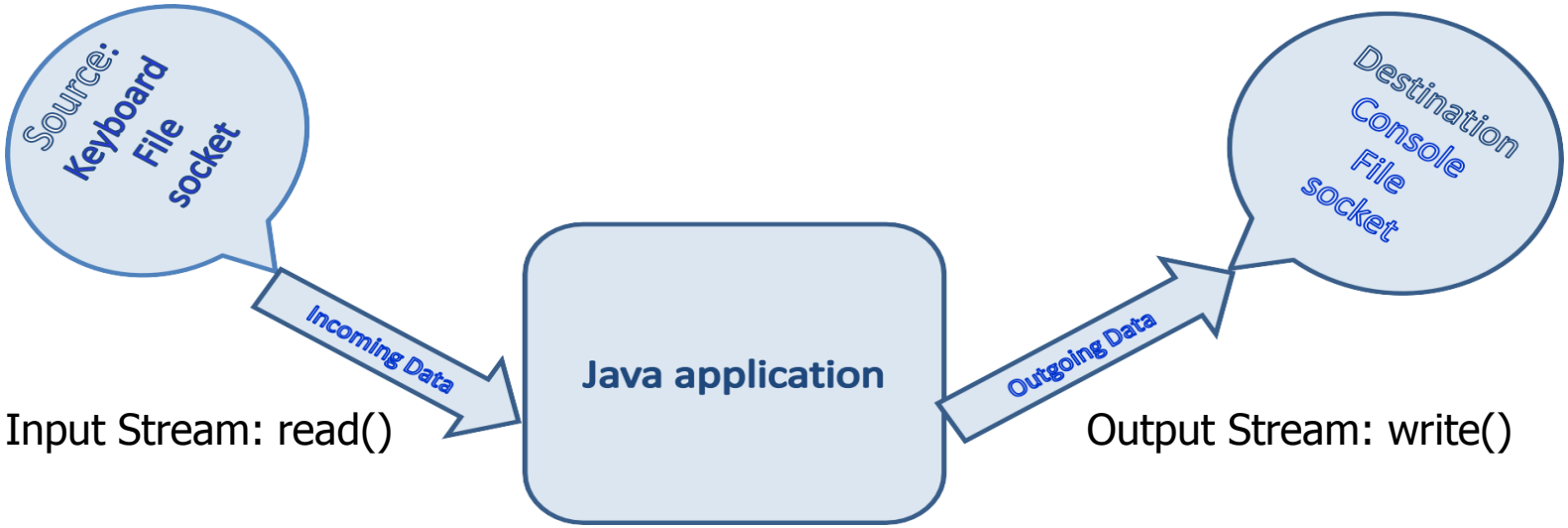
IO and Networking

Author & Presenter -Asfiya Khan
(Senior Technical Trainer)

Agenda

- Concept of Streams
- Binary Stream Vs Character Stream
- File – Reading and Writing
- Stream Layering and Handling Primitive Data
- Serialization and De-Serialization
- New File System API (NIO)
- Networking
- TCP/IP Sockets

Concept Of Streams



Flow of data is depicted as Stream

Input Stream : Flow of data from a source to java application

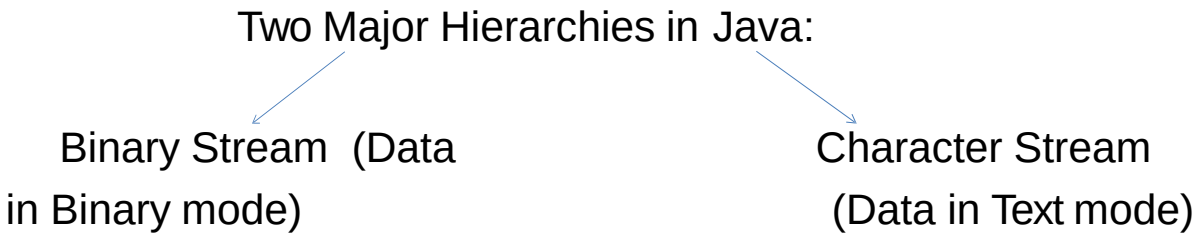
Output Stream: Flow of data from application to some destination

Concept Of Streams

I/O Fundamentals

- A stream can be thought of as a flow of data from a source or to a sink.
- A source stream initiates the flow of data, also called an input stream.
- A sink stream terminates the flow of data, also called an output stream.
- Sources and sinks are both node streams.

Binary Stream Vs Character Stream



- Java technology supports **two** types of streams: **character and byte.**
- Input and output of *character* data is handled by **readers and writers.**
- Input and output of byte data is handled by **Input streams and output streams:**
- Normally, the term stream refers to a byte stream.
- The terms reader and writer refer to character streams.

