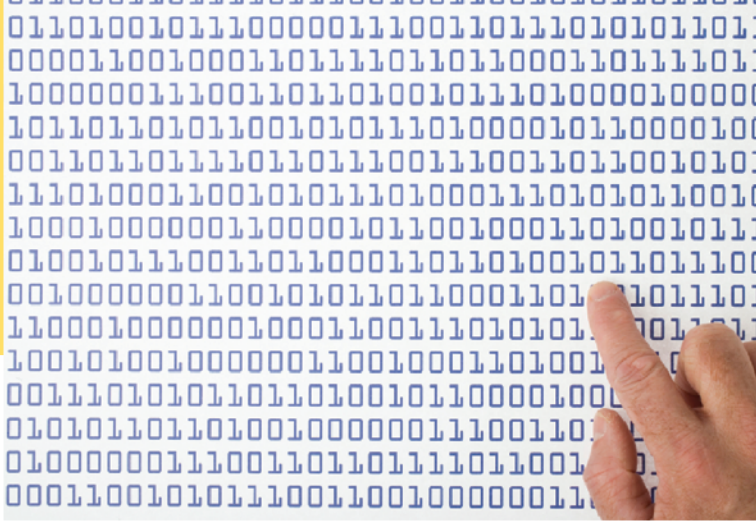
CHAPTER 21

ADVANCED INPUT/OUTPUT


© Claude Dagenais/iStockphoto.

CHAPTER GOALS

To become familiar with text and binary file formats

To learn about encryption

To understand when to use sequential and random file access

To read and write objects using serialization

CHAPTER CONTENTS

© Claude Dagenais/iStockphoto.

In this chapter you will learn more about how to write Java programs that interact with files and other sources of bytes and characters. You will learn about reading and writing text and binary data, and the differences between sequential and random access to data in a file. As an application of file processing, you will study a program that encrypts and decrypts data stored in a binary format. Next, you will see how you can use object serialization to save and load complex objects with very little effort. Finally, you will learn how to work with files and directories.

21.1 Readers, Writers, and Input/Output Streams

There are two fundamentally different ways to store data: in *text* format or *binary* format. In text format, data items are represented in human-readable form, as a sequence of *characters*. For example, in text form, the integer 12,345 is stored as the sequence of five characters:

```
'1' '2' '3' '4' '5'
```

In binary form, data items are represented in **bytes**. A byte is composed of 8 **bits** and can denote one of 256 values ($256 = 2^8$). For example, in binary format, the integer 12,345 is stored as a sequence of four bytes:

```
0 0 48 57
```

(because $12{,}345 = 48 \cdot 256 + 57$).

Input and output streams access sequences of bytes. Readers and writers access sequences of characters.

The Java library provides two sets of classes for handling input and output. **Input and output streams** handle binary data. **Readers** and **writers** handle data in text form. Figure 1 shows a part of the hierarchy of the Java classes for input and output.

Text input and output are more convenient for humans, because it is easier to produce input (just use a text editor) and it is easier to check that output is correct (just look at the output file in an editor). However, binary storage is more compact and more efficient.

The `Reader` and `Writer` classes were designed to process information in text form. You have already used the `PrintWriter` class in Chapter 11. However, for reading

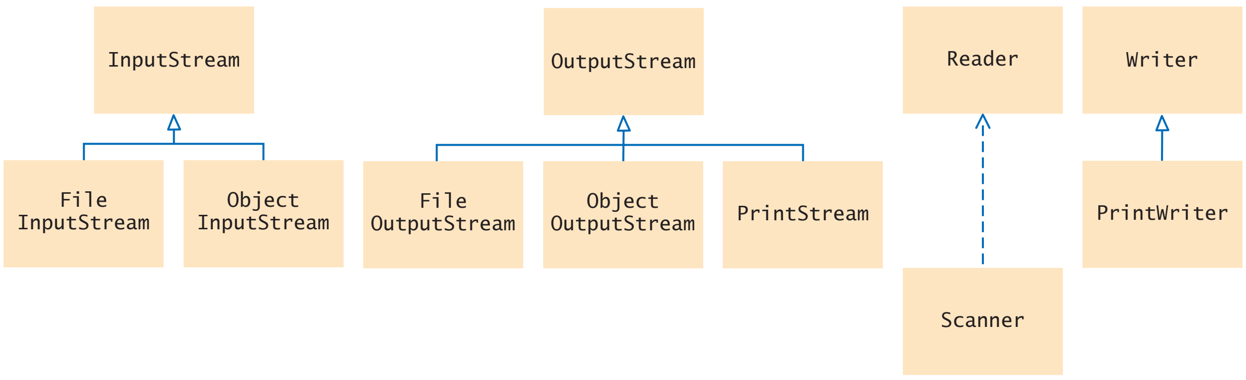


Figure 1 Java Classes for Input and Output

text, the `Scanner` class is more convenient than the `Reader` class. Internally, the `Scanner` class makes use of readers to read characters.

If you store information in binary form, as a sequence of bytes, use the `InputStream` and `OutputStream` classes and their subclasses.

Why use two sets of classes? Characters are made up of bytes, but there is some variation in how each character is represented. For example, the character `'é'` is encoded as a single byte with value `223` in the ISO-8859-1 encoding that has been commonly used in North America and Western Europe. However, in the UTF-8 encoding that is capable of encoding all Unicode characters, the character is represented by two bytes, `195` and `169`. In the UTF-16 encoding, another encoding for Unicode, the same character is encoded as `0 223`.

The `Reader` and `Writer` classes have the responsibility of converting between bytes and characters. By default, these classes use the character encoding of the computer executing the program. You can specify a different encoding in the constructor of the `Scanner` or `PrintWriter`, like this:

```
Scanner in = new Scanner(input, "UTF-8");
   // input can be a File or InputStream
PrintWriter out = new PrintWriter(output, "UTF-8");
   // output can be a File or OutputStream
```

Unfortunately, there is no way of automatically determining the character encoding that is used in a particular text. You need to know which character encoding was used when the text was written. If you only exchange data with users from the same country, then you can use the default encoding of your computer. Otherwise, it is a good idea to use the UTF-8 encoding.

You learned in Chapter 11 how to process text files. In the remainder of this chapter, we will focus on binary files.

SELF CHECK

1. Suppose you need to read an image file that contains color values for each pixel in the image. Will you use a `Reader` or an `InputStream`?
2. Special Topic 11.1 introduced the `openStream` method of the `URL` class, which returns an `InputStream`:

```
URL locator = new URL("http://bigjava.com/index.html");
InputStream in = locator.openStream();
```

Why doesn't the `URL` class provide a `Reader` instead?

