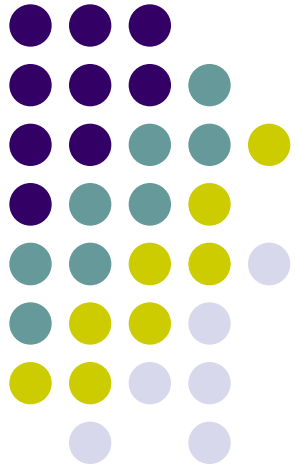
Modular Software Development

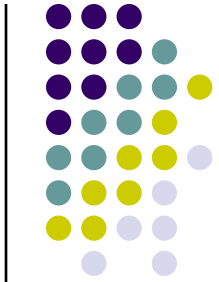
Dr. Chinthana Wimalasuriya

Department of Computer Science & Engineering

March 14, 2013

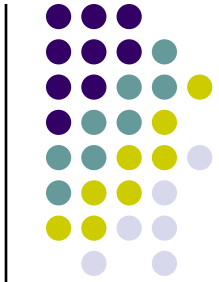


Today's Lecture



- Reading & Writing Data
- File Handling
- Object Serialization
- Using Databases in Java

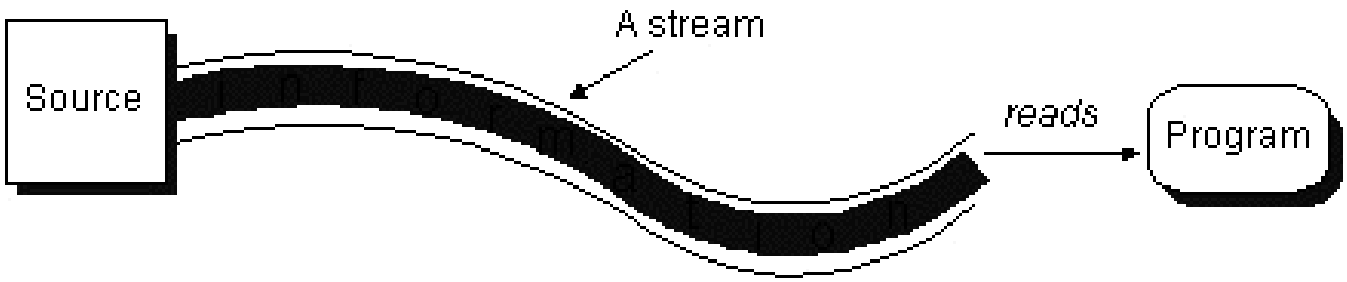
Reading & Writing Data



- Data can come from many sources & go to many destinations
  - Memory
  - Disk
  - Network

- Whatever the source or destination, a stream has to be opened to read/write data

