LESSON XI.
Input and Output Operations

Trinh Thanh TRUNG (MSc)
trungtt@soict.hust.edu.vn
094.666.8608

Objectives

- Become familiar with the concept of an I/O stream
- Understand the difference between binary files and text files
- Learn how to read data from a file
- Learn how to save data into a file

Content

I. **Basic input and output**
II. File Manipulation
III. Input and Output with Scanner and printf

I. Basic input and output

1. **Overview**
2. Model to do I/O

1. Overview – I/O

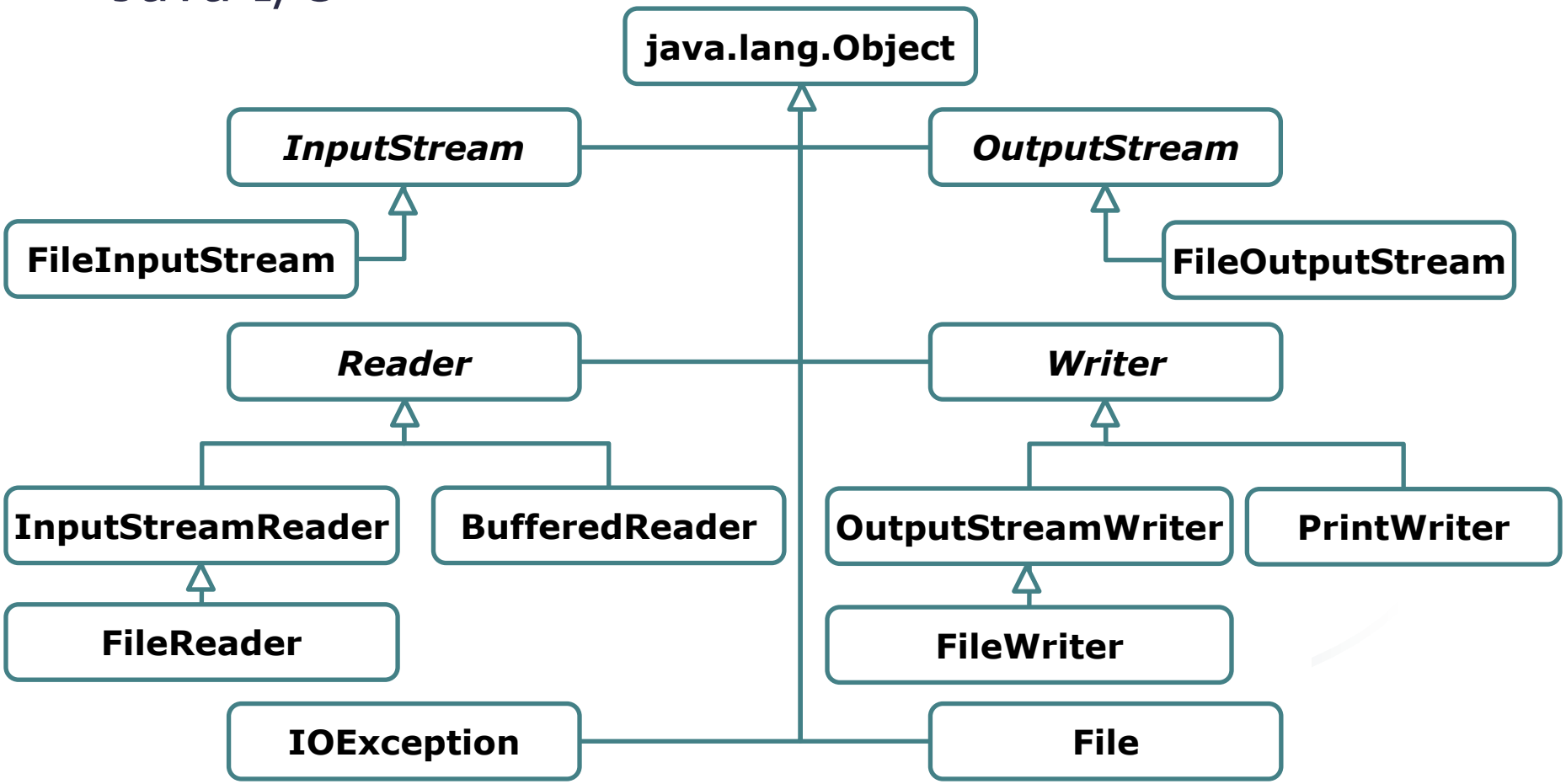
- I/O = Input/Output
- In this context it is input to and output from programs
- Input can be from keyboard or a file
- Output can be to display (screen) or a file
- Advantages of file I/O
  - permanent copy
  - output from one program can be input to another
  - input can be automated (rather than entered manually)

