Files and Streams

Advanced Programming

ICOM 4015

Lecture 16

Reading: Java Concepts Chapter 16

Chapter Goals

- To be able to read and write text files
- To become familiar with the concepts of text and binary formats
- To learn about encryption
- To understand when to use sequential and random file access
- To be able to read and write objects using serialization

Reading Text Files

- Simplest way to read text: use `Scanner` class
- To read from a disk file, construct a `FileReader`
- Then, use the `FileReader` to construct a `Scanner` object

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

Use the `Scanner` methods to read data from file

- `next`, `nextLine`, `nextInt`, and `nextDouble`

Writing Text Files

- To write to a file, construct a `PrintWriter` object

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

- If file already exists, it is emptied before the new data are written into it
- If file doesn't exist, an empty file is created

Continued...

Writing Text Files

- Use `print` and `println` to write into a `PrintWriter`:

```
out.println(29.95);
out.println(new Rectangle(5, 10, 15, 25));
out.println("Hello, World!");
```

- You must close a file when you are done processing it:

```
out.close();
```

- Otherwise, not all of the output may be written to the disk file

A Sample Program

- **Reads all lines of a file and sends them to the output file, preceded by line numbers**
- **Sample input file:**

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```


Continued…

A Sample Program

- **Program produces the output file:**

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

- **Program can be used for numbering Java source files**

File LineNumberer.java

```
01: import java.io.FileReader;
02: import java.io.IOException;
03: import java.io.PrintWriter;
04: import java.util.Scanner;
05:
06: public class LineNumberer
07: {
08:    public static void main(String[] args)
09:    {
10:       Scanner console = new Scanner(System.in);
11:       System.out.print("Input file: ");
12:       String inputFileName = console.next();
13:       System.out.print("Output file: ");
14:       String outputFileName = console.next();
15:
16:       try
17:       {
```

Continued…

File LineNumberer.java

```
18:          FileReader reader = new FileReader(inputFileName);
19:          Scanner in = new Scanner(reader);
20:          PrintWriter out = new PrintWriter(outputFileName);
21:          int lineNumber = 1;
22:
23:          while (in.hasNextLine())
24:          {
25:             String line = in.nextLine();
26:             out.println("/* " + lineNumber + " */ " + line);
27:             lineNumber++;
28:          }
29:
30:          out.close();
31:       }
32:       catch (IOException exception)
33:       {
```

Continued…

File LineNumberer.java

```
34:             System.out.println("Error processing file:"
                   + exception);
35:         }
36:     }
37: }
```

Self Check

1. What happens when you supply the same name for the input and output files to the `LineNumberer` program?
2. What happens when you supply the name of a nonexistent input file to the `LineNumberer` program?

Answers

1. When the `PrintWriter` object is created, the output file is emptied. Sadly, that is the same file as the input file. The input file is now empty and the while loop exits immediately.
2. The program catches a `FileNotFoundException`, prints an error message, and terminates.

