

File, flussi e pacchetto java.io

Formato binario e formato di testo

- I dati sono memorizzati nei files in due formati:
 - testo (successione di caratteri)
 - binario (successione di bytes)
- Ad es. numero 12345
 - testo: sequenza di caratteri 1,2,3,4,5
 - binario: sequenza di 4 byte corrispondenti a interi 0,0,48,57 ($12345 = 48 \times 256 + 57$)

Flussi

- In Java input e output sono definiti in termini di **flussi** (stream)
 - ❑ Sequenze ordinate di dati
 - ❑ Flussi di input hanno una sorgente
 - ❑ Flussi di output hanno una destinazione
- Due tipi di flussi
 - ❑ Flussi di dati binari (byte streams)
 - ❑ Flussi di caratteri (character streams)
- Ciascun tipo di flusso è gestito da apposite classi

Pacchetto java.io

- Le classi per gestire i flussi sono nel package `java.io`
 - `import java.io.*;`
- Problemi collegati al flusso input/output sono segnalati con il lancio di una `IOException`
Es.
 - Tentativo di aprire un flusso da file può lanciare `FileNotFoundException`
(sottoclasse di `IOException`)
 - Lettura/scrittura da/su flusso chiuso lancia `IOException`

Flussi Standard

Definiti dalla classe `System` in `java.lang`

- Standard input (tastiera): `System.in`
 - di tipo `InputStream`
- Standard output (monitor): `System.out`
 - di tipo `PrintStream` (discendenza di `OutputStream`)
- Standard error (per messaggi di errore): `System.err`
 - di tipo `PrintStream`

Flussi di byte da sorgente file

- Classe `FileInputStream`
- Apertura file in lettura:

```
FileInputStream in = new  
    FileInputStream("nomefile.bin");
```
- Lettura prossimo byte (un int tra 0 e 255):

```
int x = in.read();
```

 - `read` restituisce -1 se flusso terminato
- Chiusura flusso (rilascia risorse):

```
in.close();
```

Esempio: Contare i byte in un flusso

```
import java.io.*;

public class ContaByte {

    public static void main(String[] args) throws
        IOException {
        InputStream in = new
            FileInputStream("nomefile.bin");

        int totale = 0;
        while (in.read() != -1)
            totale++;
        in.close();
        System.out.println("Il numero di byte è" + totale);
    }
}
```

Specificare il path di un file

- Quando si digita il path di un file ogni barra rovesciata (“\”) va inserita due volte
 - Una singola barra rovesciata è un carattere di escape

```
InputStream in = new  
    FileInputStream("C:\\nomedir\\nomefile.est");
```


Flussi di byte verso destinazione file

- Classe `FileOutputStream`
- Apertura file in scrittura:
`FileOutputStream out= new
FileOutputStream("nomefile.bin");`
- Scrittura prossimo byte (x è di tipo byte):
`out.write(x);`
(write non restituisce valore)
- Chiusura flusso (rilascia risorse, **garantisce scrittura**):
`out.close();`

Classe `PrintStream`

- Aggiunge a `FileOutputStream` tutti i metodi per stampare convenientemente vari tipi di dati
 - Ad es. i metodi `print` e `println`
- Metodi non lanciano `IOException`
- Oggetto `System.out` è di tipo `PrintStream` (flusso standard di output, flusso binario)
- Possiamo direzionare flusso byte su un file

```
PrintStream out =  
    new PrintStream("nomefile.est");
```

Flussi di caratteri da file

- Classe `FileReader`
- Apertura file in lettura:

```
FileReader in = new  
FileReader("nomefile.txt");
```
- Lettura prossimo carattere (un int tra 0 e 65535):

```
int x = in.read();
```

 - `read` restituisce -1 se flusso terminato
- Chiusura flusso (rilascia risorse):

```
in.close();
```

Esempio: Contare i caratteri in un flusso

```
import java.io.*;

public class ContaCaratteri {

    public static void main(String[] args)
                                throws IOException {
        Reader reader = new    FileReader("nomefile.txt");

        int totale = 0;
        while (reader.read() != -1)
            totale++;
        reader.close();
        System.out.println("Il numero di caratteri è" +
            totale);
    }
}
```

Uso agevole di FileReader

- Gli oggetti di `FileReader` leggono un carattere per volta, ma spesso serve leggere intere linee.
- Un oggetto `Scanner` compone stringhe a partire dai caratteri letti da un `FileReader`

```
FileReader reader = new  
                    FileReader("file.txt");  
Scanner in = new Scanner(reader);  
String inputLine = in.nextLine();
```