1. Overview-Streams

- All modern I/O is stream-based
- Stream: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer between the data source and destination
- Input stream: a stream that provides input to a program
  - System.in is an input stream
- Output stream: a stream that accepts output from a program
  - System.out is an output stream
- A stream connects a program to an I/O object
  - System.out connects a program to the screen
  - System.in connects a program to the keyboard

1. Overview - Package java.io

- This package offers the below classes that handle Java I/O



I. Basic input output

1. Over view
2. **Model to do I/O**

2. Model to do I/O

```
import java.io.*;
```

- ***Open*** the stream
- ***Use*** the stream (read, write, or both)
- ***Close*** the stream

Opening a stream

- There is data external to your program that you want to get, or you want to put data somewhere outside your program
- When you open a stream, you are making a connection to that external place
- Once the connection is made, you forget about the external place and just use the stream

Example of opening a stream

- A FileReader is used to connect to a file that will be used for input:

```
FileReader fileReader =
    new FileReader(fileName);
```

- The fileName specifies where the (external) file is to be found
- You never use fileName again; instead, you use fileReader

Using a stream

- Some streams can be used only for input, others only for output, still others for both
- *Using* a stream means doing input from it or output to it
- But it’s not usually that simple--you need to manipulate the data in some way as it comes in or goes out

Example of using a stream

```
int charAsInt;
charAsInt = fileReader.read( );
```

- fileReader.read():
  - reads one character
  - returns it as an integer,
  - returns -1 if there are no more characters to read

- You can *cast* from int to char:

```
char ch = (char) fileReader.read( );
```

Manipulating the input data

- BufferedReader class:
  - convert integers to characters
  - read whole lines

```
BufferedReader bufferedReader =
    new BufferedReader(fileReader);
```

Reading lines

```
String s;
s = bufferedReader.readLine( );
```

- A BufferedReader will return null if there is nothing more to read

Closing

```
bufferedReader.close( );
```

- You should not have more than one opened stream on the same file.
- You must close a stream before you can open it again
- Java will normally close your streams for you when your program ends, but it isn’t good style to depend on this

II. File manipulation

1. **Text files versus binary files**
2. `File` Class
3. Text files
4. Byte stream files
5. Object stream files-Serialization

1. Binary Versus Text Files

- *All* data and programs are ultimately just zeros and ones
  - each digit can have one of two values, hence *binary*
  - *bit* is one binary digit
  - *byte* is a group of eight bits
- *Text files*: the bits represent printable characters
  - one byte per character for ASCII, the most common code
  - for example, Java source files are text files
  - so is any file created with a "text editor"
- *Binary files*: the bits represent other types of encoded information, such as executable instructions or numeric data
  - these files are easily read by the computer but not humans
  - they are *not* "printable" files
    - actually, you *can* print them, but they will be unintelligible
    - "printable" means "easily readable by humans when printed"

Java: Text Versus Binary Files

- Text files are more readable by humans
- Binary files are more efficient
  - computers read and write binary files more easily than text
- Java binary files are portable
  - they can be used by Java on different machines
  - Reading and writing binary files is normally done by a program
  - text files are used only to communicate with humans

