Chapter 3

File Input/Output

King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI

Chapter 3: Objectives

- After you have read and studied this chapter, you should be able to
  - Include a JFileChooser object in your program to let the user specify a file.
  - Write bytes to a file and read them back from the file, using FileOutputStream and FileInputStream.
  - Write values of primitive data types to a file and read them back from the file, using DataOutputStream and DataInputStream.
  - Write text data to a file and read them back from the file, using PrintWriter and BufferedReader
  - Read a text file using Scanner
  - Write objects to a file and read them back from the file, using ObjectOutputStream and ObjectInputStream

Files

- Storage of data in variables and arrays is temporary—the data is lost when a local variable goes out of scope or when the program terminates.
- Computers use files for long-term retention of large amounts of data, even after programs that create the data terminate. We refer to data maintained in files as persistent data, because the data exists beyond the duration of program execution.
- Computers store files on secondary storage devices such as magnetic disks, optical disks and magnetic tapes.

Files

There are two general types of files you need to learn about: ***text*** files and ***binary*** files…

- A **text**, or character-based, file stores information using ASCII character representations. Text files can be viewed with a standard editor or word processing program but cannot be manipulated arithmetically without requiring special conversion routines.

- A **binary** file stores numerical values using the internal numeric binary format specified by the language in use. A Java program can read a binary file to get numeric data, manipulate the data arithmetically, and write the data to a binary file without any intermediate conversions.

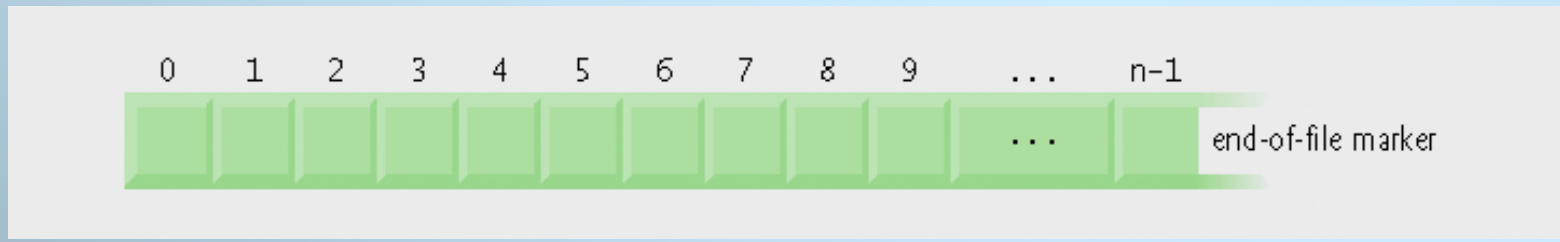
File Operations

There are three basic operations that you will need to perform when working with disk files:

- Open the file for input or output.
- Process the file, by reading from or writing to the file.
- Close the file.

Files and Streams

- Java views each files as a **sequential stream of bytes**
- Operating system provides mechanism to determine end of file
  - End-of-file marker
  - Count of total bytes in file
  - Java program processing a stream of bytes receives an indication from the operating system when program reaches end of stream



Java's view of a file of *n* bytes.

Files and Streams

- File streams
  - Byte-based streams – stores data in binary format
    - **Binary files** – created from byte-based streams, read by a program that converts data to human-readable format
  - Character-based streams – stores data as a sequence of characters
    - **Text files** – created from character-based streams, can be read by text editors
- Java opens file by creating an object and associating a stream with it
- Standard streams – each stream can be redirected
  - `System.in` – standard input stream object, can be redirected with method setIn
  - `System.out` – standard output stream object, can be redirected with method setOut
  - `System.err` – standard error stream object, can be redirected with method setErr

The Class File

- Class `File` useful for retrieving information about files and directories from disk
- Objects of class `File` do not open files or provide any file-processing capabilities
- File objects are used frequently with objects of other `java.io` classes to specify files or directories to manipulate.

Creating File Objects

- To operate on a file, we must first create a File object (from java.io).

Class `File` provides constructors:

1. Takes `String` specifying name and path (location of file on disk)

```
File filename = new File("sample.dat");
```

Opens the file sample.dat in the current directory.

```
File filename = new File("C:/SamplePrograms/test.dat");
```

Opens the file test.dat in the directory C:\SamplePrograms using the generic file separator / and providing the full pathname.