Flussi di caratteri verso file

- Classe `PrintWriter`

- Apertura file in scrittura:

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

- Contiene tutti i metodi `print` e `println` come `PrintStream`

```
PrintWriter out = new  
    PrintWriter("output.txt");  
out.println(2.75);  
out.println(new BankAccount());  
out.println("Hello, world!");
```

Esempio

- Scrivere un programma che legge tutte le righe di un file e le scrive in un altro file facendo precedere ogni riga dal suo numero

- File di input:

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

- File di output desiderato:

```
/* 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!
```

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:         {
```



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

Nota su apertura file in scrittura

- Quando si istanzia un oggetto che gestisce un flusso di output verso un file:
 - se il file passato come parametro del costruttore esiste, allora viene svuotato del suo contenuto (scrittura flusso parte da prima posizione)
 - se il file non esiste viene creato un file nuovo (vuoto) con il nome passato come parametro del costruttore (creazione file nel file system)

La classe File

- Astrazione del concetto di file
- Può essere utilizzato per manipolare file esistenti
- Creiamo un oggetto di tipo **File**

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

(input.txt può non esistere, e questo comando non lo crea)
- Non possiamo leggere/scrivere direttamente dati da un oggetto di tipo **File**
- Dobbiamo istanziare un oggetto di tipo: **FileReader**, **PrintWriter**, **FileInputStream**, **FileOutputStream** o **PrintStream**

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

La classe File: alcuni metodi

■ `public boolean delete()`

- ❑ Cancella il file restituendo true se la cancellazione ha successo

■ `public boolean renameTo(File newname);`

- ❑ Rinomina il file restituendo true se la ridenominazione ha successo

■ `public long length()`

- ❑ Restituisce la lunghezza del file in byte (zero se il file non esiste)

■ `public boolean exists()`

- ❑ Testa se il file o la directory denotata dal parametro implicito esiste

Ricapitoliamo con un esempio

```
import java.io.*;

public class Esempio {
    public static void main(String[] args)
                                throws IOException {

        // SCRITTURA
        PrintWriter pw = new PrintWriter("C:\\\\HelloWorld.txt");
        pw.println("HELLO WORLD alla fine del file");
        // Chiusura File
        pw.close();

        // LETTURA
        FileReader fr = new FileReader("C:\\\\HelloWorld.txt");
        Scanner sc = new Scanner(fr);
        String s = sc.nextLine();
        // Chiusura File
        fr.close();      System.out.println(s);
    }
}
```

Esempio

```
File f1 = new File("C:\\HelloWorld.txt");  
File f2 = new File("C:\\HelloWorld2.txt");  
f1.renameTo(f2);
```

```
// Attenzione NON crea il file  
File f3 = new File("C:\\HelloWorld3.txt");  
// Per crearlo dovete darlo ad un writer  
PrintWriter fw2 = new PrintWriter(f3);
```

