STREAMS

(fluxos)

Objetivos

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

Two Ways to Store Data

- Text format
- Binary format

Text Format

- **Human-readable form**
- **Sequence of characters**
  - **Integer 12345 stored as characters "1" "2" "3" "4" "5"**
- **Use `Reader` and `Writer` and their subclasses**

Binary Format

- **More compact and efficient**
- **Integer 12345 stored as 00 00 48 57**
- **Use** `InputStream` and `OutputStream` **and their subclasses**

To Read Text Data From a Disk File

- Create a `FileReader`
- Use its read method to read a single character
  - returns the next char as an int
  - or the integer -1 at end of input
- Test for -1
- If not -1, cast to char
- Close the file when done

Code to Read a Character From Disk

```
FileReader reader = new FileReader("input.txt");
int next = reader.read() ;
char c;
if (next != -1)  c = (char)next();
 ... reader.close()
```

Code to Write a Character to Disk

```
FileWriter writer = new FileWriter("output.txt");
...
char c='';
...
writer.write(c); ...
write.close();
```

To Read Binary Data From a Disk File

- Create a `FileInputStream`
- Use its `read` method to read a single byte
  - returns the next byte as an int
  - or the integer -1 at end of input
- Test for -1
- If not -1, cast to byte
- Close the file when done

Code to Read a Byte From Disk

```
FileInputStream inputStream = new
FileInputStream("input.dat");
int next = inputStream.read();
byte b;
if (next != -1)
  b = (byte)next;
 ...
inputStream.close();
```

Code to Write a Byte to Disk

```
FileOutputStream output = new
    FileOutputStream("output.txt");
...
byte b = 0;
...
output.write(b);
...
write.close();
```

Writing Text Files

- Use a `PrintWriter` to
  - Break up numbers and strings into individual characters
  - Send them one at a time to a `FileWriter`
- Code to create a `PrintWriter`

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

Writing Text Files

Use `print` and `println` to print numbers, objects, or strings

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

Reading Text Files

- Use a `BufferedReader`
  - Reads a character at a time from a `FileReader`
  - Assembles the characters into a line and returns it
- To convert the strings to numbers use
  - Integer.parseInt
  - Integer.parseDouble

Reading Text Files

Code to read a text file

```
FileReader reader =
                  new FileReader ("input.txt");
BufferedReader in =
                  new BufferedReader(reader);
String inputLine = in.ReadLine();
double x = Double.parseDouble(inputLine);
```

`File` Class

- `File` class describes disk files and directories
- Create a `File` object

  File inputFile = new File("input.txt");

`File` Class

- **Some File methods**
  - `delete`
  - `renameTo`
  - `exists`
- **Construct a `FileReader` from a `File` object**

```
FileReader reader = new FileReader(inputFile);
```

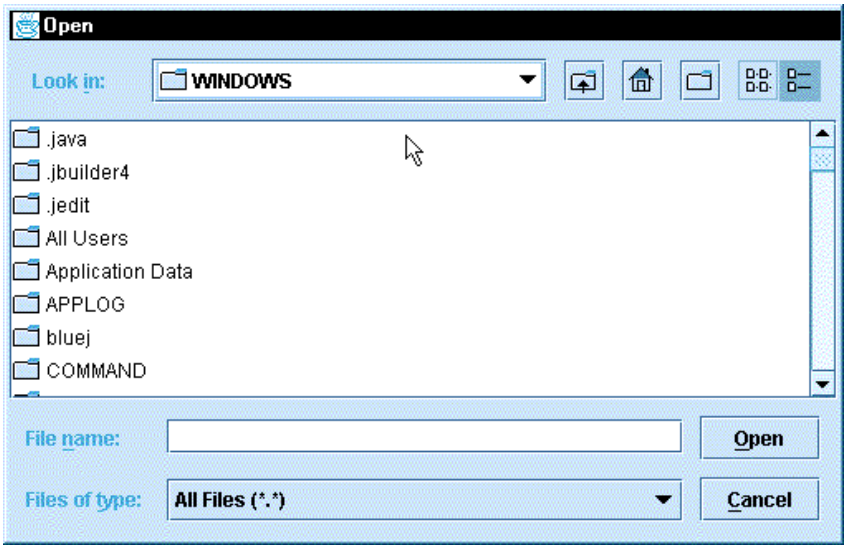
`File` Dialogs

- Use `JFileChooser` to let a user supply a file name through a file dialog
- Construct a file chooser object
- Call its `showOpenDialog` or `showSaveDialog` method
- Specify null or the user interface component over which to pop up the dialog