Practice It Now you can try these exercises at the end of the chapter: R21.1, R21.2, R21.4.

21.2 Binary Input and Output

Use `FileInputStream` and `FileOutputStream` classes to read and write binary data from and to disk files.

In this section, you will learn how to process binary data. To read binary data from a disk file, you create a `FileInputStream` object:

```
InputStream inputStream = new FileInputStream("input.bin");
```

Similarly, you use `FileOutputStream` objects to write data to a disk file in binary form:

```
OutputStream outputStream = new FileOutputStream("output.bin");
```

The `InputStream` class has a method, `read`, to read a single byte at a time. (The `FileInputStream` class overrides this method to read the bytes from a disk file.) The

The InputStream.read method returns an integer, either –1 to indicate end of input, or a byte between 0 and 255.

InputStream.read method returns an int, not a byte, so that it can signal either that a byte has been read or that the end of input has been reached. It returns the byte read as an integer between 0 and 255 or, when it is at the end of the input, it returns –1.

You should test the return value. Only process the input when it is not –1:

```
InputStream in = . . .;
int next = in.read();
if (next != -1)
{
   Process next.  // A value between 0 and 255
}
```

The OutputStream.write method writes a single byte.

The OutputStream class has a write method to write a single byte. The parameter variable of the write method has type int, but only the lowest eight bits of the argument are written to the output stream:

```
OutputStream out = . . .;
int value = . . .; // Should be between 0 and 255
out.write(value);
```

Use the try-with-resources statement to ensure that all opened files are closed:

```
try (InputStream in = . . ., OutputStream out = . . .)
{
   Read from in.
   Write to out.
}
```

These basic methods are the only input and output methods that the input and output stream classes provide. The Java input/output package is built on the principle that each class should have a very focused responsibility. The job of an input stream is to get bytes, not to analyze them. If you want to read numbers, strings, or other objects, you have to combine the class with other classes whose responsibility it is to group individual bytes or characters together into numbers, strings, and objects. You will see an example of those classes in Section 21.4.

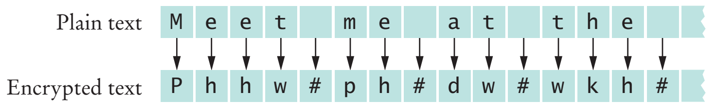
As an application of a task that involves reading and writing individual bytes, we will implement an encryption program. The program scrambles the bytes in a file so that the file is unreadable except to those who know the decryption method and the secret keyword. We will use the Caesar cipher that you saw in Section 11.3, but now we will encode all bytes. The person performing any encryption chooses an *encryption key*; here the key is a number between 1 and 255 that indicates the shift to be used in encrypting each byte. (Julius Caesar used a key of 3, replacing A with D, B with E, and so on—see Figure 2).

To decrypt, simply use the negative of the encryption key. For example, to decrypt a message encoded with a key of 3, use a key of –3.

In this program we read each value separately, encrypt it, and write the encrypted value:

```
int next = in.read();
if (next == -1)
{
   done = true;
}
else
{
   int encrypted = encrypt(next);
   out.write(encrypted);
}
```

Figure 2
The Caesar Cipher



In a more complex encryption program, you would read a block of bytes, encrypt the block, and write it out.

Try out the program on a file of your choice. You will find that the encrypted file is unreadable. In fact, because the newline characters are transformed, you may not be able to read the encrypted file in a text editor. To decrypt, simply run the program again and supply the negative of the encryption key.

section_2/CaesarCipher.java

```java
import java.io.InputStream;
import java.io.OutputStream;
import java.io.IOException;

/**
   This class encrypts files using the Caesar cipher.
   For decryption, use an encryptor whose key is the
   negative of the encryption key.
*/
public class CaesarCipher
{
   private int key;

   /**
      Constructs a cipher object with a given key.
      @param aKey the encryption key
   */
   public CaesarCipher(int aKey)
   {
      key = aKey;
   }

   /**
      Encrypts the contents of an input stream.
      @param in the input stream
      @param out the output stream
   */
   public void encryptStream(InputStream in, OutputStream out)
         throws IOException
   {
      boolean done = false;
      while (!done)
      {
         int next = in.read();
         if (next == -1)
         {
            done = true;
         }
         else
         {
            int encrypted = encrypt(next);
            out.write(encrypted);
         }
      }
   }

   /**
      Encrypts a value.
      @param b the value to encrypt (between 0 and 255)
      @return the encrypted value
   */
   public int encrypt(int b)
   {
      return (b + key) % 256;
   }
}
```

section_2/CaesarEncryptor.java

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Scanner;

/**
   This program encrypts a file, using the Caesar cipher.
*/
public class CaesarEncryptor
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      System.out.print("Input file: ");
      String inFile = in.next();
      System.out.print("Output file: ");
      String outFile = in.next();
      System.out.print("Encryption key: ");
      int key = in.nextInt();

      try (InputStream inStream = new FileInputStream(inFile);
         OutputStream outStream = new FileOutputStream(outFile))
      {
         CaesarCipher cipher = new CaesarCipher(key);
         cipher.encryptStream(inStream, outStream);
      }
      catch (IOException exception)
      {
         System.out.println("Error processing file: " + exception);
      }
   }
}
```

SELF CHECK

3. Why does the `read` method of the `InputStream` class return an `int` and not a `byte`?
4. Decrypt the following message: `Khoor/#Zruog$`.
5. Can you use the sample program from this section to encrypt a binary file, for example, an image file?

Practice It Now you can try these exercises at the end of the chapter: R21.6, E21.1, P21.1.

Common Error 21.1

Negative byte Values

The `read` method of the `InputStream` class returns –1 or the byte that was read, a value between `0` and 255. It is tempting to place this value into a variable of type `byte`, but that turns out not to be a good idea.