II. File manipulation

1. Text files versus binary files
2. **`File` Class**
3. Text files
4. Byte stream files
5. Object stream files-Serialization

2. `File` Class

- A `File` object represents an *abstract pathname*.
  - Represents both files and directories (folders).
  - Constructor takes the file's name:
    - `File info = new File("Letter.txt");`
  - No exception thrown if the file does not exist.
- Instances of the File class are immutable

2. File Class - Methods

```
File info = new File("Letter.txt");
if(info.exists()){
    System.out.println("Size of "+info.getName()+
                       " is "+info.length());
    if(info.isDirectory()){
        System.out.println("The file is a directory.");
    }
    if(info.canRead()){
        System.out.println("The file is readable.");
    }
    if(info.canWrite()){
        System.out.println("The file is writeable.");
    }
}
```

II. File manipulation

1. Text files versus binary files
2. `File` Class
3. **Text files**
4. Byte stream files
5. Object stream files-Serialization

3. Text files: `FileReader` and `FileWriter`

- FileReader: reading streams of characters
  - FileReader(String fileName)
  - read(): deals with `char` and `char[]`
- FileWriter: writing streams of characters.
  - FileWriter(String fileName)
  - write(): deal with `char` and `char[]`.

Opening and Closing a File

```
try{
    // Try to open the file.
    FileReader inputFile = new FileReader(filename);
    // Process the file's contents.
    ...
    // Close the file now that it is finished with.
    inputFile.close();
}
catch(FileNotFoundException e){
    System.out.println("Unable to open "+filename);
}
catch(IOException e){
    // The file could not be read or closed.
    System.out.println("Unable to process "+filename);
}
```

Copying a Text File (a block of chars)

```
static void copyFile(FileReader inputFile,
                     FileWriter outputFile)
                                  throws IOException {
    final int bufferSize = 1024;
    char[] buffer = new char[bufferSize];
    // Read the first chunk of characters.
    int numberRead = inputFile.read(buffer);
    while(numberRead > 0){
        // Write out what was read.
        outputFile.write(buffer,0,numberRead);
        numberRead = inputFile.read(buffer);
    }
    outputFile.flush();
}
```

Copying a Text File (char by char)

```
static void copyFile(FileReader inputFile,
                     FileWriter outputFile){
    try{
        // Read the first character.
        int nextChar = inputFile.read();
        // Have we reached the end of file?
        while(nextChar != -1){
            outputFile.write(nextChar);
            // Read the next character.
            nextChar = inputFile.read();
        }
        outputFile.flush();
    }
    catch(IOException e){
        System.out.println("Unable to copy a file.");
    }
}
```

Buffered Reader and Writers

```
try{
    FileReader in = new FileReader(infile);
    BufferedReader reader = new BufferedReader(in);
    FileWriter out = new FileWriter(outfile);
    BufferedWriter writer = new BufferedWriter(out);
    ...
    reader.close();
    writer.close();
}
catch(FileNotFoundException e){
    System.out.println(e.getMessage());
}
catch(IOException e){
    System.out.println(e.getMessage());
}
```

Line-by-Line Copying

```
BufferedReader reader = new BufferedReader(...);
// Read the first line.
String line = reader.readLine();
// null returned on EOF.
while(line != null){
    // Write the whole line.
    writer.write(line);
    // Add the newline character.
    writer.newLine();
    // Read the next line.
    line = reader.readLine();
}
```

PrintWriter

```
try{
    FileWriter out = new FileWriter(outfile);
    PrintWriter writer = new PrintWriter(out);

    writer.println(…);
    writer.close();
}
catch(IOException e){
    System.out.println(e.getMessage());
}
```

II. File manipulation

1. Text files versus binary files
2. `File` Class
3. Text files
4. **Byte stream files**
5. Object stream files-Serialization

4. Byte stream files: FileInputStream & FileOutputStream

- FileInputStream/FileOutputStream associates a binary input/output stream with an external file.
- All methods in FileInputStream/FileOuptputStream are inherited from its superclasses