`File` Dialogs

- If the user chooses a file:
  `JFileChooser.APPROVE_OPTION` is returned
- If the user cancels the selection:
  `JFileChooser.CANCEL_OPTION` is returned
- If a file is chosen, use `GetSelectedFile` method
  to obtain a `File` object describing the file

A `JFileChooser` Dialog



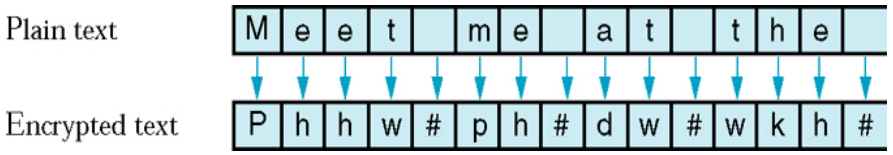
Code to Use a `JFileChooser`

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

Encryption

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

Caesar Cipher



To Encrypt Binary Data

```
int next - in.read();
if (next == -1)
        done = true;
else
{
        byte b = (byte)next;
        byte c = encrypt(b); // call the method to
                             // encrypt the byte
        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:    */
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(File inFile, File outFile)
30:       throws IOException
31:    {
32:       InputStream in = null;
33:       OutputStream out = null;
34:
35:       try
36:       {
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:    */
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:
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 EncrytorTest.java

```
01: import java.io.File;
02: import java.io.IOException;
03: import javax.swing.JFileChooser;
04: import javax.swing.JOptionPane;
05:
06: /**
07:    A program to test the Caesar cipher encryptor.
08: */
09: public class EncryptorTest
10: {
11:    public static void main(String[] args)
12:    {
13:     try
14:     {
15:     JFileChooser chooser = new JFileChooser();
16:     if (chooser.showOpenDialog(null) != JFileChooser.APPROVE_OPTION) System.exit(0);
17:
```

```
18:          File inFile = chooser.getSelectedFile();
19:          if (chooser.showSaveDialog(null) != JFileChooser.APPROVE_OPTION) System.exit(0)
20:          File outFile = chooser.getSelectedFile();
21:          String input = JOptionPane.showInputDialog("Key");
22:          int key = Integer.parseInt(input);
23:          Encryptor crypt = new Encryptor(key);
24:          crypt.encryptFile(inFile, outFile);
25:       }
26:       catch (NumberFormatException exception)
27:       {
28:          System.out.println("Key must be an integer: " + exception);
29:       }
30:       catch (IOException exception)
31:       {
32:          System.out.println("Error processing file: " + exception);
33:       }
34:       System.exit(0);
35:    }
36: }
37:
```

Command Line Arguments

- Launch a program from the command line
- Specify arguments after the program name
- In the program, access the strings by processing the `args` parameter of the `main method`

Example of Command Line Arguments

– **Java MyProgram -d file.txt**

```
class MyProgram
{
     public static void main(String[] args)
    {
        ...
    }
}
```

args[0] "-d"

args[1] "file.txt"

File Crypt.java

```
01: import java.io.File;
02: import java.io.IOException;
03:
04: /**
05:    A program to run the Caesar cipher encryptor with
06:    command line arguments.
07: */
08: public class Crypt
09: {
10:    public static void main(String[] args)
11:    {
12:       boolean decrypt = false;
13:       int key = DEFAULT_KEY;
14:       File inFile = null;
15:       File outFile = null;
16:
17:       if (args.length < 2 || args.length > 4) usage();
```

```
18:
19:       try
20:       {
21:          for (int i = 0; i < args.length; i++)
22:          {
23:             if (args[i].charAt(0) == '-')
24:             {
25:                // it is a command line option
26:                char option = args[i].charAt(1);
27:                if (option == 'd')
28:                   decrypt = true;
29:                else if (option == 'k')
30:                   key = Integer.parseInt(args[i].substring(2));
31:             }
32:             else
33:             {
34:                // it is a file name
35:                if (inFile == null)
36:                   inFile = new File(args[i]);
37:                else if (outFile == null)
```