```
int next = in.read();
if (next != -1)
{
   byte input = (byte) next; // Not recommended
   . . .
}
```

In Java, the `byte` type is a *signed* type. There are 256 values of the `byte` type, from –128 to 127. When converting an `int` value between 128 and 255 to a byte, the result is a negative value. This can be inconvenient. For example, consider this test:

```
int next = in.read();
byte input = (byte) next;
if (input == 'é') . . .
```

The condition is never true, even if `next` is equal to the Unicode value for the `'é'` character. That Unicode value happens to be 233, but a single byte is always a value between –128 and 127.

The remedy is to work with `int` values. Don't use the `byte` type.

21.3 Random Access

Reading a file sequentially from beginning to end can be inefficient. In this section, you will learn how to directly access arbitrary locations in a file. Consider a file that contains a set of bank accounts. We want to change the balances of some of the accounts. We could read all account data into an array list, update the information that has changed, and save the data out again. But if the data set in the file is very large, we may end up doing a lot of reading and writing just to update a handful of records. It would be better if we could locate the changed information in the file and simply replace it.

In sequential file access, a file is processed one byte at a time.

This is quite different from the file access you programmed in Chapter 11, where you read from a file, starting at the beginning and reading the entire contents until you reached the end. That access pattern is called **sequential access**. Now we would like to access specific locations in a file and change only those locations. This access pattern is called **random access** (see Figure 3). There is nothing "random" about random access—the term simply means that you can read and modify any byte stored at any location in the file.

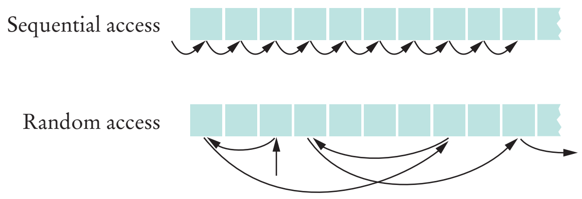


Figure 3 Sequential and Random Access

Random access allows access at arbitrary locations in the file, without first reading the bytes preceding the access location.

Only disk files support random access; the input and output streams that correspond to the keyboard and the terminal window do not. Each disk file has a special **file pointer** position. Normally, the file pointer is at the end of the file, and any output is appended to the end. However, if you move the file pointer to the middle of the file and write to the file, the output overwrites what is already there. The next read command starts reading input at the file pointer location. You can move the file pointer just beyond the last byte currently in the file but no further.

In Java, you use a `RandomAccessFile` object to access a file and move a file pointer. To open a random access file, you supply a file name and a string to specify the *open mode*. You can open a file either for reading only (`"r"`) or for reading and writing (`"rw"`). For example, the following command opens the file `bank.dat` for both reading and writing:

```
RandomAccessFile f = new RandomAccessFile("bank.dat", "rw");
```

The method call

```
f.seek(position);
```

moves the file pointer to the given position, counted from the beginning of the file. The first byte of a file has position `0`. To find out the current position of the file pointer (counted from the beginning of the file), use

```
long position = f.getFilePointer();
```

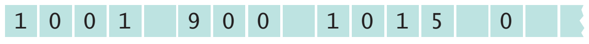
A file pointer is a position in a random access file. Because files can be very large, the file pointer is of type `long`.

Because files can be very large, the file pointer values are `long` integers. To determine the number of bytes in a file, use the `length` method:

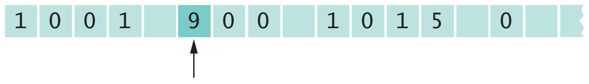
```
long fileLength = f.length();
```

In the example program at the end of this section, we use a random access file to store a set of bank accounts, each of which has an account number and a current balance. The test program lets you pick an account and deposit money into it.

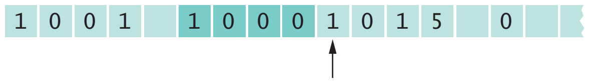
If you want to manipulate a data set in a file, you have to pay special attention to the formatting of the data. Suppose you just store the data as text. Say account 1001 has a balance of $900, and account 1015 has a balance of 0.



We want to deposit $100 into account 1001. Suppose we move the file pointer to the first character of the old value:



If we now simply write out the new value, the result is



That is not working too well. The update is overwriting the space that separates the values.

In order to be able to update values in a file, you must give each value a *fixed* size that is sufficiently large. As a result, every record in the file has the same size. This has another advantage: It is then easy to skip quickly to, say, the 50th record, without having to read the first 49 records in. Just set the file pointer to 49 × the record size.

The RandomAccessFile class reads and writes numbers in binary form.

When storing numbers in a file with fixed record sizes, it is easier to access them in binary form, rather than text form. For that reason, the RandomAccessFile class stores binary data. The readInt and writeInt methods read and write integers as four-byte quantities. The readDouble and writeDouble methods process double-precision floating-point numbers as eight-byte quantities.

```
double x = f.readDouble();
f.writeDouble(x);
```

If we save the account number as an integer and the balance as a double value, then each bank account record consists of 12 bytes: 4 bytes for the integer and 8 bytes for the double-precision floating-point value.

Now that we have determined the file layout, we can implement our random access file methods. In the program at the end of this section, we use a BankData class to translate between the random access file format and bank account objects. The size method determines the total number of accounts by dividing the file length by the size of a record.

```
public int size() throws IOException
{
   return (int) (file.length() / RECORD_SIZE);
}
```

To read the nth account in the file, the read method positions the file pointer to the offset n * RECORD_SIZE, then reads the data, and constructs a bank account object:

```
public BankAccount read(int n) throws IOException
{
   file.seek(n * RECORD_SIZE);
   int accountNumber = file.readInt();
   double balance = file.readDouble();
   return new BankAccount(accountNumber, balance);
}
```

Writing an account works the same way:

```
public void write(int n, BankAccount account) throws IOException
{
   file.seek(n * RECORD_SIZE);
   file.writeInt(account.getAccountNumber());
   file.writeDouble(account.getBalance());
}
```

The test program asks the user to enter an account number and an amount to deposit. If the account does not currently exist, it is created. The money is deposited, and then the user can choose to continue or quit. The bank data are saved and reloaded when the program is run again.

section_3/BankSimulator.java

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

/**
   This program demonstrates random access. You can access
   existing accounts and deposit money, or create new accounts.
   The accounts are saved in a random access file.
*/
public class BankSimulator
{
   public static void main(String[] args) throws IOException
   {
      try (Scanner in = new Scanner(System.in);
         BankData data = new BankData())
      {
         data.open("bank.dat");

         boolean done = false;
         while (!done)
         {
            System.out.print("Account number: ");
            int accountNumber = in.nextInt();
            System.out.print("Amount to deposit: ");
            double amount = in.nextDouble();

            int position = data.find(accountNumber);
            BankAccount account;
            if (position >= 0)
            {
               account = data.read(position);
               account.deposit(amount);
               System.out.println("New balance: " + account.getBalance());
            }
            else // Add account
            {
               account = new BankAccount(accountNumber, amount);
               position = data.size();
               System.out.println("Adding new account.");
            }
            data.write(position, account);

            System.out.print("Done? (Y/N) ");
            String input = in.next();
            if (input.equalsIgnoreCase("Y")) { done = true; }
         }
      }
   }
}
```