InputStream and OutputStream Methods

The InputStream Methods:

The three basic read methods are:

int read()

int read(byte[] buffer)

int read(byte[] buffer, int offset, int length)

The OutputStream Methods:

The three basic write methods are:

void write(int c)

void write(byte[] buffer)

void write(byte[] buffer, int offset, int length)

Other methods include:

void close()

void flush()

Reader and Writer Methods

The Reader Methods

- The three basic read methods are:

```
int read()

int read(char[] cbuf)

int read(char[] cbuf, int offset, int
length)
```

The Writer Methods

- The basic write methods are:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- Other methods include:

```
void close()
void flush()
```

File Stream I/O

File Stream I/O

- For file input:

  Use the FileReader class to read characters.

  Use the BufferedReader class to use the readLine method.

- For file output:

  Use the FileWriter class to write characters.

  Use the PrintWriter class to use the print and println methods.

File – Reading and Writing

The java.io package enables you to do the following steps:

- Create File objects
- Manipulate File objects
- Read and write to file streams

Standard Streams

Standard Streams : feature provided by many operating systems.

- By default, they read input from the *keyboard*
- write output to display.
- They also support I/O operations on files.

- System.in : used to read input from the keyboard.
- System. out : used to write output to display.
- System. err : used to write error output

Standard Streams Cont....

System.in is a byte stream that has no character stream features.

To use Standard Input as a character stream, wrap System.in within the InputStreamReader as an argument.

```
InputStreamReader inp = new InputStreamReader(system.in);
```

Stream Layering & Handling Primitive Data

Apart from Binary or Text mode, data can also be read or written in ***primitive data type format*** like : int ,float ..
DataInput & DataOutput Streams are provided for this purpose.

If you want to read or write ***primitive data*** to file then, choose appropriate File Stream object & Enwrap it into DataStream object.
This is known as ***enlayering***.

Methods in DataInputStream: readInt() ,readFloat() ,readUTF()…. Methods in

DataOutputStream: writeInt() ,writeFloat() , writeUTF()….

Layering Demo

```
public static void main(String[] args) throws IOException {
  DataOutputStream dos=new DataOutputStream(new
    FileOutputStream("MyData"));

    dos.writeInt(1);
    dos.writeFloat(2.2f);
    dos.writeUTF("hi");

  DataInputStream dis=new DataInputStream(new
    FileInputStream("MyData"));

   System.out.println(dis.readInt());
  System.out.println(dis.readFloat());
  System.out.println(dis.readUTF());
  }
```

Serialization and De-Serialization

Java supports Persistence

The state of Java Objects can be stored permanently (to some file….)
Advantage: Future Reuse.

Serialization & Deserialization :

It is the process of saving an object's state to a sequence of bytes
&
the process of rebuilding those bytes into a live object at some future time

Streams & methods provided:

ObjectOutputStream : writeObject() ……. Serialization

ObjectInputStream : readObject() ……. Deserialization

Serializable : A tagging interface

- We need to implement Serializable interface to a class Object which we need to persist.

- Object Serialization is not possible if class does not implement ***Serializable*** interface.

- In such case, ***NotSerializableException*** will be thrown.

New File System API –NIO 2.0

- Those who worked with Java IO may still remember the headaches that framework caused. It was never easy to work seamlessly across operating systems or multi-file systems. There were methods such as delete or rename that behaved unexpected in most cases. Working with symbolic links was another issue.

- With the intention of solving the above problems with Java IO, Java 7 introduced an overhauled and in many cases new API.

- The NIO 2.0 has come forward with many enhancements. It's also introduced new classes to ease the life of a developer when working with multiple file systems.

Working With Path

- A new java.nio.file package consists of classes and interfaces such as Path, Paths, FileSystem,FileSystems and others.

- A Path is simply a reference to a file path. It is the equivalent (and with more features) to java.io.File. The following snippet shows how to obtain a path reference to the "temp" folder:

```
public voidpathInfo() {
  Path path= Paths.get("c:\Temp\temp");
   System.out.println("Number of Nodes:"+ path.getNameCount());
   System.out.println("File Name:"+ path.getFileName());
   System.out.println("File Root:"+ path.getRoot()); System.out.println("File
   Parent:"+ path.getParent());
}
```

Files In NIO

- Deleting a file or directory is as simple as invoking a delete method on Files (note the plural) class. The Files class exposes two delete methods, one that throws NoSuchFileException and the other that does not.

- The following delete method invocation throws NoSuchFileException, so you have to handle it:

- Files.delete(path);

  Where as Files.deleteIfExists(path) does not throw exception (as expected) if the file/directory does not exist.

