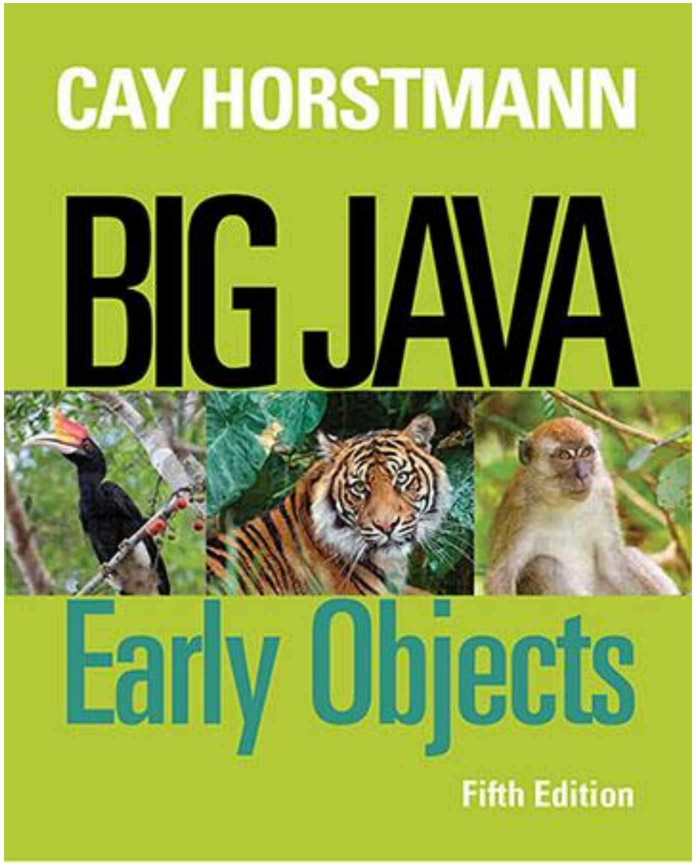


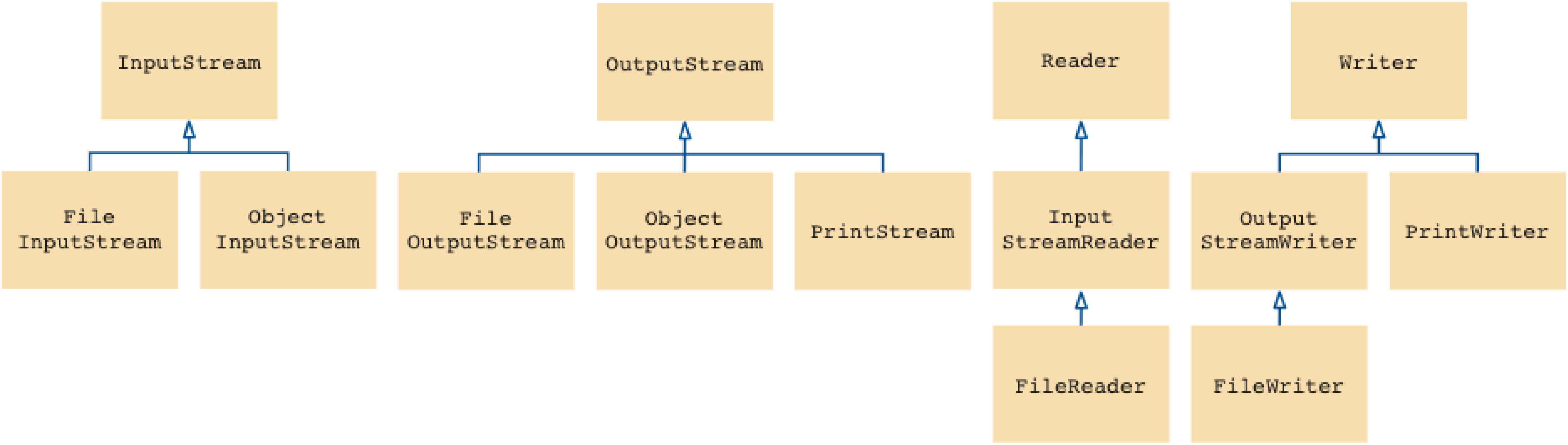
Chapter 20 – Streams and Binary Input/Output

