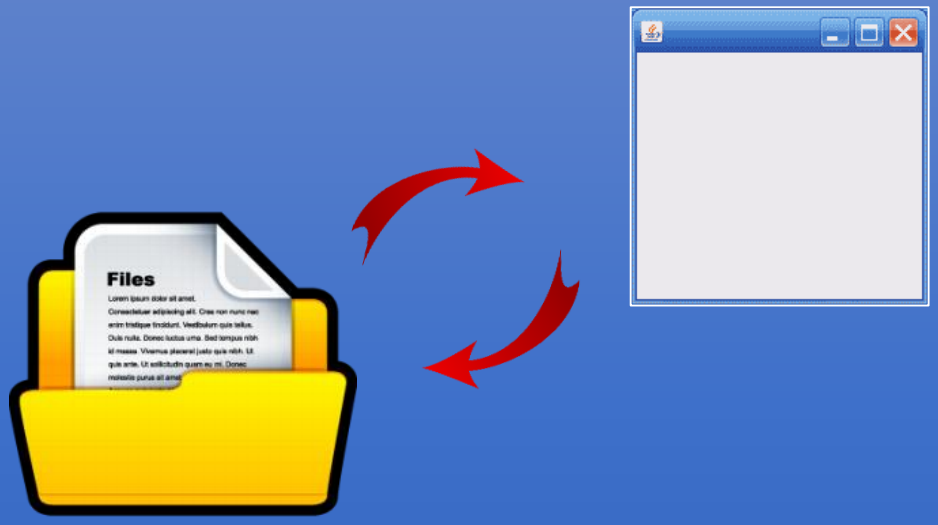



# File I/O

**Lecture 11**

Dr. Tamer ABUHMED
Java Programming Course (SWE2023)
College of Computing

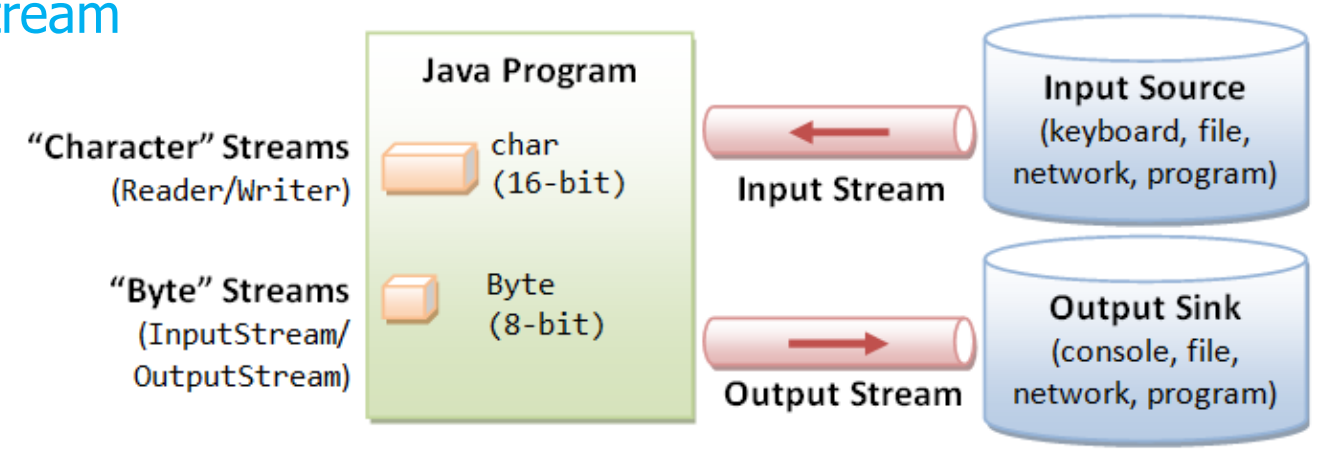
SUNG KYUN KWAN UNIVERSITY
1398

# Outline


- Streams
- Files Types
  - Text Files
  - Binary Files
- Text files Processing
  - Write / Appending
  - Read
  - Path Names
- Binary files Processing
  - Serializable objects
  - Write
  - Read

# Streams


- A *stream* is an object that enables the flow of data between a program and some I/O device or file
  - 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




