Unit 9:

File I/O and File Manipulation

Object-Oriented Programming (OOP)
CCIT4023, 2025-2026

U9: File I/O and File Manipulation

- File Input and Output
  - File Stream
  - Text vs. Binary File
- Common File Manipulation
  - With The **`JFileChooser`** Class
  - Copy, Rename and Delete Files
- Text File Input and Output
- Low-Level File I/O (in Bytes)
- High-Level File I/O (in Primitive Types)
- Object-Level File I/O

File Input and Output

- Data stored in memory for processing during program execution are not permanent. They will be deleted / released when they are out of scope (e.g. a local variable within a method will be deleted after the method execution) or the program terminates.
- To permanently retain data even after the program execution, we store data in files, such as in secondary storage: hard disks, optical disks
- File data are often categorized into two forms when processing:
  - *Character-based data* stored in a **text file**, in human-readable form
    - Numeric data could be mapped to character symbols with character coding schemes (e.g. ASCII and Unicode), such as most *.txt files
  - *Byte-based data* stored in a **binary file**, which is better read by programs
    - Data are stored as raw numeric values, often could not be directly mapped to characters symbols, such as image JPEG files
- The advantage of binary files is that very often they are more efficient to process than text files

File Input and Output

Reference Only

- Example of ASCII code table (e.g. code value 65 is `'A'`)

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

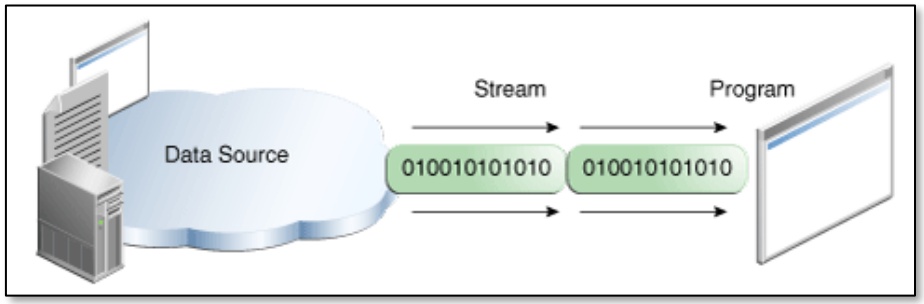
File Input and Output

Reference Only

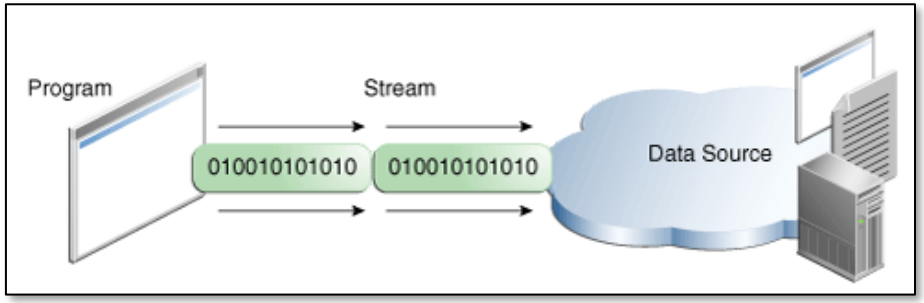
- Example of *partial* Unicode code table

File Input and Output
(File and File Streams)

- Java views file as a sequential **stream** of bytes or characters
- File streams can be used to input and output data as bytes or characters



Reading information into a program,
via an *input* stream



Writing information from a program,
via an *output* stream

File Input and Output
(Text vs. Binary File I/O)

- Java Text File I/O requires encoding and decoding (via Character-based streams)
  - Write : Unicode → File
  - Read : File → Unicode
- Java Binary File I/O does not require conversions (via Byte-based streams)
  - Write : original byte is copied into the file
  - Read : exact byte in the file is returned
- Many basic File handling resources could be found in the `java.io` package (and in newer version, `java.nio.file` package)
- File manipulation in Java often involves *exception handling*, e.g.
  - `IOException`, `FileNotFoundException`