FileInputStream

- To construct a FileInputStream, use the following constructors:
  - public FileInputStream(String filename)
  - public FileInputStream(File file)

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

FileInputStream

1. *read(): int*
2. read(b: byte[]): int
3. read(b: byte[], off: int, len: int): int
4. available(): int
5. close(): void
6. skip(n: long): long
7. markSupported(): boolean
8. mark(readlimit: int): void
9. reset(): void

FileOutputStream

- To construct a FileOutputStream, use the following constructors:
  - public FileOutputStream(String filename)
  - public FileOutputStream(File file)
  - public FileOutputStream(String filename, boolean append)
  - public FileOutputStream(File file, boolean append)

- If the file does not exist, a new file would be created.
- If the file already exists, the first two constructors would delete the current contents in the file. To retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter.

FileOutputStream

1. *write(int b): void*
2. write(b: byte[]): void
3. write(b: byte[], off: int, len: int): void
4. close(): void
5. *flush(): void*

FileOutputStream-Example

```
import java.io.*;
class FileOutputStreamDemo {
   public static void main(String args[]) throws Exception {
        String source = "Now is the time for all good men\\n"
                + " to come to the aid of their country\\n"
                + " and pay their due taxes.";
        byte buf[] = source.getBytes();

        OutputStream f0 = new FileOutputStream("file1.txt");
        for (int i=0; i < buf.length; i += 2) {
                f0.write(buf[i]);
        }
        f0.close();

        OutputStream f1 = new FileOutputStream("file2.txt");
        f1.write(buf);
        f1.close();

        OutputStream f2 = new FileOutputStream("file3.txt");
        f2.write(buf,buf.length-buf.length/4,buf.length/4);
        f2.close();
   }
}
```

FileOutputStream-Example

- **file1.txt**:
  Nwi h iefralgo e
  t oet h i ftercuty n a hi u ae.

- **file2.txt**:
  Now is the time for all good men
  to come to the aid of their country
  and pay their due taxes.

- **file3.txt**:
  nd pay their due taxes.

II. File manipulation

1. Text files versus binary files
2. `File` Class
3. Text files
4. Byte stream files
5. **Object stream files-Serialization**

5. Object stream files-Serialization

- You can also read and write *objects* to files
- Object I/O goes by the awkward name of serialization
- Serialization in other languages can be *very* difficult, because objects may contain references to other objects
- Java makes serialization (almost) easy

Conditions for serializability

- If an object is to be serialized:
  - The class must be declared as public
  - The class must implement Serializable
  - The class must have a no-argument constructor
  - All fields of the class must be serializable: either primitive types or serializable objects

Implementing Serializable

- To "implement" an interface means to define all the methods declared by that interface, but...
- The Serializable interface does not define any methods!
  - Question: What possible use is there for an interface that does not declare any methods?
  - Answer: Serializable is used as flag to tell Java it needs to do extra work with this class

Writing objects to a file

```
FileOutputStream fos = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(new Date());
oos.close();
```

Reading objects from a file

```
FileInputStream fis = new FileInputStream("t.tmp");
ObjectInputStream ois = new ObjectInputStream(fis);
int i = ois.readInt();
String today = (String) ois.readObject();
Date date = (Date) ois.readObject(); ois.close()
```

Content

I. Basic input output
II. File Manipulation
III. Input and Output with Scanner and printf

Predefined Streams in Java

- All Java programs automatically import the java.lang package.
- The java.lang package defines a class called System which encapsulates several aspects of the runtime environment.
- The System class contains 3 predefined stream variables called:
  - in, out, and err (System.in, System.out, System.err)
- These variables are declared as public and static with the System class.

Predefined Streams in Java: System class

- System.out refers to the standard output stream which is the console by default. (Monitor)
- System.in refers to the standard input stream which is the keyboard by default. (Keyboard)
- System.err refers to the standard error stream which is also the console by default. (Monitor)
- These streams may be redirected to any compatible I/O device.