Internal Data Formats:
- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:
- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

# Text Files and Binary Files


- Files that are designed to be read by human beings, and that can be read or written with an editor are called text files
  - Text files can also be called ASCII files because the data they contain uses an ASCII encoding scheme and we can move these files from one computer to another

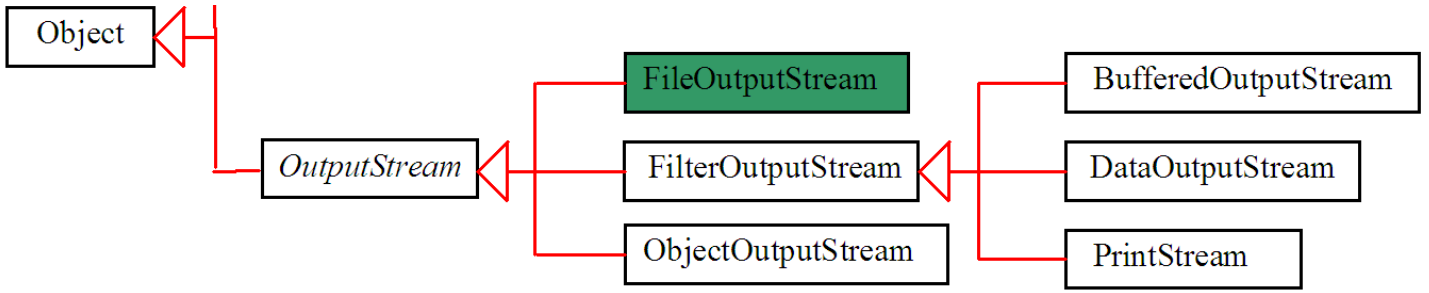
- Files that are designed to be read by programs and that consist of a sequence of binary digits are called binary files.
  - Efficient to process than text files


MyBooks - Notepad
File Edit Format View Help
Title Author Publisher ISBN
Harry Potter and the Sorcerer's Stone J.K. Rowling Arthur A. Levine Books 0590353403
The Da Vinci Code Dan Brown DoubleDay 0385504209
Cracking The Da Vinci Code Simon Cox Barnes & Noble 0760759316
Illuminating Angels and Demons Simon Cox Barnes & Noble 0760767270
Bunnicula: A Rabbit-Tale of Mystery Deborah Howe Aladdin 0689806590
Bunnicula Strikes Again! James Howe Aladdin 0689814623
Red Dragon Thomas Harris Dutton Adult 0525945563
The Silence of the Lambs Thomas Harris St. Martin's Paperbacks 0312924585
I'm OK--You're OK Thomas Harris Harper Paperbacks 0060724277

# Text File

# Writing to a Text File using `PrintWriter`


- The class **`PrintWriter`** is a stream class that can be used to write to a text file
  - An object of the class **`PrintWriter`** has the methods **`print`** and **`println`**
  - These are similar to the **`System.out`** methods of the same names, but are used for text file output, not screen output

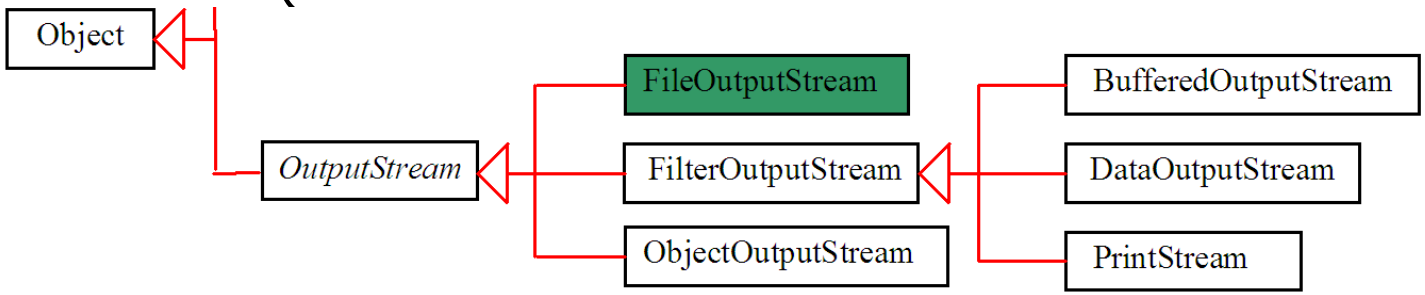


# Writing to a Text File


- All the file I/O classes that follow are in the package **`java.io`**, so a program that uses **`PrintWriter`** will start with a set of **`import`** statements:

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
```

- The class **`PrintWriter`** has no constructor that takes a file name as its argument
  - It uses another class, **`FileOutputStream`**, to convert a file name to an object that can be used as the argument to its (the **`PrintWriter`**) constructor



# Writing to a Text File


- A stream of the class **`PrintWriter`** is created and connected to a text file for writing as follows:

```
PrintWriter outputStreamName;
outputStreamName = new PrintWriter(new
                        FileOutputStream(FileName));
```

  - The class **`FileOutputStream`** takes a string representing the file name as its argument
  - The class **`PrintWriter`** takes the anonymous **`FileOutputStream`** object as its argument

# Appending to a Text File


- To create a **`PrintWriter`** object and connect it to a text file for *appending*, a second argument, set to **`true`**, must be used in the constructor for the **`FileOutputStream`** object

```
outputStreamName = new PrintWriter(new
  FileOutputStream(FileName, true));
```

  - After this statement, the methods **`print`**, **`println`** and/or **`printf`** can be used to write to the file

  - The new text will be written *after the old text* in the file

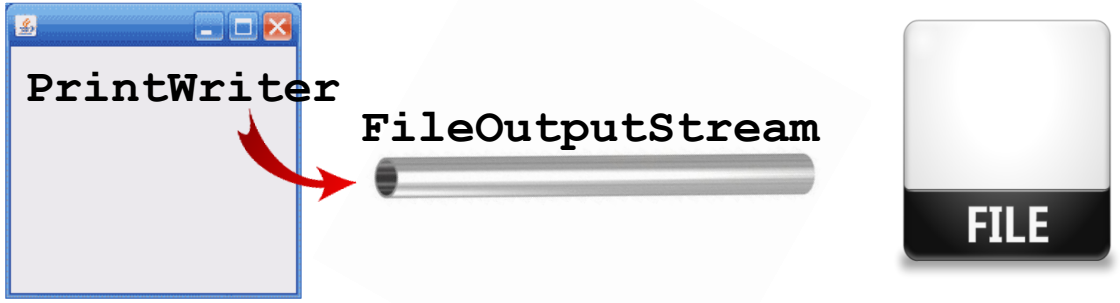
# Example

```
private void DoWrite() {

try {
FileOutputStream fileObject =new FileOutputStream("data.txt",true);

PrintWriter x = new PrintWriter(fileObject);
x.println(textField.getText()+"\n");
x.close();
} catch (Exception e) {
// TODO Auto-generated catch block
e.printStackTrace();
    }

}
```

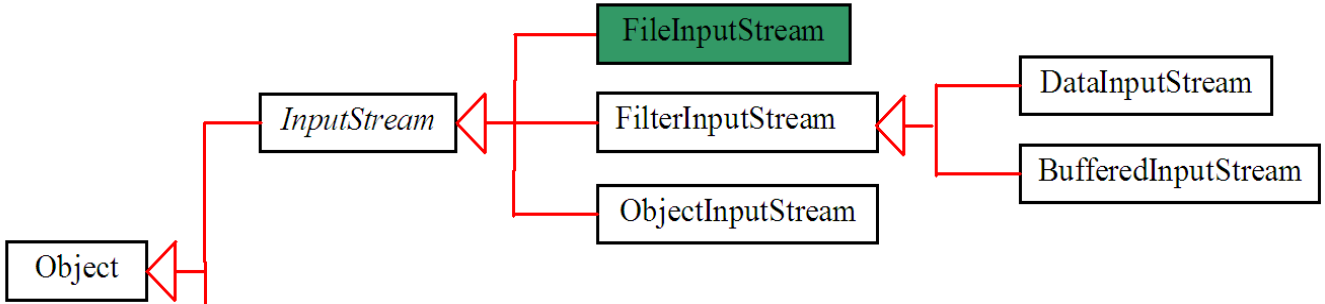



# Reading From a Text File Using `Scanner`


- The class **`Scanner`** can be used for reading from the keyboard as well as reading from a text file
  - Simply replace the argument **`System.in`** (to the **`Scanner`** constructor) with a suitable stream that is connected to the text file

