



# Computer Science Workshop-2 (CSE3141) (Input and Output)

by,  
Smita Mohanty  
Assistant Professor  
Department of Computer Science & Engineering  
ITER, SOA Deemed To Be University  
Email Id: [smitamohanty@soa.ac.in](mailto:smitamohanty@soa.ac.in)

# Java IO – Input Output

- Java input output deals with the java.io package and the various predefined classes defined in this API.
- It deals with the process of storing data permanently into a file, reading a file, also reading data from the keyboard and writing the data in the console.
- File handling in Java can be performed by Java I/O API.
- Uses the concept of Stream to make i/o operations fast.
- The java.io package contains all the classes required for input and output operations.

# Basic of File Handling



Here source can be a text file or networking socket program or a file or a command program.

And destination can be a file or console or socket.

# Stream

- Stream is a sequence of data.
- It is composed of bytes.
- `System.in`, `System.out` and `System.err` are the input , output and error stream respectively.
- All these streams are attached with the console.
- `InputStream` and `OutputStream` are the predefined abstract classes with some important methods defined inside them.

# InputStream class

- Java application uses an input stream to read data from a source; it may be a file, console or socket.
- `Int read()` – reads the next byte, returns -1 at the end of file.
- `Int available()` – returns the no. of available bytes
- `Void close()`- closes current input stream

# OutputStream class

- Java application uses an output stream to write data to a destination; it may be a file, peripheral device or socket.
- Void write(int) – writes byte
- Void write(byte[ ]) – writes array of byte
- Void flush() – flushes current stream
- Void close() – closes current stream

# Java I/O class

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode.

Table 10.1: Java I/O Class

Byte Based		Character Based	
Input	Output	Input	Output
FileInputStream	FileOutputStream	FileReader	FileWriter
BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
RandomAccessFile	RandomAccessFile		
StreamTokenizer	StreamTokenizer		

# Reading Standard Input

- Most desktop platforms support the notion of standard input (a keyboard, a file, or the output from another program) and standard output (a terminal window, a printer, a file on disk, or the input to yet another program).
- Most such systems also support a standard error output so that error messages can be seen by the user even if the standard output is being redirected.
- When programs on these platforms start up, the three streams are preassigned to particular platform-dependent handles, or file descriptors.



# Reading Standard Input

- Java continues this tradition and enshrines it in the `System` class. The static variables `System.in`, `System.out`, and `System.err` are connected to the three operating system streams before your program begins execution.
- So, to read the standard input, you need only refer to the variable `System.in` and call its methods.
- To read bytes, wrap a **`BufferedInputStream()`** around `System.in` . For the more common case of reading text, use an **`InputStreamReader`** and a **`BufferedReader`**.

# Reading Standard Input

- The standard input(stdin) can be represented by System.in in Java.
- The System.in is an instance of the InputStream class. It means that all its methods work on bytes, not Strings. To read any data from a keyboard, we can use either a Reader class or Scanner class.
- The particular subclass of Reader class that allows you to read lines of characters is a BufferedReader.
- To read from a Stream to a Reader, A “crossover” class called InputStreamReader is tailor-made for this purpose.

# Reading Standard Input

- Just pass your Stream (like `System.in`) to the `InputStreamReader` constructor and you get back a `Reader`, which you in turn pass to the `BufferedReader` constructor.
- You can then read lines of text from the standard input using the `readLine()` method.
- This method takes no argument and returns a `String` that is made up for you by `readLine()` containing the characters (converted to Unicode) from the next line of text in the file.
- If there are no more lines of text, the constant `null` is returned:

# Reading from the Console or Controlling Terminal; Reading Passwords Without Echoing

- The `Console` class is intended for reading directly from a program's controlling terminal.
- When you run an application from a “terminal window” or “command prompt window” on most systems, its console and its standard input are both connected to the terminal, by default.
- You cannot instantiate `Console` yourself; you must get an instance from the `System` class's `console()` method. You can then call methods such as `readLine()`, which behaves largely like the method of the same name in the `BufferedReader` class.

# Reading from the Console or Controlling Terminal; Reading Passwords Without Echoing

- The Console class is quite useful for reading a password without having it echo.
- The Console class has a readPassword() method that takes a prompt argument, intended to be used like: `cons.readPassword("Password:")`.
- This method returns an array of bytes, which can be used directly in some encryption and security APIs, or can easily be converted into a String.

# Writing Standard Output or Standard Error

- You want your program to write to the standard output or the standard error stream.
- Use `System.out` or `System.err` as appropriate.
- `System.out` is a `PrintStream` connected to the “standard output” – This is the standard output stream that is used to produce the result of a program on an output device like the computer screen.
- `System.err` is a `PrintStream` connected to “the standard error output”— This is the standard error stream that is used to output all the error data that a program might throw, on a computer screen or any standard output device.