Big Java Early Objects by Cay Horstmann

20.1 Readers, Writers, and Streams

- Two ways to store data:
  - *Text format:* human-readable form, as a sequence of *characters*
    - E.g. Integer 12,345 stored as characters `'1' '2' '3' '4' '5'`
    - More convenient for humans: easier to produce input and to check output
    - *Readers* and *writers* handle data in text form
  - *Binary format:* data items are represented in *bytes*
    - E.g. Integer 12,345 stored as sequence of four bytes `0 0 48 57`
    - More compact and more efficient
    - *Streams* handle binary data

Java Classes for Input and Output



Text Data

- Reader and Writer and their subclasses were designed to process text input and output
- PrintWriter was used in Chapter 7
- Scanner class is more convenient than Reader class
- By default, these classes use the character encoding of the computer executing the program
  - OK, when only exchanging data with users from same country
  - Otherwise, good idea to use UTF-8 encoding:

```
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
```

20.2 Binary Input and Output

- Use `InputStream` and `OutputStream` and their subclasses to process binary input and output
- To read:

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

- To write:

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

- `System.out` is a `PrintStream` object

Binary Input

- Use `read` method of `InputStream` class to read a single byte
  - returns the next byte as an `int` between 0 and 255
  - or, the integer `-1` at end of file

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

Binary Output

- Use `write` method of `OutputStream` class to write a single byte:

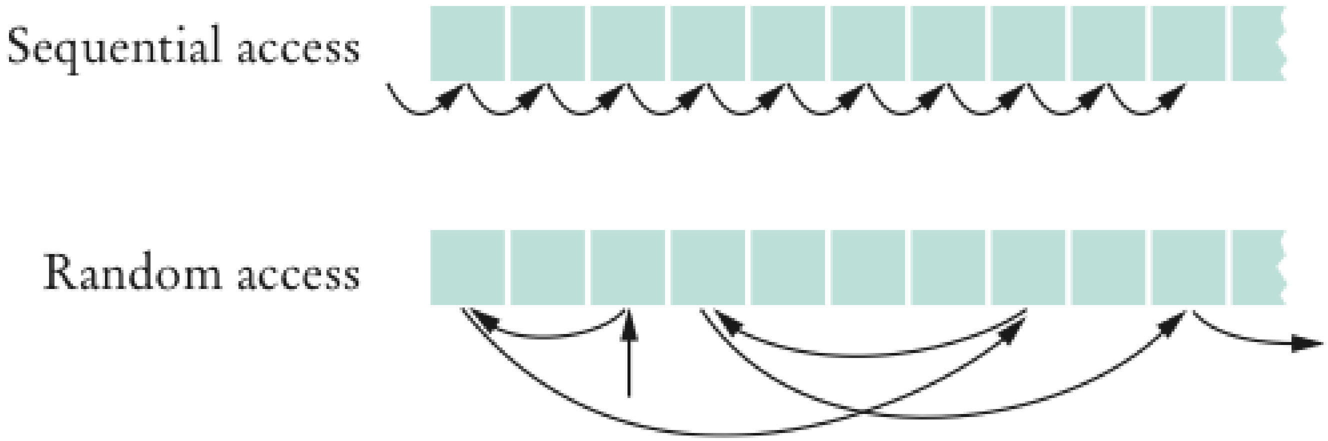
```
OutputStream out = . . .;
int value= . . .; // should be between 0 and 255
out.write(value);
```

- When finished writing to the file, close it:

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

20.3 Random Access

- **Sequential access:** process file one byte at a time
- **Random access:** access file at arbitrary locations
  - Only disk files support random access
    - `System.in` and `System.out` do not
  - Each disk file has a special **file pointer** position
    - Read or write at pointer position