```
Scanner StreamObject =
    new Scanner(new FileInputStream(FileName));
```

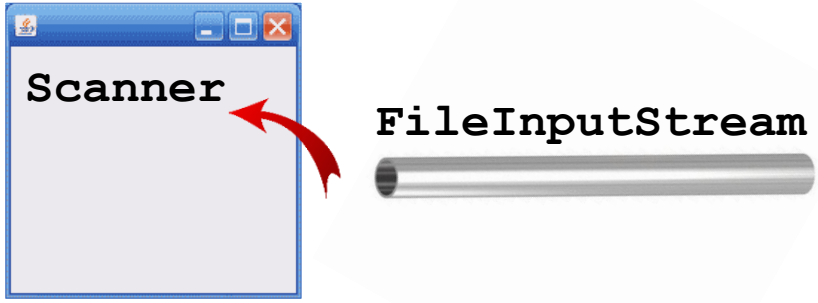
- Methods of the **`Scanner`** class for reading input behave the same whether reading from the keyboard or reading from a text file
  - For example, the **`nextInt`** and **`nextLine`** methods



# Example Read from file using scanner


```
private void doRead() {


try {
FileInputStream fileObject = new FileInputStream("data.txt");
Scanner x = new Scanner(fileObject);
String s =new String();
while(x.hasNext()){
s = s+""+x.nextLine();
 }
JOptionPane.showMessageDialog(null, s);
} catch (FileNotFoundException e) {
// TODO Auto-generated catch block
e.printStackTrace();
   }
}
```

# Testing for the End of a Text File with `Scanner`


- A program that tries to read beyond the end of a file using methods of the `Scanner` class will cause an exception to be thrown
- However, instead of having to rely on an exception to signal the end of a file, the `Scanner` class provides methods such as `hasNextInt` and `hasNextLine`
  - These methods can also be used to check that the next token to be input is a suitable element of the appropriate type

# Reading From a Text File Using **`BufferedReader`**


- The class **`BufferedReader`** is a stream class that can be used to read from a text file
  - An object of the class **`BufferedReader`** has the methods **`read`** and **`readLine`**
- A program using **`BufferedReader`**, like one using **`PrintWriter`**, will start with a set of **`import`** statements:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
```

# Reading From a Text File Using BufferedReader


- Like the classes **`PrintWriter`** and **`Scanner`**, **`BufferedReader`** has no constructor that takes a file name as its argument
  - It needs to use another class, **`FileReader`**, to convert the file name to an object that can be used as an argument to its (the **`BufferedReader`**) constructor
- A stream of the class **`BufferedReader`** is created and connected to a text file as follows:

```
BufferedReader readerObject;
readerObject = new BufferedReader(new
                        FileReader(FileName));
```

  - This opens the file for reading

# Reading From a Text File


- After these statements, the methods **`read`** and **`readLIne`** can be used to read from the file
  - The **`readLine`** method is the same method used to read from the keyboard, but in this case it would read from a file
  - The **`read`** method reads a single character, and returns a value (of type **`int`**) that corresponds to the character read
  - Since the read method does not return the character itself, a type cast must be used:

```
char next = (char)(readerObject.read());
```

# Reading From a Text File


- A program using a **`BufferedReader`** object in this way may throw two kinds of exceptions
  - An attempt to open the file may throw a **`FileNotFoundException`** (which in this case has the expected meaning)
  - An invocation of **`readLine`** may throw an **`IOException`**
  - Both of these exceptions should be handled

# Testing for the End of a Text File


- The method **`readLine`** of the class **`BufferedReader`** returns **`null`** when it tries to read beyond the end of a text file
  - A program can test for the end of the file by testing for the value **`null`** when using **`readLine`**
- The method **`read`** of the class **`BufferedReader`** returns **`-1`** when it tries to read beyond the end of a text file
  - A program can test for the end of the file by testing for the value **`-1`** when using **`read`**