File Input and Output
(The `File` Class)

- To operate on a file, we first create a **`File` object** (from package `java.io`). For example:

```
File inFile = new File("sample.txt");
```

Opens the file `sample.txt` in the current directory

```
File inFile = new File
        ("C:/OOP/test.txt");
```

Opens the file `test.txt` in the directory `C:\OOP` using the generic file separator '/' and providing the full pathname

Some `File` Methods


```
if ( inFile.exists( ) ) {
```

To see if **`inFile`** is associated to a real file correctly

```
if ( inFile.isFile() ) {
```

To see if **`inFile`** is associated to a file (`true`) or a directory (`false`)

List the name of all files in the directory `C:/OOP`

Example: TestMain.java

Reference Only

```
// import statements
import javax.swing.*;
import java.io.*;

public class TestMain {
  public static void main(String[] args) {

    String fileName = "sample.txt";
    File inFile = new File(fileName);
    // If inFile is associated to a real file correctly,
    // show a message dialog "sample.txt exists"
    if ( inFile.exists() ) {
      JOptionPane.showMessageDialog(null, fileName + " exists.");
    } else {
      JOptionPane.showMessageDialog(null, fileName + " doesn't exist!");
    }

    // List the name of all files in the directory "D:/"
    File directory = new File("D:/");
    String fileName2[] = directory.list();
    for (int i = 0; i < fileName2.length; i++) {
      System.out.println(fileName2[i]);
    }
  }
}
```

Common File Manipulation

- Apart from basic reading and writing data, other file manipulation functions may support better file handling, including:
  - Choosing files
  - Copying, Renaming, and Deleting files
- It is common to use classes `java.io.`**`File`** (and in newer version, `java.nio.file.`**`Files`**) for basic file manipulation such as copying, renaming, and deleting files
- We may also use `javax.swing.JFileChooser` to choose/select a file via Graphical User Interface

Simple Copy, Rename and Delete File

With `java.io.File` (and in newer version, `java.nio.file.Files`), and the string name of source and target files, we can:

- **Copy** (from original file name String `srcFName` to `targetFName`)

```
java.nio.file.Files.copy(new File(srcFName).toPath(),
      new File(targetFName).toPath());
```

- **Rename** (from file name String `srcFName` to `targetFName`)

```
new File(srcFName).renameTo(new File(targetFName))
```

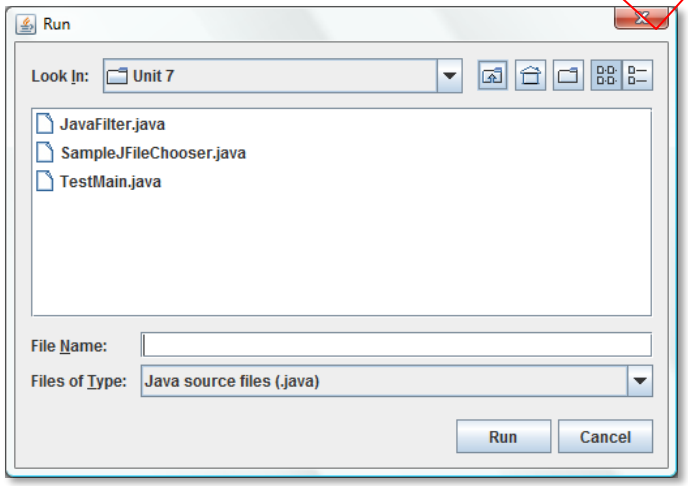
- **Delete** (the file name String `srcFName`)

```
new File(srcFName).delete();
```

The `JFileChooser` Class

Reference Only

- `JFileChooser` allows the user to choose/select a file via Graphical User Interface
  - In package `javax.swing`



Default directory:

```
JFileChooser chooser = new JFileChooser( );
chooser.showOpenDialog(null);
```

To start the listing from a specific directory with customized label:

```
JFileChooser chooser = new JFileChooser("D:/Unit7");
chooser.showDialog(null, "Run");
```