File Dialog Boxes

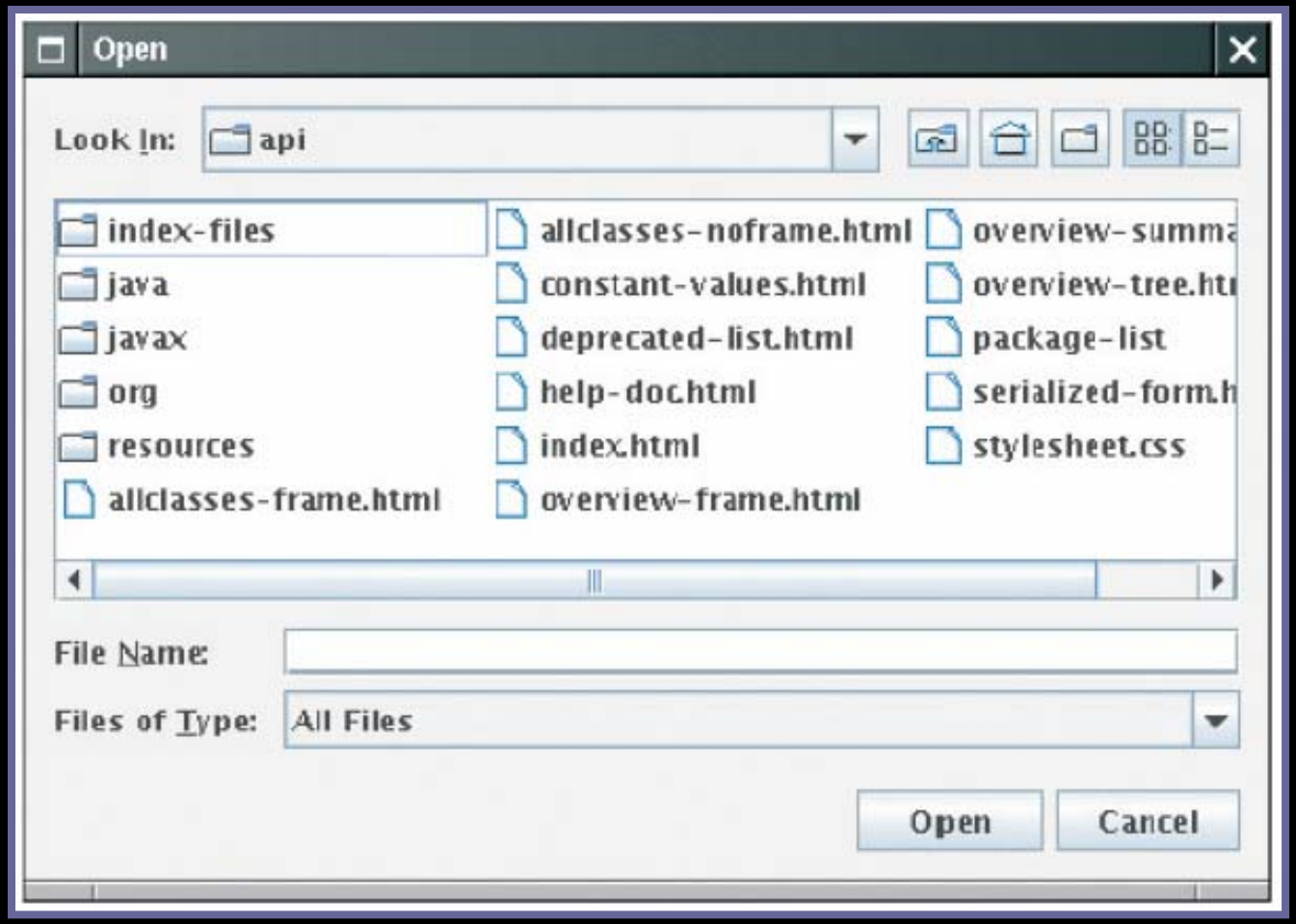


Figure 1: A `JFileChooser` Dialog Box

File Dialog Boxes

```
JFileChooser chooser = new JFileChooser();
FileReader in = null;
if (chooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION)
   { File selectedFile = chooser.getSelectedFile();
   reader = new FileReader(selectedFile);
   . . .
}
```

Text and Binary Formats

- **Two ways to store data:**
  - Text format
  - Binary format

Text Format

- **Human-readable form**
- **Sequence of characters**
  - Integer 12,345 stored as characters '1' '2' '3' '4' '5'
- **Use `Reader` and `Writer` and their subclasses to process input and output**
- **To read:**

```
FileReader reader = new FileReader("input.txt");
```

- **To write**

```
FileWriter writer = new FileWriter("output.txt");
```

Binary Format

- Data items are represented in *bytes*
- Integer 12,345 stored as a sequence of four bytes `0 0 48 57`
- Use `InputStream` and `OutputStream` and their subclasses
- More compact and more efficient

Continued…

Binary Format

- **To read:**

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

- **To write**

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

Reading a Single Character from a File in Text Format

- **Use `read` method of `Reader` class to read a single character**
  - returns the next character as an `int`
  - or the integer `-1` at end of file

```
Reader reader = . . .;
int next = reader.read();
char c;
if (next != -1)
   c = (char) next;
```