```
    } // Fine main  
} // Fine classe
```

Accesso sequenziale e casuale

- **Accesso sequenziale**

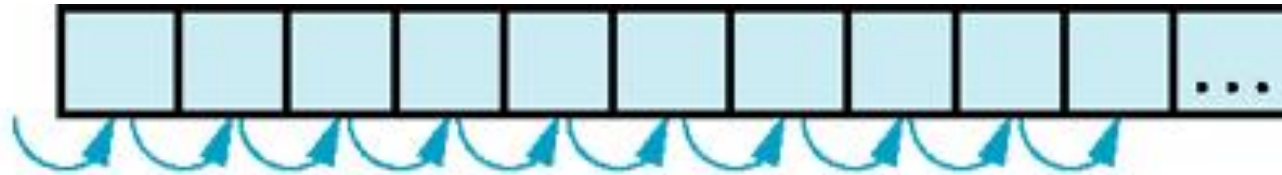
- Un file viene elaborato un byte alla volta, in sequenza
- può essere inefficiente

- **Accesso casuale**

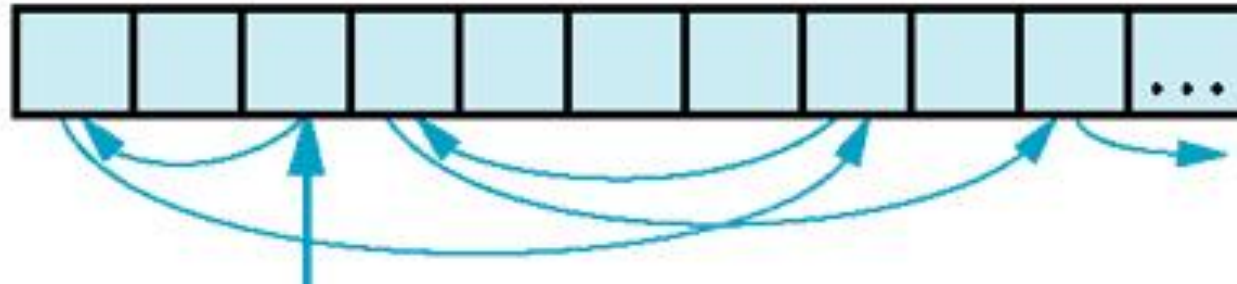
- Possiamo accedere a posizioni arbitrarie nel file
- Soltanto i file su disco supportano l'accesso casuale: **System.in** e **System.out** no
- Ogni file su disco ha un **puntatore di file** che individua la posizione dove leggere o scrivere.

Accesso sequenziale e casuale

Sequential access



Random access



Accesso casuale

- Per l'accesso casuale al file, usiamo un oggetto di tipo `RandomAccessFile`
- Possiamo aprire il file in diverse modalità:
 - "r" apre il file in sola lettura; se viene usato un metodo di scrittura viene invocata una `IOException`
 - "rw" apre il file per lettura e scrittura. Se il file non esiste prova a crearlo.
- Es.:

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

Accesso casuale: metodi

- **f.read()**
 - come read di InputStream, un byte alla volta
 - readLine(), readInt(), readDouble(), ...
- **f.write(b)**
 - scrive il byte b a partire dalla posizione indicata dal puntatore
 - writeChars(String), writeDouble(double), writeInt(int), ...
- **f.close()** //chiude il file
- **f.seek(n)** //sposta il puntatore al byte di indice n
- **long n = f.getFilePointer();**
 - Fornisce la posizione corrente del puntatore nel file
- **long fileLength = f.length();**
 - Fornisce il numero di byte di un file

Esempio

- Si vuole usare un `RandomAccessFile` per mantenere un insieme di oggetti `BankAccount`
- Il programma deve permettere di selezionare un conto e di effettuare un versamento
- Per manipolare un insieme di dati in un file occorre prestare attenzione a come i dati sono formattati
 - Supponiamo che memorizziamo un conto come un testo (`String`), ad esempio: conto 1001 ha saldo 900 e conto 1015 ha saldo 0

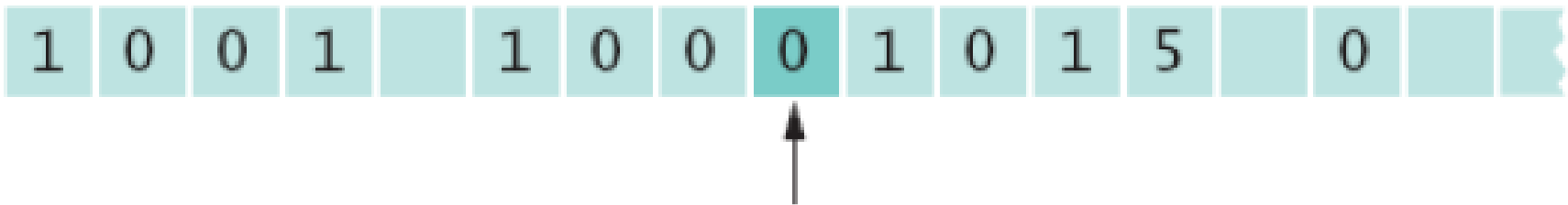
1	0	0	1		9	0	0		1	0	1	5		0		
---	---	---	---	--	---	---	---	--	---	---	---	---	--	---	--	--

Esempio

Vogliamo versare 100 nel conto 1001



Se semplicemente scriviamo il nuovo valore si ha



Soluzione

- Per aggiornare un file:
 - Ogni valore deve avere uno spazio fissato sufficientemente grande
 - Ogni record ha la stessa taglia
 - E' facile individuare ogni record
 - E' facile aggiornare i campi di un record
 - Se memorizziamo i campi dei record in binario (in bytes) in base al tipo, allora i record dello stesso tipo hanno la stessa taglia

Note su RandomAccessFile

- `RandomAccessFile` memorizza i dati in binario
- `readInt` legge interi come sequenze di 4 bytes
- `writeInt` scrive interi come sequenze di 4 bytes
- `readDouble` e `writeDouble` usano 8 bytes

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

Esempio

■ Determinare il numero di conti nel 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 }
}
```