```
38:                outFile = new File(args[i]);
39:             else usage();
40:           }
41:         }
42:         if (decrypt) key = -key;
43:         Encryptor crypt = new Encryptor(key);
44:         crypt.encryptFile(inFile, outFile);
45:       }
46:       catch (NumberFormatException exception)
47:       {
48:         System.out.println("Key must be an integer: " + exception);
49:       }
50:       catch (IOException exception)
51:       {
52:         System.out.println("Error processing file: " + exception);
53:       }
54:    }
55:
56:    /**
57:       Prints a message describing proper usage and exits.
```

```
58:    */
59:    public static void usage()
60:    {
61:       System.out.println
62:          ("Usage: java Crypt [-d] [-kn] infile outfile");
63:       System.exit(1);
64:    }
65:
66:    public static final int DEFAULT_KEY = 3;
67: }
```

Object Streams

- `ObjectOutputStream` class can save 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 `Coin` Object to a File

```
Coin c = ...;
ObjectOutputStream out =
    new ObjectOutputStream
     (new FileOutputStream("coins.dat"));
out.writeObject(c);
```

Reading a `Coin` Object from a File

```
ObjectInputStream in =
    new ObjectInputStream
       (new FilInputStream("coins.dat"));
Coin c = (Coin)in.readObject();
```

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

Write and Read an ArrayList to a File

- Write

```
ArrayList a = new ArrayList();
//add many objects to the array list
out.writeObject(a);
```

- Read

```
ArrayList a = (ArrayList)in.readObject();
```

Serializable

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

```
class Coin implements Serializable
{
   ...
}
```

- `Serializable` **interface has no methods.**

File PurseTest.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: import javax.swing.JOptionPane;
08:
09: /**
10:    This program tests serialization of a Purse object.
11:    If a file with serialized purse data exists, then it is
12:    loaded. Otherwise the program starts with a new purse.
13:    More coins are added to the purse. Then the purse data
14:    are saved.
15: */
16: public class PurseTest
17: {
```

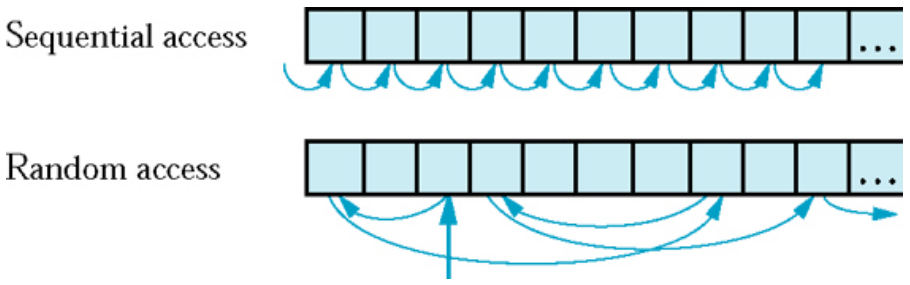
```
18:   public static void main(String[] args)
19:      throws IOException, ClassNotFoundException
20:   {
21:      Purse myPurse;
22:
23:      File f = new File("purse.dat");
24:      if (f.exists())
25:      {
26:         ObjectInputStream in = new ObjectInputStream
27:            (new FileInputStream(f));
28:         myPurse = (Purse)in.readObject();
29:         in.close();
30:      }
31:      else myPurse = new Purse();
32:
33:      // add coins to the purse
34:      myPurse.add(new Coin(NICKEL_VALUE, "nickel"));
35:      myPurse.add(new Coin(DIME_VALUE, "dime"));
36:      myPurse.add(new Coin(QUARTER_VALUE, "quarter"));
37:
```

```
38:      double totalValue = myPurse.getTotal();
39:      System.out.println("The total is " + totalValue);
40:
41:      ObjectOutputStream out = new ObjectOutputStream
42:         (new FileOutputStream(f));
43:      out.writeObject(myPurse);
44:      out.close();
45:   }
46:
47:   private static double NICKEL_VALUE = 0.05;
48:   private static double DIME_VALUE = 0.1;
49:   private static double QUARTER_VALUE = 0.25;
50: }
```

Random and Sequential Access

- Sequential access
  - A file is processed a byte at a time.
- Random access
  - Allows access at arbitrary locations in the file

Random and Sequential Access



- **RandomAccessFile**
  - **To open a random-access file for reading and writing**

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

RandomAccessFile

- To move the file pointer to a specific byte

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

- To get the current position of the file pointer.