Reading a Single Character from a File in Text Format

- **Use `read` method of `InputStream` class to read a single byte**
  - returns the next byte as an `int`
  - or the integer `-1` at end of file

```
InputStream in = . . .;
int next = in.read();
byte b; if
(next != -1)
   b = (byte) next;
```

Text and Binary Format

- Use `write` method to write a single character or byte
- `read` and `write` are the only input and output methods provided by the file input and output classes
- Java stream package principle: each class should have a very focused responsibility

Continued…

Text and Binary Format

- Job of `FileInputStream`: interact with files and get bytes
- To read numbers, strings, or other objects, combine class with other classes

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. Why do the read methods of the `Reader` and `InputStream` classes return an `int` and not a `char` or `byte`?

Answers

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. They return a special value of -1 to indicate that no more input is available. If the return type had been `char` or `byte`, no special value would have been available that is distinguished from a legal data value.

An Encryption Program

- **File encryption**
  - To scramble it so that it is readable only to those who know the encryption method and secret keyword
- **To use Caesar cipher**
  - Choose an encryption key–a number between 1 and 25
  - Example: If the key is 3, replace A with D, B with E, . . .

An Encryption Program

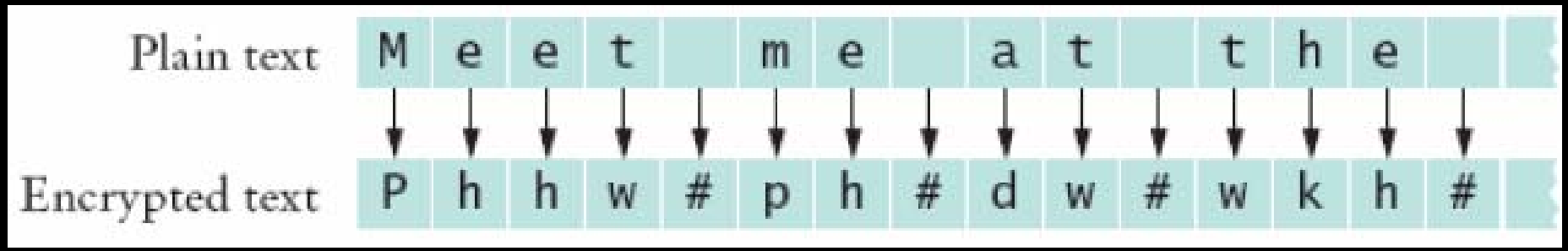


Figure 2:
The Caesar Cipher

- **To decrypt, use the negative of the encryption key**

To Encrypt Binary Data

```
int next = in.read();
if (next == -1) done = true;
else
{
   byte b = (byte) next; //call the method to encrypt the byte
   byte c = encrypt(b);
   out.write(c);
}
```

File Encryptor.java

```
01: import java.io.File;
02: import java.io.FileInputStream;
03: import java.io.FileOutputStream;
04: import java.io.InputStream;
05: import java.io.OutputStream;
06: import java.io.IOException;
07:
08: /**
09:    An encryptor encrypts files using the Caesar cipher.
10:    For decryption, use an encryptor whose key is the
11:    negative of the encryption key.
12: */
13: public class Encryptor
14: {
15:    /**
16:       Constructs an encryptor.
17:       @param aKey the encryption key
18:    */
```

Continued…

File Encryptor.java

```
19:    public Encryptor(int aKey)
20:    {
21:       key = aKey;
22:    }
23:
24:    /**
25:       Encrypts the contents of a file.
26:       @param inFile the input file
27:       @param outFile the output file
28:    */
29:    public void encryptFile(String inFile, String outFile)
30:          throws IOException
31:    {
32:       InputStream in = null;
33:       OutputStream out = null;
34:
35:       try
36:       {
```

Continued…

File Encryptor.java

```
37:          in = new FileInputStream(inFile);
38:          out = new FileOutputStream(outFile);
39:          encryptStream(in, out);
40:       }
41:       finally
42:       {
43:          if (in != null) in.close();
44:          if (out != null) out.close();
45:       }
46:    }
47:
48:    /**
49:       Encrypts the contents of a stream.
50:       @param in the input stream
51:       @param out the output stream
52:    */
```

