File Input/Output & Networking

CS 18000
Sunil Prabhakar
Department of Computer Science
Purdue University

Objectives

- This week we will study
  - File input and output
    - Choosing files using JFileChooser
    - Reading from and writing to files
      - **Bytes**, **Primitive data types**,**Text** and **Objects**
    - Reading a text file using Scanner
  - Network I/O
    - Exchanging messages over the internet between independent Java programs
    - Creating client server solutions

Important Note

- The File IO API in Java was significantly revised in Jave SE 7.
  - The new version is an improvement
- However, in this course, we will continue to study the earlier version of this API.
- You can see how the two are related at the following set of webpages from Oracle:
  - http://docs.oracle.com/javase/tutorial/essential/io/legacy.html#mapping

File Input Output (IO)

Persistent Data

- Suppose we write a Bank application.
- How do we remember the account balances?
- What if the bank application stops running?
  - Runtime exception
  - Power failure?
  - …
- How do we ensure that when we restart the application, the balances are set correctly?

File I/O

- Remember that a computer system has two types of memory
  - Fast, volatile main memory
  - Slower, non-volatile secondary memory (disk)
- When a program runs, its variables (primitive and objects) are stored in main memory.
- When the program terminates, all these variables are lost!
- If we would like to preserve values of variables across multiple executions we must save them on disk and read them back in -- data on disks is stored in files.

File I/O

- Files can also be used to prevent loss of data due to system failures.
- Files can serve as a means for sharing data between different programs.
- Different operating systems manage files differently.
- Since Java is a HLL, we are mostly shielded from these differences.
- Java provides various classes for file I/O

The File Class

- To operate on a file, we must first create a File object (from *java.io*).

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

Opens the file myFile.txt in the current directory.

```
File inFile = new File
              ("/homes/sunil/Data/myFile");
```

Opens the file myFile in the directory /homes/ sunil/Data/ on a Unix, Linux, or Mac

```
File inFile = new File
    ("C:\\Users\\sunil\\MyDocuments\\myFile");
```

Opens the file myFile in the directory C:\Users\sunil\MyDocuments on a PC.

File names

- The rules for file names are determined by the operating system on which the Java program is run.
- Thus the name may or may not be case sensitive, or require filename extensions…
- Java simply passes the string to the operating system.

Some File Methods

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

To test if inFile is associated to an existing file correctly.

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

To test if inFile is associated to a file or not. If false, it is a directory.

```
File directory = new File("/users/sunil/Data");

String filename[] = directory.list();

for (int i = 0; i < filename.length; i++) {
    System.out.println(filename[i]);
}
```

List the name of all files in the directory /users/sunil/Data

The **JFileChooser** Class

- A **javax.swing.JFileChooser** object allows the user to select a file.

```
JFileChooser chooser = new JFileChooser( );

chooser.showOpenDialog(null);
```

To start the listing from a specific directory:

```
JFileChooser chooser = new JFileChooser("/users/sunil");

chooser.showOpenDialog(null);
```

Getting Info from JFileChooser

```
int status = chooser.showOpenDialog(null);

if (status == JFileChooser.APPROVE_OPTION) {
    JOptionPane.showMessageDialog(null, "Open was clicked");

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

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

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

Input/Output from a file

- Once we have a File object we can perform I/O on the file through that object
- I/O is handled using streams
- Create a *stream* object and attach to file
  - To read create an input stream object
  - To write create an output stream object
- Different classes of stream objects allows input/output of different types of data

I/O for Various Data Types

- Binary Files
  - Low-level
    - Bytes - arbitrary content can be written or read
- Text files
  - Similar to reading from System.in or writing to System.out
    - Data represented in Unicode format
    - Can be edited with a text editor
- High-Level
  - Primitive types and entire objects are read from or written

Opening and closing files

- When we create the stream object we open the file and connect it to the stream for input or output.
- Once we are done, we must close the stream. Otherwise, we may see corrupt data in the file.
- If a program does not close a file and terminates normally, the system closes the files for it.
- We must close a file after writing before we can read from it in the same program.

Streams for Low-Level File I/O

- To read bytes we create a *FileInputStream* object
  - A read returns an array of Bytes

- To write bytes, we create a *FileOutputStream* object
  - A write takes an array of Bytes

Sample: Low-Level File Output

```
byte[] byteArray = {10, 20, 30, 40, 50, 60, 70, 80};
File  outFile   = new File("myData");
FileOutputStream outStream;

try {
    outStream = new FileOutputStream( outFile );

    outStream.write( byteArray );

    outStream.write(90);

} catch (IOException e) {
    System.out.println("Error writing to file");

} finally {
    outStream.close();
}
```

Reading A Byte at a Time

```
File inFile   = new File("myData");
FileInputStream inStream;
byte readByte;

try {
    inStream = new FileInputStream(inFile);

    while( (readByte = inStream.read()) != -1)
        System.out.println(readByte);

} catch (IOException e) {
    System.out.println("Error writing to file");

} finally {
    outStream.close();
}
```

read() returns a byte or -1 if the end of the file has been reached.

Reading Multiple Bytes

```
File inFile   = new File("myData");
FileInputStream inStream;
int    fileSize  = (int)inFile.length();
byte[] byteArray = new byte[fileSize];
int numBytesRead;

try {
    inStream = new FileInputStream(inFile)
    numBytesRead = inStream.read(byteArray);
    for (int i = 0; i < numBytesRead; i++) {
        System.out.println(byteArray[i]);
} catch (IOException e) {
    System.out.println("Error writing to file");

} finally {
    outStream.close();
}
```

read(byte[] b) attempts to read as many as b.length bytes from the file.

read(byte[]) returns the total number of bytes read or -1 if the end of the file has been reached.

Text File Input and Output

- To output data as a string to file,
  - create a *FileOutputStream* object and attach it to the file
  - create a *PrintWriter* object for this stream
  - this object behaves like System.out
- To input data from a text file, we simply create a *Scanner* object and attach it to the file
  - behaves just like the scanner objects we have seen so far.

Sample Textfile Output