RandomAccessFile Class

- Open a file with *open mode*:
  - 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(position);
```

- To get the current position of the file pointer:

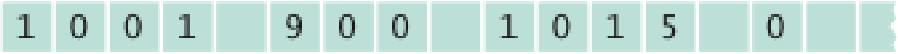
```
long position = f.getFilePointer();
// of type "long" because files can be very large
```

- To find the number of bytes in a file:

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

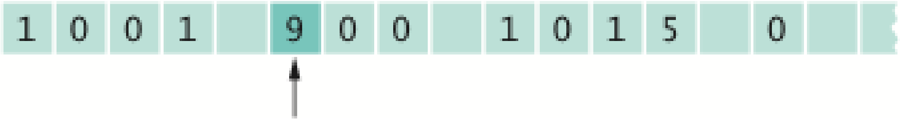
Bank Account Program (1)

- 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:



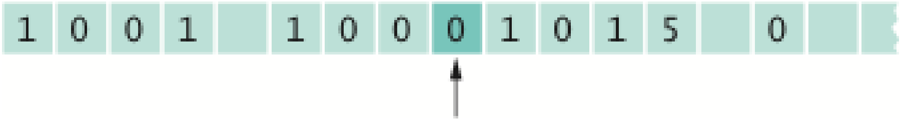
1	0	0	1		9	0	0		1	0	1	5		0	

    - Want to deposit $100 into account 1001:



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

    - Writing out the new value:



1	0	0	1		1	0	0	0	1	0	1	5		0	

Bank Account Program (2)

- 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

Bank Account Program (3)

- RandomAccessFile class stores binary data
- readInt and writeInt methods read/write integers as four-byte quantities
- readDouble and writeDouble methods use eight-byte quantities
- 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 account number plus
            // 8 bytes for balance
}
```

Bank Account Program (4)

- To read the *n*th 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);
}
```

Bank Account Program (5)

- To write the *n*th 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());
}
```

BankSimulator.java

```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
   {
      Scanner in = new Scanner(System.in);
      BankData data = new BankData();
      try
      {
         data.open("bank.dat");

```

Continued

BankSimulator.java (cont.)

```
      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());
         }
```

Continued

BankSimulator.java (cont.)

```java
            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;
         }
      }
      finally
      {
         data.close();
      }
   }
}
```

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
{
   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;
   }

```

Continued

BankData.java (cont.)

```
   /**
      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);
   }

```

Continued

BankData.java (cont.)

```
   /**
      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);
   }

```

Continued

BankData.java (cont.)

```
   /**
      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
   }

```

Continued

BankData.java (cont.)

```
   /**
      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());
   }
}
```

Continued

BankData.java (cont.)

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
```

20.4 Object Streams

- `ObjectOutputStream` class can save entire objects to disk
- `ObjectInputStream` class can read them back in
- Use streams, not writers because objects are saved in binary format

Writing an Object to File

- The object output stream saves all instance variables:

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

Reading an Object From File

- `readObject` method 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();
```

- `readObject` method can throw `ClassNotFoundException`
  - Checked exception ⇒ you must catch or declare it

Write and Read Array List

- 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();
```