Getting Info from `JFileChooser`

Reference Only

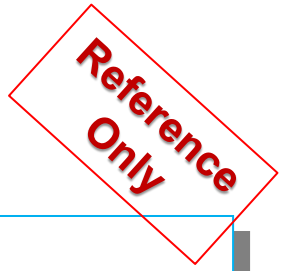
```
int status = chooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION) {
    JOptionPane.showMessageDialog(null, "Open is clicked");

} else { //== JFileChooser.CANCEL_OPTION
    JOptionPane.showMessageDialog(null, "Cancel is clicked");
}
```

```
File selectedFile  = chooser.getSelectedFile();
```

```
String fileName = selectedFile.getName();
```

```
File currentDirectory = chooser.getCurrentDirectory();
```

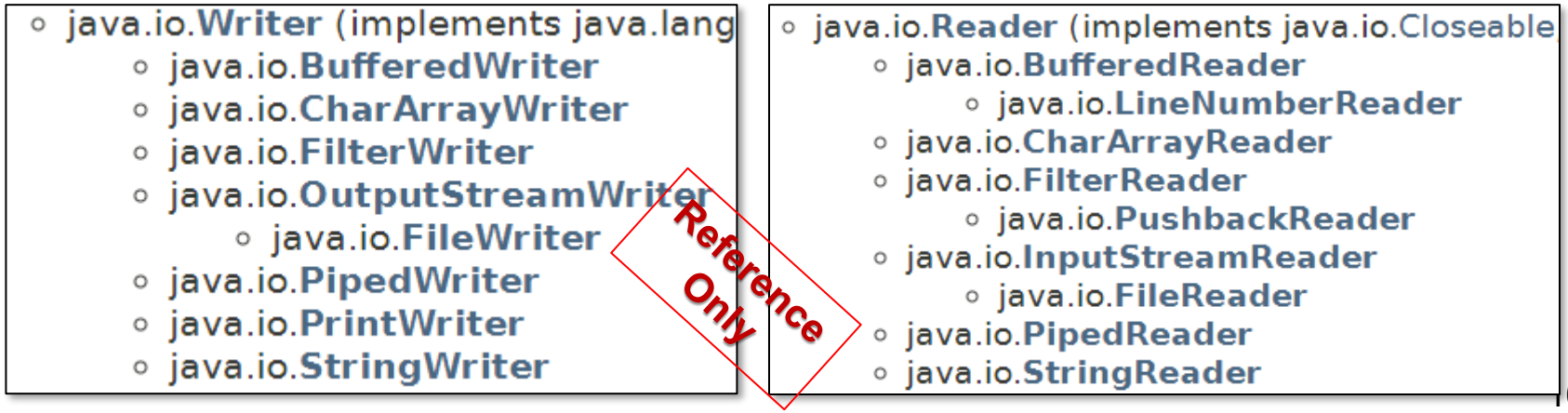
Example: SampleJFileChooser.java




```
import javax.swing.*;
import java.io.*;

public class SampleJFileChooser {
    public static void main(String[] args) {
        JFileChooser chooser;
        File file;
        int status;
    // start the listing from a specific directory "D:/"
        chooser = new JFileChooser("./");


    // show the dialog with customized label "Browse"
        status = chooser.showDialog(null,"Browse");

        if (status == JFileChooser.APPROVE_OPTION) {
           file = chooser.getSelectedFile();
    // show the message dialog "Open File: <Your selected file name>"
           JOptionPane.showMessageDialog(null, "Open File: " + file.getName());
        } else {
           JOptionPane.showMessageDialog(null, "Open File dialog cancelled");
        }
    }
}
```

Text File Input and Output

- **Handling data as string text**
  - This allows us to view the file content using any text editor
- To output/write data as a string to file, we may use `PrintWriter` or `FileWriter` classes (in package `java.io`)
- To input/read data from a text file, we may use `FileReader` and `BufferedReader` classes (in package `java.io`)
- From Java 5.0 (SDK 1.5), we can also use the `Scanner` class for inputting text files