2. Takes two Strings, first specifying path and second specifying name of file

```
File filename = new File(String pathToName, String Name);
```

File Methods

Method	Description
`boolean canRead()`	Returns `true` if a file is readable by the current application; `false` otherwise.
`boolean canWrite()`	Returns `true` if a file is writable by the current application; `false` otherwise.
`boolean exists()`	Returns `true` if the name specified as the argument to the `File` constructor is a file or directory in the specified path; `false` otherwise.
`boolean isFile()`	Returns `true` if the name specified as the argument to the `File` constructor is a file; `false` otherwise.
`boolean isDirectory()`	Returns `true` if the name specified as the argument to the `File` constructor is a directory; `false` otherwise.
`boolean isAbsolute()`	Returns `true` if the arguments specified to the `File` constructor indicate an absolute path to a file or directory; `false` otherwise.
`String getAbsolutePath()`	Returns a string with the absolute path of the file or directory.
`String getName()`	Returns a string with the name of the file or directory.
`String getPath()`	Returns a string with the path of the file or directory.
`String getParent()`	Returns a string with the parent directory of the file or directory (i.e., the directory in which the file or directory can be found).
`long length()`	Returns the length of the file, in bytes. If the `File` object represents a directory, `0` is returned.
`long lastModified()`	Returns a platform-dependent representation of the time at which the file or directory was last modified. The value returned is useful only for comparison with other values returned by this method.
`String[] list()`	Returns an array of strings representing the contents of a directory. Returns `null` if the `File` object does not represent a directory.

Some File Methods

```
if (filename.exists( ) ) {
```

To see if `filename` is associated to a real file correctly.

```
if (filename.isFile() ) {
```

To see if `filename` is associated to a file or not. If false, it is a directory.

```
File directory = new
    File("C:/JavaPrograms/Ch4");

String Arrayfilename[] = directory.list();

for (int i = 0; i < Arrayfilename.length; i++)
{
    System.out.println(Arrayfilename[i]);
}
```

List the name of all files in the directory C:\JavaProjects\Ch4

Demonstrating Class File

```
1
2  // Demonstrating the File class.
3  import java.io.File;
4
5  public class FileDemonstration
6  {
7     // display information about file user specifies
8     public void analyzePath( String path )
9     {
10       // create File object based on user input
11       File name = new File( path );
12
13       if ( name.exists() ) // if name exists, output information about it
14       {
15          // display file (or directory) information
16          System.out.printf(
17             "%s%s\n%s\n%s\n%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s",
18             name.getName(), " exists",
19             ( name.isFile() ? "is a file" : "is not a file" ),
20             ( name.isDirectory() ? "is a directory" :
21                "is not a directory" ),
22             ( name.isAbsolute() ? "is absolute path" :
23                "is not absolute path" ), "Last modified: ",
24             name.lastModified(), "Length: ", name.length(),
25             "Path: ", name.getPath(), "Absolute path: ",
26             name.getAbsolutePath(), "Parent: ", name.getParent() );
27
```

Create new `File` object; user specifies file name and path

Returns `true` if file or directory specified exists

Retrieve name of file or directory

Returns `true` if `name` is a file, not a directory

Returns `true` if `name` is a directory, not a file

Returns `true` if `path` was an absolute path

Retrieve length of file in bytes

Retrieve time file or directory was last modified (system-dependent value)

Retrieve absolute path of file or directory

Retrieve path entered as a string

Retrieve parent directory (path where `File` object's file or directory can be found)

```
28          if ( name.isDirectory() ) // output directory listing
29          {
30             String directory[] = name.list();
31             System.out.println( "\n\nDirectory contents:\n" );
32
33             for ( String directoryName : directory )
34                System.out.printf( "%s\n", directoryName );
35          } // end else
36       } // end outer if
37       else // not file or directory, output error message
38       {
39          System.out.printf( "%s %s", path, "does not exist." );
40       } // end else
41    } // end method analyzePath
42 } // end class FileDemonstration
```

Returns true if `File` is a directory, not a file

Retrieve and display contents of directory

```
1
2  // Testing the FileDemonstration class.
3  import java.util.Scanner;
4
5  public class FileDemonstrationTest
6  {
7     public static void main( String args[] )
8     {
9        Scanner input = new Scanner( System.in );
10       FileDemonstration application = new FileDemonstration();
11
12       System.out.print( "Enter file or directory name here: " );
13       application.analyzePath( input.nextLine() );
14    } // end main
15 } // end class FileDemonstrationTest
```

```
Enter file or directory name here: C:\Program Files\Java\jdk1.5.0\demo\jfc
jfc exists
is not a file
is a directory
is absolute path
Last modified: 1083938776645
Length: 0
Path: C:\Program Files\Java\jdk1.5.0\demo\jfc
Absolute path: C:\Program Files\Java\jdk1.5.0\demo\jfc
Parent: C:\Program Files\Java\jdk1.5.0\demo

Directory contents:

CodePointIM
FileChooserDemo
Font2DTest
Java2D
Metalworks
Notepad
SampleTree
Stylepad
SwingApplet
SwingSet2
TableExample
```

```
Enter file or directory name here:
C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
readme.txt exists
is a file
is not a directory
is absolute path
Last modified: 1083938778347
Length: 7501
Path: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
Absolute path: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D\readme.txt
Parent: C:\Program Files\Java\jdk1.5.0\demo\jfc\Java2D
```