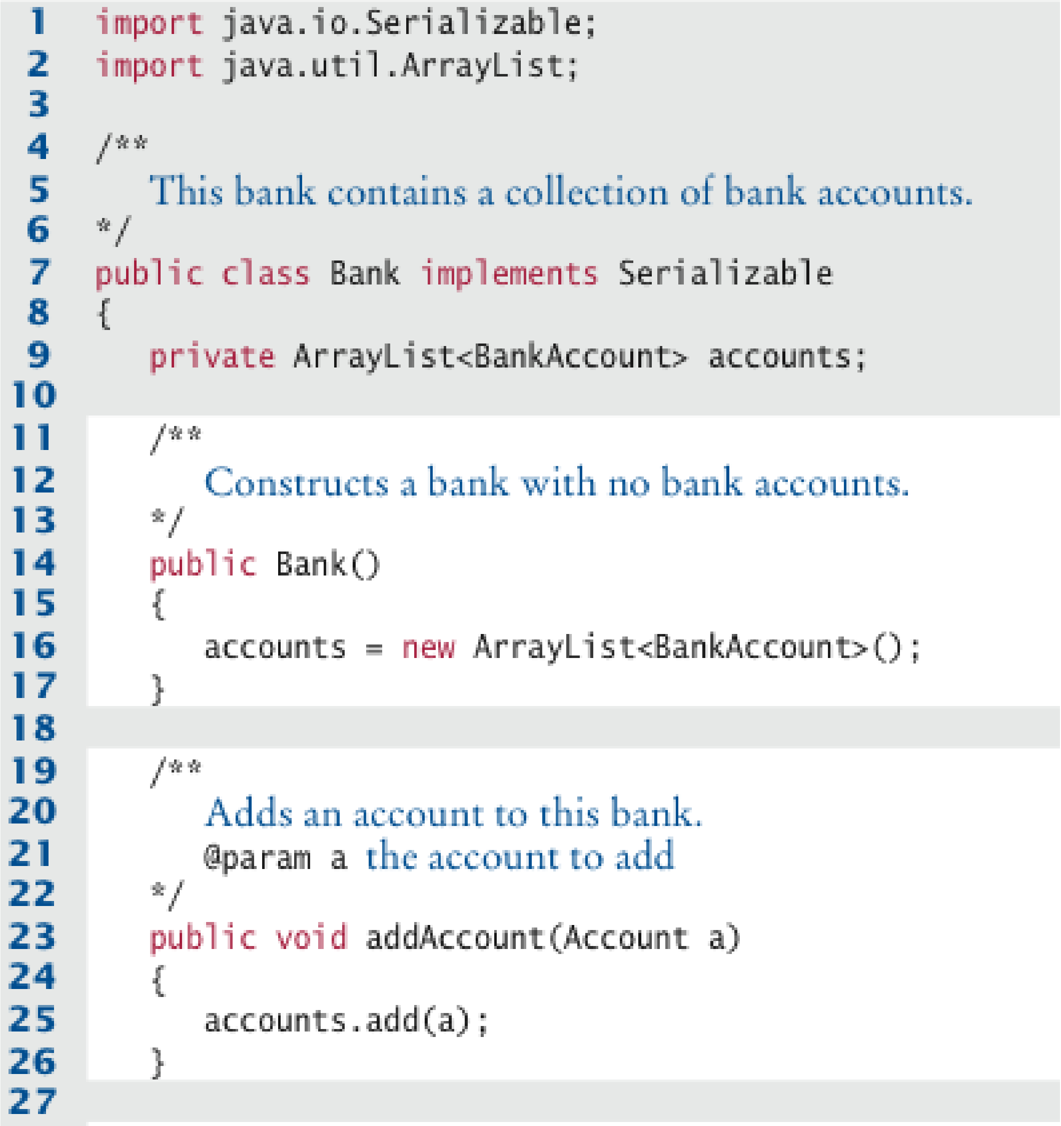
Serializable Interface

- 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
- **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