section_3/BankData.java

```
import java.io.IOException;
import java.io.RandomAccessFile;

/**
   This class is a conduit to a random access file
   containing bank account records.
*/
public class BankData implements AutoCloseable
{
   private RandomAccessFile file;

   public static final int INT_SIZE = 4;
   public static final int DOUBLE_SIZE = 8;
   public static final int RECORD_SIZE = INT_SIZE + DOUBLE_SIZE;

   /**
      Constructs a BankData object that is not associated with a file.
   */
   public BankData()
   {
      file = null;
   }

   /**
      Opens the data file.
      @param filename the name of the file containing bank
      account information
   */
   public void open(String filename)
         throws IOException
   {
      if (file != null) { file.close(); }
      file = new RandomAccessFile(filename, "rw");
   }

   /**
      Gets the number of accounts in the file.
      @return the number of accounts
   */
   public int size()
         throws IOException
   {
      return (int) (file.length() / RECORD_SIZE);
   }

   /**
      Closes the data file.
   */
   public void close()
         throws IOException
   {
      if (file != null) { file.close(); }
      file = null;
   }

   /**
      Reads a bank account record.
      @param n the index of the account in the data file
      @return a bank account object initialized with the file data
   */
   public BankAccount read(int n)
         throws IOException
   {
      file.seek(n * RECORD_SIZE);
      int accountNumber = file.readInt();
      double balance = file.readDouble();
      return new BankAccount(accountNumber, balance);
   }

   /**
      Finds the position of a bank account with a given number.
      @param accountNumber the number to find
      @return the position of the account with the given number,
      or –1 if there is no such account
   */
   public int find(int accountNumber)
         throws IOException
   {
      for (int i = 0; i < size(); i++)
      {
         file.seek(i * RECORD_SIZE);
         int a = file.readInt();
         if (a == accountNumber) { return i; }
            // Found a match
      }
      return -1; // No match in the entire file
   }

   /**
      Writes a bank account record to the data file.
      @param n the index of the account in the data file
      @param account the account to write
   */
   public void write(int n, BankAccount account)
         throws IOException
   {
      file.seek(n * RECORD_SIZE);
      file.writeInt(account.getAccountNumber());
      file.writeDouble(account.getBalance());
   }
}
```

Program Run

```
Account number: 1001
Amount to deposit: 100
Adding new account.
Done? (Y/N) N
Account number: 1018
Amount to deposit: 200
Adding new account.
Done? (Y/N) N
Account number: 1001
Amount to deposit: 1000
New balance: 1100.0
Done? (Y/N) Y
```

SELF CHECK

6. Why doesn't `System.out` support random access?
7. What is the advantage of the binary format for storing numbers? What is the disadvantage?

Practice It Now you can try these exercises at the end of the chapter: R21.12, R21.13, E21.6, E21.7.

21.4 Object Input and Output Streams

In the program in Section 21.3 you read `BankAccount` objects by reading each input value separately. Actually, there is an easier way. The `ObjectOutputStream` class can save entire objects out to disk. Objects are saved in binary format; hence, you use output streams and not writers. You use the `ObjectInputStream` class to read the saved objects.

For example, you can write a BankAccount object to a file as follows:

```
BankAccount b = . . .;
ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("bank.dat"));
out.writeObject(b);
```

Use object output streams to save all instance variables of an object automatically, and use object input streams to load the saved objects.

The object output stream automatically saves all instance variables of the object to the output stream. When reading the object back in, you use the readObject method of the ObjectInputStream class. That method returns an Object reference, so you need to remember the types of the objects that you saved and use a cast:

```
ObjectInputStream in = new ObjectInputStream(
      new FileInputStream("bank.dat"));
BankAccount b = (BankAccount) in.readObject();
```

The readObject method can throw a ClassNotFoundException—it is a checked exception, so you need to catch or declare it.

You can do even better than that, though. You can store a whole bunch of objects in an array list or array, or inside another object, and then save that object:

```
ArrayList<BankAccount> a = new ArrayList<>();
// Now add many BankAccount objects into a
out.writeObject(a);
```

With one instruction, you can save the array list and *all the objects that it references*. You can read all of them back with one instruction:

```
ArrayList<BankAccount> a = (ArrayList<BankAccount>) in.readObject();
```

Of course, if the Bank class contains an ArrayList of bank accounts, then you can simply save and restore a Bank object. Then its array list, and all the BankAccount objects that it contains, are automatically saved and restored as well. The sample program at the end of this section uses this approach.

This is a truly amazing capability that is highly recommended.

Objects saved to an object output stream must belong to classes that implement the Serializable interface.

To place objects of a particular class into an object output stream, the class must implement the Serializable interface. That interface has no methods, so there is no effort involved in implementing it:

```
public class BankAccount implements Serializable
{
   . . .
}
```

The process of saving objects to an object output stream is called **serialization** because each object is assigned a serial number. If the same object is saved twice, only the serial number is written out the second time. When the objects are read back in, duplicate serial numbers are restored as references to the same object.

Following is a sample program that puts serialization to work. The Bank class manages a collection of bank accounts. Both the Bank and BankAccount classes implement the Serializable interface. Run the program several times. Whenever the program exits, it saves the Bank object (and all bank account objects that the bank contains) into a file bank.dat. When the program starts again, the file is loaded, and the changes from the preceding program run are automatically reflected. However, if the file is missing (either because the program is running for the first time, or because the file was erased), then the program starts with a new bank.