Low-Level File I/O

- To read data from or write data to a file, we must create one of the Java stream objects and attach it to the file.
- A *stream* is a sequence of data items (sequence of characters or bytes) used for program input or output. Java provides many different input and output stream classes in the **java.io** API.
- A ***file stream*** is an *object* that enables the flow of data between a program and some I/O device or file

Low-Level File I/O

- Java has two types of streams: an *input stream* and an *output stream*.
- If the data flows into a program, then the stream is called an **input stream**
- If the data flows out of a program, then the stream is called an **output stream**

Streams for Low-Level File I/O

Binary File Stream Classes

FileInputStream	To open a binary input stream and connect it to a physical disk file
FileOutputStream	To open a binary output stream and connect it to a physical disk file
DataInputStream	To read binary data from a stream
DataOutputStream	To write binary data to a stream

A File Has Two Names

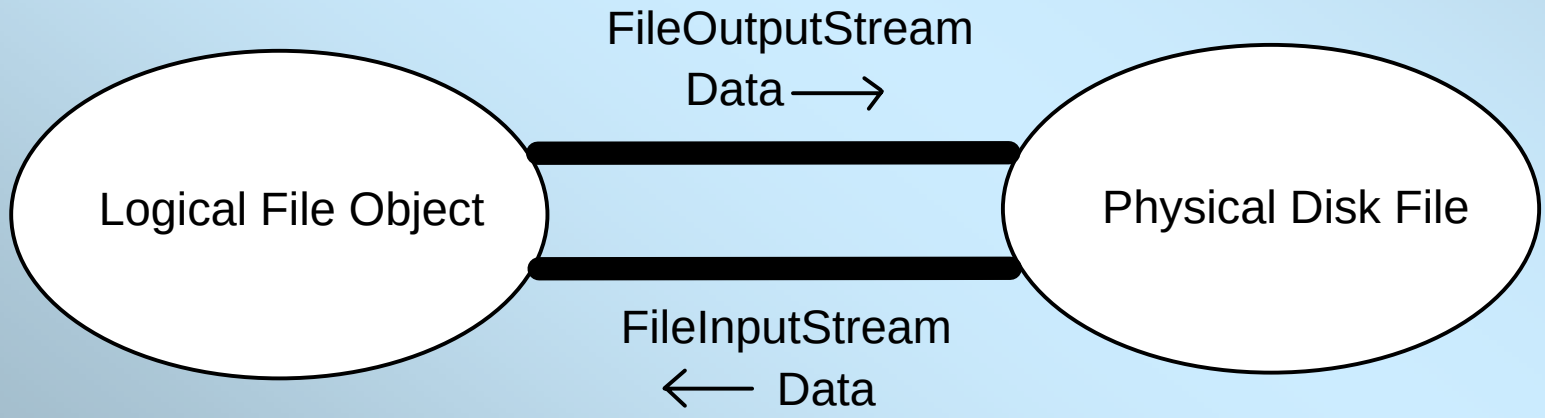
- Every input file and every output file used by a program has two names:

  1. The real file name used by the operating system
  2. The name of the stream that is connected to the file

- The actual file name is used to connect to the stream
- The stream name serves as a temporary name for the file, and is the name that is primarily used within the program

Opening a File

A ***file stream*** provides a connection between your program and the outside world. Opening a file makes the connection between a logical program object and a physical file via the file stream.



Opening a Binary File for Output

Using the FileOutputStream class, create a file stream and connect it to a physical disk file to open the file. We can output only a sequence of bytes.

```
Import java.io.*
Class TestFileOuputStream {
Public static void main (String [] args) throws IOException
{
  //set up file and stream
  File  F   = new File("sample1.data");

  FileOutputStream OutF = new FileOutputStream( F );

  //data to save
  byte[] A = {10, 20, 30, 40,50, 60, 70, 80};

//write the whole byte array at once to the stream
  OutF.write( A );

  //output done, so close the stream
  OutF.close();
 }
}
```

To ensure that all data are saved to a file, close the file at the end of the file access.

Opening a Binary File for Input

Using the FileInputStream class, create a file stream and connect it to a physical disk file to open the file.

```
Import java.io.*
Class TestFileInputStream {
Public static void main (String [] args) throws IOException
{
 //set up file and stream
  File  G = new File("sample1.data");
  FileInputStream InG = new FileInputStream(G);

  //set up an array to read data in
  int fileSize  = (int)G.length();
  byte[] B = new byte[fileSize];

  //read data in and display them
  InG.read(B);
  for (int i = 0; i < fileSize; i++) {
        System.out.println(B[i]);
    }
  //input done, so close the stream
  InG.close();
 }
}
```