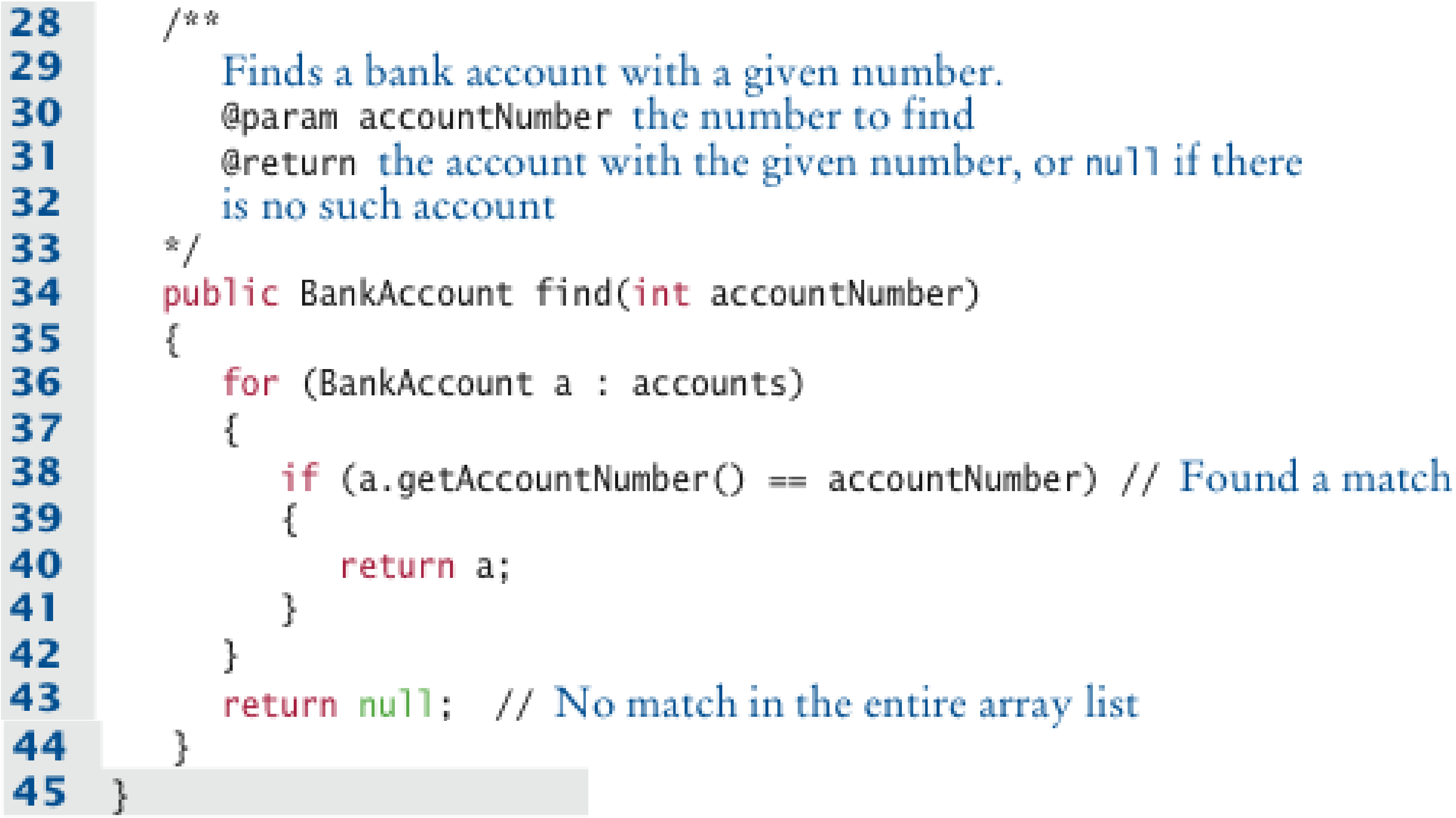
Bank.java



```
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<BankAccount>();
   }

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