- You can use other utility methods such as Files.copy(..) and Files.move(..) to act on a file system efficiently. Similarly, use the createSymbolicLink(..) method to create symbolic links using your code.

NIO 2.0 Enhancement – File System Change Notification

File System Change Notification

- The java.nio.file package has a WatchService API to support file change notification.
- User can register directory/directories to watch and specify type of events to monitor. Various types of events are -

ENTRY_CREATE - A directory entry is created.

ENTRY_DELETE - A directory entry is deleted.

ENTRY_MODIFY - A directory entry is modified.

OVERFLOW - Indicates that events might have been lost or discarded. You do not have to register for the OVERFLOW event to receive it.

Once the event occurs, it is forwarded to registered process which is thread or pool of threads for handling the event.

NIO 2.0 Enhancement – File System Change Notification

```
FileSystem fileSystem = FileSystems.getDefault()
WatchService wService = fileSystem.newWatchService();  Path
dir = ...;
try {
     dir.register(wService, ENTRY_CREATE, ENTRY_DELETE);
     WatchKey key = wService.take(); //or wService.poll();
     :
} catch (IOException x) {
     System.err.println(x);
}
```

- First step is to create WatchService using FileSystem.newWatchService() method.
- Next step is to registered instance of WatchService with object that implements Watchable interface. Java.nio.file.Path implement Watchable interface.
- WatchService has methods take(), poll() and close()

NIO 2.0 Enhancement – File System Change Notification

```
for (;;) {
  WatchKey key = watcher.take();
  for (WatchEvent event : key.pollEvents()) {
    String file = object.context().toString();  if
    (event.kind() == ENTRY_DELETE) {
      System.out.println("Delete: " + file);
    } if (event.kind() == ENTRY_CREATE) {
      System.out.println("Created: " + file);
    }
  }
}
```

- Infinite for loop waits for incoming events. When event occurs key is signaled and place into watcher queue.
- Inner for loop retrieves pending events for the WatchKey & process as needed.
- Reset key and resume waiting for the events.

Networking

- The Internet is all about connecting machines together.
- One of the most exciting aspects of Java is that it incorporates an easy-to- use, cross-platform model for network communications.

Sockets

- The Internet Protocol breaks all communications into packets, finite- sized chunks of data which are separately and individually routed from source to destination.

- Socket is an abstraction that is provided to an application programmer to send or receive data to another process.

- Data can be sent to or received from another process running on the same machine or a different machine.

Sockets Contd..

- Socket is a combination of IP Address & Port number
- A port number is a logical no. assigned to a process to identify it on a given system
- Port number up to 1024 are reserved port numbers. Also known as well-known ports.

TCP/IP

- Transmission Control Protocol, TCP, establishes a connection between the two endpoints.

- The socket remains open throughout the duration of the communications.
- TCP is responsible for

  - breaking the data into packets,

  - buffering the data,

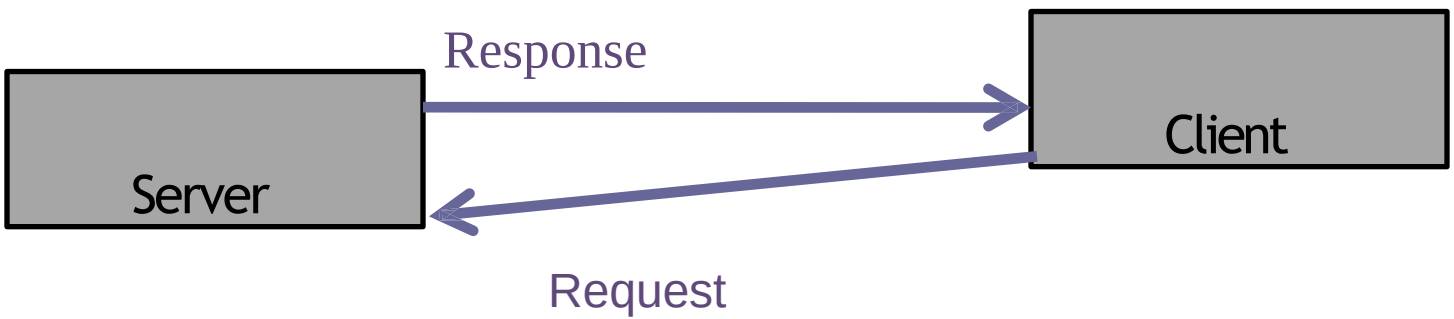
  - resending lost or out of order packets,

  -acknowledging receipt,

  -controlling rate of data flow.