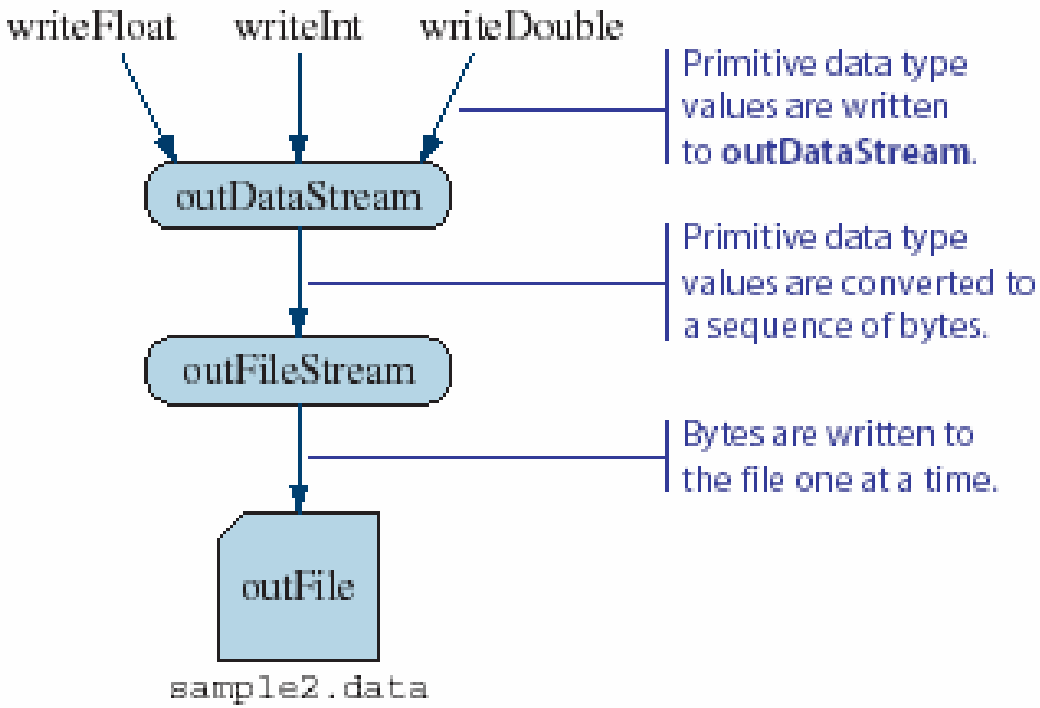
Streams for High-Level File I/O

- FileOutputStream and DataOutputStream are used to output primitive data values
- FileInputStream and DataInputStream are used to input primitive data values
- To read the data back correctly, we must know the order of the data stored and their data types

Setting up DataOutputStream

A standard sequence to set up a DataOutputStream object:

```
File             outFile       = new File("sample2.data");
FileOutputStream outFileStream = new FileOutputStream(outFile);
DataOutputStream outDataStream = new DataOutputStream(outFileStream);
```



Sample Output

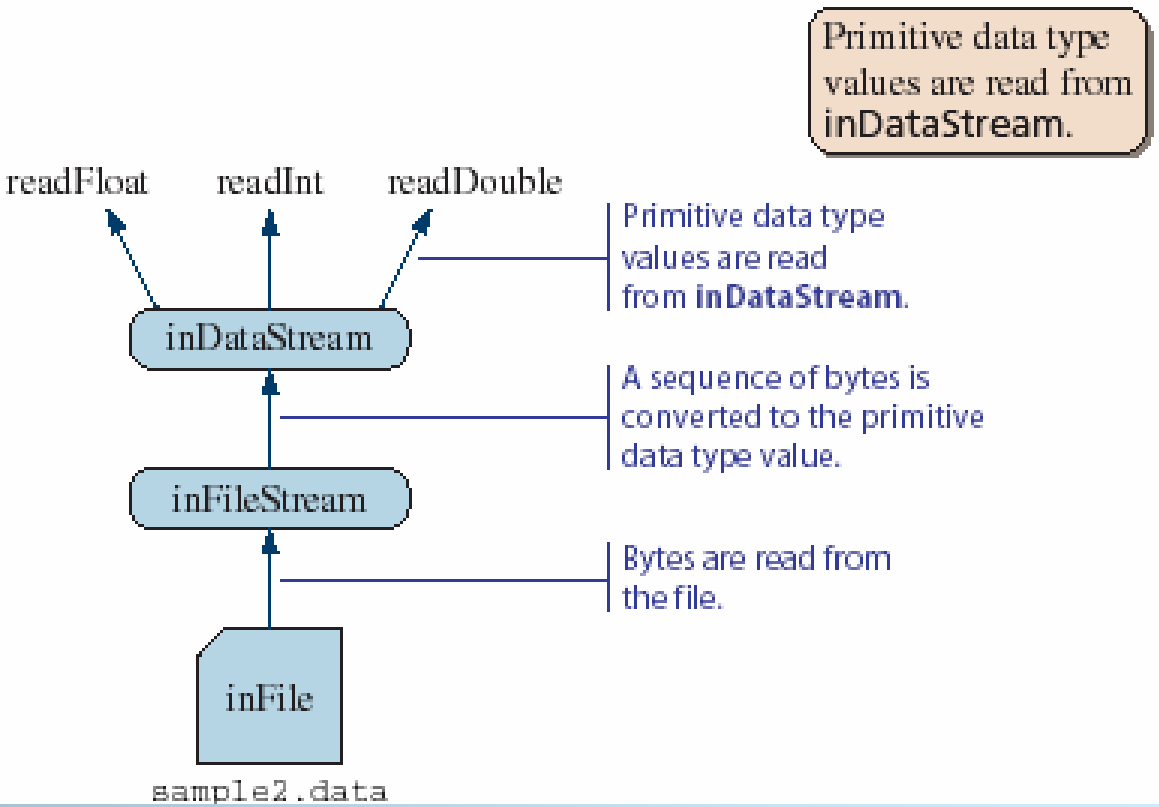
```
import java.io.*;
class TestDataOutputStream {
 public static void main (String[] args) throws IOException {

  //set up file and stream

  File  F   = new File("sample3.data");

  FileOutputStream OutF = new FileOutputStream( F );

  DataOutputStream DF = new   DataOutputStream(OutF);

        //write values of primitive data types to the stream
        DF.writeByte(12);
        DF.writeInt(1234);
        DF.writeLong(9876543);
        DF.writeFloat(1234F);
        DF.writeDouble(1234.4565345);
        DF.writeChar('A');
        DF.writeBoolean(false);

        //output done, so close the stream
        DF.close();
    }
}
```

```
/*========= run============

inside the file "sample3.data" is:

 Ò   –´?Dš@ @“IÓ}Ç«ü A

**********************/
```