section_4/Bank.java

```
import java.io.Serializable;
import java.util.ArrayList;

/**
   This bank contains a collection of bank accounts.
*/
public class Bank implements Serializable
{
   private ArrayList<BankAccount> accounts;

   /**
      Constructs a bank with no bank accounts.
   */
   public Bank()
   {
      accounts = new ArrayList<>();
   }

   /**
      Adds an account to this bank.
      @param a the account to add
   */
   public void addAccount(BankAccount a)
   {
      accounts.add(a);
   }

   /**
      Finds a bank account with a given number.
      @param accountNumber the number to find
      @return the account with the given number, or null if there
      is no such account
   */
   public BankAccount find(int accountNumber)
   {
      for (BankAccount a : accounts)
      {
         if (a.getAccountNumber() == accountNumber) // Found a match
         {
            return a;
         }
      }
      return null;   // No match in the entire array list
   }
}
```

section_4/SerialDemo.java

```
import java.io.File;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
   This program demonstrates serialization of a Bank object.
   If a file with serialized data exists, then it is loaded.
```

```java
   Otherwise the program starts with a new bank.
   Bank accounts are added to the bank. Then the bank
   object is saved.
*/
public class SerialDemo
{
   public static void main(String[] args)
         throws IOException, ClassNotFoundException
   {
      Bank firstBankOfJava;

      File f = new File("bank.dat");
      if (f.exists())
      {
         try (ObjectInputStream in = new ObjectInputStream(
               new FileInputStream(f)))
         {
            firstBankOfJava = (Bank) in.readObject();
         }
      }
      else
      {
         firstBankOfJava = new Bank();
         firstBankOfJava.addAccount(new BankAccount(1001, 20000));
         firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      }

      // Deposit some money
      BankAccount a = firstBankOfJava.find(1001);
      a.deposit(100);
      System.out.println(a.getAccountNumber() + ":" + a.getBalance());
      a = firstBankOfJava.find(1015);
      System.out.println(a.getAccountNumber() + ":" + a.getBalance());

      try (ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream(f)))
      {
         out.writeObject(firstBankOfJava);
      }
   }
}
```

Program Run

```
1001:20100.0
1015:10000.0
```

Second Program Run

```
1001:20200.0
1015:10000.0
```

SELF CHECK

8. Why is it easier to save an object with an `ObjectOutputStream` than a `RandomAccess-File`?
9. What do you have to do to the `Country` class from Section 10.1 so that its objects can be saved in an `ObjectOutputStream`?

Practice It Now you can try these exercises at the end of the chapter: R21.8, R21.9, E21.8.

HOW TO 21.1 Choosing a File Format

Many programs allow users to save their work in files. Program users can later load those files and continue working on the data, or they can send the files to other users. When you develop such a program, you need to decide how to store the data.

This How To shows you how to choose the appropriate mechanisms for saving and loading your program's data.

Step 1 Select a data format.

The most important questions you need to ask yourself concern the format to use for saving your data:

- Does your program manipulate text, such as plain text files? If so, use readers and writers.
- Does your program update portions of a file? Then use random access.
- Does your program read or write individual bytes of binary data, such as image files or encrypted data? Then use input and output streams.
- Does your program save and restore objects? Then use object output and input streams.

Step 2 Use scanners and writers if you are processing text.

Use a scanner to read the input.

```
Scanner in = new Scanner(new File("input.txt"));
```

Then use the familiar methods `next`, `nextInt`, and so on. See Chapter 11 for details.

To write output, turn the file output stream into a `PrintWriter`:

```
PrintWriter out = new PrintWriter("output.txt");
```

Then use the familiar `print` and `println` methods:

```
out.println(text);
```

Step 3 Use the `RandomAccessFile` class if you need random access.

The `RandomAccessFile` class has methods for moving a file pointer to an arbitrary position:

```
file.seek(position);
```

You can then read or write individual bytes, characters, binary integers, and binary floating-point numbers.

Step 4 Use input and output streams if you are processing bytes.

Use this loop to process input one byte at a time:

```
InputStream in = new FileInputStream("input.bin");
boolean done = false;
while (!done)
{
   int next = in.read();
   if (next == -1)
   {
      done = true;
   }
   else
   {
      Process next. // next is between 0 and 255
   }
}
```

Similarly, write the output one byte at a time:

```
try (OutputStream out = new FileOutputStream("output.bin"))
{
   . . .
   while (. . .)
   {
      int b = . . .; // b is between 0 and 255
      out.write(b);
   }
}
```

Use input and output streams only if you are ready to process the input one byte at a time. This makes sense for encryption/decryption or processing the pixels in an image.

Step 5 Use object input and output streams if you are processing objects.

First go through your classes and tag them with `implements Serializable`. You don't need to add any additional methods.

Also go to the online API documentation to check that the library classes that you are using implement the `Serializable` interface. Fortunately, many of them do. In particular, `String` and `ArrayList` are serializable.

Next, put all the objects you want to save into a class (or an array or array list—but why not make another class containing that?).

Saving all program data is a trivial operation:

```
ProgramData data = . . .;
try (ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream("program.dat")))
{
   out.writeObject(data);
}
```

Similarly, to restore the program data, you use an `ObjectInputStream` and call

```
ProgramData data = (ProgramData) in.readObject();
```

The `readObject` method can throw a `ClassNotFoundException`. You must catch or declare that exception.

21.5 File and Directory Operations

You have now seen how to read and write data in files. In the following sections, you will learn how to manipulate files and directories.

21.5.1 Paths

A `Path` describes the location of a file or directory.

To work with files and directories, you need to specify them. Up to now, you have specified file names using strings, or, when constructing a `Scanner`, you used a `File` object. The `Path` interface provides a more sophisticated way of working with names for files and directories.

You obtain a `Path` object with the static `Paths.get` method:

```
Path inputPath = Paths.get("input.txt");
Path dirPath = Paths.get("/home/myname");
```

FULL CODE EXAMPLE
Go to wiley.com/go/bjeo6code to download a program that demonstrates paths.

You combine paths with the resolve method:

```
Path fullPath = dirPath.resolve(inputPath); // The path /home/myname/input.txt
```

The argument to resolve can also be a string: dirPath.resolve("output.txt").

A path can be *absolute* (starting at the root of the file system) or *relative* (only meaningful when resolved against some other path). For example, /home/myname is an absolute path, and input.txt is relative. When you have a relative path, you can turn it into an absolute path by calling the toAbsolutePath method. For example,

```
Path absolutePath = Paths.get("input.txt").toAbsolutePath();
```

yields a path that resolves input.txt against the directory from which the program was started; something similar to /home/myname/cs2/project10/input.txt.

There are many methods for taking paths apart. Some of the most useful are:

```
Path parent = fullPath.getParent(); // The path /home/myname
Path fileName = fullPath.getFileName(); // The path input.txt
```

To get all components of a Path, you can use an enhanced for loop because the Path interface extends the Iterable<Path> interface:

```
for (Path p : fullPath)
{
   Analyze p.
}
```

In our example, p is set to home, myname, and input.txt.

Path objects are not strings. If you need to convert them to strings (for example, to change the file extension), use the toString method.