```
int i = 34;
File outFile = new File("myFile");
FileOutputStream outStream;
PrintWriter filePrinter;

try{
    outStream = new FileOutputStream(outFile);

    filePrinter = new PrintWriter(outStream);

    filePrinter.println("value is:" + i);

} catch (FileNotFoundException e){
    System.out.println("File " + inFile.getName()
            + " was not found or writeable");

} finally {
    outStream.close();
}
```

PrintWriter

- If a file with the given name exists it is opened for output and its current contents are lost.
- If we want to retain the old data, and append to the end of the file:

```
FileOutputStream outFileStream
        = new FileOutputStream(outFile, true);
```

then create the PrintWriter object with this stream object.

Sample Textfile Input

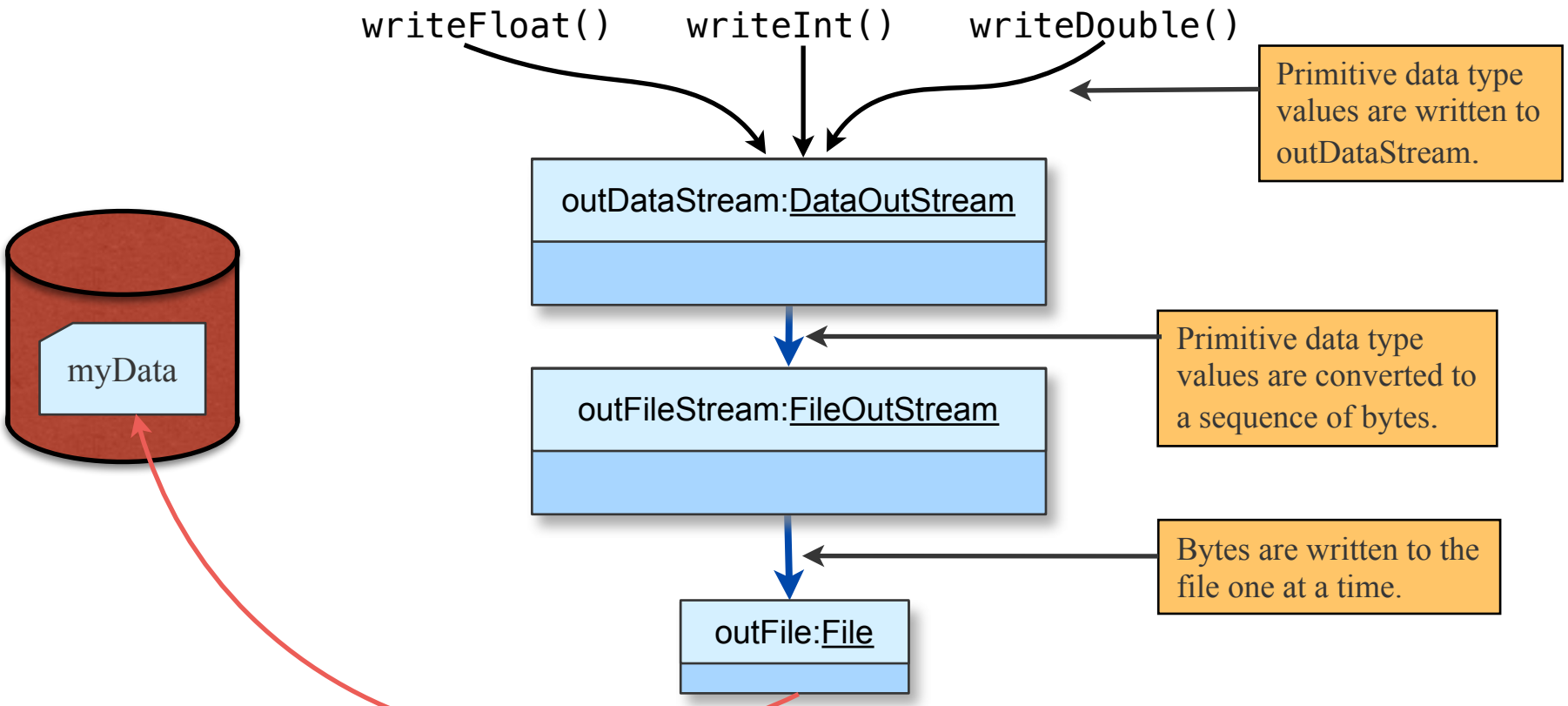
```
Scanner sc;

try{
    sc = new Scanner(new File(“myFile"));

    while(sc.hasNextInt())
        i = sc.nextInt();

} catch (FileNotFoundException e){
    System.out.println("File " + inFile.getName() +
            " was not found or was unreadable”);

} finally {
    sc.close();
}
```

Primitive Data IO

- In order to write/read primitive data types, we use DataOutputStream and DataInputStream objects
- Each of these attaches to a FileOutputStream (FileInputStream) object attached to a file.
- To read the data back correctly, we must know the order of the data stored and their data types

Writing Primitive Data