Setting up DataInputStream

A standard sequence to set up a DataInputStream object:

```
File            inFile       = new File("sample2.data");
FileInputStream inFileStream = new FileInputStream(inFile);
DataInputStream inDataStream = new DataInputStream(inFileStream);
```



Sample Input

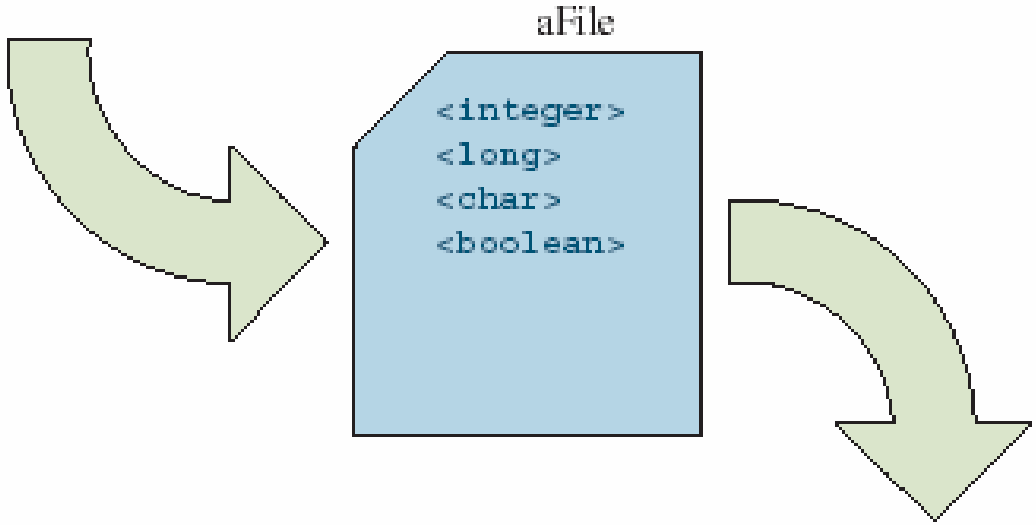
```
import java.io.*;
class TestDataInputStream {
    public static void main (String[] args) throws IOException {
        //set up inDataStream

        File  G   = new File("sample3.data");

        FileInputStream InF = new FileInputStream( G );

        DataInputStream DF = new   DataInputStream(InF);


        //read values back from the stream and display them
        System.out.println(DF.readByte());
        System.out.println(DF.readInt());
        System.out.println(DF.readLong());
        System.out.println(DF.readFloat());
        System.out.println(DF.readDouble());
        System.out.println(DF.readChar());
        System.out.println(DF.readBoolean());

        //input done, so close the stream
        DF.close();
    }
}
```

```
/*output after reading file sample3.dtat"
12
1234
9876543
1234.0
1234.4565345
A
true
************************
```

Reading Data Back in Right Order

The order of write and read operations must match in order to read the stored primitive data back correctly.

```
outStream.writeInteger(...);
outStream.writeLong(...);
outStream.writeChar(...);
outStream.writeBoolean(...);
```




```
inStream.readInteger(...);
inStream.readLong(...);
inStream.readChar(...);
inStream.readBoolean(...);
```