Text File Input and Output
(Simple Sample Codes, for Writing / Appending Text File)

- To **write/append text data as strings into a text file**, we may use `PrintWriter` or `FileWriter` classes, with methods below:

1) To **create and open** an output stream **for writing** text, e.g.:

```
String fileName = "myFile.txt"; // a file name (string)
PrintWriter outStream = new PrintWriter(fileName);
```

– To create and open an output stream **for appending** text, e.g.:

```
String fileName = "myFile.txt";
PrintWriter outStream = new PrintWriter(
        new FileWriter(fileName, true)); // true for appending
```

2) To **write a string line** to text output stream (with newline), e.g.:

```
String aStr = "This is a string line.";
outStream.println(aStr); // end with a newline '\n' added
```

3) To **close the stream**, e.g.:

```
outStream.close(); //close the stream
```

Text File Input and Output
(Simple Sample Codes , for Reading Text File)

- To **input/read string data from a text file**, we may use `FileReader` and `BufferedReader` classes, with methods below:

1) To **create and open** an input stream for **reading text**, e.g.:

```
BufferedReader bufferReader =
        new BufferedReader(new FileReader(fileName));
```

2) To **read a string line** from text input stream, e.g.:

```
// below read a line, NOT including the newline '\n'
String strLine = bufferReader.readLine();
```

3) To **close the stream**, e.g.:

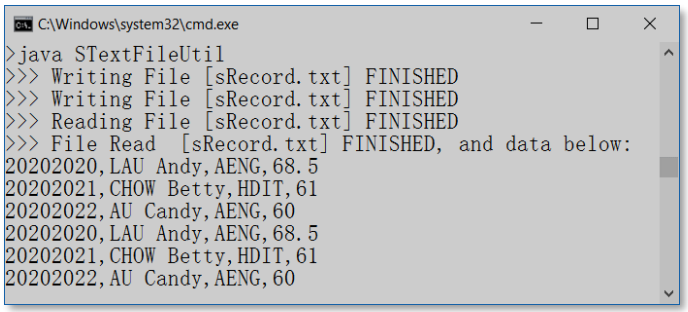
```
bufferReader.close(); //close the stream
```

Example: Text File Input / Output
(`main()` Method for Testing)

Reference Only

STextFileUtil.java

```
// STextFileUtil.java
import java.io.*;
import java.util.*;
public class STextFileUtil  {
    public static void main(String[] args){  // main, for testing
        String fName = "sRecord.txt";
        String [] fData = {"20202020,LAU Andy,AENG,68.5",
                           "20202021,CHOW Betty,HDIT,61",
                           "20202022,AU Candy,AENG,60"};

        STextFileUtil.writeTextFile(fName, fData);
        STextFileUtil.appendTextFile(fName, fData);
        String [] fDataStrs = STextFileUtil.readTextFile(fName);
        System.out.println(">>> File Read  [" + fName
                    + "] FINISHED, and data below:" );
        for (int i=0; i<fDataStrs.length; i++)
            System.out.println(fDataStrs[i]);
    }

// continue ...
// More coming

}
```



Text file, e.g. sRecord.txt

```
20202020,LAU Andy,AENG,68.5
20202021,CHOW Betty,HDIT,61
20202022,AU Candy,AENG,60
20202020,LAU Andy,AENG,68.5
20202021,CHOW Betty,HDIT,61
20202022,AU Candy,AENG,60
```

Example: Text File Output
(Write – may overwrite all current data, if file exists)

Reference Only

STextFileUtil.java

```
// continue ...

    public static boolean writeTextFile(String fileName,
                                String [] fileContent){
        try{
            PrintWriter outStream = new PrintWriter(fileName);
            for (int i=0; i<fileContent.length; i++)
                outStream.println(fileContent[i]); //write into file, in a line
            outStream.close(); //close the stream
            System.out.println(">>> Writing File [" + fileName + "] FINISHED" );
        } catch (FileNotFoundException fnfE){
            System.out.println(">>> Exception, FileNotFoundException");
            return false;
        }
        return true;
    }
// continue ...
```