```

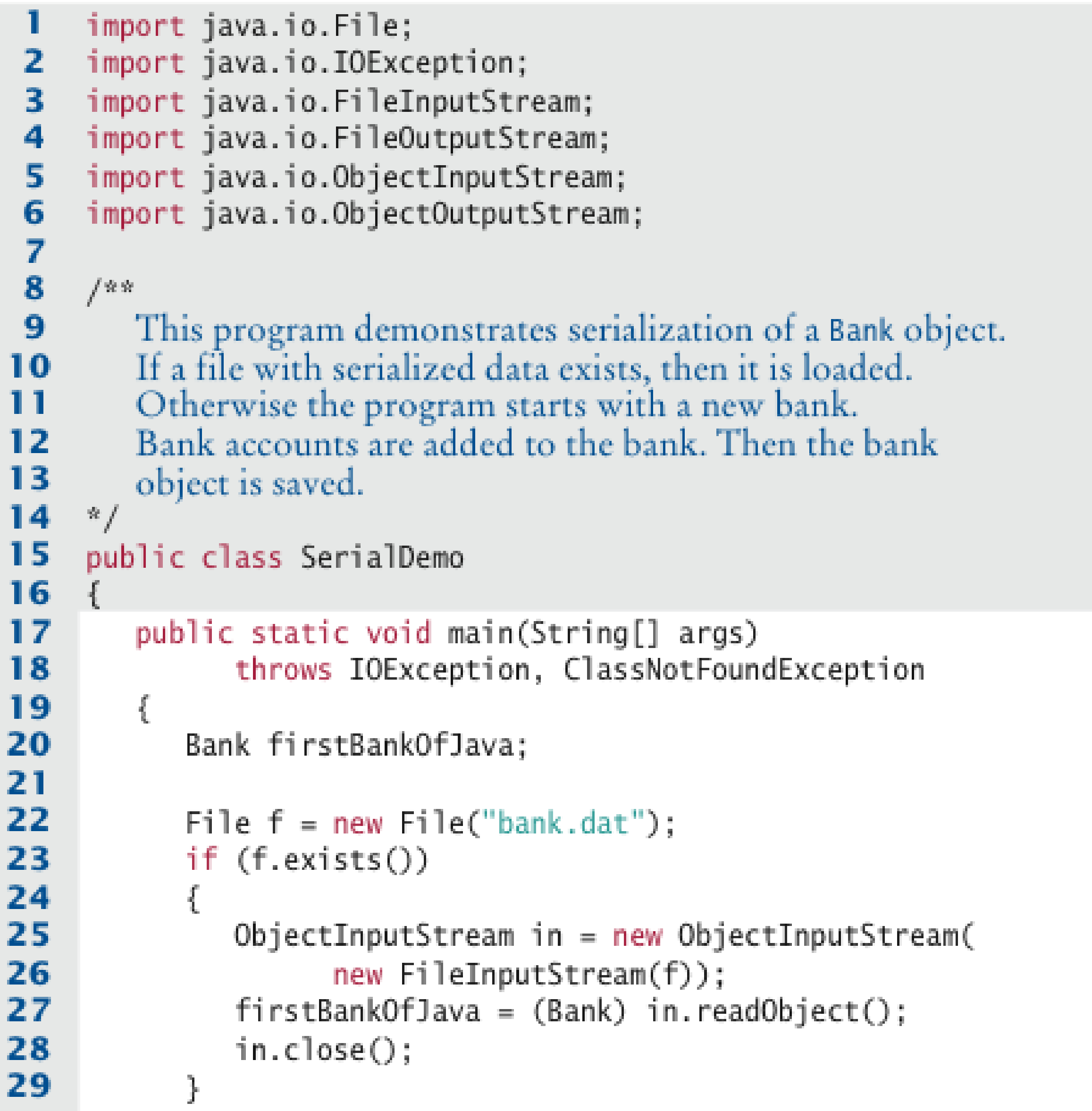
Continued

Bank.java (cont.)



```java
   /**
      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
   }
}
```