# Reading Numbers


- Unlike the **`Scanner`** class, the class **`BufferedReader`** has no methods to read a number from a text file
  - Instead, a number must be read in as a string, and then converted to a value of the appropriate numeric type using one of the wrapper classes
  - To read in a single number on a line by itself, first use the method **`readLine`**, and then use **`Integer.parseInt`**, **`Double.parseDouble`**, etc. to convert the string into a number
  - If there are multiple numbers on a line, **`StringTokenizer`** can be used to decompose the string into tokens, and then the tokens can be converted as described above

# Buffered(Writer/Reader)


**FileWriter("your_file.txt", true);** true is to append the content to file

```
FileWriter fw = new FileWriter("your_file.txt");
BufferedWriter bw = new BufferedWriter(fw);
bw.writeData(content);
bw.flush();
bw.close();
```

**Write**

```
FileReader fr = new FileReader("your_file.txt");
BufferedReader br = new BufferedReader(fr);

String sCurrentLine;
// read until the end of file
while ((sCurrentLine = br.readLine()) != null) {
        System.out.println(sCurrentLine);
}
br.close();
```

**Read**

# Path Names


- When a file name is used as an argument to a constructor for opening a file, it is assumed that the file is in the same directory or folder as the one in which the program is run
- If it is not in the same directory, the full or relative path name must be given

# Path Names


- The way path names are specified depends on the operating system
  - A typical UNIX path name that could be used as a file name argument is

```
"/user/sallyz/data/data.txt"
```

  - A **`BufferedReader`** input stream connected to this file is created as follows:

```
BufferedReader inputStream =
  new BufferedReader(new
  FileReader("/user/sallyz/data/data.txt"));
```

# Path Names


- The Windows operating system specifies path names in a different way
  - A typical Windows path name is the following:

    ```
    C:\dataFiles\goodData\data.txt
    ```

  - A **BufferedReader** input stream connected to this file is created as follows:

    ```
    BufferedReader inputStream = new
      BufferedReader(new FileReader
      ("C:\\dataFiles\\goodData\\data.txt"));
    ```

  - Note that in Windows **** must be used in place of ****, since a single backslash denotes an the beginning of an escape sequence

# Path Names

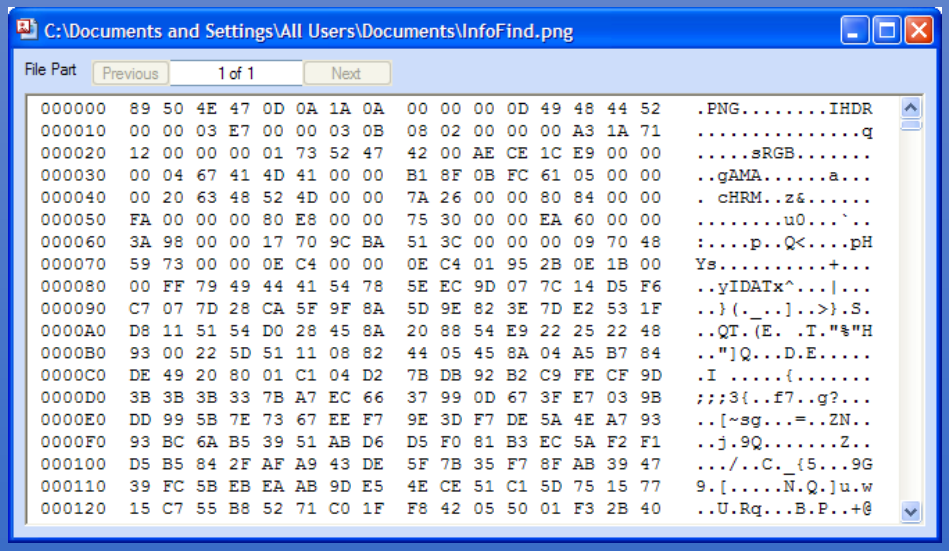

- A double backslash (`\\`) must be used for a Windows path name enclosed in a quoted string
  - This problem does not occur with path names read in from the keyboard

- Problems with escape characters can be avoided altogether by always using UNIX conventions when writing a path name
  - A Java program will accept a path name written in either Windows or Unix format regardless of the operating system on which it is run