Text file, e.g.
sRecord.txt

```
20202020,LAU Andy,AENG,68.5
20202021,CHOW Betty,HDIT,61
20202022,AU Candy,AENG,60
```

Example: Text File Output
(Append – Append data at file end, if file exists)

Reference Only

STextFileUtil.java

```
// continue ...

    public static boolean appendTextFile // class method to append
                  (String fileName, String [] fileContent) {
        try{
            //Creates a new PrintWriter, with specified file name
            PrintWriter outStream = new PrintWriter(
                 new FileWriter(fileName, true)); // true for appending
            for (int i=0; i<fileContent.length; i++)
                outStream.println(fileContent[i]);// write into file, in a line
            outStream.close(); //close the stream
            System.out.println(">>> Writing File [" + fileName + "] FINISHED" );
        } catch (IOException ioE){
            System.out.println(">>> Exception, FileNotFoundException");
            return false;
        }
        return true;
    }

// continue ...
```

Text file, e.g. sRecord.txt

```
20202020,LAU Andy,AENG,68.5
20202021,CHOW Betty,HDIT,61
20202022,AU Candy,AENG,60
20202020,LAU Andy,AENG,68.5
20202021,CHOW Betty,HDIT,61
20202022,AU Candy,AENG,60
```

Text File Input (Read)
(Use `FileReader`)

Reference Only

STextFileUtil.java

```
// continue ...

    public static String[] readTextFile(String fileName){
        ArrayList<String> retAList = new ArrayList<String>(); // arraylist
        try{
            String strLine;
            BufferedReader bufferReader =
                                    new BufferedReader(new FileReader(fileName));
            // read file to string until end, one line a time
            while ((strLine = bufferReader.readLine()) != null) {
                retAList.add(strLine); // add the read line data to arraylist
            }
            bufferReader.close(); //close the stream
            System.out.println(">>> Reading File [" + fileName + "] FINISHED" );
        } catch (FileNotFoundException fnfE){
            System.out.println(">>> Exception, FileNotFoundException");
            return null;
        } catch (IOException ioE){
            System.out.println(">>> Exception, IOException");
            return null;
        }
        // ArrayList to String array with method toArray()
        return retAList.toArray(new String[retAList.size()]);
    }
```

Low-Level File I/O (in Bytes)

- Java offers many Byte-based stream classes to handle binary file I/O:
  - **Byte Streams** (`FileInputStream` / `FileOutputStream`) handle I/O of raw binary data.
  - **Data Streams** (`DataInputStream` / `DataOutputStream`) handle binary I/O of primitive data type and string values
  - **Object Streams** (`ObjectInputStream` / `ObjectOutputStream`) handle binary I/O of objects

- java.io.**InputStream** (implements java.io.Closeable)
  - java.io.**ByteArrayInputStream**
  - java.io.**FileInputStream**
  - java.io.**FilterInputStream**
    - java.io.**BufferedInputStream**
    - java.io.**DataInputStream** (implements java.io.DataInput)
    - java.io.**LineNumberInputStream**
    - java.io.**PushbackInputStream**
  - java.io.**ObjectInputStream** (implements java.io.ObjectInput, java.io.ObjectStreamConstants)

- java.io.**OutputStream** (implements java.io.Closeable, java.io.Flushable)
  - java.io.**ByteArrayOutputStream**
  - java.io.**FileOutputStream**
  - java.io.**FilterOutputStream**
    - java.io.**BufferedOutputStream**
    - java.io.**DataOutputStream** (implements java.io.DataOutput)
    - java.io.**PrintStream** (implements java.lang.Appendable, java.io.Closeable)
  - java.io.**ObjectOutputStream** (implements java.io.ObjectOutput, java.io.ObjectStreamConstants)

Low-Level File I/O (in Bytes)

- 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*** (byte stream) is a sequence of data items, usually 8-bit bytes
- Java has two types of streams: an *input stream* and an *output stream*
- An *input stream* has a source form which the data items come, and an *output stream* has a destination to which the data items are going

Streams for Low-Level File I/O (in Bytes)

- `FileOutputStream` and `FileInputStream` are two stream objects that facilitate file access
- `FileOutputStream` allows us to output ***a sequence of bytes*** (values of data type byte)
- `FileInputStream` allows us to read in an array of bytes.