Predefined Streams in Java: System class

- System.in is an object of type InputStream. (byte stream)
- System.out is an object of type PrintStream. (byte stream)
- System.err is an object of type PrintStream. (byte stream)
- They are all byte streams and are a part of the original Java specification.
- They are NOT character streams ! (Unicode character = 2 bytes)

java.util.Scanner

- Java finally has a fairly simple way to read input
- First, you must create a Scanner object
  - To read from the keyboard (System.in), do:
    - Scanner scanner = new Scanner(System.in);
  - To read from a file, do:
    - File myFile = new File("myFileName.txt");
      Scanner scanner = new Scanner(myFile);
    - You have to be prepared to handle a FileNotFound exception
  - You can even "read" from a String:
    - Scanner scanner = new Scanner(myString);
    - This can be handy for parsing a string

Using the Scanner

- First, you should make sure there is something to scan
  - scanner.hasNext() → boolean
  - You wouldn't use this when reading from the keyboard
- You can read a line at a time
  - scanner.nextLine() → String
- Or, you can read one "token" at a time
  - A token is any sequence of nonwhitespace characters
  - scanner.next () → String
- You must be prepared to deal with exceptions
  - An IDE will tell you what you need to do
- nextLine and next return Strings, which you can convert to numbers or other types if you like
- There are also methods to check for and return primitives directly

Scanning for primitives

- You can read in and convert text to primitives:
  - boolean b = sc.nextBoolean();
  - byte by = sc.nextByte();
  - short sh = sc.nextShort();
  - int i = sc.nextInt();
  - long l = sc.nextLong();
  - float f = sc.nextFloat();
  - double d = sc.nextDouble();
- And test if you have something to read:
  - hasNextBoolean()
  - hasNextByte()
  - hasNextShort()
  - hasNextInt()
  - hasNextLong()
  - hasNextFloat()
  - hasNextDouble()

Formatted output

- Java 5 has a printf method, similar to that of C
- Each format code is % width code
- Some format codes are s for strings, d for integers, f for floating point numbers
- Example:
  - ```
    double pi = Math.PI;
    System.out.printf("%8s %-8s %6d %-6d %8f %-8.2f :)\n",
                      "abc", "def", 123, 456, pi, pi);
    System.out.printf("%8s %-8s %6d %-6d",
                      "abcdef", "ghijkl", 12345, 6789);
    ```
  - Output:
    ```
        abc def          123 456    3.141593 3.14     :)
     abcdef ghijkl    12345 6789
    ```
    ```

Quick quiz (1/2)

- 1. Which import function we usually used when doing I/O with java
- 2. What is the superclass of all classes representing an input stream of bytes?
- 3. Why Java I/O is hard?
- 4. What data type the FileReader.read() method will return?
  a. short
  b. char
  c. int
  d. boolean
  e. String
  f. long

Quick quiz (2/2)

- 5. Which value will be return if we use method BufferedReader.readLine() method when there's nothing more to read?
  a. 0
  b. -1
  c. false
  d. null
- 6. Why we should wrap a BufferedReader around the FileReader ?
  a. It is faster
  b. It simplifies the file-reading proccess

Quiz

- Write 3 applications:
  - 1. Read from a keyboard to create a ArrayList of Triangle objects (you may modified several classes in the previous lesson). Write all objects of this list to a binary file
  - 2. Read all objects from your previous file and write to a text file
  - 3. Read the text file and display all data to the screen

Review

- Input and output
  - I/O model: open, use and then close the stream
- File manipulation
  - Text files:
    - Reading: FileReader, BufferedReader, Scanf
    - Writing: FileWriter, BufferedWriter, PrintWriter
  - Binary files:
    - Reading: FileInputStream
    - Writing: FileOutputStream
  - Serializable interface: for reading objects from and writing to binary files
- Input and Output with Scanner and printf
  - Scanner: read from the keyboard, from a file or even from a String.
  - printf: write a formatted string to an output stream