21.5.2 Creating and Deleting Files and Directories

The Files class has many static methods for working with files and directories.

The Files class has a large number of useful static methods for working with files and directories. You create an empty file or directory with

```
Files.createFile(path);
Files.createDirectory(path);
```

Here, path is a Path object. You saw in the preceding section how to obtain such an object.

If you try to create a file or directory that already exists, an exception is thrown. Also, the parent of the path must already exist. You can test whether a path exists by calling

```
boolean pathExists = Files.exists(path);
```

FULL CODE EXAMPLE
Go to wiley.com/go/bjeo6code to download a program that demonstrates creating and deleting files.

To find out whether an existing path is a file or a directory, call Files.isRegularFile or Files.isDirectory.

If you want to delete a file or an empty directory, call

```
Files.delete(path);
```

Sometimes, you want to create a temporary file or directory. You don't care what it is called, but you want a unique fresh name. Call

```
Path tempFile = Files.createTempFile(prefix, extension);
Path tempDir = Files.createTempDirectory(prefix);
```

Here, prefix is a string that helps you find the file or directory. It is located in the temporary directory of your operating system (for example, /tmp in Linux). You don't need to delete the file or directory when you are done. The operating system automatically cleans the temporary directory.

21.5.3 Useful File Operations

FULL CODE EXAMPLE
Go to wiley.com/go/bjeo6code to download a program that demonstrates file operations.

The Files class has several useful operations for common tasks. The Files.size method yields the size of a file in bytes:

```
long size = Files.size(path);
```

You can read an entire file into a list of lines (if it is a text file) or a byte array (if it is a binary file):

```
List<String> lines = Files.readAllLines(path);
byte[] bytes = Files.readAllBytes(path);
```

To write a collection of lines or an array of bytes to a file, call

```
Files.write(path, lines);
Files.write(path, bytes);
```

If you want to read a file into a single string, call

```
String contents = new String(Files.readAllBytes(path), "UTF-8");
```

Conversely, here is how you can save a string to a file:

```
Files.write(path, contents.getBytes("UTF-8"));
```

You can also obtain the lines of a text file as a stream (see Chapter 19). Then the lines are read lazily, as needed by the stream operations. You need to be careful to use a try-with-resources block so that the underlying file is closed after stream processing is complete. For example, this code snippet gets the first ten lines containing a given string and then automatically closes the file without reading more lines:

```
String target = " and ";
final int MAX_LINES = 10;
List<String> result = null;

try (Stream<String> lines = Files.lines(path))
{
   result = lines
      .filter(s -> s.contains(target))
      .limit(MAX_LINES)
      .collect(Collectors.toList());
}
```

To copy or move a file, call

```
Files.copy(fromPath, toPath);
Files.move(fromPath, toPath);
```

If fromPath doesn't exist or toPath exists, the methods throw an exception.

You can use the Files.move method to move an empty directory. But to move a nonempty directory, you need to move all descendants—see Exercise P21.11.

21.5.4 Visiting Directories

In order to read all files in a directory, call the Files.list method. It returns a Stream<Path> with the files and directories of the given directory.

The `Files.list` and `Files.walk` methods yield the children and descendants of a directory.

If you just want to have a list of files, call

```
try (Stream<Path> entries = Files.list(dirPath))
{
   List<Path> paths = entries.collect(Collectors.toList());
   Process the list paths.
}
```

FULL CODE EXAMPLE
Go to wiley.com/go/bjeo6code to download a program that demonstrates how to visit files in a directory.

If you are familiar with streams, you can instead use stream operations such as `filter`, `map`, and `collect`.

```
try (Stream<Path> entries = Files.list(dirPath))
{
   Process the stream entries.
}
```

The API uses streams because some applications have directories with huge numbers of files. A stream visits them lazily, but a list must be big enough to hold them all.

The `Files.list` method does not visit subdirectories. In order to get all descendant files and directories, call `Files.walk` instead. It also returns a `Stream<Path>`, containing the descendants in depth-first order.

SELF CHECK

10. Construct a `Path` object `downloads` with the path to the directory on your computer that contains downloaded files.
11. Call a method of the `Files` class to create a subdirectory `bigjava` in your downloads directory.
12. Given a `Path` object `p` for `/home/cay/output.txt`, how do you get a path to `/home/cay/output.bak`?
13. How can you make a backup copy of a file with path `p` before writing to it?
14. What happens in Self Check 13 if the backup file already exists? How can you overcome that problem?

Practice It Now you can try these exercises at the end of the chapter: R21.15, E21.9, E21.10.

CHAPTER SUMMARY

Describe the Java class hierarchy for handling input and output.

- Input and output streams access sequences of bytes. Readers and writers access sequences of characters.

Write programs that carry out input and output of binary data.

- Use `FileInputStream` and `FileOutputStream` classes to read and write binary data from and to disk files.
- The `InputStream.read` method returns an integer, either –1 to indicate end of input, or a byte between 0 and 255.
- The `OutputStream.write` method writes a single byte.

Describe random access and use the `RandomAccessFile` class.

- In sequential file access, a file is processed one byte at a time.
- Random access allows access at arbitrary locations in the file, without first reading the bytes preceding the access location.
- A file pointer is a position in a random access file. Because files can be very large, the file pointer is of type `long`.
- The `RandomAccessFile` class reads and writes numbers in binary form.

Use object streams to automatically read and write entire objects.

- Use object output streams to save all instance variables of an object automatically, and use object input streams to load the saved objects.
- Objects saved to an object output stream must belong to classes that implement the `Serializable` interface.

Use paths to manipulate files and directories.

- A `Path` describes the location of a file or directory.
- The `Files` class has many static methods for working with files and directories.
- The `Files.list` and `Files.walk` methods yield the children and descendants of a directory.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

- `java.io.FileInputStream`
- `java.io.FileOutputStream`
- `java.io.InputStream`
  - `close`
  - `read`
- `java.io.ObjectInputStream`
  - `readObject`
- `java.io.ObjectOutputStream`
  - `writeObject`
- `java.io.OutputStream`
  - `close`
  - `write`
- `java.io.RandomAccessFile`
  - `getFilePointer`
  - `length`
  - `readChar`
  - `readDouble`
  - `readInt`
  - `seek`
  - `writeChar`
  - `writeChars`
  - `writeDouble`
  - `writeInt`
- *`java.io.Serializable`*
- `java.nio.file.Files`
  - `copy`
  - `createDirectory`
  - `createFile`
  - `delete`
  - `exists`
  - `isDirectory`
  - `isRegularFile`
  - `lines`
  - `list`
  - `move`
  - `readAllBytes`
  - `readAllLines`
  - `walk`
- *`java.nio.file.Path`*
  - `getFileName`
  - `getParent`
  - `resolve`
- `java.nio.file.Paths`
  - `get`

REVIEW EXERCISES

■ **R21.1** What is the difference between an input stream and a reader?