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

- To find the number of bytes in a file

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

File BankDataTest.java

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

```
18:       data.open("bank.dat");
19:
20:       boolean done = false;
21:       while (!done)
22:       {
23:          String input = JOptionPane.showInputDialog(
24:             "Account number or " + data.size()
25:             + " for new account");
26:          if (input == null) done = true;
27:          else
28:          {
29:             int pos = Integer.parseInt(input);
30:
31:             if (0 <= pos && pos < data.size()) // add interest
32:             {
33:                SavingsAccount account = data.read(pos);
34:                System.out.println("balance="
35:                   + account.getBalance() + ",interest rate="
36:                   + account.getInterestRate());
37:                account.addInterest();
```

```
38:                data.write(pos, account);
39:             }
40:             else // add account
41:             {
42:                input = JOptionPane.showInputDialog("Balance");
43:                double balance = Double.parseDouble(input);
44:                input = JOptionPane.showInputDialog("Interest Rate");
45:                double interestRate = Double.parseDouble(input);
46:                SavingsAccount account
47:                   = new SavingsAccount(interestRate);
48:                account.deposit(balance);
49:                data.write(data.size(), account);
50:             }
51:          }
52:       }
53:    }
54:    finally
55:    {
56:       data.close();
57:       System.exit(0);
```

```
57:          System.exit(0);
58:       }
59:    }
60: }
```

File BankData.java

```
01: import java.io.IOException;
02: import java.io.RandomAccessFile;
03:
04: /**
05:    This class is a conduit to a random access file
06:    containing savings account data.
07: */
08: public class BankData
09: {
10:    /**
11:       Constructs a BankData object that is not associated
12:       with a file.
13:    */
14:    public BankData()
15:    {
16:       file = null;
17:    }
```

```
18:
19:    /**
20:       Opens the data file.
21:       @param filename the name of the file containing savings
22:       account information.
23:    */
24:    public void open(String filename)
25:       throws IOException
26:    {
27:       if (file != null) file.close();
28:       file = new RandomAccessFile(filename, "rw");;
29:    }
30:
31:    /**
32:       Gets the number of accounts in the file.
33:       @return the number of accounts.
34:    */
35:    public int size()
36:       throws IOException
37:    {
```

```
38:       return (int)(file.length() / RECORD_SIZE);
39:    }
40:
41:    /**
42:       Closes the data file.
43:    */
44:    public void close()
45:       throws IOException
46:    {
47:       if (file != null) file.close();
48:       file = null;
49:    }
50:
51:    /**
52:       Reads a savings account record.
53:       @param n the index of the account in the data file
54:       @return a savings account object initialized with the file data
55:    */
56:    public SavingsAccount read(int n)
57:       throws IOException
```

```
58:    {
59:       file.seek(n * RECORD_SIZE);
60:       double balance = file.readDouble();
61:       double interestRate = file.readDouble();
62:       SavingsAccount account = new SavingsAccount(interestRate);
63:       account.deposit(balance);
64:       return account;
65:    }
66:
67:    /**
68:       Writes a savings account record to the data file
69:       @param n the index of the account in the data file
70:       @param account the account to write
71:    */
72:    public void write(int n, SavingsAccount account)
73:       throws IOException
74:    {
75:       file.seek(n * RECORD_SIZE);
76:       file.writeDouble(account.getBalance());
77:       file.writeDouble(account.getInterestRate());
```

```
78:    }
79:
80:    private RandomAccessFile file;
81:
82:    public static final int DOUBLE_SIZE = 8;
83:    public static final int RECORD_SIZE
84:       = 2 * DOUBLE_SIZE;
85: }
```