Client Server Computing

- You can use the Java language to communicate with remote file systems using a client/server model.
- A server listens for connection requests from clients across the network or even from the same machine.
- Clients know how to connect to the server via an IP address and port number.



Client Server Computing Contd..

Creating TCP Clients

To create a TCP client, do the following:

1. Create a Socket object attached to a remote host, port.
2. Socket client = new Socket(host, port);

   When the constructor returns, you have a connection.

3. Get input and output streams associated with the socket.

Client Server Computing Contd..

Creating TCP Servers

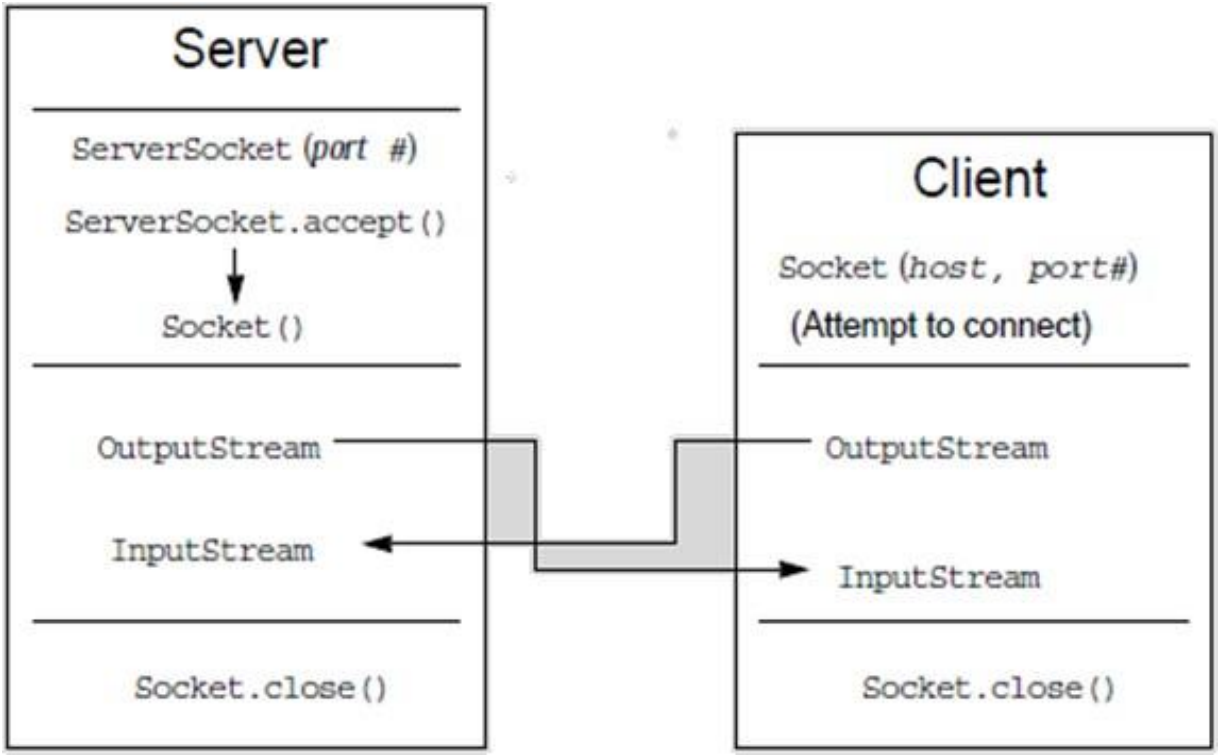
To create a TCP server, do the following:

1. Create a ServerSocket attached to a port number.
   ServerSocket server = new ServerSocket(port);

2. Wait for connections from clients requesting connections to that port.
   Socket channel = server.accept();
   You'll get a Socket object as a result of the connection.

3. Get input and output streams associated with the socket.

Client Server Computing Contd..



Client Server Computing Contd..

Client must contact server:

- server process must first be running.
- server must have created socket (door) that welcomes client's contact.

Client contacts server by:

- creating client-local TCP socket.
- specifying IP address, port number of server process.
- When client creates socket: client TCP establishes connection to server TCP.

When contacted by client, server TCP creates new socket for server process to communicate with client and allows server to talk with multiple clients on the basis of source port numbers.

Java TCP Socket Programming

Part of the java.net package

- import java.net.*;
- Provides two classes of sockets for TCP
- Socket – client side of socket
- ServerSocket – server side of socket

Java TCP Socket Programming

ServerSocket performs functions bind and listen

- Bind – fix to a certain port number
- Listen – wait for incoming requests on the port

Socket performs function connect

- Connect – begin TCP session

CYBAGE
Delivering Value. Scientifically.

Thank You!

Any Questions ?