# Nested Constructor Invocations


```
new BufferedReader(new FileReader("stuff.txt"))
```

- Above, the anonymous **`FileReader`** object establishes a connection with the **`stuff.txt`** file
  - However, it provides only very primitive methods for input
- The constructor for **`BufferedReader`** takes this **`FileReader`** object and adds a richer collection of input methods
  - This transforms the inner object into an instance variable of the outer object




# Binary Files

# Writing Simple Data to a Binary File


- The class **`ObjectOutputStream`** is a stream class that can be used to write to a binary file
  - An object of this class has methods to write strings, values of primitive types, and objects to a binary file

- A program using **`ObjectOutputStream`** needs to import several classes from package **`java.io`**:

```
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

# Opening a Binary File for Output


- An **`ObjectOutputStream`** object is created and connected to a binary file as follows:

```
ObjectOutputStream outputStreamName = new
                        ObjectOutputStream(new
                        FileOutputStream(FileName));
```

  - The constructor for **`FileOutputStream`** may throw a **`FileNotFoundException`**
  - The constructor for **`ObjectOutputStream`** may throw an **`IOException`**
  - Each of these must be handled

# Opening a Binary File for Output


- After opening the file, **`ObjectOutputStream`** methods can be used to write to the file
  - Methods used to output primitive values include **`writeInt`**, **`writeDouble`**, **`writeChar`**, and **`writeBoolean`**
- *UTF* is an encoding scheme used to encode Unicode characters that favors the ASCII character set
  - The method **`writeUTF`** can be used to output values of type **`String`**
- The stream should always be closed after writing

# Reading Simple Data from a Binary File


- The class **`ObjectInputStream`** is a stream class that can be used to read from a binary file
  - An object of this class has methods to read strings, values of primitive types, and objects from a binary file
- A program using **`ObjectInputStream`** needs to import several classes from package **`java.io`**:

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
```

# Opening a Binary File for Reading


- An **`ObjectInputStream`** object is created and connected to a binary file as follows:

```
ObjectInputStream inStreamName = new
                                 ObjectInputStream(new
                                 FileInputStream(FileName));
```

  - The constructor for **`FileInputStream`** may throw a **`FileNotFoundException`**
  - The constructor for **`ObjectInputStream`** may throw an **`IOException`**
  - Each of these must be handled

# Opening a Binary File for Reading


- After opening the file, **`ObjectInputStream`** methods can be used to read to the file
  - Methods used to input primitive values include **`readInt`**, **`readDouble`**, **`readChar`**, and **`readBoolean`**
  - The method **`readUTF`** is used to input values of type **`String`**
- If the file contains multiple types, each item type must be read in exactly the same order it was written to the file
- The stream should be closed after reading

# Checking for the End of a Binary File the Correct Way


- All of the **`ObjectInputStream`** methods that read from a binary file throw an **`EOFException`** when trying to read beyond the end of a file
  - This can be used to end a loop that reads all the data in a file
- Note that different file-reading methods check for the end of a file in different ways
  - Testing for the end of a file in the wrong way can cause a program to go into an infinite loop or terminate abnormally

# Binary I/O of Objects


- Objects can also be input and output from a binary file
  - Use the **`writeObject`** method of the class **`ObjectOutputStream`** to write an object to a binary file
  - Use the **`readObject`** method of the class **`ObjectInputStream`** to read an object from a binary file
  - In order to use the value returned by **`readObject`** as an object of a class, it must be type cast first:

```
SomeClass someObject =
   (SomeClass)objectInputStream.readObject();
```

# Binary I/O of Objects


- It is best to store the data of only one class type in any one file
  - Storing objects of multiple class types or objects of one class type mixed with primitives can lead to loss of data

- In addition, the class of the object being read or written must implement the ***`Serializable`*** *interface*
  - The **`Serializable`** interface is easy to use and requires no knowledge of interfaces
  - A class that implements the **`Serializable`** interface is said to be a *serializable class*

# The `Serializable` Interface


- In order to make a class serializable, simply add **`implements Serializable`** to the heading of the class definition

```
public class SomeClass implements Serializable
```

- When a serializable class has instance variables of a class type, then all those classes must be serializable also
  - A class is not serializable unless the classes for all instance variables are also serializable for all levels of instance variables within classes