Continued…

File Encryptor.java

```
53:    public void encryptStream(InputStream in, OutputStream out)
54:          throws IOException
55:    {
56:       boolean done = false;
57:       while (!done)
58:       {
59:          int next = in.read();
60:          if (next == -1) done = true;
61:          else
62:          {
63:             byte b = (byte) next;
64:             byte c = encrypt(b);
65:             out.write(c);
66:          }
67:       }
68:    }
69:
```

Continued…

File Encryptor.java

```
70:    /**
71:       Encrypts a byte.
72:       @param b the byte to encrypt
73:       @return the encrypted byte
74:    */
75:    public byte encrypt(byte b)
76:    {
77:       return (byte) (b + key);
78:    }
79:
80:    private int key;
81: }
```

File EncryptorTester.java

```
01: import java.io.IOException;
02: import java.util.Scanner;
03:
04: /**
05:    A program to test the Caesar cipher encryptor.
06: */
07: public class EncryptorTester
08: {
09:    public static void main(String[] args)
10:    {
11:       Scanner in = new Scanner(System.in);
12:       try
13:       {
14:          System.out.print("Input file: ");
15:          String inFile = in.next();
16:          System.out.print("Output file: ");
17:          String outFile = in.next();
```

Continued…

File EncryptorTester.java

```
18:          System.out.print("Encryption key: ");
19:          int key = in.nextInt();
20:          Encryptor crypt = new Encryptor(key);
21:          crypt.encryptFile(inFile, outFile);
22:       }
23:       catch (IOException exception)
24:       {
25:          System.out.println("Error processing file: "
                + exception);
26:       }
27:    }
28: }
29:
30:
```

Self Test

1. **Decrypt the following message:**
   `Khoor/#Zruog$.`
2. **Can you use this program to encrypt a binary file, for example, an image file?**