```
File             outFile       = new File( "myData" );
FileOutputStream outFileStream = new FileOutputStream(outFile);
DataOutputStream outDataStream = new DataOutputStream(outFileStream);
```

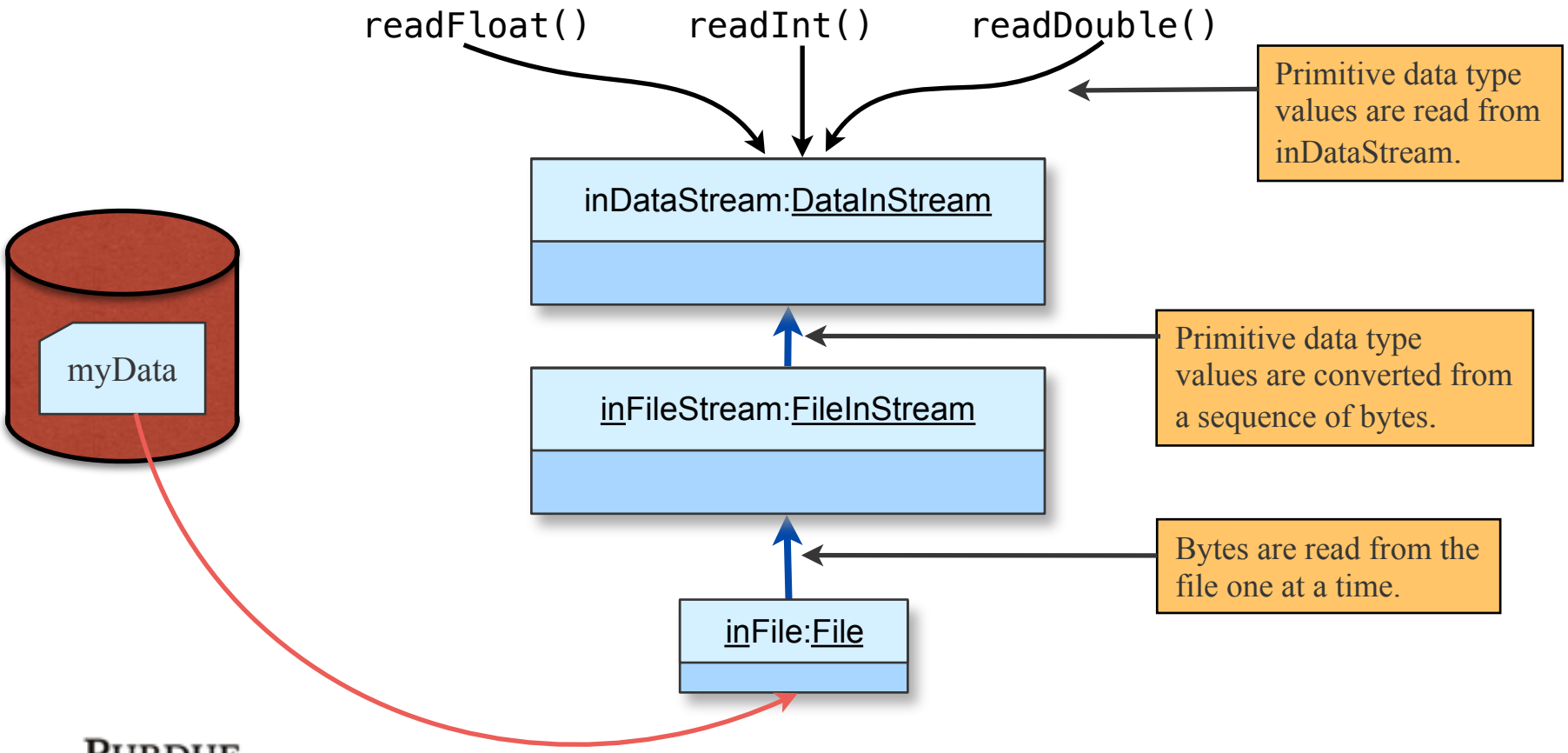


Sample Primitive Data Output

```
FileOutputStream outFileStream;
DataOutputStream outDataStream;
try {
    outFileStream = new
           FileOutputStream(new File( "myData" ));
    outDataStream
        = new DataOutputStream(outFileStream);

    outDataStream.writeInt(2345);
    outDataStream.writeLong(111111L);
    outDataStream.writeFloat(22222222F);
    outDataStream.writeDouble(3333333D);
    outDataStream.writeChar('A');
    outDataStream.writeBoolean(true);
} catch (FileNotFoundException e){
    System.out.println("File " + inFile.getName() +
              " was not found or was unreadable”);
} …
```

Reading Primitive Data

```
File            inFile       = new File( "myData" );
FileInputStream inFileStream = new FileInputStream(inFile);
DataInputStream inDataStream = new DataInputStream(inFileStream);
```

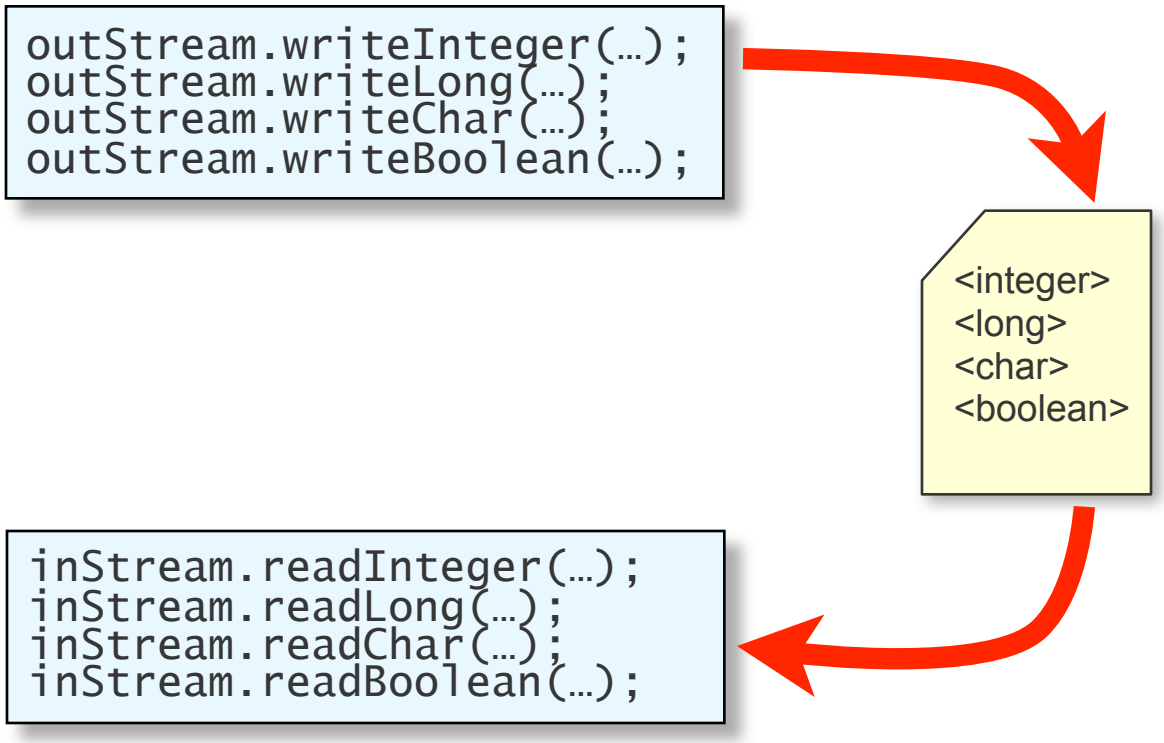


Sample Primitive Data Input