Reading & Writing Data



Reading

Open a Stream

While more Information

Read

Close the Stream

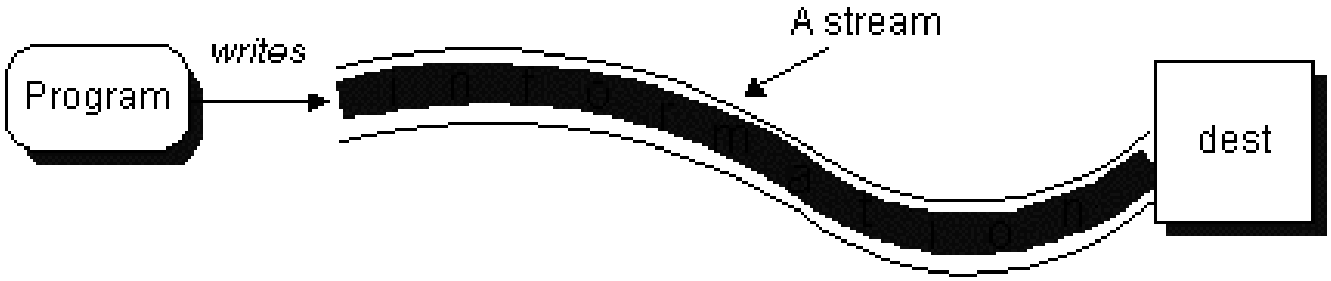
Writing

Open a Stream

While more Information

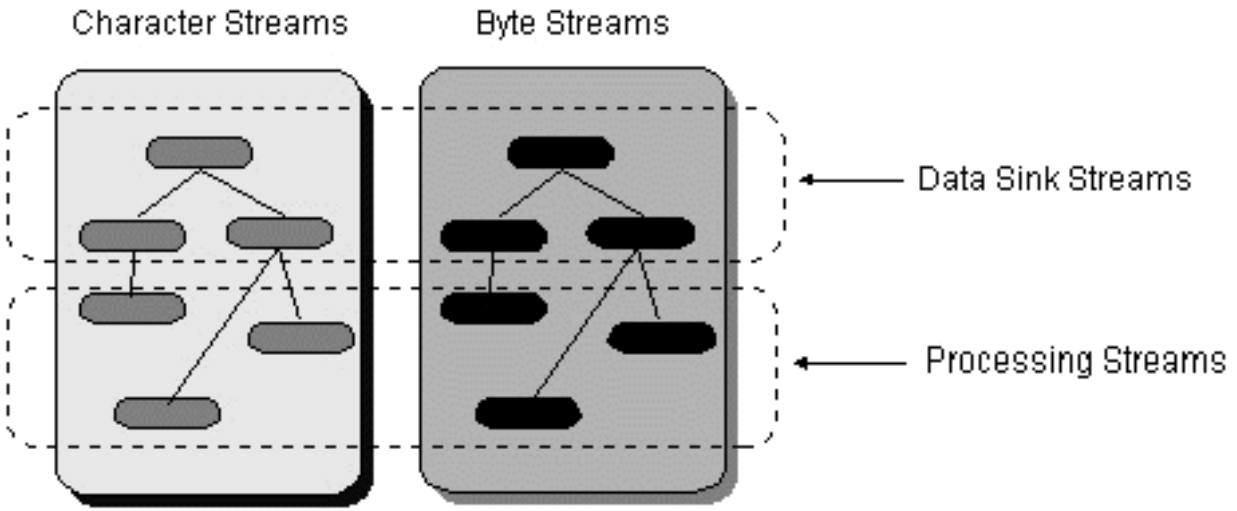
Write

Close the Stream

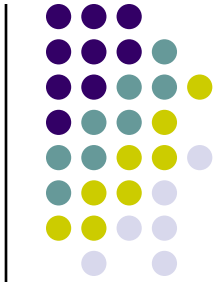


Reading & Writing Data

- java.io package includes these stream classes
- *Character Streams* are used for 16-bit Characters – Uses *Reader* & *Writer* Classes