Answers

1. It is "Hello, World!", encrypted with a key of 3.
2. Yes–the program uses streams and encrypts each byte.

Public Key Encryption

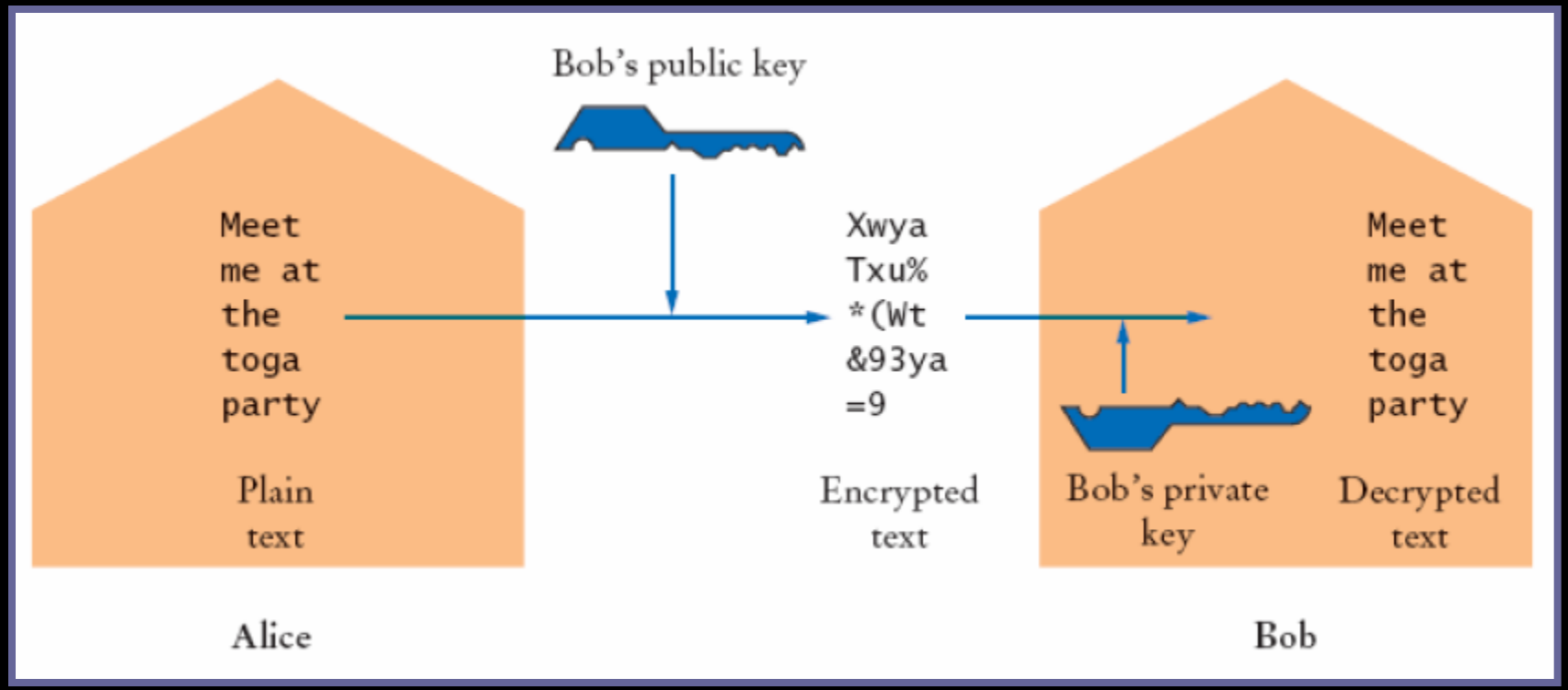


Figure 3:
Public Key Encryption

Random Access vs. Sequential Access

- **Sequential access**
  - A file is processed a byte at a time
  - It can be inefficient
- **Random access**
  - Allows access at arbitrary locations in the file
  - Only disk files support random access
    - `System.in` and `System.out` do not
  - Each disk file has a special file pointer position
    - You can read or write at the position where the pointer is

Continued…

Random Access vs. Sequential Access

- Each disk file has a special file pointer position
  - You can read or write at the position where the pointer is