■ **R21.2** Write a few lines of text to a `new PrintWriter("output1.txt", "UTF-8")` and the same text to a `new PrintWriter("output2.txt", "UTF-16")`. How do the output files differ?

■ **R21.3** How can you open a file for both reading and writing in Java?

■ ■ **R21.4** What happens if you try to write to a file reader?

■ ■ **R21.5** What happens if you try to write to a random access file that you opened only for reading? Try it out if you don't know.

■ **R21.6** How can you break the Caesar cipher? That is, how can you read a document that was encrypted with the Caesar cipher, even though you don't know the key?

■■ **R21.7** What happens if you try to save an object that is not serializable in an object output stream? Try it out and report your results.

■■ **R21.8** Of the classes in the `java.lang` and `java.io` packages that you have encountered in this book, which implement the `Serializable` interface?

■■ **R21.9** Why is it better to save an entire `ArrayList` to an object output stream instead of programming a loop that writes each element?

■ **R21.10** What is the difference between sequential access and random access?

■ **R21.11** What is the file pointer in a file? How do you move it? How do you tell the current position? Why is it a long integer?

■ **R21.12** How do you move the file pointer to the first byte of a file? To the last byte? To the exact middle of the file?

■■ **R21.13** What happens if you try to move the file pointer past the end of a file? Try it out and report your result.

■■ **R21.14** Can you move the file pointer of `System.in`?

■■ **R21.15** Paths can be absolute or relative. An absolute path name begins with a *root element* (/ on Unix-like systems, a drive letter on Windows). Look at the `Path` API to see how one can create and recognize absolute and relative paths. What does the `resolve` method do when its argument is an absolute path?

■■ **R21.16** Look up the `relativize` method in the `Path` API and explain in which sense it is the opposite of `resolve`. Give two examples when it is useful; one for files and one for directories.

■■ **R21.17** What exactly does it mean that `Files.walk` yields the results in depth-first order? Are directory children listed before or after parents? Are files listed before directories? Are either listed alphabetically? Run some experiments to find out.

PRACTICE EXERCISES

■ **E21.1** Write a program that opens a binary file and prints all ASCII characters from that file, that is, all bytes with values between 32 and 126. Print a new line after every 64 characters. What happens when you use your program with word processor documents? With Java class files?

■■ **E21.2** Write a method `public static void copy(String infile, String outfile)` that copies all bytes from one file to another, without using `Files.copy`.

■ **E21.3** Write a method that reverses all lines in a file. Read all lines, reverse each line, and write the result.

■■ **E21.4** Repeat Exercise E21.3 by using a random access file, reversing each line in place.

■■ **E21.5** Repeat Exercise E21.3, reading one line at a time and writing the reversed lines to a temporary file. Then erase the original and move the temporary file into its place.

■■ **E21.6** Modify the `BankSimulator` program in Section 21.3 so that it is possible to delete an account. To delete a record from the data file, fill the record with zeroes.

■■ **E21.7** The data file in Exercise E21.6 may end up with many deleted records that take up space. Write a program that compacts such a file, moving all active records to the beginning and shortening the file length. *Hint:* Use the `setLength` method of the `RandomAccessFile` class to truncate the file length. Look up the method's behavior in the API documentation.

■■ **E21.8** Enhance the `SerialDemo` program from Section 21.4 to demonstrate that it can save and restore a bank that contains a mixture of savings and checking accounts.

■ **E21.9** Write a method that, given a `Path` to a file that doesn't yet exist, creates all intermediate directories and the file.

■■ **E21.10** Write a method `public static void swap(Path p, Path q)` that swaps two files. *Hint:* Use a temporary file.

PROGRAMMING PROJECTS

■■ **P21.1** *Random monoalphabet cipher*. The Caesar cipher, which shifts all letters by a fixed amount, is far too easy to crack. Here is a better idea. For the key, don't use numbers but words. Suppose the keyword is `FEATHER`. Then first remove duplicate letters, yielding `FEATHR`, and append the other letters of the alphabet in reverse order. Now encrypt the letters as follows:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
F	E	A	T	H	R	Z	Y	X	W	V	U	S	Q	P	O	N	M	L	K	J	I	G	D	C	B

Write a program that encrypts or decrypts a file using this cipher. The keyword is specified with the `-k` command line option. The `-d` command line option specifies decryption. For example,

```
java Encryptor -d -k FEATHER encrypt.txt output.txt
```

decrypts a file using the keyword `FEATHER`. It is an error not to supply a keyword.

■ **P21.2** *Letter frequencies*. If you encrypt a file using the cipher of Exercise P21.1, it will have all of its letters jumbled up, and will look as if there is no hope of decrypting it without knowing the keyword. Guessing the keyword seems hopeless, too. There are just too many possible keywords. However, someone who is trained in decryption will be able to break this cipher in no time at all. The average letter frequencies of English letters are well known. The most common letter is E, which occurs about 13 percent of the time. Here are the average frequencies of English letters:

A	8%	F	3%	K	<1%	P	3%	U	3%	X	<1%
B	<1%	G	2%	L	4%	Q	<1%	V	1%	Y	2%
C	3%	H	4%	M	3%	R	8%	W	2%	Z	<1%
D	4%	I	7%	N	8%	S	6%				
E	13%	J	<1%	O	7%	T	9%				

Write a program that reads an input file and prints the letter frequencies in that file. Such a tool will help a code breaker. If the most frequent letters in an encrypted file are H and K, then there is an excellent chance that they are the encryptions of E and T.

■■ P21.3 *Vigenère cipher*. The trouble with a monoalphabetic cipher is that it can be easily broken by frequency analysis. The so-called Vigenère cipher overcomes this problem by encoding a letter into one of several cipher letters, depending on its position in the input document. Choose a keyword, for example `TIGER`. Then encode the first letter of the input text like this:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S

That is, the encoded alphabet is just the regular alphabet shifted to start at `T`, the first letter of the keyword `TIGER`. The second letter is encrypted according to this map:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H

The third, fourth, and fifth letters in the input text are encrypted using the alphabet sequences beginning with characters `G`, `E`, and `R`. Because the key is only five letters long, the sixth letter of the input text is encrypted in the same way as the first.