Textfile Input and Output

- Instead of storing primitive data values as binary data in a file, we can convert and store them as a string data.
  - This allows us to view the file content using any text editor
- To output data as a string to file, we use a **PrintWriter** object.
- To input data from a textfile, we use **FileReader** and **BufferedReader** classes
  - From Java 5.0 (SDK 1.5), we can also use the Scanner class for inputting textfiles

Text File Stream Classes

FileReader	To open a character input stream and connect it to a physical disk file
FileWriter	To open a character output stream and connect it to a physical disk file
BufferedReader	To provide buffering and to read data from an input stream
BufferedWriter	To provide output buffering
PrintWriter	To write character data to an output stream

Sample Textfile Output

A test program to save data to a file using PrintWriter for high-level IO

```
import java.io.*;
class TestPrintWriter {
   public static void main (String[] args) throws IOException {

            //set up file and stream
            File outFile = new File("sample3.data");
            FileOutputStream SF = new FileOutputStream(outFile);
            PrintWriter PF = new PrintWriter(SF);

            //write values of primitive data types to the stream
            PF.println(987654321);
            PF.println("Hello, world.");
            PF.println(true);

            //output done, so close the stream
            PF.close();
   }
}
```

We use println and print with PrintWriter. The print and println methods convert primitive data types to strings before writing to a file.

Sample Textfile Input

To read the data from a text file, we use the FileReader and BufferedReadder objects.

To read back from a text file:

- we need to associate a BufferedReader object to a file,

```
File inF = new File("sample3.data");
FileReader FR = new FileReader(inF);
BufferedReader BFR = new BufferedReader(FR);
```

- read data using the readLine method of BufferedReader,

```
String str;
str = bufReader.readLine();
```

- convert the string to a primitive data type as necessary.

```
int i = Integer.parseInt(str);
```

Sample Textfile Input

```
import java.io.*;
class TestBufferedReader {

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

  //set up file and stream
  File inF = new File("sample3.data");
  FileReader FR = new FileReader(inF);
  BufferedReader BFR = new BufferedReader(FR);
  String str;
  //get integer
  str = BFR.readLine();
  int i = Integer.parseInt(str);

   //get long
    str = BFR.readLine();
    long l = Long.parseLong(str);

   //get float
    str = BFR.readLine();
    float f = Float.parseFloat(str);

   //get double
    str = BFR.readLine();
    double d = Double.parseDouble(str);

    //get char
    str = BFR.readLine();
    char c = str.charAt(0);

    //get boolean
    str = BFR.readLine();
    Boolean boolObj = new Boolean(str);
    boolean b = boolObj.booleanValue( );

    System.out.println(i);
    System.out.println(l);
    System.out.println(f);
    System.out.println(d);
    System.out.println(c);
    System.out.println(b);

    //input done, so close the stream
    BFR.close();
  }
}
```

Sample Textfile Input with Scanner

```
import java.util.*;
import java.io.*;
class TestScanner {

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

        //open the Scanner
     try{
         Scanner input = new Scanner(new File("sample3.data"));
        } catch (FileNotFoundException e) {System.out.println("Error opening file");
                                           System. Exit(1);}
        int i = input.nextInt();
        long l = input.nextLong();
        float f = input.nextFloat();
        double d = input.nextDouble();
        char c = input.next().charAt(0);
        boolean b = input.nextBoolean();

        System.out.println(i);
        System.out.println(l);
        System.out.println(f);
        System.out.println(d);
        System.out.println(c);
        System.out.println(b);

        input.close();
   }
}
```

We can associate a new Scanner object to a File object. For example:

Scanner scanner = new File ("sample3.data"));

Will associate scanner to the file sample3.data. Once this association is made, we can use scanner methods such as nexInt, next, and others to input data from the file.

The code is the same as TestBufferedReader but uses the Scanner class instead of BufferedReader. Notice that the conversion is not necessary with the Scanner class by using appropriate input methods such as nexInt and nexDouble.

Saving Objects

To save objects to a file, we first create an ObjectOutputStream object. We use the method writeObject to write an object.

```
import java.io.*;
Class TestObjectOutputStream {
 public static void main (String[] args) throws IOException {

    File outFile = new File("objects.data");
    FileOutputStream   outFileStream = new FileOutputStream(outFile);
    ObjectOutputStream outObjectStream = new ObjectOutputStream(outFileStream);
    Person p;
    for (int i =0; i<10; i++) {
       s=input.next();
       p = new Person ();
       p.setName(input.next()+input.nextLine());
       p.setAge(input.nextInt());
       p.setGender(s.charAt(0));


       outObjecttStream.writeObject(p);
     }
      outObjectStream.close();
  }
}
```

Saving Objects

It is possible to save different type of objects to a single file. Assuming the Account and Bank classes are defined properly, we can save both types of objects to a single file:

```
File outFile = new File("objects.data");

FileOutputStream   outFileStream = new FileOutputStream(outFile);

ObjectOutputStream outObjectStream = new ObjectOutputStream(outFileStream);
```

```
Person person = new Person("Mr. Ali", 20, 'M');

outObjectStream.writeObject( person );
```

```
account1    = new Account();
bank1       = new Bank();

outObjectStream.writeObject( account1 );
outObjectStream.writeObject( bank1    );
```

Could save objects from the different classes.