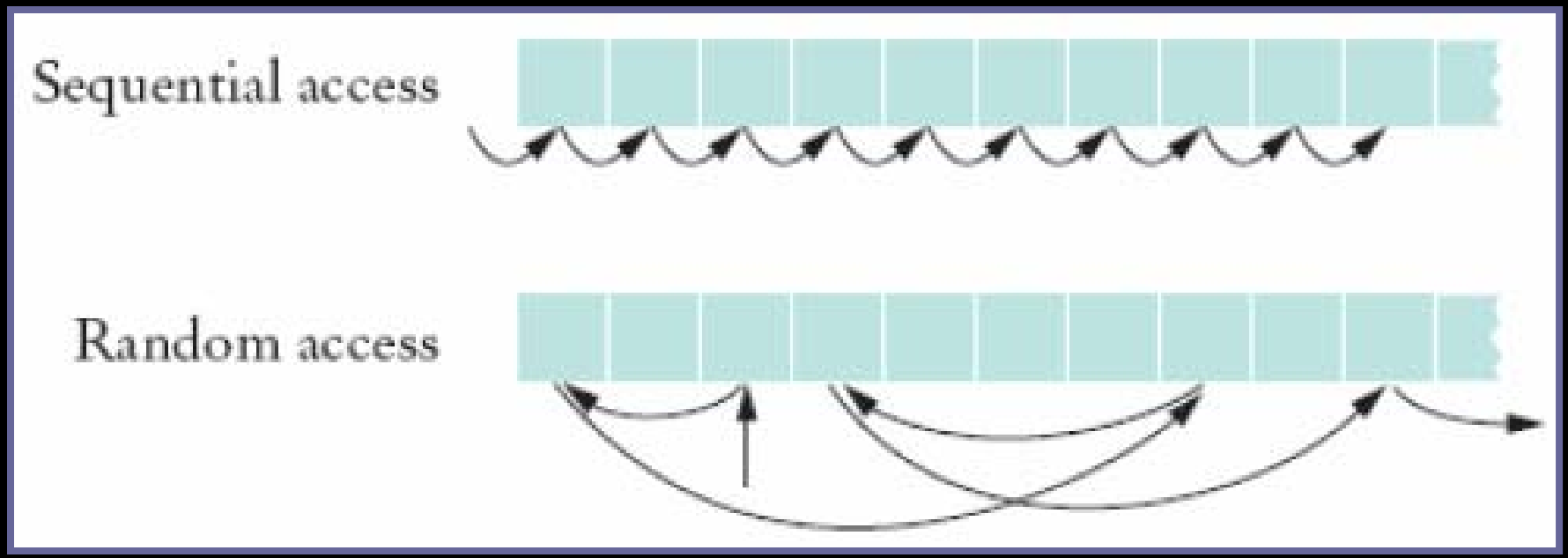


Figure 4:
Random and Sequential Access

RandomAccessFile

- **You can open a file either for**
  - Reading only ("r")
  - Reading and writing ("rw")

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

- **To move the file pointer to a specific byte**

```
f.seek(n);
```


Continued…

RandomAccessFile

- To get the current position of the file pointer.

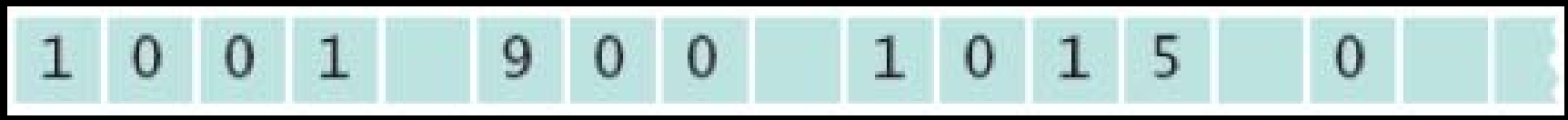
```
long n = f.getFilePointer();
      // of type "long" because files can be very large
```

- To find the number of bytes in a file long

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

A Sample Program

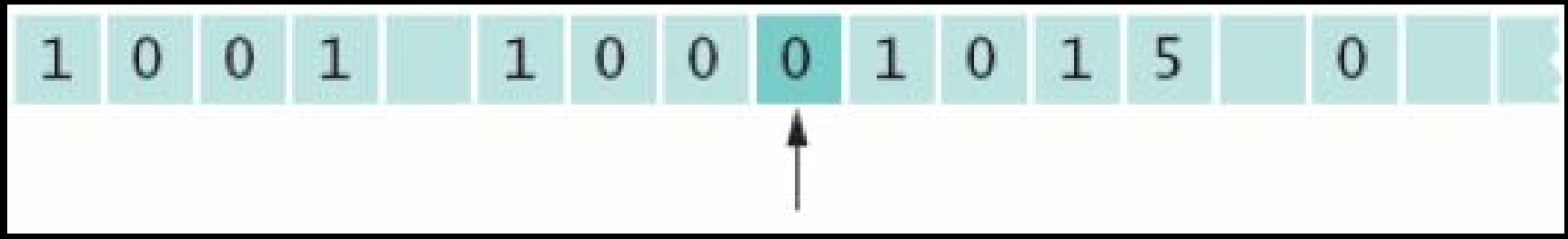
- **Use a random access file to store a set of bank accounts**
- **Program lets you pick an account and deposit money into it**
- **To manipulate a data set in a file, pay special attention to data formatting**
  - Suppose we store the data as text
    Say account 1001 has a balance of $900, and account 1015 has a balance of 0



A Sample Program

We want to deposit $100 into account 1001

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



A Sample Program

- **Better way to manipulate a data set in a file:**
  - Give each value a fixed size that is sufficiently large
  - Every record has the same size
  - Easy to skip quickly to a given record
  - To store numbers, it is easier to store them in binary format

A Sample Program

- `RandomAccessFile` class stores binary data
- `readInt` and `writeInt` read/write integers as four-byte quantities
- `readDouble` and `writeDouble` use 8 bytes

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


Continued…

A Sample Program

- To find out how many bank accounts are in the file

```
public int size() throws IOException
{
   return (int) (file.length() / RECORD_SIZE);
         // RECORD_SIZE is 12 bytes:
         // 4 bytes for the account number and
         // 8 bytes for the balance }
```