  * A `java.io.FileNotFoundException` would occur if you attempt to create a `FileInputStream` with a nonexistent file

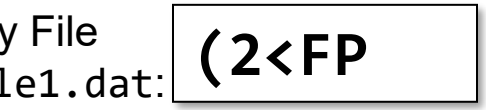
API references:
https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/io/FileInputStream.html
https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/io/FileOutputStream.html

Sample: Low-Level File Output (in Bytes)

Reference Only

TestFileOutputStream.java

Binary File sample1.dat:




```
//set up file and stream
FileOutputStream outStream = null;
try {
        outStream = new FileOutputStream ("sample1.dat");
        //data to save
       byte[] byteArray = {10, 20, 30, 40, 50, 60, 70, 80};
        //write data to the stream
        outStream.write( byteArray );
        //output done, so close the stream
        outStream.close();
} catch (IOException ioe){
        System.out.println("ERR: IOException");
}
```

Sample: Low-Level File Input (in Bytes)

TestFileInputStream.java

```
//set up file and stream
File inFile = new File( "sample1.dat" );
FileInputStream inStream = null;
try {
        inStream = new FileInputStream(inFile);
// we may also create directly, with specified file name
// outStream = new FileInputStream("sample1.txt");
        // set up an array to read data in
        int    fileSize  = (int) inFile.length();
        byte[] byteArray = new byte[fileSize];
        //read data in and display them
        inStream.read(byteArray);
        for (int i = 0; i < fileSize; i++){
                System.out.println(byteArray[i]);
        }
        //input done, so close the stream
        inStream.close();
} catch (IOException ioe){
        System.out.println("ERR: IOException");
}
```

Console output:

```
10
20
30
40
50
60
70
80
```

High-Level File I/O (in Primitive Types)

- `DataOutputStream` are used to output ***primitive data values***
  - take care of the details of converting the primitive data type values to a sequence of bytes
- `DataInputStream` is the reverse operation that read the data back from the files
- Set up a `DataOutputStream`, to write data to a file `"aFile.dat"`

```
dataOutS = new DataOutputStream(
               new FileOutputStream("aFile.dat"));
```

- Set up a `DataInputStream`, to read data from a file `"cFile.dat"`

```
dataInS = new DataInputStream(
              new FileInputStream("aFile.dat"));
```

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, e.g.

```
# writing data from an array of Circle, to DataOutputStream File
for (int i = 0; i < circleArr.length; i ++) {
    dataOutS.writeInt(circleArr[i].x);
    dataOutS.writeInt(circleArr[i].y);
    dataOutS.writeDouble(circleArr[i].radius);
}
```

Circle
x: int y: int radius: double

Writing different primitive type data to a `DataOutputStream` *object -* **`dataOutS`***:*

Reading different primitive type data frome a `DataInputStream` *object -* **`dataInS`***:*

```
# reading data from DataInputStream file, to an array of Circle
for (int i = 0; i < cInArr.length; i ++) {
    cInArr[i].x = dataInS.readInt();
    cInArr[i].y = dataInS.readInt();
    cInArr[i].radius = dataInS.readDouble();
}
```

Example: High-Level File Output
(in Primitive Types)

TestDataOutputStream.java

Binary File sample2.dat:

```
:Þh±K©ŠÇAInj€
```

```
import java.io.*;
public class TestDataOutputStream {
  public static void main (String[] args) throws IOException {
      //set up DataOutputStream object to write data to sample2.txt
      DataOutputStream outDataStream  = new DataOutputStream(
         new FileOutputStream("sample2.dat") );

      //write values of primitive data types to the stream
      outDataStream.writeInt(987654321);
      outDataStream.writeFloat(22222222F);
      outDataStream.writeDouble(3333333D);

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

Example: High-Level File Input
(in Primitive Types)

Reference Only

TestDataInputStream.java

Console output:

```
987654321
2.2222222E7
3333333.0
```

```
import java.io.*;
public class TestDataInputStream {
    public static void main (String[] args) throws IOException {
        // set up the DataInputStream object
        DataInputStream inDataStream = new DataInputStream(
            new FileInputStream("sample2.dat") );

        //read values back from the stream and display them
        System.out.println(inDataStream.readInt());
        System.out.println(inDataStream.readFloat());
        System.out.println(inDataStream.readDouble());

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

Object-Level File I/O

- It is possible to store objects just as easily as you store primitive data values
- We use `ObjectOutputStream` / `ObjectInputStream` to save to and load ***objects*** from a file
  - Object File I/O also includes exception handling and closing the stream
- To save objects from a given class, the class declaration must include the phrase implements `Serializable` (in the `java.io` package).

  E.g.

```
public class Person implements java.io.Serializable {
 //..
}
```

** There are certain restrictions of object stream I/O in Java. Therefore, sometimes it is a bit complicated and tricky to use this Object Stream with `Serializable`. Details should be reference to: http://docs.oracle.com/javase/25/docs/api/java/io/Serializable.html*

Saving and Reading Object Arrays

- Instead of processing array elements individually, it is even possible to save and read the whole array of objects at once

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

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

```
//read the array

Person[ ] people = (Person[]) inObjectStream.readObject( );
```

Saving Objects

Reference Only

```
FileOutputStream   outFileStream
                       = new FileOutputStream("objects.obj");

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

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

outObjectStream.writeObject( person );
```

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

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

Could save objects from the different classes

Example: Object Output

Reference Only

TestObjectOutputStream.java

```
import java.io.*;
public class TestObjectOutputStream {
    public static void main (String[] args) throws IOException {
        //set up file and stream
        FileOutputStream outFileStream
                        = new FileOutputStream("objects.obj");
        ObjectOutputStream outObjectStream
                        = new ObjectOutputStream(outFileStream);
        //write serializable Person objects
        Person person;
        for (int i = 0; i < 3; i++) {
                person = new Person("Mr. Espresso" + i, 20+i, 'M');
                outObjectStream.writeObject(person);
        }
        //input done, so close the stream
        outObjectStream.close();
    }
}
```

Object File Objects.obj:

```
秒  sr  Person 濼
-?   I  ageC  genderL  namet  Ljava/lang/String;xp     Mt
Mr. Espresso0sq ~       Mt
Mr. Espresso1sq ~       Mt
Mr. Espresso2
```

Reading Objects

Reference Only

TestObjectInputStream.java

```
FileInputStream    inFileStream
                       = new FileInputStream("objects.obj");
ObjectInputStream inObjectStream
                       = new ObjectInputStream(inFileStream);
```

```
Person person

      = (Person) inObjectStream.readObject( );
```

Must type cast to the correct object type.

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

Must read in the correct order.

Console output:

```
Mr. Espresso0      20      M
Mr. Espresso1      21      M
Mr. Espresso2      22      M
```

References

- This set of slides is only for educational purpose.

- Part of this slide set is referenced, extracted, and/or modified from the followings:
  - Deitel, P. and Deitel H. (2017) “Java How To Program, Early Objects”, 11ed, Pearson.
  - Liang, Y.D. (2017) “Introduction to Java Programming and Data Structures”, Comprehensive Version, 11ed, Prentice Hall.
  - Wu, C.T. (2010) “An Introduction to Object-Oriented Programming with Java”, 5ed, McGraw Hill.
  - Oracle Corporation, “Java Language and Virtual Machine Specifications” https://docs.oracle.com/javase/specs/
  - Oracle Corporation, “The Java Tutorials” https://docs.oracle.com/javase/tutorial/
  - Wikipedia, Website: https://en.wikipedia.org/