```
File inFile = new File( "myData" );
try {
    FileInputStream inFileStream
        = new FileInputStream(inFile);
    DataInputStream inDataStream
        = new DataInputStream(inFileStream);

    System.out.println(inDataStream.readInt());
    System.out.println(inDataStream.readLong());
    System.out.println(inDataStream.readFloat());
    System.out.println(inDataStream.readDouble());
    System.out.println(inDataStream.readChar());
    System.out.println(inDataStream.readBoolean());

} catch (FileNotFoundException e){
    System.out.println("File " + inFile.getName() +
            " was not found or was unreadable");
} …
```

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.



Object IO

- Consider saving an array of Student objects to a file and reading them back in.
- We could do this using the primitive data I/O methods discussed earlier, but it is quite tedious
- Java makes it easy to read/write an entire object at a time.

Object File I/O

- It is possible to store objects just as easily as you store primitive data values.
- We use *ObjectOutputStream* and *ObjectInputStream* to write and read objects from a file.
- To write an object to a file, the class for the object must implement the *Serializable* interface.
  - this is an empty interface, i.e., there are no methods to implement, simply declare that the class implements Serializable
  - note that any references in the object must also be serializable objects!

Writing Objects to a file

```
File   outFile = new File("objectFile");

try {
    FileOutputStream   outFileStream
        = new FileOutputStream(outFile);

    ObjectOutputStream outObjectStream
        = new ObjectOutputStream(outFileStream);

    int numInts = 10;

    Student[] roster = new Student[numInts];
       . . . // setup roster array
        . . .
    outObjectStream.writeInt( numInts );
    for(int i = 0; i < numInts; i++)
        outObjectStream.writeObject( roster[i] );
} catch (FileNotFoundException e) {
…
```

Reading Objects

```
File   inFile = new File("objectFile");

try {
    FileInputStream    inFileStream
        = new FileInputStream(inFile);

    ObjectInputStream inObjectStream
        = new ObjectInputStream(inFileStream);

    int numInts = inObjectStream.readInt();

    Student roster[] = new Student[numInts];

    for(int i = 0; i < numInts; i++)
        roster[i] = (Student) inObjectStream.readObject();
} catch (FileNotFoundException e) {
…
```

Note the need to typecast the read object to the correct type.

ObjectInput(Output)Stream

- Note that the readInt(), writeInt(), readByte(), … methods are also defined for the Object streams
- We can also write different types of objects to the same file
- Remember that all references in a serializable object must also be serializable in order to be written to a file
  - objects that can't be serialized may be declared transient

Saving and Loading Arrays

- Instead of processing array elements individually, it is possible to save and load the entire array at once.

```
Student[] roster = new Student[ numStudents ];

. . .

outObjectStream.writeObject ( roster );
```

```
Student[ ] roster = (Student[]) inObjectStream.readObject( );
```

Exceptions

- File I/O methods throw various types of exceptions including
  - IOException
  - FileNotFoundException
- Please see Java API to become familiar with these.
- Many of these need to be handled in order to make programs more robust.
- Labs and projects will cover some

Knowing when to stop reading

- It is possible to try to read beyond the end of the file.
- Different reader classes signal the end of file in different ways.
- Primitive Data (DataInputStrem) readers throw the EOFException.
- Text file readers return null or -1.
- You should be aware of how the common classes indicate the end of file condition.

Networking

Networking in Java

- Computer networks allow programs running on different computers to communicate (exchange data) with each other.
- Surprisingly, networking in Java is very similar to File I/O
  - both are based upon streams
- The key difference is that one uses File objects, the other uses Socket objects.

Networking Essentials

- Each computer that is connected to the Internet has an address, called the IP Address (try the *host* command)
  - 128.10.19.13 (data.cs.purdue.edu)
- Each computer uses one or more ports to exchange data.
  - Also known as sockets
  - A socket can receive or send data from/to the network
  - ports are numbered (e.g., 80 -- web server).

Client-Server

- A simple model for network communication is the client-server model
- A server is a well known network location
- The server is always listening for connections
- A client that wishes to communicate connects with the well-known server
- Once connected two-way communication is possible

A Server (Listener)

- A server program is typically one that runs on a well-known computer and listens to a specific port number
- This allows any computer on the Internet to communicate with this program.
- To be a server, A Java program
  - creates a ServerSocket object for a given port
  - waits for a connection by calling the accept() method on the ServerSocket

Being a server

- The accept() method returns a Socket object
- This object behaves like a File object
  - A socket has input and output streams
- Once we have a Socket object, we can get its input/output Stream object by calling:
  - getInputStream() or getOutputStream()
  - rest is similar to File I/O

Connecting to a Server

- In order to initiate a network connection, a Client program creates a Socket object by specifying the server:
  - IP Address and Port number
- Once the Socket object is created, situation is similar to a server socket
- Creating the socket object involves requesting and making a connection to the server which must accept the connection.

Server Receiving Data