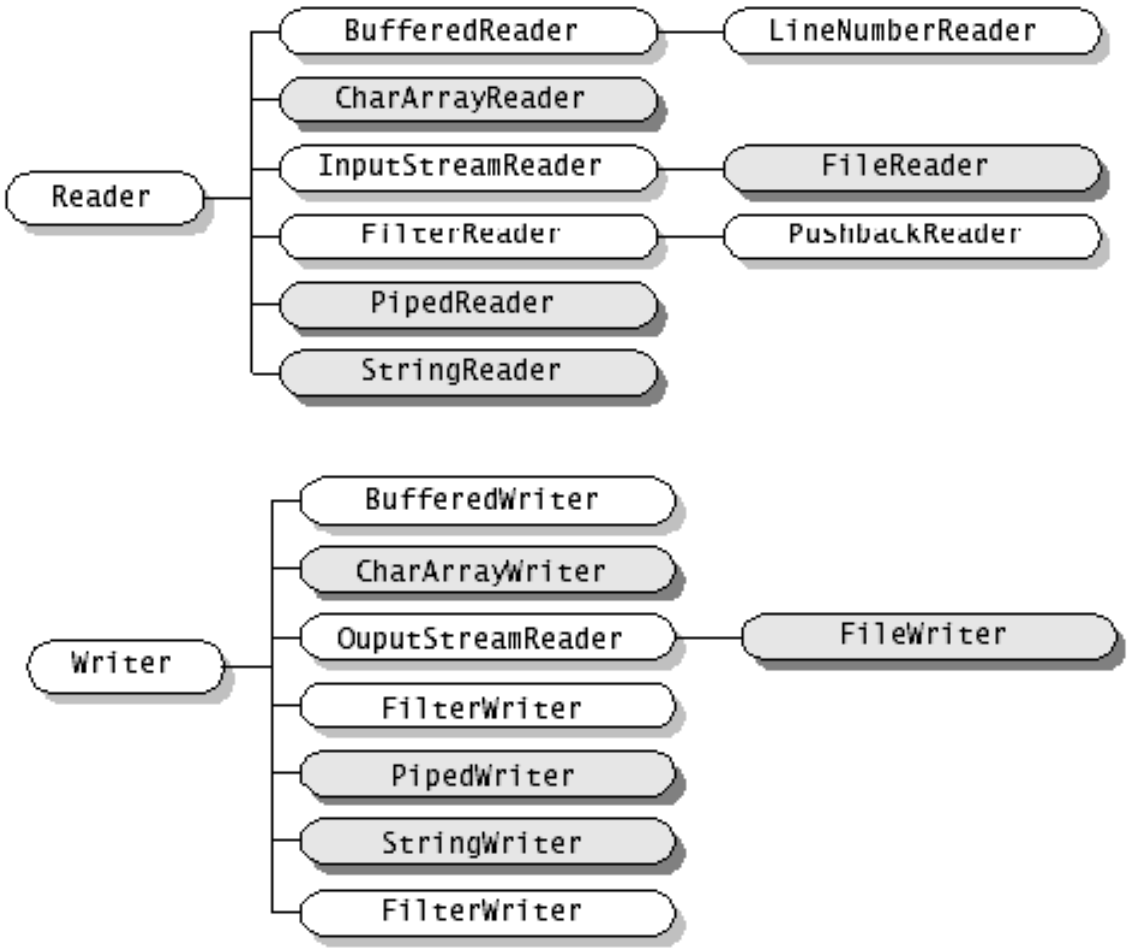
Reading & Writing Data



- *Byte Streams* are used for 8-bit Bytes – Uses *InputStream* & *OutputStream* Classes
  - Used for Image, Sound Data etc.
- Data Sinks
  - Files
  - Memory
  - Pipes
- Processing
  - Buffering
  - Filtering