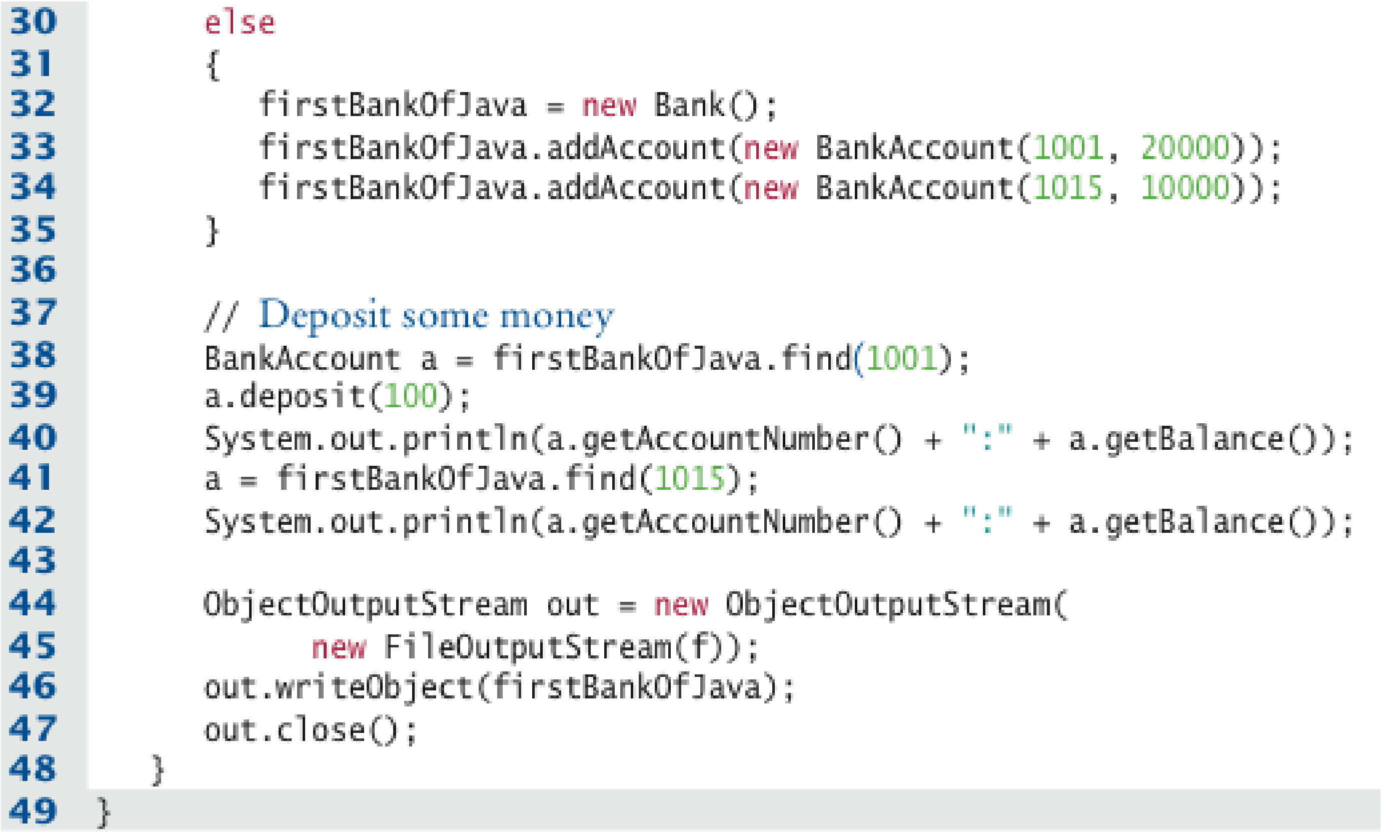
SerialDemo.java



```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.
   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())
      {
         ObjectInputStream in = new ObjectInputStream(
               new FileInputStream(f));
         firstBankOfJava = (Bank) in.readObject();
         in.close();
      }
```

Continued

SerialDemo.java (cont.)

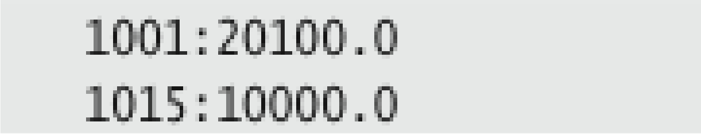


```java
      else
      {
         firstBankOfJava = new Bank();
         firstBankOfJava.addAccount(new BankAccount(1001, 20000));
         firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      }

      // 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());

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

Continued

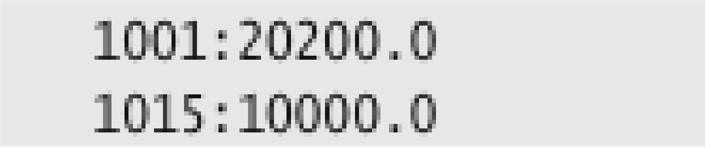
SerialDemo.java (cont.)

Program Run



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

Second Program Run



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

Summary: Java Class Hierarchy for Handling Input and Output

- Streams access sequences of bytes. Readers and writers access sequences of characters.

Summary: 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.

Summary: Random Access

- 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.

Summary: Object Streams

- Use object streams to save and restore all instance variables of an object automatically.
- Objects saved to an object stream must belong to classes that implement the `Serializable` interface.