```
public static void main(String[] args) throws IOException {

    ServerSocket serverSocket = null;
    Socket clientSocket = null;
    PrintWriter outToClient = null;
    BufferedReader inFromClient = null;


    try {
        System.out.println("Creating Socket");
        serverSocket = new ServerSocket(4444);

        System.out.println("Listening");
        clientSocket = serverSocket.accept();

        System.out.println("Got a request: " + clientSocket.getPort());

        outToClient = new PrintWriter(clientSocket.getOutputStream(), true);
        inFromClient = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));

        outToClient.println("Message from Server");

        inputLine = inFromClient.readLine();
          System.out.println("Client sent" + inputLine);

     . . .
```

Server Receiving Data

```
    try{
         . . .  //from previous slide

   } catch (MalformedURLException e) {
        System.err.println("Unable to connect:\n" + e.getMessage());
        System.exit(3);
    } catch (IOException e) {
        System.err.println("Error with IO:\n" + e.getMessage());
        System.exit(4);
    } finally {
        if (serverSocket != null)
            serverSocket.close();
        if (clientSocket != null)
            clientSocket.close();
        if (outToClient != null)
            outToClient.close();
        if (inFromClient != null)
            inFromClient.close();
    }
}
```

Client Connecting and Sending

```
public static void main(String args[]) throws IOException {
     int serverPort;
     Socket socket = null;
     PrintWriter outToServer = null;
     BufferedReader inFromServer = null;
     String inputLine;
     int inCount, outCount;

     try{
          System.out.println("trying to connect to localhost, port: 4444");
          socket = new Socket("localhost", 4444);

          System.out.println(“trying to create input and output”);

          outToServer = new PrintWriter (socket.getOutputStream(), true);
          inFromServer = new BufferedReader( new InputStreamReader (socket.getInputStream()));

          while ( (inputLine = inFromServer.readLine()) != null) {
               System.out.println("Recieved message  " + ++inCount + " from Server:\n" +
inputLine);
               if(outCount < 10)
                    outToServer.println("Message " + ++outCount + " to server from client");
               else
                    outToServer.println("LEAVE");
     }
     System.out.println("Closing up client");