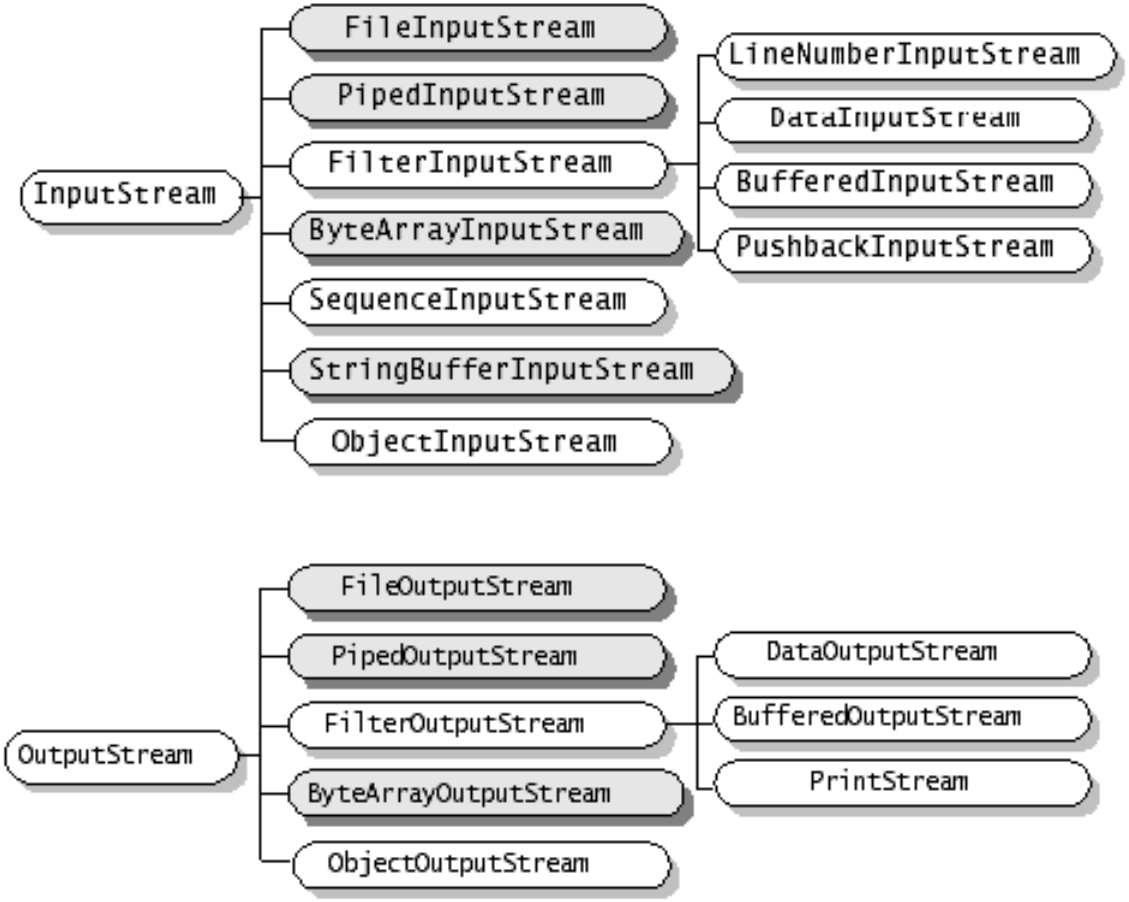
Character Streams

- Reader and Writer are abstract super classes for character streams (16-bit data)
- Sub classes provide specialized behavior

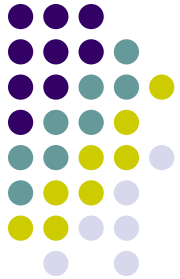


Byte Streams

- InputStream and OutoutStream are abstract super classes for byte streams (8-bit data)
- Sub classes provide specialized behavior



I/O Super Classes



- Reader and InputStream define similar APIs but for different data types

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

Reader

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

InputStream

I/O Super Classes

- Writer and OutputStream define similar APIs but for different data types

int write()
int write(char cbuf[])
int write(char cbuf[], int offset, int length)

Writer

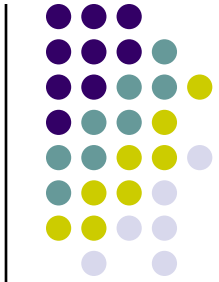
int write()
int write(byte cbuf[])
int write(byte cbuf[], int offset, int length)

OutputStream

File Handling

- To Read from & Write to Files
  - FileReader / FileInputStream
  - FileWriter / FileOutputStream
- The Streams are Opened when they are Created
- They can be Closed by using the close() Method

File Handling – Character Streams



```
import java.io.*;
public class CopyCharacters {
public static void main(String[] args) throws IOException {
    File inputFile = new File("InputFile.txt");
    File outputFile = new File("OutputFile.txt");
    FileReader in = new FileReader(inputFile);
    FileWriter out = new FileWriter(outputFile);
    int c;

    while ((c = in.read()) != -1)      // Read from Stream
        out.write(c);                  // Write to Stream
    in.close();
    out.close();
    }
}
```

Create File Objects

Create File Streams

Close the Streams

File Handling – Byte Streams

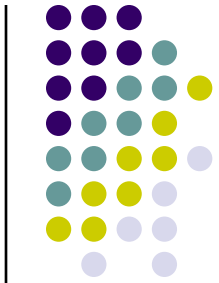
```
import java.io.*;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("picture1.jpg");
        File outputFile = new File("picture2.jpg");

        FileInputStream in = new FileInputStream(inputFile);
        FileOutputStream out = new FileOutputStream(outputFile);
        int c;

        while ((c = in.read()) != -1)  // Read from Stream
           out.write(c);               // Write to Stream

        in.close();
        out.close();
    }
}
```

Create File Objects

Create File Streams

Close the Streams

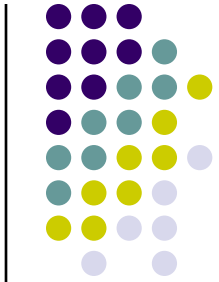
File Handling



- The *File* Object represents the File that is being read or written to

- FileStreams can even be created without the File Object
  - **FileReader**(String fileName)

Character Files



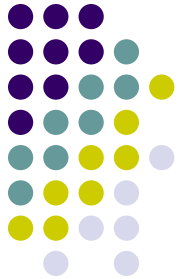
- BufferedReader class can be used for efficient reading of characters, arrays and lines

```
BufferedReader in = new BufferedReader(new FileReader("foo.in"));
```

- BufferedWriter and PrintWriter classes can be used for efficient writing of characters, arrays and lines and other data types

```
BufferedWriter out = new BufferedWriter(new FileWriter("foo.out"));
PrintWriter out
    = new PrintWriter(new BufferedWriter(new FileWriter("foo.out")));
```