■ Leggere l'(n+1)-esimo conto nel 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);
}
```

Esempio

- Scrivere nell'(n+1)-esimo conto del file

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

- Cerca l'indice di un conto nel file

```
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;
    }
    return -1; // conto non trovato
}
```


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
055:        // the file data
056:    */
057:    public BankAccount read(int n)
058:        throws IOException
059:    {
060:        file.seek(n * RECORD_SIZE);
061:        int accountNumber = file.readInt();
062:        double balance = file.readDouble();
063:        return new BankAccount(accountNumber, balance);
064:    }
065:    /**
066:        Finds the position of a bank account with a given
067:        // 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: }
```

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:
```

Flussi di Oggetti

- Consentono di operare su interi oggetti
 - Per scrivere un oggetto non dobbiamo prima decomporlo
 - Per leggere un oggetto non dobbiamo leggere i dati separatamente e poi ricomporre l'oggetto
- Flussi in scrittura
Classe `ObjectOutputStream`
- Flussi in lettura
Classe `ObjectInputStream`

Serializzazione

- La memorizzazione di oggetti in un flusso è detta **serializzazione**
 - Ogni oggetto riceve un numero di serie nel flusso
 - Se lo stesso oggetto viene salvato due volte la seconda volta salviamo solo il numero di serie
 - Numeri di serie ripetuti sono interpretati come riferimenti allo stesso oggetto
- Gli oggetti in un flusso sono **Serializable**
 - Non è possibile inserire in un flusso oggetti che non sono di tipo **Serializable**

Interfaccia Serializable

- Interfaccia di marcatura (non contiene metodi)
- Serve solo a verificare che siamo a conoscenza della serializzazione (come **Cloneable** rispetto alla clonazione)
- Non tutti gli oggetti implementano **Serializable** (ad es. **Object**)
 - Non tutti i dati del programma necessitano di persistere tra differenti esecuzioni del programma

Identificativo universale di serializzazione

- Ogni classe serializzabile ha un identificativo universale di serializzazione (**UID**)
 - ❑ viene utilizzato nella deserializzazione per controllare che un oggetto corrisponde ad una classe caricata nella JVM
 - ❑ in caso non vi è corrispondenza viene lanciata una **InvalidClassException**
 - ❑ è possibile assegnare un UID esplicitamente definendo la costante di classe **serialVersionUID**
ad es:

```
private static final long serialVersionUID = 42L;
```

Identificativo universale di serializzazione

- Se `serialVersionUID` non viene definita, la JVM calcola l'UID automaticamente
 - di fatto il numero calcolato è il risultato di un hashing di alcuni aspetti della classe
- Diverse implementazioni della JVM possono calcolare l'UID in maniera differente
- Per evitare evitare lanci inattesi dell'eccezione è preferibile definire la costante `serialVersionUID`

Oggetti composti nella serializzazione

- Per poter serializzare un oggetto, tutti le variabili di istanza non primitive devono essere di tipo serializzabile
- Cosa succede se una variabile di istanza non è serializzabile, ma vogliamo serializzare?
 - possiamo fare in modo che la variabile di istanza sia ignorata usando lo specificatore **transient**

Flusso di oggetti in output

- `writeObject` esegue serializzazione dell'oggetto
- L'oggetto da inserire nel flusso deve essere serializzabile altrimenti viene sollevata la `NotSerializableException`

```
MyClass mc = new MyClass(...);  
  
ObjectOutputStream out =  
    new ObjectOutputStream(new FileOutputStream("mc.dat"));  
  
out.writeObject(mc); //MyClass implementa Serializable  
  
out.close();        // chiude il flusso e rilascia risorse
```

Flusso di oggetti in input

- `readObject` legge un **Object** da un flusso (deserializzazione) e restituisce un riferimento ad esso
 - l'output necessita di un cast
- `readObject` può lanciare un'eccezione controllata di tipo **ClassNotFoundException** se classe non caricata in JVM (oltre ad altre eccezioni quali ad es. **InvalidClassException**)

```
ObjectInputStream in =  
    new ObjectInputStream(new FileInputStream("mc.dat"));  
MyClass mc = (MyClass) in.readObject();  
in.close();           // chiude il flusso e rilascia risorse
```

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: {
```

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 =
26:                 new ObjectInputStream(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));
```

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: }
```