. . .
```

Client Connecting and Sending

```
try{
        . . . //from previous slide

} catch (MalformedURLException e) {
    System.err.println("Error with URL:\n" + e.getMessage());
    System.exit(2);
} catch (IOException e) {
    System.err.println("Error with IO:\n" + e.getMessage());
    System.exit(3);
} finally {
    if(socket != null)
        socket.close();
    if(outToServer != null)
        outToServer.close();
    if(inFromServer != null)
        inFromServer.close();
}
```

Client-Server Setup

Server Machine

Create a ServerSocket

"Listen" for client connections on a specified port

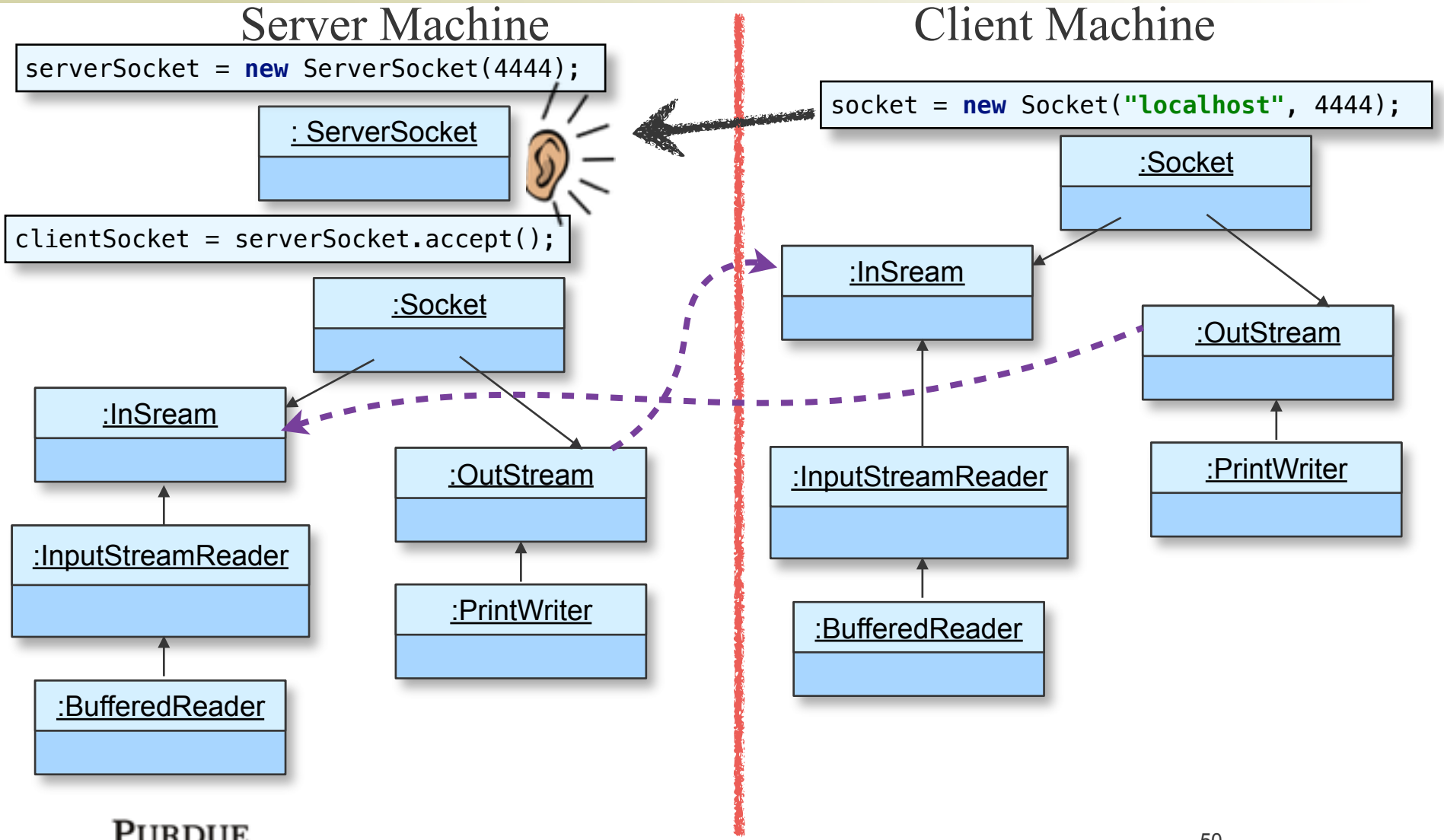
Create a socket for the connected client

Client Machine

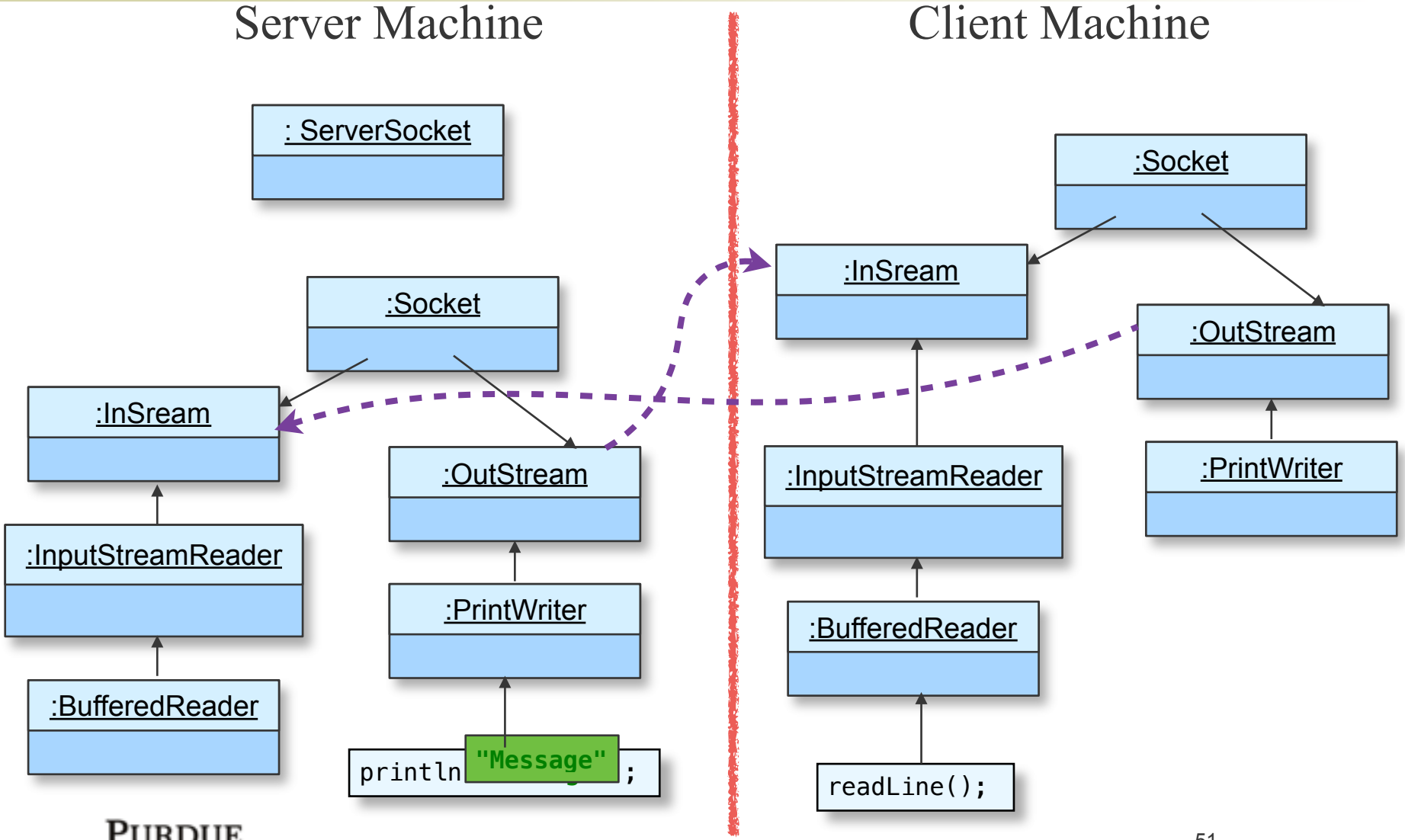