# Array Objects in Binary Files


- Since an array is an object, arrays can also be read and written to binary files using **`readObject`** and **`writeObject`**
  - If the base type is a class, then it must also be serializable, just like any other class type
  - Since **`readObject`** returns its value as type **`Object`** (like any other object), it must be type cast to the correct array type:

```
SomeClass[] someObject =
  (SomeClass[])objectInputStream.readObject();
```

# Random Access to Binary Files


- The streams for sequential access to files are the ones most commonly used for file access in Java
- However, some applications require very rapid access to records in very large databases
  - These applications need to have random access to particular parts of a file

# Reading and Writing to the Same File


- The stream class **`RandomAccessFile`**, which is in the **`java.io`** package, provides both read and write random access to a file in Java
- A random access file consists of a sequence of numbered bytes
  - There is a kind of marker called the *file pointer* that is always positioned at one of the bytes
  - All reads and writes take place starting at the *file pointer location*
  - The file pointer can be moved to a new location with the method **`seek`**

# Reading and Writing to the Same File


- Although a random access file is byte oriented, there are methods that allow for reading or writing values of the primitive types as well as string values to/from a random access file
  - These include **`readInt`**, **`readDouble`**, and **`readUTF`** for input, and **`writeInt`**, **`writeDouble`**, and **`writeUTF`** for output
  - It does no have **`writeObject`** or **`readObject`** methods, however

# Opening a File


- The constructor for **`RandomAccessFile`** takes either a string file name or an object of the class **`File`** as its first argument
- The second argument must be one of four strings:
  - **`"rw"`**, meaning the code can both read and write to the file after it is open
  - **`"r"`**, meaning the code can read form the file, but not write to it
  - **`"rws"`** or **`"rwd"`** (See Table of methods from **`RandomAccessFile`**)

# A Random-Access File Need Not Start Empty


- If the file already exists, then when it is opened, the length is not reset to 0, and the file pointer will be positioned at the start of the file
  - This ensures that old data is not lost, and that the file pointer is set for the most likely position for reading (not writing)

- The length of the file can be changed with the **`setLength`** method
  - In particular, the **`setLength`** method can be used to empty the file

# Example Random Access files 1/2

```
public class DataCollection implements
Serializable{

private int x;

private int y;


 DataCollection(int xpoint, int ypoint){

 x=xpoint;

 y=ypoint;   }

public void setX(int x) {

this.x = x; }

public void setY(int y) {

this.y = y; }

public int getX() {

return x;   }

public int getY() {

return y;   }
```

```
private void doBWrite() {

ArrayList<DataCollection> list = new
ArrayList<DataCollection>();

list.add(new DataCollection(0, 0));

list.add(new DataCollection(0, 1));

list.add(new DataCollection(0, 2));

list.add(new DataCollection(0, 3));

list.add(new DataCollection(1, 0));

list.add(new DataCollection(2, 0));

list.add(new DataCollection(3, 0));

//send this list to Binary file

File f = new File("file.dat");

ObjectOutputStream objectToStream;

FileOutputStream BinaryOutFileStream;

try {

 BinaryOutFileStream = new FileOutputStream(f);

 objectToStream = new ObjectOutputStream(BinaryOutFileStream);

for (DataCollection d : list){

objectToStream.writeObject(d); }

BinaryOutFileStream.close();

objectToStream.close();

} catch (IOException e) {

e.printStackTrace();

} }
```

# Example Random Access files 2/2

```
private void doBRead()  {
File f = new File("file.dat");
FileInputStream BinaryInFileStream;
ObjectInputStream objectToStream;
try {
 BinaryInFileStream = new FileInputStream(f);
 objectToStream = new ObjectInputStream(BinaryInFileStream);
DataCollection obj =(DataCollection)objectToStream.readObject();;
while (obj!=null){
readData.append("("+obj.getX()+","+obj.getY()+")\n");
obj = (DataCollection)objectToStream.readObject();
}
BinaryInFileStream.close();
objectToStream.close();

} catch (Exception e) {
// TODO Auto-generated catch block
//e.printStackTrace();
}

}
```

# Summery


- Streams
- Files Types
  - Text Files
  - Binary Files
- Text files Processing
  - Write / Appending
  - Read
  - Path Names
- Binary files Processing
  - Serializable objects
  - Write
  - Read