Getting User Input in Command Line

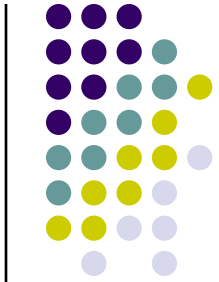


- Read as reading from the standard input device which is treated as an input stream represented by System.in

```
BufferedReader input= new
        BufferedReader(newInputStreamReader(System.in));
System.out.println("Enter the name :"  );
String name =input.readLine();
```

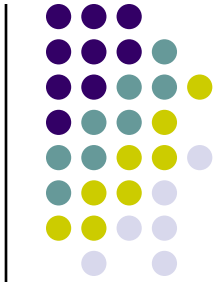
- Throws java.io.IOException

Object Serialization



- To allow to *Read & Write Objects*
- The State of the Object is represented in a *Serialized* form sufficient to reconstruct it later
- Streams to be used
  - ObjectInputStream
  - ObjectOutputStream

Object Serialization

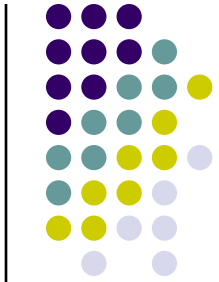


- An Object of any Class that implements the *Serializable Interface* can be serialized
  - public class MyClass implements Serializable {

    ...

    }
- Serializable is an Empty Interface, no methods have to be implemented

Object Serialization

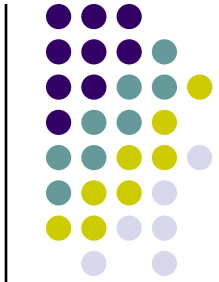


- Writing to an ObjectOutputStream

```
FileOutputStream out = new FileOutputStream(“Time");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

- ObjectOutputStream must be constructed on another Stream

Object Serialization



- Reading from an ObjectInputStream

```
FileInputStream in = new FileInputStream("Time");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

- The objects must be read from the stream in the same order in which they were written

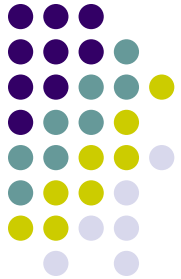
Object Serialization

- Specialized behavior can be provided in serilazation and deserialization by implementing the following methods

```
private void writeObject(java.io.ObjectOutputStream out) throws IOException
private void readObject(java.io.ObjectInputStream in) throws
        IOException, ClassNotFoundException;
```

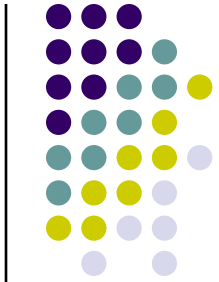
- Object Serialization is used in
  - Remote Method Invocation (RMI) : communication between objects via sockets
  - Lightweight persistence : the archival of an object for use in a later invocation of the same program

JDBC



- Java Database Connectivity
- A *Driver* is needed to connect to a Database
- The Driver depends on the type of Database used
- *SQL* (Structured Query Language) Queries can be used to interact with the Database
- The Package *java.sql* should be imported
  - import java.sql.*;

Set up a Connection



- Load the Driver
  - Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  - Class.forName("jdbc.DriverXYZ");
- Make the Connection
  - String url = "jdbc:odbc:DatabaseName";

    Connection con =
    DriverManager.getConnection(url, “UserName",
    “Password");

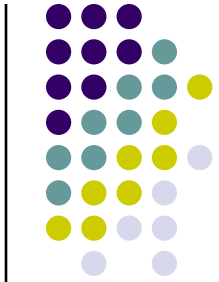
Connecting to a Microsoft Access Database

- DriverManager can be set up to use JDBC:ODBC bridge

  ```
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  ```

- Connection can be established by directly accessing the MS Access database file
- Empty strings can be provided for user name and password

Connecting to a Microsoft Access Database



```
String filename = "D:\\java\\mdbTEST.mdb";
String url = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=";
url += filename.trim() + ";DriverID=22;READONLY=true}";
Connection con = DriverManager.getConnection( url ,"","");
```

- Alternatively a DSN (Data Source Name) can be set up and the connection can be made through the DSN

References

- These slides have been prepared by Ms. Sudanthi Wijewickrema for the “CS201: Principles of Object-Oriented Programming” course module.