# Printing with Formatter and printf

- The underlying Formatter class in `java.util` works on a String containing format codes.
- For each item that you want to format, you put a format code. The format code consists of a percent sign (%), optionally an argument number followed by a dollar sign (\$), optionally a field width or precision, and a format type (d for decimal integer, that is, an integer with no decimal point, f for floating point, and so on)



# Printing with Formatter and printf

% - *format code*  
1\$ - *use first arg (1951)*  
0 - *leading with 0 if needed*  
4 - *field width (4 digits)*  
d - *decimal integer (int)*

```
format("%1$04d - the year of %2$f", 1956, Math.PI);
```

% - *format code*  
2\$ - *use second arg (PI)*  
f - *floating point*



# Scanning Input with the Scanner Class

- Scanner class in Java is found in the `java.util` package.
- Java provides various ways to read input from the keyboard, the `java.util.Scanner` class is one of them.
- The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default.
- It provides many methods to read and parse various primitive values.
- By the help of Scanner in Java, we can get input from the user in primitive types such as `int`, `long`, `double`, `byte`, `float`, `short`, etc.

# Scanning Input with the Scanner Class

- The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.
- The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc.

# Various Ways to read data from keyboard

- Using InputStreamReader class
  - Connects input stream of keyboard
  - Converts byte oriented stream to character oriented stream
  - `InputStreamReader ir = new InputStreamReader(System.in);`
  - `BufferedReader br = new BufferedReader(ir);`
  - `Br.readLine();`
- Using Scanner class
- Using Console class

# Opening a File by Name

- Construct a `FileReader`, `FileWriter`, `FileInputStream`, or `FileOutputStream`
- To read a text file, you'd create, in order, a `FileReader` and a `BufferedReader`.
- To write a file a byte at a time, you'd create `FileOutputStream` and probably a `BufferedOutputStream` for efficiency.
- Remember that you need to handle **IOExceptions** around these calls.

# FileInputStream and FileOutputStream class

- Used to read input bytes from a file and write data to a file.
- Preferred when primitive data type values are to be written.  
(both byte and character oriented data can be written)
- `FileReader` and `FileWriter` classes work on character oriented data.

# FileWriter and FileReader Class

- Java FileWriter class is used to write character-oriented data to a file.
- It is character-oriented class which is used for file handling in java.
- Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

# Copy a File

- You need to copy a file in its entirety.

# BufferedInputStream and BufferedOutputStream Class

- Used to read information from stream.
- It internally uses **buffer mechanism** to make the performance fast.
- used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.



# Reading/Writing Binary Data

- Problem:
- You need to read or write binary data, as opposed to text.
- Solution:
- Use a `DataInputStream` or `DataOutputStream`.
- Java `DataInputStream` class allows an application to read primitive data from the input stream in a **machine-independent way**.
- Java application generally uses the data output stream to write data that can later be read by a data input stream.

# Seeking to a Position within a File

- **Problem**
- You need to read from or write to a particular location in a file, such as an indexed file.
- **Solution**
- Use a **RandomAccessFile**.
- **Discussion**
- The class `java.io.RandomAccessFile` allows you to move the read or write position when writing to any location within a file or past the end.

# Seeking to a Position within a File

- For controlling the file pointer, the `RandomAccessFile` class provides the following methods:
- The primary methods of interest are **`void seek(long where)`**, which moves the position for the next read or write to where;
- **`int skip-Bytes(int howmany)`** , which moves the position forward by how many bytes;
- **`long getFilePointer()`**, which returns the position.

# Saving and Restoring Java Objects

- Problem
- You need to write and (later) read objects.
- Solution
- Use the object stream classes, `ObjectInputStream` and `ObjectOutputStream`.
- These classes are used to serialize objects and store them as a file or any other storage accessible by Output Stream , read them again, deserialize it into an object and use it.

# Reading and Writing JAR or ZIP Archive

- Problem
- You need to create and/or extract from a JAR archive or a file in the well-known ZIP Archive format.
- Solution
- Use `ZipInputStream/ZipOutputStream` (in `java.util.zip`) and `ZipEntry` (in `java.util`) classes.

# Reading and Writing Compressed Files

- Problem
- You need to read or write files that have been compressed using GNU zip, or *gzip*. These files are usually saved with the extension *.gz*.
- Solution
- Use a `GZipInputStream` or `GZipOutputStream` as appropriate



THANK YOU