Write a program that encrypts or decrypts an input text using this cipher. Use command line arguments as in Exercise P21.1.

■■ P21.4 *Playfair cipher*. Another way of thwarting a simple letter frequency analysis of an encrypted text is to encrypt pairs of letters together. A simple scheme to do this is the Playfair cipher. You pick a keyword and remove duplicate letters from it. Then you fill the keyword, and the remaining letters of the alphabet, into a 5 × 5 square. (Because there are only 25 squares, I and J are considered the same letter.) Here is such an arrangement with the keyword `PLAYFAIR`:

```
P L A Y F
I R B C D
E G H K M
N O Q S T
U V W X Z
```

To encrypt a letter pair, say `AT`, look at the rectangle with corners `A` and `T`:

```
P L A Y F
I R B C D
E G H K M
N O Q S T
U V W X Z
```

The encoding of this pair is formed by looking at the other two corners of the rectangle—in this case, `FQ`. If both letters happen to be in the same row or column, such as `GO`, simply swap the two letters. Decryption is done in the same way.

Write a program that encrypts or decrypts an input text using this cipher. Use command line arguments as in Exercise P21.1.

■■■ Business P21.5 Write a program that manipulates a database of product records. Records are stored in a binary file. Each record consists of these items:

- Product name: 30 characters at two bytes each = 60 bytes
- Price: one `double` = 8 bytes
- Quantity: one `int` = 8 bytes

The program should allow the user to add a record, find a record that matches a product name, and change the price and quantity of a product by a given amount.

■■ Graphics P21.6 Implement a graphical user interface for the `BankSimulator` program in Section 21.3.

■■■ Graphics P21.7 Write a graphical application in which the user clicks on a panel to add shapes (rectangles, ellipses, cars, etc.) at the mouse click location. The shapes are stored in an array list. When the user selects File->Save from the menu, save the selection of shapes in a file. When the user selects File->Open, load in a file. Use serialization.

■■■ P21.8 Write a toolkit that helps a cryptographer decrypt a file that was encrypted using a monoalphabet cipher. A monoalphabet cipher encrypts each character separately. Examples are the Caesar cipher and the cipher in Exercise P21.1. Analyze the letter frequencies as in Exercise P21.2. Use brute force to try all Caesar cipher keys, and check the output against a dictionary file. Allow the cryptographer to enter some substitutions and show the resulting text, with the unknown characters represented as ?. Try out your toolkit by decrypting files that you get from your classmates.

■■ P21.9 In the BMP format for 24-bit true-color images, an image is stored in binary format. The start of the file has the following information:

- The position of the first pixel of the image data, starting at offset `10`
- The width of the image, starting at offset 18
- The height of the image, starting at offset 22

Each of these is a 4-byte integer value stored in *little-endian format*. That is, you need to get the integer value as $n = b_k + 256 \cdot b_{k+1} + 256^2 \cdot b_{k+2} + 256^3 \cdot b_{k+3}$, where k is the starting offset.

Each pixel of the image occupies three bytes; one each for red, green, and blue. At the end of each row are between `0` and 3 padding bytes to make the row lengths multiples of four. For example, if an image has a width of `101` pixels, there is one padding byte per row.

Using a `RandomAccessFile`, turn each pixel of such a BMP file into its negative.

■■ P21.10 Write a method `public static copyFiles(Path fromDir, Path toDir)` that copies all files (but none of the directories) from one directory to another.

■■■ P21.11 Write a method `public static copyDirectories(Path fromDir, Path toDir)` that copies all files and directories from one directory to another.

■■ P21.12 Write a method `public static clearDirectory(Path dir)` that removes all files (but none of the directories) from a directory. Be careful when testing it!

■■■ P21.13 Write a method `public static clearAllDirectories(Path dir)` that removes all files and all subdirectories from a directory. Be very careful when testing it!

■■■ P21.14 You can use a *zip file system* to look into the contents of a `.zip` file. Call

```
FileSystem zipfs = FileSystems.newFileSystem(path, null);
```

where `path` is the `Path` to the zip file. Then call `zipfs.getPath(p)` to get any `Path` inside the zip file, as if it were a path in a regular directory. You can read, copy, or move it. To inspect all files and directory trees, call `Files.walk(zipfs.getPath("/"))`.

Write a program that opens a zip file and shows the names and the first ten lines of all files in it.

ANSWERS TO SELF-CHECK QUESTIONS

1. Image data is stored in a binary format—try loading an image file into a text editor, and you won't see much text. Therefore, you should use an `InputStream`.
2. For HTML files, a reader would be useful. But URLs can also point to binary files, such as `http://horstmann.com/bigjava/duke.gif`.
3. It returns a special value of `-1` to indicate that no more input is available. If the return type were `byte`, no special value would be available that could be distinguished from a legal data value.
4. It is `"Hello, World!"`, encrypted with a key of 3.
5. Yes—the program uses input and output streams and encrypts each byte.
6. Suppose you print something, and then you call `seek(0)`, and print again to the same location. It would be difficult to reflect that behavior in the console window.
7. Advantage: The numbers use a fixed amount of storage space, making it possible to change their values without affecting surrounding data. Disadvantage: You cannot read a binary file with a text editor.
8. You can save the entire object with a single `writeObject` call. With a `RandomAccessFile`, you have to save each instance variable separately.
9. Add `implements Serializable` to the class definition.
10. The details depend on your operating system. On my computer, it is

    ```
    Path downloads = Paths.get(
       "/home/cay/Downloads");
    ```
11.
    ```
    Files.createDirectory(
       downloads.resolve("bigjava"));
    ```
12.
    ```
    Path q = Paths.get(p.toString().replace(
       ".txt", ".bak"));
    ```

    Or, if you are worried that the directory name of `p` might contain the string `.txt`,

    ```
    Path q = p.getParent().resolve(
       p.getFileName().replace(".txt", ".bak"));
    ```
13. First get a path for the backup file and change the suffix as in the preceding answer or, as is common in Linux, add a `~` to the filename:

    ```
    Path q = Paths.get(p.toString() + "~");
    ```

    Then copy the file:

    ```
    Files.copy(p, q);
    ```
14. Then the call to `copy` throws an exception. Delete the file first if it exists:

    ```
    if (Files.exists(q)) { Files.delete(q); }
    Files.copy(p, q);
    ```