A Sample Program

- To read the nth account in the file

```
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);
}
```

A Sample Program

- To write the nth account in the file

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

File BankDatatester.java

```
01: import java.io.IOException;
02: import java.io.RandomAccessFile;
03: import java.util.Scanner;
04:
05: /**
06:    This program tests random access. You can access existing
07:    accounts and deposit money, or create new accounts. The
08:    accounts are saved in a random access file.
09: */
10: public class BankDataTester
11: {
12:    public static void main(String[] args)
13:          throws IOException
14:    {
15:       Scanner in = new Scanner(System.in);
16:       BankData data = new BankData();
17:       try
```

Continued…

File BankDatatester.java

```
18:         {
19:            data.open("bank.dat");
20:
21:            boolean done = false;
22:            while (!done)
23:            {
24:               System.out.print("Account number: ");
25:               int accountNumber = in.nextInt();
26:               System.out.print("Amount to deposit: ");
27:               double amount = in.nextDouble();
28:
29:               int position = data.find(accountNumber);
30:               BankAccount account;
31:               if (position >= 0)
32:               {
33:                  account = data.read(position);
34:                  account.deposit(amount);
```

Continued…

File BankDatatester.java

```
35:                System.out.println("new balance="
36:                      + account.getBalance());
37:             }
38:             else // Add account
39:             {
40:                account = new BankAccount(accountNumber,
41:                      amount);
42:                position = data.size();
43:                System.out.println("adding new account");
44:             }
45:             data.write(position, account);
46:
47:             System.out.print("Done? (Y/N) ");
48:             String input = in.next();
49:             if (input.equalsIgnoreCase("Y")) done = true;
50:          }
51:       }
```

Continued…

File BankDatatester.java

```
52:         finally
53:         {
54:            data.close();
55:         }
56:     }
57: }
58:
59:
60:
61:
62:
63:
64:
65:
66:
67:
68:
69:
```

File BankData.java

```
001: import java.io.IOException;
002: import java.io.RandomAccessFile;
003:
004: /**
005:    This class is a conduit to a random access file
006:    containing savings account data.
007: */
008: public class BankData
009: {
010:    /**
011:       Constructs a BankData object that is not associated
012:       with a file.
013:    */
014:    public BankData()
015:    {
016:       file = null;
017:    }
```

Continued…

File BankData.java

```
018:
019:    /**
020:       Opens the data file.
021:       @param filename the name of the file containing savings
022:       account information
023:    */
024:    public void open(String filename)
025:          throws IOException
026:    {
027:       if (file != null) file.close();
028:       file = new RandomAccessFile(filename, "rw");
029:    }
030:
031:    /**
032:       Gets the number of accounts in the file.
033:       @return the number of accounts
034:    */
```

Continued…

File BankData.java

```
035:    public int size()
036:          throws IOException
037:    {
038:       return (int) (file.length() / RECORD_SIZE);
039:    }
040:
041:    /**
042:       Closes the data file.
043:    */
044:    public void close()
045:          throws IOException
046:    {
047:       if (file != null) file.close();
048:       file = null;
049:    }
050:
```

Continued…

File BankData.java

```
051:    /**
052:       Reads a savings account record.
053:       @param n the index of the account in the data file
054:       @return a savings account object initialized with
           // the file data
055:    */
056:    public BankAccount read(int n)
057:          throws IOException
058:    {
059:       file.seek(n * RECORD_SIZE);
060:       int accountNumber = file.readInt();
061:       double balance = file.readDouble();
062:       return new BankAccount(accountNumber, balance);
063:    }
064:
065:    /**
066:       Finds the position of a bank account with a given
           // number
```

Continued…

File BankData.java

```
067:          @param accountNumber the number to find
068:          @return the position of the account with the given
                 // number,
069:          or -1 if there is no such account
070:       */
071:       public int find(int accountNumber)
072:             throws IOException
073:       {
074:          for (int i = 0; i < size(); i++)
075:          {
076:             file.seek(i * RECORD_SIZE);
077:             int a = file.readInt();
078:             if (a == accountNumber) // Found a match
079:                return i;
080:          }
081:          return -1; // No match in the entire file
082:       }
```