Saving Objects

We can even mix objects and primitive data type values, for example,

```
Account account1, account2;
Bank bank1, bank2;

account1 = new Account();
account2 = new Account();
bank1 = new Bank();
bank2 = new Bank();


outObjectStream.writeInt( 15 );

outObjectStream.writeObject( account1 );

outObjectStream.writeChar( 'X' );
```

Reading Objects

To read objects from a file, we use FileInputStream and ObjectInputStream. We use the method readObject to read an object.

```
import java.io.*;
Class TestObjectInputStream {
 public static void main (String[] args) throws IOException {
   File  inFile = new File("objects.data");
   FileInputStream    inFileStream = new FileInputStream(inFile);
   ObjectInputStream inObjectStream = new ObjectInputStream(inFileStream);
   Person p;
   for (int i =0; i<10; i++) {
     p =  (Person) inObjectStream.readObject();
     System.out.println(p.getName() + “    “ + p.getAge() + “  “ +p.getGender());
   }
   inObjectStream.close();
  }
}
```

Reading Objects

If a file contains objects from different classes, we must read them in the correct order and apply the matching typecasting. For example, if the file contains two Account and two Bank objects, then we must read them in the correct order:

```
account1 = (Account) inObjectStream.readObject( );
account2 = (Account) inObjectStream.readObject( );
bank1 = (Bank) inObjectStream.readObject( );
bank2 = (Bank) inObjectStream.readObject( );
```