Create a Socket to connect to the server using the name of the server's machine and the port on which it is listening

Use this Socket to communicate with the Server

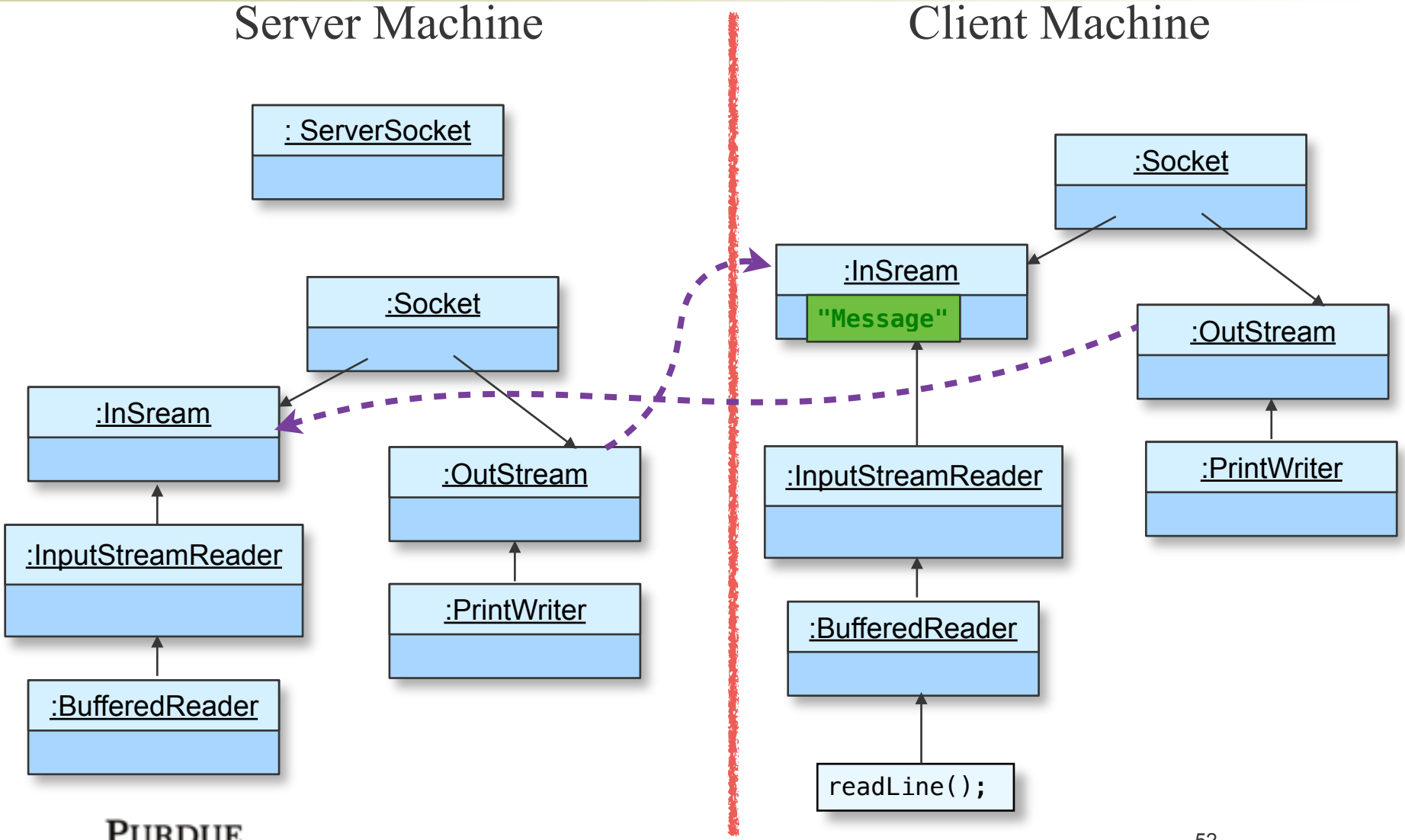
Client-Server Setup



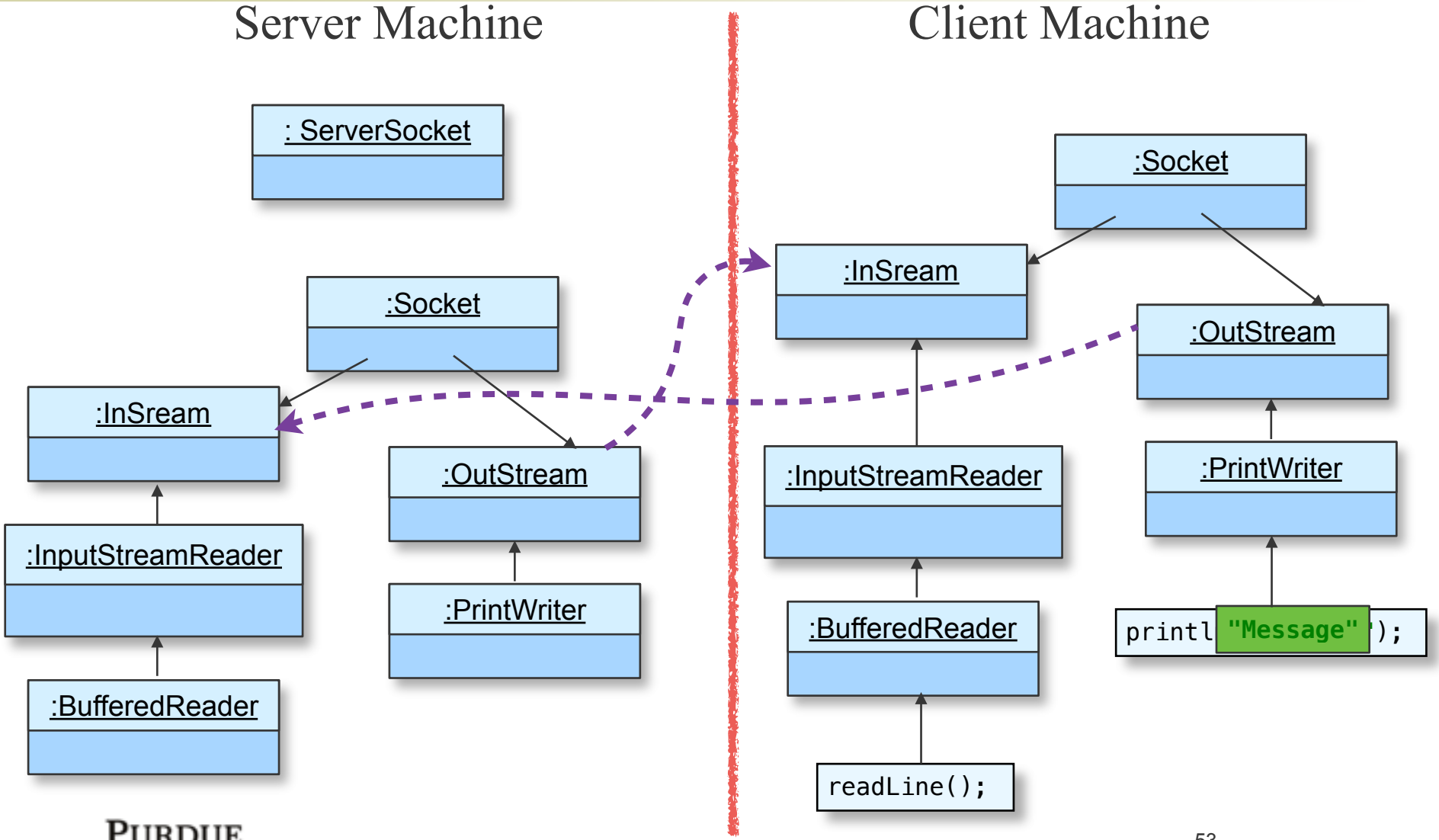
Server Sending a Message



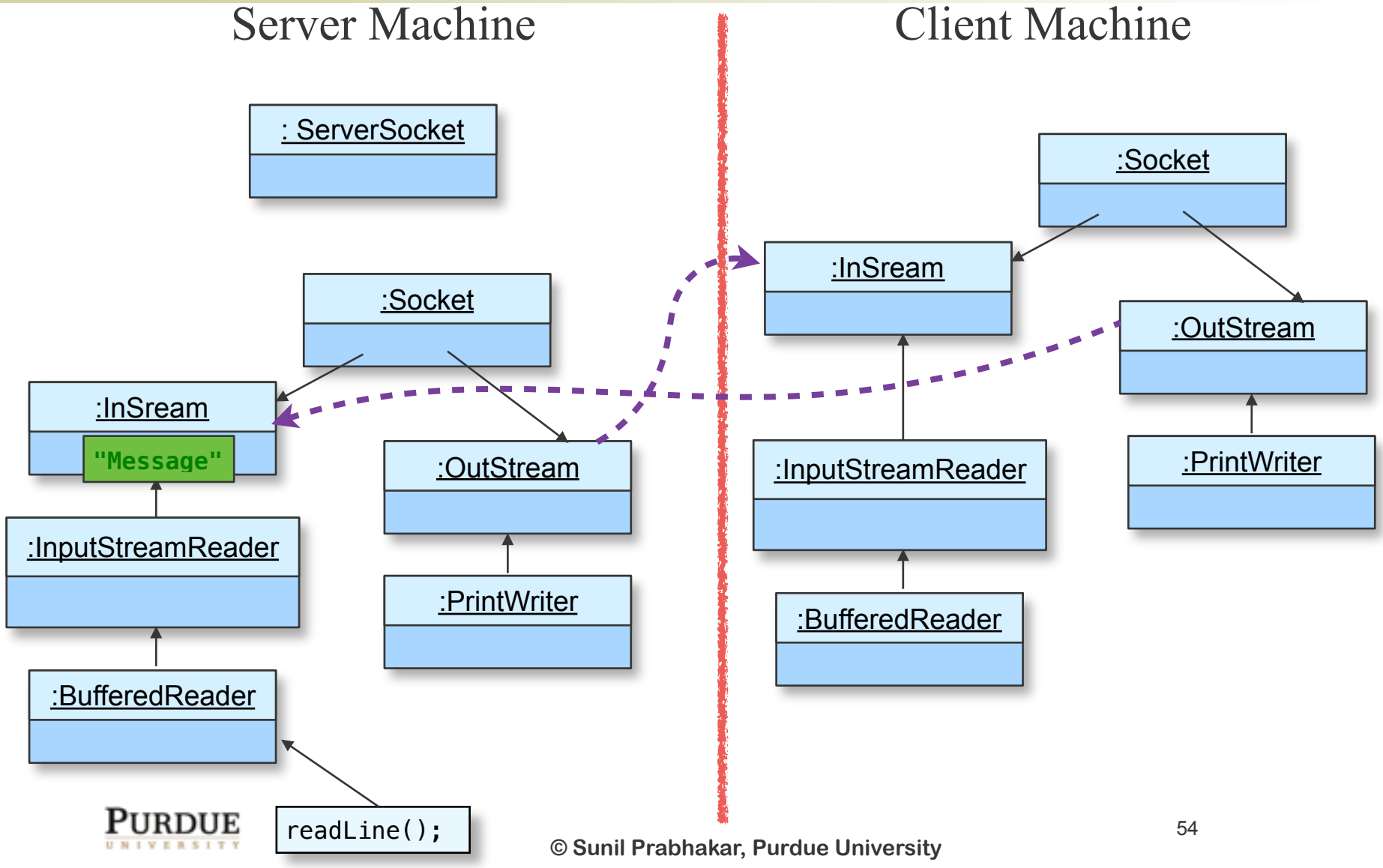
Client Reading a Message



Client Sending a Message



Server Reading a Message



Sockets

- We can terminate a connection by calling close() on the socket.
- For efficiency, output streams on sockets do not immediately send data
  - wait for data to collect
  - we can force immediate transmission by calling flush() on the output stream
- We can test if a socket is closed by calling the isClosed() method

Ports

- Several ports on most computers are reserved for special use
  - E.g., 80 for http; 20 for ftp;
- We must not use these in our programs
  - Safe to use port numbers greater that 4000
  - Must be less than 65535
- Some computers block connections to ports for security reasons
- Hackers try to exploit open ports to attack computers.

Blocking Calls

- Calls to the accept() method of ServerSocket, and readLine() method of Socket are blocking
  - i.e., they do not return until either a client connects, or some data is received on the socket's input stream
- We can prevent blocking by setting a timeout:
  - A SocketTimeoutException is thrown when time runs out
  - Timeout is set using setSoTimeout() with a non zero time out interval in milliseconds

Multiple Clients