Continued…

File BankData.java

```
083:
084:    /**
085:       Writes a savings account record to the data file
086:       @param n the index of the account in the data file
087:       @param account the account to write
088:    */
089:    public void write(int n, BankAccount account)
090:          throws IOException
091:    {
092:       file.seek(n * RECORD_SIZE);
093:       file.writeInt(account.getAccountNumber());
094:       file.writeDouble(account.getBalance());
095:    }
096:
097:    private RandomAccessFile file;
098:
```

Continued…

File BankData.java

```
099:    public static final int INT_SIZE = 4;
100:    public static final int DOUBLE_SIZE = 8;
101:    public static final int RECORD_SIZE
102:          = INT_SIZE + DOUBLE_SIZE;
103: }
```

Output

```
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

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

Answers

1. 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.
2. 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.

Object Streams

- `ObjectOutputStream` class can save a entire objects to disk
- `ObjectInputStream` class can read objects back in from disk
- Objects are saved in binary format; hence, you use streams

Writing a BankAccount Object to a File

- The object output stream saves all instance variables

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

Reading a `BankAccount` Object From a File

- `readObject` returns an `Object` reference
- 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();
```


Continued…

Reading a `BankAccount` Object From a File

- `readObject` method can throw a `ClassNotFoundException`
- It is a checked exception
- You must catch or declare it

Write and Read an `ArrayList` to a File

- **Write**

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

- **Read**

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

Serializable

- Objects that are written to an object stream must belong to a class that implements the `Serializable` interface.

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

- `Serializable` interface has no methods.

Continued…

Serializable

- **Serialization: process of saving objects to a stream**
  - Each object is assigned a serial number on the stream
  - If the same object is saved twice, only serial number is written out the second time
  - When reading, duplicate serial numbers are restored as references to the same object

File Serialtester.java

```
01: import java.io.File;
02: import java.io.IOException;
03: import java.io.FileInputStream;
04: import java.io.FileOutputStream;
05: import java.io.ObjectInputStream;
06: import java.io.ObjectOutputStream;
07:
08: /**
09:    This program tests serialization of a Bank object.
10:    If a file with serialized data exists, then it is
11:    loaded. Otherwise the program starts with a new bank.
12:    Bank accounts are added to the bank. Then the bank
13:    object is saved.
14: */
15: public class SerialTester
16: {
```

Continued…

File Serialtester.java

```
17:    public static void main(String[] args)
18:          throws IOException, ClassNotFoundException
19:    {
20:       Bank firstBankOfJava;
21:
22:       File f = new File("bank.dat");
23:       if (f.exists())
24:       {
25:          ObjectInputStream in = new ObjectInputStream
26:                (new FileInputStream(f));
27:          firstBankOfJava = (Bank) in.readObject();
28:          in.close();
29:       }
30:       else
31:       {
32:          firstBankOfJava = new Bank();
33:          firstBankOfJava.addAccount(new
                BankAccount(1001, 20000));
```

Continued…

File Serialtester.java

```
34:             firstBankOfJava.addAccount(new
                      BankAccount(1015, 10000));
35:         }
36:
37:         // Deposit some money
38:         BankAccount a = firstBankOfJava.find(1001);
39:         a.deposit(100);
40:         System.out.println(a.getAccountNumber()
                  + ":" + a.getBalance());
41:         a = firstBankOfJava.find(1015);
42:         System.out.println(a.getAccountNumber()
                  + ":" + a.getBalance());
43:
44:         ObjectOutputStream out = new ObjectOutputStream
45:               (new FileOutputStream(f));
46:         out.writeObject(firstBankOfJava);
47:         out.close();
48:     }
49: }
```

Continued…

Output

First Program Run

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

Second Program Run

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

Self Check

1. Why is it easier to save an object with an `ObjectOutputStream` than a `RandomAccessFile`?
2. What do you have to do to the `Coin` class so that its objects can be saved in an `ObjectOutputStream`?

Answers

1. You can save the entire object with a single `writeObject` call. With a `RandomAccessFile`, you have to save each field separately.
2. Add implements `Serializable` to the class definition.