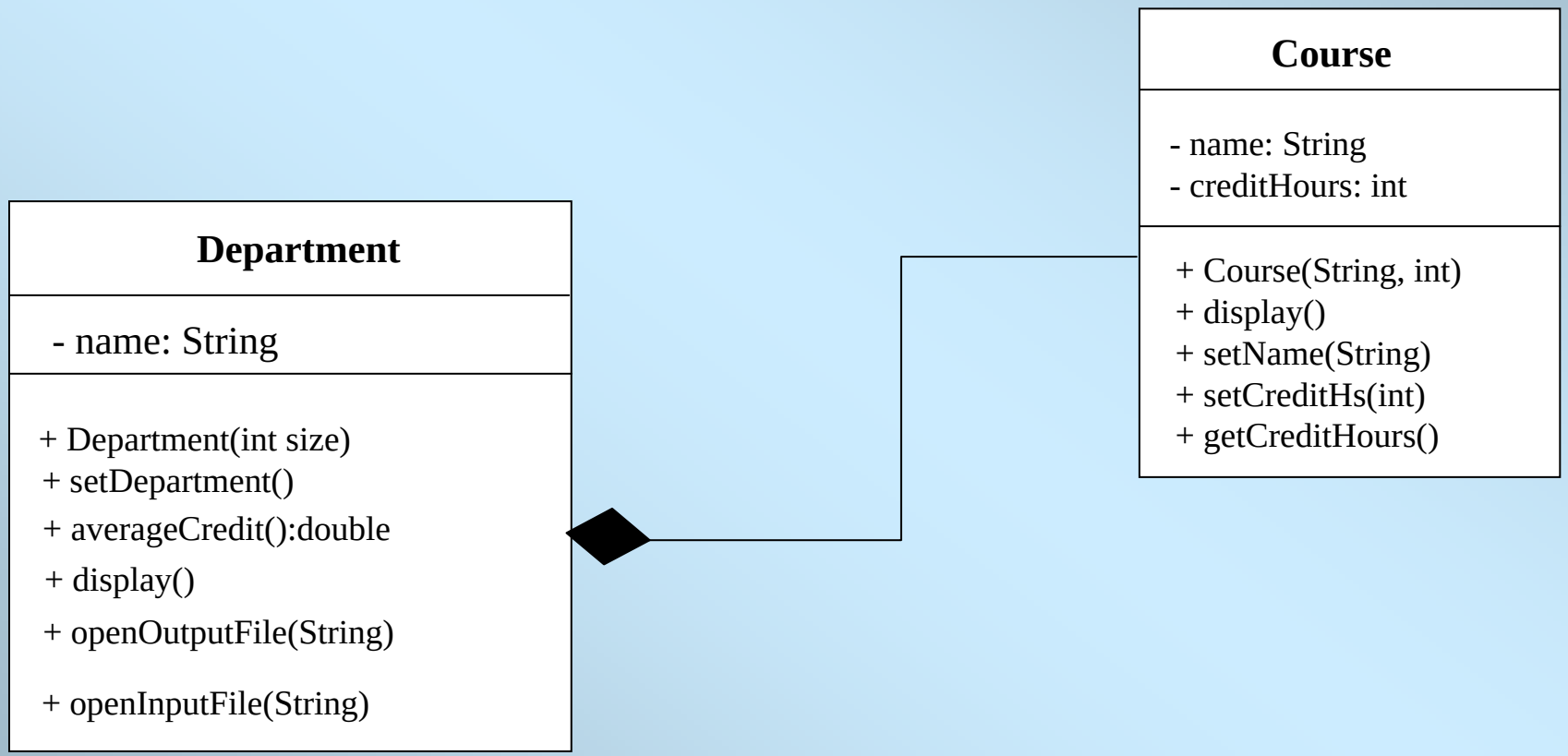
Saving and Loading Arrays

- Instead of processing array elements individually, it is possible to save and load the whole array at once.

```
Person[] p = new Person[ N ];
            //assume N already has a value

//build the people array
. . .
//save the array
outObjectStream.writeObject ( p );
```

```
//read the array
Person[ ] p = (Person[]) inObjectStream.readObject( );
```

Example: Class Department



Example: Class Department

Implementation of Class Course

```
import java.io.*;
public class Course implements Serializable
{
 private String name;
 private int creditHours;
 public Course (String na, int h)
 {
  name=na;
  creditHours=h;
 }
 public void display()
 {
  System.out.println("Name : "+name);
  System.out.println("Credit Hours : "+ creditHours);
 }
```

```
 public void setName(String na)
 {
   name=na;
 }
 public void setCreditHs(int h)
 {
    creditHours=h;
 }
 public double getCreditHours()
 {
  return creditHours;
 }
}
```

Example: Class Department

Implementation of Class Department

```
import java.io.*;
import java.util.Scanner;
public class Department
{
  private String name;
  private Course []c;
  public Department(int size)
  {
     name= " ";
     c= new Course[size];
  }
```

```
public void setDepartment()
  {
    Scanner input = new Scanner(System.in);
    System.out.print("Please enter the name of Department :");
    name =input.next()+input.nextLine();
   for (int i=0; i<c.length; i++)
  {
    System.out.print("Please enter the name of the course :");
   c[i]=new course();
    c[i].setName(input.next()+ input.nextLine());
    System.out.print("Please enter the credit hours : ");
    c[i].setCreditHs(input.nextInt());
    }
  }
```

Example: Class Department

Implementation of Class Department

```
public void openOutputFile(String fileName) throws
IOException
{
  File  f = new File(fileName);
  FileOutputStream  g = new FileOutputStream(f);
  ObjectOutputStream obj = new ObjectOutputStream(g);
  obj.writeBytes(name);
  obj.writeObject(c);
  obj.close();
 }
```

```
public void openInputFile(String fileName) throws
                    ClassNotFoundException, IOException
{
  File  f = new File(fileName);
  FileInputStream  g = new FileInputStream(f);
 ObjectInputStream obj = new ObjectInputStream(g);
 name=obj.readLine();
 c = (Course [])obj.readObject();
 obj.close();
}
```

Example: Class Department

Implementation of Class Department

```
public double averageCredit()
{
  double s=0.0;
  for (int i=0; i<c.length; i++)
  s+=c[i].getCreditHours();
  return (s/c.length);
 }
 public void display()
{
  System.out.println("=======================");
  System.out.println("The name of the department is :" + name);
  for (int i=0; i<c.length; i++)
      c[i].display();
  System.out.println("The average of credit hours is :" + averageCredit());
                    }
}
```

Implementation of DepartmentTest1

```
import java.io.*;
public class DepartmentTest1
{
  public static void main(String[] args) throws IOException
 {
   Department dep = new Department(3);
   dep.setDepartment();
  dep.openOutputFile("computer.data");
   Department dep2 = new Department(2);
   dep2.setDepartment();
   dep2.openOutputFile("engineering.data");
  }
}
```

```
/*    run
Please enter the name of Department :Computer science
Please enter the name of the course :csc107
Please enter the credit hours : 3
Please enter the name of the course :csc112
Please enter the credit hours : 3
Please enter the name of the course :csc113
Please enter the credit hours : 4
Please enter the name of Department :Engineering
Please enter the name of the course :eng123
Please enter the credit hours : 4
Please enter the name of the course :eng125
Please enter the credit hours : 3
*/
```

Implementation of DepartmentTest2

```
import java.io.*;
public class DepartmentTest2
{
  public static void main(String[] args) throws
                    ClassNotFoundException, IOException
 {
   Department d1 = new Department(3);
  d1.openInputFile("computer.data");
  d1.display();
  Department d2 = new Department(2);
  d2.openInputFile("engineering.data");
  d2.display();
  }
}
```

```
/*
=====================================
The name of the department is :Computer science
Name : csc107
Credit Hours : 3
Name : csc112
Credit Hours : 3
Name : csc113
Credit Hours : 4
The average of credit hours is :3.3333333333333335
=====================================
The name of the department is :Engineering
Name : eng123
Credit Hours : 4
Name : eng125
Credit Hours : 3
The average of credit hours is :3.5
*/
```